mathematical

Release 0.5.0

Mathematical tools for Python

Dominic Davis-Foster

Apr 20, 2021

API REFERENCE

Includes tools for calculating mean, median and standard deviation of rows in data frames, detection of outliers, and statistical calculations

CHAPTER

ONE

INSTALLATION

1.1 from PyPI

```
$ python3 -m pip install mathematical --user
```

1.2 from Anaconda

First add the required channels

```
$ conda config --add channels https://conda.anaconda.org/conda-forge
$ conda config --add channels https://conda.anaconda.org/domdfcoding
```

Then install

```
$ conda install mathematical
```

1.3 from GitHub

```
$ python3 -m pip install git+https://github.com/domdfcoding/mathematical@master --user
```

CHAPTER

TWO

API REFERENCE

2.1 mathematical.data_frames

Mathematical operations for Data Frames.

Data:

ColumnLabelList	Type hint for the column_label_list parameter in the df_*() functions.

Functions:

df_count(row[, column_label_list])	Count the number of occurrences of a non-NaN value in the specified columns of a data frame.
df_data_points(row, column_label_list)	Compile the values for the specified columns in each row into a list.
df_delta(row, left_column, right_column)	Calculate the difference between values in the two columns for each row of a data frame.
df_delta_relative(row, left_column, right_column)	Calculate the relative difference between values in the two columns for each row of a data frame.
df_log(row, column_label_list[, base])	Calculate the logarithm of the values in each row for the specified columns of a data frame.
df_log_stdev(row[, column_label_list])	Calculate the standard deviation of the log10 values in each row for the specified columns of a data frame.
df_mean(row[, column_label_list])	Calculate the mean of each row for the specified columns of a data frame.
df_median(row[, column_label_list])	Calculate the median of each row for the specified columns of a data frame.
df_outliers(row[, column_label_list, ...])	Identify outliers in each row.
df_percentage(row, column_label, total)	Returns the value of the specified column as a percentage of the given total.
df_stdev(row[, column_label_list])	Calculate the standard deviation of each row for the specified columns of a data frame.
set_display_options([desired_width, ...])	Set the display options for numpy and pandas.

ColumnLabelList

Type hint for the column_label_list parameter in the df_*() functions.

Alias of Optional[Sequence[str]]

df_count(*row*, *column_label_list=None*)

Count the number of occurrences of a non-NaN value in the specified columns of a `data frame`.

Do not call this function directly; use it with `df.apply()` instead:

```
data_frame["Count"] = data_frame.apply(
    func=df_count,
    args=[["Bob", "Alice"]],
    axis=1,
    )
```

Parameters

- **row** (`Series`) – Row of the data frame.
- **column_label_list** (`Optional`[`Sequence`[`str`]]) – List of column labels to count occurrences in. Default `None`.

Return type `int`

Returns Count of the occurrences of non-NaN values.

df_data_points(*row*, *column_label_list*)

Compile the values for the specified columns in each row into a list.

Do not call this function directly; use it with `df.apply()` instead:

```
data_frame["Data Points"] = data_frame.apply(
        func=df_data_points,
        args=[["Bob", "Alice"]],
        axis=1,
        )
```

Parameters

- **row** (`Series`) – Row of the data frame.
- **column_label_list** (`Sequence`[`str`]) – List of column labels to calculate standard deviation for.

Return type `List`

Returns The number of data points.

df_delta(*row*, *left_column*, *right_column*)

Calculate the difference between values in the two columns for each row of a `data frame`.

Do not call this function directly; use it with `df.apply()` instead:

```
data_frame["Delta"] = data_frame.apply(
        func=df_delta,
        args=["Bob", "Alice"],
        axis=1,
        )
```

Parameters

- **row** (`Series`) – Row of the data frame.
- **left_column** (`str`)
- **right_column** (`str`)

Return type float

Returns The difference between left_column and right_column.

New in version 0.4.0.

df_delta_relative(*row*, *left_column*, *right_column*)

Calculate the relative difference between values in the two columns for each row of a data frame:

```
(left - right) / right
```

Do not call this function directly; use it with df.apply() instead:

```
data_frame["Rel. Delta"] = data_frame.apply(
        func=df_delta_relative,
        args=["Bob", "Alice"],
        axis=1,
        )
```

Parameters

- **row** (Series) – Row of the data frame.
- **left_column** (str)
- **right_column** (str)

Return type float

Returns The relative difference between left_column and right_column.

New in version 0.4.0.

df_log(*row*, *column_label_list*, *base=10*)

Calculate the logarithm of the values in each row for the specified columns of a data frame.

Do not call this function directly; use it with df.apply() instead:

```
data_frame["Bob Log10"] = data_frame.apply(
        func=df_log,
        args=[["Bob"], 10],
        axis=1,
        )
```

Parameters

- **row** (Series) – Row of the data frame.
- **column_label_list** (Sequence[str]) – List of column labels to calculate log for.
- **base** (float) – The logarithmic base. Default 10.

Return type float

Returns The logarithmic value.

df_log_stdev(*row*, *column_label_list=None*)

Calculate the standard deviation of the log10 values in each row for the specified columns of a data frame.

Do not call this function directly; use it with df.apply() instead:

```
data_frame["Log Stdev"] = data_frame.apply(
        func=df_log_stdev,
        args=[["Bob", "Alice"]],
        axis=1,
        )
```

Parameters

- **row** (Series) – Row of the data frame.
- **column_label_list** (Optional[Sequence[str]]) – List of column labels to calculate standard deviation for. Default None.

Return type float

Returns The standard deviation

df_mean(*row*, *column_label_list=None*)

Calculate the mean of each row for the specified columns of a data frame.

Do not call this function directly; use it with df.apply() instead:

```
data_frame["Mean"] = data_frame.apply(
        func=df_mean,
        args=[["Bob", "Alice"]],
        axis=1,
        )
```

Parameters

- **row** (Series) – Row of the data frame.
- **column_label_list** (Optional[Sequence[str]]) – List of column labels to calculate the mean for. Default None.

Return type float

Returns The mean

df_median(*row*, *column_label_list=None*)

Calculate the median of each row for the specified columns of a data frame.

Do not call this function directly; use it with df.apply() instead:

```
data_frame["Median"] = data_frame.apply(
        func=df_median,
        args=[["Bob", "Alice"]],
        axis=1,
        )
```

Parameters

- **row** (Series) – Row of the data frame.
- **column_label_list** (Optional[Sequence[str]]) – List of column labels to calculate median for. Default None.

Return type float

Returns The median

df_outliers(*row*, *column_label_list=None*, *outlier_mode=1*)

Identify outliers in each row.

This function only returns the list of outliers (if any). If you want the list of values without the outliers see the functions in *mathematical.outliers*.

Do not call this function directly; use it with `df.apply()` instead:

```
data_frame["Outliers"] = data_frame.apply(
        func=df_outliers,
        args=[["Bob", "Alice"]],
        axis=1,
        )
```

Parameters

- **row** (`Series`) – Row of the data frame.
- **column_label_list** (`Optional`[`Sequence`[`str`]]) – List of column labels to determine outliers for. Default `None`.
- **outlier_mode** (`int`) – outlier detection method to use. Default `1`.

The supported outlier modes are:

- `1` or :py:data`mathematical.data_frames.MAD` – Use the Median Absolute Deviation
- `2` or :py:data`mathematical.data_frames.QUARTILES` – Treat values more than `3×` the inter-quartile range away from the upper or lower quartile as outliers.
- `3` or :py:data`mathematical.data_frames.STDEV2` – Treat values more than `rng × stdev` away from mean as outliers

Return type `List`

Returns The outliers.

df_percentage(*row*, *column_label*, *total*)

Returns the value of the specified column as a percentage of the given total.

The total is usually the sum of the specified column.

Do not call this function directly; use it with `df.apply()` instead:

```
data_frame["Bob Percentage"] = data_frame.apply(
        func=df_percentage,
        args=[13, "Bob"],
        axis=1,
        )
```

Parameters

- **row** (`Series`) – Row of the data frame.
- **column_label** (`str`) – The column to calculate percentage for.
- **total** (`float`) – The total value.

Return type `float`

Returns Percentage * 100

df_stdev(*row*, *column_label_list=None*)

Calculate the standard deviation of each row for the specified columns of a `data frame`.

Do not call this function directly; use it with `df.apply()` instead:

```
data_frame["Stdev"] = data_frame.apply(
        func=df_stdev,
        args=[["Bob", "Alice"]],
        axis=1,
        )
```

Parameters

- **row** (`Series`) – Row of the data frame.
- **column_label_list** (`Optional`[`Sequence`[`str`]]) – List of column labels to calculate standard deviation for. Default `None`.

Return type `float`

Returns The standard deviation

set_display_options(*desired_width=300*, *max_columns=15*, *max_rows=20*)

Set the display options for numpy and pandas.

Parameters

- **desired_width** (`int`) – The desired maximum output width, in characters. Default `300`.
- **max_columns** (`int`) – The maximum number of columns to display in a `pandas.DataFrame`. Default `15`.
- **max_rows** (`int`) – The maximum number of rows to display in a `pandas.DataFrame`. Default `20`.

New in version 0.3.0.

2.2 mathematical.linear_regression

Functions for performing linear regression.

Data:

ArrayLike_Float	Type hint for arguments that take either a sequence of floats or a numpy array.

Functions:

linear_regression_perpendicular(x[, y])	Calculate coefficients of a linear regression y = a * x + b.
linear_regression_vertical(x[, y, a, b])	Calculate coefficients of a linear regression y = a * x + b.

ArrayLike_Float

Type hint for arguments that take either a sequence of floats or a numpy array.

Alias of Union[Sequence[float], ndarray]

linear_regression_perpendicular(*x*, *y=None*)

Calculate coefficients of a linear regression y = a * x + b. The fit minimizes *perpendicular* distances between the points and the line.

Parameters

- **x** (Union[Sequence[float], ndarray]) – 1-D array of floats.
- **y** (Union[Sequence[float], ndarray, None]) – 1-D array of floats. Default None.

If *y* is omitted, *x* must be a 2-D array of shape (N, 2).

Return type Tuple[float, float, float, float]

Returns (a, b, r, stderr), where a – slope coefficient, b – free term, r – Peason correlation coefficient, stderr – standard deviation.

linear_regression_vertical(*x*, *y=None*, *a=None*, *b=None*)

Calculate coefficients of a linear regression y = a * x + b. The fit minimizes *vertical* distances between the points and the line.

Parameters

- **x** (Union[Sequence[float], ndarray]) – 1-D array of floats
- **y** (Union[Sequence[float], ndarray, None]) – 1-D array of floats. Default None.
- **a** (Optional[float]) – If specified then the slope coefficient is fixed as this value. Default None.
- **b** (Optional[float]) – If specified then the free term is fixed as this value. Default None.

If *y* is omitted, *x* must be a 2-D array of shape (N, 2).

Return type Tuple[float, float, float, float]

Returns (a, b, r, stderr), where a – slope coefficient, b – free term, r – Pearson correlation coefficient, stderr – standard deviation.

2.3 mathematical.outliers

Outlier detection functions.

Functions:

mad_outliers(dataset[, strip_zero, threshold])	Identifies outlier values using the Median Absolute Deviation.
quartile_outliers(dataset[, strip_zero])	Identifies outlier values that are more than 3× the interquartile range from the upper or lower quartile.
spss_outliers(dataset[, strip_zero, mode])	Identifies outlier values using the IBM SPSS method.
stdev_outlier(dataset[, strip_zero, rng])	Identifies outlier values that are greater than rng × stdev from mean.
two_stdev(dataset[, strip_zero])	Identifies outlier values that are greater than 2× stdev from the mean.

mad_outliers(*dataset*, *strip_zero=True*, *threshold=3*)

Identifies outlier values using the Median Absolute Deviation.

Parameters

- **dataset** (Sequence)
- **strip_zero** (bool) – Default True.
- **threshold** (int) – The multiple of MAD above which values are considered to be outliers. Default 3.

  Leys et al. (2013) make the following recommendations:

  1. In univariate statistics, the Median Absolute Deviation is the most robust dispersion/scale measure in presence of outliers, and hence we strongly recommend the median plus or minus 2.5 times the MAD method for outlier detection.
  2. The threshold should be justified and the justification should clearly state that other concerns than cherry-picking degrees of freedom guided the selection. By default, we suggest a threshold of 2.5 as a reasonable choice.
  3. We encourage researchers to report information about outliers, namely: the number of outliers removed and their value (or at least the distance between outliers and the selected threshold)

  See also:

  https://dipot.ulb.ac.be/dspace/bitstream/2013/139499/1/Leys_MAD_final-libre.pdf

Return type Tuple[List[float], List[float]]

Returns A list of the outlier values, and the remaining data points.

quartile_outliers(*dataset*, *strip_zero=True*)

Identifies outlier values that are more than 3× the inter-quartile range from the upper or lower quartile.

Parameters

- **dataset** (Sequence)
- **strip_zero** (bool) – Default True.

Return type Tuple[List[float], List[float]]

Returns A list of the outlier values, and the remaining data points.

spss_outliers(*dataset*, *strip_zero=True*, *mode='all'*)

Identifies outlier values using the IBM SPSS method.

Outlier values are more than 1.5 × IQR from Q1 or Q3.

"Extreme values" are more than 3 × IQR from Q1 or Q3.

Parameters

- **dataset** (Sequence)
- **mode** (str) – str. Default 'all'.

Return type Tuple[List[float], List[float], List[float]]

Returns A list of extreme outliers, a list of other outliers, and the remaining data points.

stdev_outlier(*dataset*, *strip_zero=True*, *rng=2*)

Identifies outlier values that are greater than rng × stdev from mean.

Parameters

- **dataset** (Sequence)

- **strip_zero** (bool) – Default True.
- **rng** (int) – Default 2.

Return type Tuple[List[float], List[float]]

Returns A list of the outlier values, and the remaining data points.

two_stdev(*dataset*, *strip_zero=True*)

Identifies outlier values that are greater than 2× stdev from the mean.

Parameters

- **dataset** (Sequence)
- **strip_zero** (bool) – Default True.

Return type Tuple[List[float], List[float]]

Returns A list of the outlier values, and the remaining data points.

2.4 mathematical.stats

Functions for calculating statistics.

Functions:

absolute_deviation(x[, axis, center, nan_policy])	Compute the absolute deviations from the median of the data along the given axis.
absolute_deviation_from_median(x[, axis, ...])	Compute the absolute deviation from the median of each point in the data along the given axis, given in terms of the MAD.
d_cohen(sample1, sample2[, which, tail, pooled])	Calculates and returns Cohen's effect size index **d**.
g_durlak_bias(g, n)	Application of Durlak's bias correction to the Hedge's g statistic.
g_hedge(sample1, sample2)	Calculates and returns Hedge's g-Statistic.
interpret_d(d_or_g)	Interpret Cohen's d or Hedge's g values using Table 1 from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3444174/
iqr_none(dataset)	Calculate the interquartile range, excluding NaN, strings, boolean values, and zeros.
mean_none(dataset)	Calculate the mean, excluding NaN, strings, boolean values, and zeros.
median_absolute_deviation(x[, axis, center, ...])	Compute the median absolute deviation of the data along the given axis.
median_none(dataset)	Calculate the median, excluding NaN, strings, boolean values, and zeros.
percentile_none(dataset, percentage)	Calculate the given percentile, excluding NaN, strings, boolean values, and zeros.
pooled_sd(sample1, sample2[, weighted])	Returns the pooled standard deviation.
std_none(dataset[, ddof])	Calculate the standard deviation, excluding NaN, strings, boolean values, and zeros.
within1min(value1, value2)	Returns whether `value2` is within one minute of `value1`.

absolute_deviation(*x*, *axis=0*, *center=<function 'median'>*, *nan_policy='propagate'*)

Compute the absolute deviations from the median of the data along the given axis.

Parameters

- **x** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (Optional[int]) – Axis along which the range is computed. If None, compute the MAD over the entire array. Default 0.
- **center** (Callable) – A function that will return the central value. The default is to use numpy.median. Any user defined function used will need to have the function signature func(arr, axis). Default numpy.median().
- **nan_policy** (Literal['propagate', 'raise', 'omit']) – Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default 'propagate'.

Returns If axis=None, a scalar is returned. If the input contains integers or floats of smaller precision than numpy.float64, then the output data-type is numpy.float64. Otherwise, the output data-type is the same as that of the input.

Return type scalar or ndarray

Overloads

- *absolute_deviation*(**x**, **axis**: None, **center** = ..., **nan_policy** = ...) -> float
- *absolute_deviation*(**x**, **axis**: int = ..., **center** = ..., **nan_policy** = ...) -> ndarray

---

Note: The *center* argument only affects the calculation of the central value around which the MAD is calculated. That is, passing in center=numpy.mean will calculate the MAD around the mean - it will not calculate the *mean* absolute deviation.

---

absolute_deviation_from_median(*x*, *axis=0*, *center=<function 'median'>*, *nan_policy='propagate'*)

Compute the absolute deviation from the median of each point in the data along the given axis, given in terms of the MAD.

See also:

https://eurekastatistics.com/using-the-median-absolute-deviation-to-find-outliers/

Parameters

- **x** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (Optional[int]) – Axis along which the range is computed. If None, compute the MAD over the entire array. Default 0.
- **center** (Callable) – A function that will return the central value. The default is to use numpy.median. Any user defined function used will need to have the function signature func(arr, axis). Default numpy.median().
- **nan_policy** (Literal['propagate', 'raise', 'omit']) – Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default 'propagate'.

Returns If axis=None, a scalar is returned. If the input contains integers or floats of smaller precision than numpy.float64, then the output data-type is numpy.float64. Otherwise, the output data-type is the same as that of the input.

Return type scalar or ndarray

Overloads

- *absolute_deviation_from_median*(**x**, **axis**: None, **center** = ..., **nan_policy** = ...) -> float
- *absolute_deviation_from_median*(**x**, **axis**: int = ..., **center** = ..., **nan_policy** = ...) -> ndarray

---

Note: The *center* argument only affects the calculation of the central value around which the MAD is calculated. That is, passing in `center=numpy.mean` will calculate the MAD around the mean - it will not calculate the *mean* absolute deviation.

---

d_cohen(*sample1*, *sample2*, *which=1*, *tail=1*, *pooled=False*)

Calculates and returns Cohen's effect size index **d**.

See also:

Cohen, J. (1988). Statistical power analysis for the behavioral sciences (2nd Edition). Hillsdale, NJ: Lawrence Erlbaum Associates

Parameters

- **sample1** (Sequence[float]) – datapoints for first sample
- **sample2** (Sequence[float]) – datapoints for second sample
- **which** (Literal[1, 2]) – Use the standard deviation of the first sample (1) or the second sample (2). Default 1.
- **tail** (Literal[1, 2]) – The number of tails to consider. Default 1.
- **pooled** (bool) – Whether to use the pooled standard deviation. Default False.

Return type float

g_durlak_bias(*g*, *n*)

Application of Durlak's bias correction to the Hedge's g statistic.

n = n1+n2

Parameters

- **g** (float) – Hedge's g-Statistic, calculated using *g_hedge()*.
- **n** (float) – The total number of samples in both datasets.

See also:

https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/hedgeg.htm

Return type float

g_hedge(*sample1*, *sample2*)

Calculates and returns Hedge's g-Statistic.

Formula from https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/hedgeg.htm

Parameters

- **sample1** (Sequence[float]) – datapoints for first sample

- **sample2** (Sequence[float]) – datapoints for second sample

Return type float

interpret_d(*d_or_g*)

Interpret Cohen's d or Hedge's g values using Table 1 from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3444174/

Parameters **d_or_g** (float)

Return type str

iqr_none(*dataset*)

Calculate the interquartile range, excluding NaN, strings, boolean values, and zeros.

Parameters **dataset** (Sequence[Union[float, bool, None]]) – A list to calculate iqr from.

Return type float

Returns The interquartile range.

mean_none(*dataset*)

Calculate the mean, excluding NaN, strings, boolean values, and zeros.

Parameters **dataset** (Sequence[Union[float, bool, None]]) – list to calculate mean from

Return type float

Returns mean

median_absolute_deviation(*x*, *axis=0*, *center=<function 'median'>*, *scale=1.4826*, *nan_policy='propagate'*)

Compute the median absolute deviation of the data along the given axis. The median absolute deviation (MAD,[1]) computes the median over the absolute deviations from the median. It is a measure of dispersion similar to the standard deviation, but is more robust to outliers[2]. The MAD of an empty array is `numpy.nan`.

Parameters

- **x** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (Optional[int]) – Axis along which the range is computed. If None, compute the MAD over the entire array. Default `0`.
- **center** (Callable) – A function that will return the central value. The default is to use numpy.median. Any user defined function used will need to have the function signature `func(arr, axis)`. Default `numpy.median()`.
- **scale** (float) – The scaling factor applied to the MAD. The default scale (1.4826) ensures consistency with the standard deviation for normally distributed data. Default `1.4826`.
- **nan_policy** (Literal['propagate', 'raise', 'omit']) – Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default `'propagate'`.

Returns If `axis=None`, a scalar is returned. If the input contains integers or floats of smaller precision than `numpy.float64`, then the output data-type is `numpy.float64`. Otherwise, the output data-type is the same as that of the input.

Return type scalar or ndarray

Overloads

[1] "Median absolute deviation" https://en.wikipedia.org/wiki/Median_absolute_deviation
[2] "Robust measures of scale" https://en.wikipedia.org/wiki/Robust_measures_of_scale

- *median_absolute_deviation*(**x**, **axis**: None, **center** = ..., **scale** = ..., **nan_policy** = ...) -> float
- *median_absolute_deviation*(**x**, **axis**: int = ..., **center** = ..., **scale** = ..., **nan_policy** = ...) -> ndarray

---

Note: The *center* argument only affects the calculation of the central value around which the MAD is calculated. That is, passing in `center=numpy.mean` will calculate the MAD around the mean - it will not calculate the *mean* absolute deviation.

References

Examples

When comparing the behavior of *median_absolute_deviation* with `numpy.std`, the latter is affected when we change a single value of an array to have an outlier value while the MAD hardly changes:

```
>>> import scipy.stats
>>> import mathematical.stats
>>> x = scipy.stats.norm.rvs(size=100, scale=1, random_state=123456)
>>> x.std()
0.9973906394005013
>>> mathematical.stats.median_absolute_deviation(x)
1.2280762773108278
>>> x[0] = 345.6
>>> x.std()
34.42304872314415
>>> mathematical.stats.median_absolute_deviation(x)
1.2340335571164334
Axis handling example:
>>> x = numpy.array([[10, 7, 4], [3, 2, 1]])
>>> x
array([[10,  7,  4], [ 3,  2,  1],])
>>> mathematical.stats.median_absolute_deviation(x)
array([5.1891, 3.7065, 2.2239])
>>> mathematical.stats.median_absolute_deviation(x, axis=None)
2.9652
```

median_none(*dataset*)

Calculate the median, excluding NaN, strings, boolean values, and zeros.

Parameters **dataset** (Sequence[Union[float, bool, None]]) – list to calculate median from

Return type float

Returns standard deviation

percentile_none(*dataset*, *percentage*)

Calculate the given percentile, excluding NaN, strings, boolean values, and zeros.

Parameters

- **dataset** (Sequence[Union[float, bool, None]]) – Sequence to calculate the percentile from.
- **percentage** (float)

Raises ValueError if `dataset` contains fewer than two values

Return type float

Returns The interquartile range.

pooled_sd(*sample1*, *sample2*, *weighted=False*)
Returns the pooled standard deviation.

Parameters

- **sample1** (Sequence[float]) – datapoints for first sample
- **sample2** (Sequence[float]) – datapoints for second sample
- **weighted** (bool) – True for weighted pooled SD. Default False.

See also:

https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/hedgeg.htm

Return type float

std_none(*dataset*, *ddof=1*)
Calculate the standard deviation, excluding NaN, strings, boolean values, and zeros.

Parameters

- **dataset** (Sequence[Union[float, bool, None]]) – list to calculate mean from.
- **ddof** (int) – Means Delta Degrees of Freedom. The divisor used in calculations is N − ddof, where N represents the number of elements. Default 1.

Return type float

Returns standard deviation

within1min(*value1*, *value2*)
Returns whether value2 is within one minute of value1.

Parameters

- **value1** (float) – A time in minutes.
- **value2** (float) – Another time in minutes.

Return type bool

2.5 mathematical.utils

Utilities for mathematical operations.

Classes:

FRange()	Returns a range of floating-point numbers.

Functions:

concatenate_csv(*files[, outfile])	Concatenate multiple CSV files together and return a pandas.DataFrame representing the output.
gcd(a, b)	Returns the GCD (HCF) of a and b using Euclid's Algorithm.

continues on next page

Table 8 – continued from previous page

gcd2(numbers)	Returns the GCD (HCF) of a list of numbers using Euclid's Algorithm.
gcd_array(array)	Returns the GCD for an array of numbers using Euclid's Algorithm.
hcf(a, b)	Returns the GCD (HCF) of `a` and `b` using Euclid's Algorithm.
hcf2(numbers)	Returns the GCD (HCF) of a list of numbers using Euclid's Algorithm.
intdiv(p, q)	Integer divsions which rounds toward zero.
isint(num)	Checks whether a float is an integer value.
lcm(numbers)	Returns the LCM of a list of numbers using Euclid's Algorithm.
log_factorial(x)	Returns the natural logarithm of `x` factorial (`ln(x!)`.
magnitude(x)	Returns the magnitude of the given value.
mod_inverse(a, m)	Returns the modular inverse of `a % m`, which is the number `x` such that `a × x % m = 1`.
nanmean(ls[, dtype])	Returns the mean of the given sequence, ignoring `None` and `numpy.nan` values etc.
nanrsd(ls[, dtype])	Returns the relative standard deviation of the given sequence, ignoring `None` and `numpy.nan` values etc.
nanstd(ls[, dtype])	Returns the standard deviation of the given sequence, ignoring `None` and `numpy.nan` values etc.
remove_zero(inputlist)	Remove zero values from the given list.
represents_int(s)	Checks whether a value can be converted to an `int`.
roman(num)	Retuns the Roman numeral represtation of the given value.
rounders(val_to_round, round_format)	Round a value to the specified number format, e.g.
strip_booleans(ls)	Remove booleans from a list.
strip_none_bool_string(ls)	Remove `None`, boolean and string values from a list.
strip_nonetype(ls)	Remove `None` from a list.
strip_strings(ls)	Remove strings from a list.

class FRange(*stop: float*)
class FRange(*start: float*, *stop: float*, *step: float = '...'*)

Bases: `Sequence[float]`

Returns a range of floating-point numbers.

The arguments to the range constructor may be integers or floats.

Parameters

- **start** – Default `None`.
- **stop** – Default `None`.
- **step** – Default `1.0`.

Raises **ValueError** – If step is zero, or if any value is larger than 1×10^{14}.

New in version 0.2.0.

Methods:

__contains__(o)	Returns whether `o` is in the range.

continues on next page

Table 9 – continued from previous page

__delattr__(key)	Implement `delattr(self, name)`.
__eq__(other)	Return `self == other`.
__getitem__(item)	Returns the value in the range at index `item`.
__iter__()	Iterates over values in the range.
__len__()	Returns the number of values in the range.
__repr__()	Return a string representation of the *FRange*.
__reversed__()	Returns `reversed(self)`.
__setattr__(key, value)	Implement `setattr(self, name)`.
count(value)	Returns `1` if the value is within the range, `0` otherwise.
index(value)	Returns the index of `value` in the range.

Attributes:

start	The value of the `start` parameter (or `0.0` if the parameter was not supplied)
step	The value of the `step` parameter (or `1.0` if the parameter was not supplied)
stop	The value of the `stop` parameter

__contains__(*o*)
Returns whether `o` is in the range.

Parameters **o** (`object`)

Return type `bool`

__delattr__(*key*)
Implement `delattr(self, name)`.

__eq__(*other*)
Return `self == other`.

Return type `bool`

__getitem__(*item*)
Returns the value in the range at index `item`.

Parameters **item**

Overloads

- *__getitem__*(**i**: `int`) -> `int`
- *__getitem__*(**s**: `slice`) -> *FRange*

__iter__()
Iterates over values in the range.

Return type `Iterator`[`float`]

__len__()
Returns the number of values in the range.

Return type `int`

__repr__()
Return a string representation of the *FRange*.

Return type `str`

__reversed__()
Returns `reversed(self)`.

Return type `Iterator[float]`

__setattr__(*key*, *value*)
Implement `setattr(self, name)`.

count(*value*)
Returns `1` if the value is within the range, `0` otherwise.

Parameters **value** (`float`)

Return type `int`

index(*value*)
Returns the index of `value` in the range.

Parameters **value** (`float`)

Raises **ValueError** – if the value is not in the range.

Return type `int`

start
Type: `float`

The value of the `start` parameter (or `0.0` if the parameter was not supplied)

step
Type: `float`

The value of the `step` parameter (or `1.0` if the parameter was not supplied)

stop
Type: `float`

The value of the `stop` parameter

concatenate_csv(**files*, *outfile=None*)
Concatenate multiple CSV files together and return a `pandas.DataFrame` representing the output.

Parameters

- ***files** – The files to concatenate.
- **outfile** (`Union[str, Path, PathLike, None]`) – The file to save the output as. If `None` no file will be saved. Default `None`.

Return type `DataFrame`

Returns A `pandas.DataFrame` containing the concatenated CSV data.

New in version 0.3.0.

gcd(*a*, *b*)
Returns the GCD (HCF) of `a` and `b` using Euclid's Algorithm.

Parameters

- **a** (`int`)
- **b** (`int`)

Return type `int`

gcd2(*numbers*)
Returns the GCD (HCF) of a list of numbers using Euclid's Algorithm.

Parameters **numbers** (Sequence[int])

Return type int

gcd_array(*array*)
Returns the GCD for an array of numbers using Euclid's Algorithm.

Based on https://www.geeksforgeeks.org/python-program-for-gcd-of-more-than-two-or-array-numbers/

Parameters **array**

Return type float

hcf(*a*, *b*)
Returns the GCD (HCF) of a and b using Euclid's Algorithm.

Parameters

- **a** (int)
- **b** (int)

Return type int

hcf2(*numbers*)
Returns the GCD (HCF) of a list of numbers using Euclid's Algorithm.

Parameters **numbers** (Sequence[int])

Return type int

intdiv(*p*, *q*)
Integer divsions which rounds toward zero.

Examples >>> intdiv(3, 2) 1 >>> intdiv(-3, 2) -1 >>> -3 // 2 -2

Return type int

isint(*num*)
Checks whether a float is an integer value.

Note: This function only works with floating-point numbers

Parameters **num** (float) – value to check

Return type bool

lcm(*numbers*)
Returns the LCM of a list of numbers using Euclid's Algorithm.

Parameters **numbers** (Sequence[int])

Return type float

log_factorial(*x*)
Returns the natural logarithm of x factorial (ln(x!).

Parameters **x** (float)

Return type float

magnitude(*x*)
Returns the magnitude of the given value.

Parameters **x** (float) – Numerical value to find the magnitude of.

Changed in version 0.2.0: Now returns the absolute magnitude of negative numbers.

Return type int

mod_inverse(*a*, *m*)
Returns the modular inverse of `a % m`, which is the number `x` such that `a × x % m = 1`.

Parameters

- **a** (int)
- **m** (int)

Return type Optional[float]

nanmean(*ls*, *dtype=<class 'float'>*)
Returns the mean of the given sequence, ignoring None and `numpy.nan` values etc.

Similar to numpy.nanmean except it handles None.

Parameters

- **ls** (Sequence[Any])
- **dtype** – Default float.

Return type float

nanrsd(*ls*, *dtype=<class 'float'>*)
Returns the relative standard deviation of the given sequence, ignoring None and `numpy.nan` values etc.

Parameters

- **ls** (Sequence[Any])
- **dtype** – Default float.

Return type float

nanstd(*ls*, *dtype=<class 'float'>*)
Returns the standard deviation of the given sequence, ignoring None and `numpy.nan` values etc.

Similar to numpy.nanstd except it handles None.

Parameters

- **ls** (Sequence[Any])
- **dtype** – Default float.

Return type float

remove_zero(*inputlist*)
Remove zero values from the given list.

Also removes False and None.

Parameters **inputlist** (Sequence[Union[float, bool, None]]) – list to remove zero values from

Return type List[float]

represents_int(*s*)
Checks whether a value can be converted to an int.

Parameters **s** (Any) – value to check

Return type bool

roman(*num*)
Retuns the Roman numeral represtation of the given value.

Examples:

```
>>> roman(4)
'IV'
>>> roman(17)
'XVII'
```

Return type str

rounders(*val_to_round*, *round_format*)
Round a value to the specified number format, e.g. "0.000" for three decimal places.

Parameters

- **val_to_round** (Union[str, float, Decimal]) – The value to round
- **round_format** (str) – The rounding format

Return type Decimal

strip_booleans(*ls*)
Remove booleans from a list.

Parameters **ls** (Sequence[Any]) – the list to remove booleans from.

Return type List

Returns The list without boolean values.

strip_none_bool_string(*ls*)
Remove None, boolean and string values from a list.

Parameters **ls** (Sequence) – The list to remove values from.

Return type List

strip_nonetype(*ls*)
Remove None from a list.

Parameters **ls** (Sequence[Any]) – the list to remove None from.

Return type List

Returns The list without None values.

strip_strings(*ls*)
Remove strings from a list.

Parameters **ls** (Sequence[Any]) – the list to remove strings from.

Return type List

Returns The list without strings.

CHAPTER

THREE

CONTRIBUTING

3.1 Overview

`mathematical` uses tox to automate testing and packaging, and pre-commit to maintain code quality.

Install `pre-commit` with `pip` and install the git hook:

```
$ python -m pip install pre-commit
$ pre-commit install
```

3.2 Coding style

formate is used for code formatting.

It can be run manually via `pre-commit`:

```
$ pre-commit run formate -a
```

Or, to run the complete autoformatting suite:

```
$ pre-commit run -a
```

3.3 Automated tests

Tests are run with `tox` and `pytest`. To run tests for a specific Python version, such as Python 3.6:

```
$ tox -e py36
```

To run tests for all Python versions, simply run:

```
$ tox
```

3.4 Type Annotations

Type annotations are checked using `mypy`. Run `mypy` using `tox`:

```
$ tox -e mypy
```

3.5 Build documentation locally

The documentation is powered by Sphinx. A local copy of the documentation can be built with `tox`:

```
$ tox -e docs
```

3.6 Downloading source code

The `mathematical` source code is available on GitHub, and can be accessed from the following URL: https://github.com/domdfcoding/mathematical

If you have `git` installed, you can clone the repository with the following command:

```
$ git clone https://github.com/domdfcoding/mathematical"
> Cloning into 'mathematical'...
> remote: Enumerating objects: 47, done.
> remote: Counting objects: 100% (47/47), done.
> remote: Compressing objects: 100% (41/41), done.
> remote: Total 173 (delta 16), reused 17 (delta 6), pack-reused 126
> Receiving objects: 100% (173/173), 126.56 KiB | 678.00 KiB/s, done.
> Resolving deltas: 100% (66/66), done.
```

Alternatively, the code can be downloaded in a 'zip' file by clicking:
Clone or download –> Download Zip

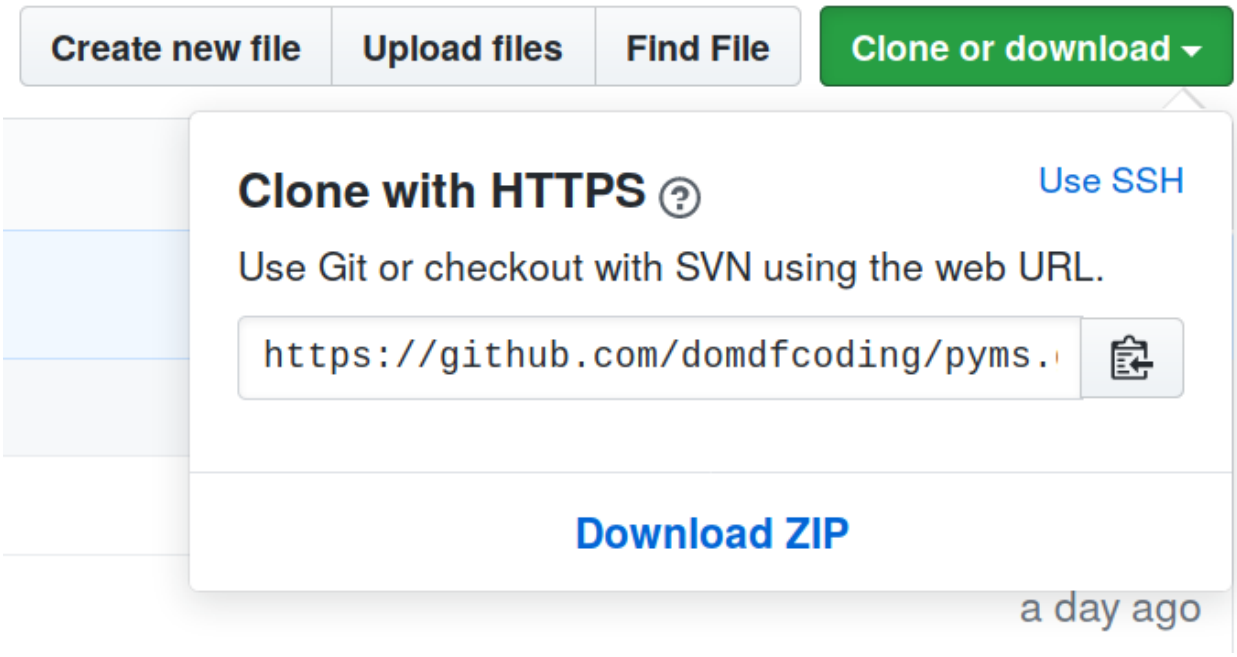

Fig. 1: Downloading a 'zip' file of the source code

3.6.1 Building from source

The recommended way to build `mathematical` is to use tox:

```
$ tox -e build
```

The source and wheel distributions will be in the directory `dist`.

If you wish, you may also use pep517.build or another **PEP 517**-compatible build tool.

PYTHON MODULE INDEX

m

PYTHON MODULE INDEX

m

INDEX

Symbols

A

C

D

F

G

H

I

L

M

N

P

Q

R

S

T

W