Notes

A minimum of 4 periods is required for the rolling calculation.

Examples

The example below will show a rolling calculation with a window size of four matching the equivalent function call using *scipy.stats*.

```
>>> arr = [1, 2, 3, 4, 999]
>>> import scipy.stats
>>> print(f"{scipy.stats.kurtosis(arr[:-1], bias=False):.6f}")
-1.200000
>>> print(f"{scipy.stats.kurtosis(arr[1:], bias=False):.6f}")
3.999946
>>> s = pd.Series(arr)
>>> s.rolling(4).kurt()
0         NaN
1         NaN
2         NaN
3   -1.200000
4    3.999946
dtype: float64
```

pandas.core.window.rolling.Rolling.apply

`Rolling.`**`apply`**(*self*, *func*, *raw=False*, *engine='cython'*, *engine_kwargs=None*, *args=None*, *kwargs=None*)

The rolling function's apply function.

Parameters

func [function] Must produce a single value from an ndarray input if `raw=True` or a single value from a Series if `raw=False`. Can also accept a Numba JIT function with `engine='numba'` specified.

Changed in version 1.0.0.

raw [bool, default None]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

engine [str, default 'cython']

- `'cython'` : Runs rolling apply through C-extensions from cython.
- `'numba'` : Runs rolling apply through JIT compiled code from numba. Only available when `raw` is set to `True`.

  New in version 1.0.0.

engine_kwargs [dict, default None]

- For `'cython'` engine, there are no accepted `engine_kwargs`
- For `'numba'` engine, the engine can accept `nopython`, `nogil` and `parallel` dictionary keys. The values must either be `True` or `False`. The default `engine_kwargs` for the `'numba'` engine is `{'nopython': True,`

`'nogil': False, 'parallel': False}` and will be applied to both the `func` and the `apply` rolling aggregation.

New in version 1.0.0.

args [tuple, default None] Positional arguments to be passed into func.

kwargs [dict, default None] Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

`Series.rolling` Series rolling.
`DataFrame.rolling` DataFrame rolling.

Notes

See *Rolling Apply* for extended documentation and performance considerations for the Numba engine.

pandas.core.window.rolling.Rolling.aggregate

`Rolling.`**`aggregate`**(*self*, *func*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series/Dataframe or when passed to Series/Dataframe.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

`Series.rolling`
`DataFrame.rolling`

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'])
>>> df
          A         B         C
0 -2.385977 -0.102758  0.438822
1 -1.004295  0.905829 -0.954544
2  0.735167 -0.165272 -1.619346
3 -0.702657 -1.340923 -0.706334
4 -0.246845  0.211596 -0.901819
5  2.463718  3.157577 -1.380906
6 -1.142255  2.340594 -0.039875
7  1.396598 -1.647453  1.677227
8 -0.543425  1.761277 -0.220481
9 -0.640505  0.289374 -1.550670
```

```
>>> df.rolling(3).sum()
          A         B         C
0       NaN       NaN       NaN
1       NaN       NaN       NaN
2 -2.655105  0.637799 -2.135068
3 -0.971785 -0.600366 -3.280224
4 -0.214334 -1.294599 -3.227500
5  1.514216  2.028250 -2.989060
6  1.074618  5.709767 -2.322600
7  2.718061  3.850718  0.256446
8 -0.289082  2.454418  1.416871
9  0.212668  0.403198 -0.093924
```

```
>>> df.rolling(3).agg({'A':'sum', 'B':'min'})
          A         B
0       NaN       NaN
1       NaN       NaN
2 -2.655105 -0.165272
3 -0.971785 -1.340923
4 -0.214334 -1.340923
5  1.514216 -1.340923
6  1.074618  0.211596
7  2.718061 -1.647453
8 -0.289082 -1.647453
9  0.212668 -1.647453
```

pandas.core.window.rolling.Rolling.quantile

Rolling.**quantile**(*self*, *quantile*, *interpolation='linear'*, ***kwargs*)

Calculate the rolling quantile.

Parameters

quantile [float] Quantile to compute. 0 <= quantile <= 1.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.23.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i* + (*j* - *i*) * *fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

****kwargs** For compatibility with other rolling methods. Has no effect on the result.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation.

See also:

Series.quantile Computes value at the given quantile over all data in Series.
DataFrame.quantile Computes values at the given quantile over requested axis in DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
0    NaN
1    1.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

pandas.core.window.rolling.Window.mean

Window.**mean**(*self*, **args*, ***kwargs*)

Calculate the window mean of the values.

Parameters

***args** Under Review.

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the window calculation.

See also:

Series.window Calling object with Series data.
DataFrame.window Calling object with DataFrames.
Series.mean Equivalent method for Series.
DataFrame.mean Equivalent method for DataFrame.

Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0    NaN
1    NaN
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.rolling.Window.sum

Window.**sum**(*self*, **args*, ***kwargs*)

Calculate window sum of given DataFrame or Series.

Parameters

***args, **kwargs** For compatibility with other window methods. Has no effect on the computed value.

Returns

Series or DataFrame Same type as the input, with the same index, containing the window sum.

See also:

Series.sum Reducing sum for Series.
DataFrame.sum Reducing sum for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0     NaN
1     NaN
2     6.0
3     9.0
4    12.0
dtype: float64
```

```
>>> s.expanding(3).sum()
0     NaN
1     NaN
2     6.0
3    10.0
4    15.0
dtype: float64
```

```
>>> s.rolling(3, center=True).sum()
0     NaN
1     6.0
2     9.0
3    12.0
4     NaN
dtype: float64
```

For DataFrame, each window sum is computed column-wise.

```
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A   B
0  1   1
1  2   4
2  3   9
3  4  16
4  5  25
```

```
>>> df.rolling(3).sum()
      A     B
0   NaN   NaN
1   NaN   NaN
2   6.0  14.0
3   9.0  29.0
4  12.0  50.0
```

pandas.core.window.rolling.Window.var

Window.**var**(*self*, *ddof=1*, **args*, ***kwargs*)

Calculate unbiased window variance.

New in version 1.0.0.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the window calculation.

See also:

`Series.window` Calling object with Series data.
`DataFrame.window` Calling object with DataFrames.
`Series.var` Equivalent method for Series.
`DataFrame.var` Equivalent method for DataFrame.
`numpy.var` Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in `Series.var()` is different than the default *ddof* of 0 in `numpy.var()`.

A minimum of 1 period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0         NaN
1         NaN
2    0.333333
3    1.000000
4    1.000000
5    1.333333
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).var()
0         NaN
1         NaN
2    0.333333
3    0.916667
4    0.800000
5    0.700000
6    0.619048
dtype: float64
```

pandas.core.window.rolling.Window.std

Window.**std**(*self*, *ddof=1*, **args*, ***kwargs*)

Calculate window standard deviation.

New in version 1.0.0.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the window calculation.

See also:

`Series.window` Calling object with Series data.
`DataFrame.window` Calling object with DataFrames.
`Series.std` Equivalent method for Series.
`DataFrame.std` Equivalent method for DataFrame.
`numpy.std` Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in Series.std is different than the default *ddof* of 0 in numpy.std.

A minimum of one period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0         NaN
1         NaN
2    0.577350
3    1.000000
4    1.000000
5    1.154701
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).std()
0         NaN
1         NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

3.10.2 Standard expanding window functions

Expanding.count(self, **kwargs)	The expanding count of any non-NaN observations inside the window.
Expanding.sum(self, *args, **kwargs)	Calculate expanding sum of given DataFrame or Series.
Expanding.mean(self, *args, **kwargs)	Calculate the expanding mean of the values.
Expanding.median(self, **kwargs)	Calculate the expanding median.
Expanding.var(self[, ddof])	Calculate unbiased expanding variance.
Expanding.std(self[, ddof])	Calculate expanding standard deviation.
Expanding.min(self, *args, **kwargs)	Calculate the expanding minimum.
Expanding.max(self, *args, **kwargs)	Calculate the expanding maximum.
Expanding.corr(self[, other, pairwise])	Calculate expanding correlation.
Expanding.cov(self[, other, pairwise, ddof])	Calculate the expanding sample covariance.
Expanding.skew(self, **kwargs)	Unbiased expanding skewness.
Expanding.kurt(self, **kwargs)	Calculate unbiased expanding kurtosis.
Expanding.apply(self, func[, raw, args, kwargs])	The expanding function's apply function.
Expanding.aggregate(self, func, *args, **kwargs)	Aggregate using one or more operations over the specified axis.
Expanding.quantile(self, quantile[, ...])	Calculate the expanding quantile.

pandas.core.window.expanding.Expanding.count

Expanding.**count**(*self*, ***kwargs*)

The expanding count of any non-NaN observations inside the window.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.
DataFrame.expanding Calling object with DataFrames.
DataFrame.count Count of the full DataFrame.

Examples

```
>>> s = pd.Series([2, 3, np.nan, 10])
>>> s.rolling(2).count()
0    1.0
1    2.0
2    1.0
3    1.0
dtype: float64
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.rolling(4).count()
0    1.0
1    2.0
```

(continues on next page)

(continued from previous page)

```
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.expanding.Expanding.sum

Expanding.**sum**(*self*, **args*, ***kwargs*)

Calculate expanding sum of given DataFrame or Series.

Parameters

***args, **kwargs** For compatibility with other expanding methods. Has no effect on the computed value.

Returns

Series or DataFrame Same type as the input, with the same index, containing the expanding sum.

See also:

Series.sum Reducing sum for Series.
DataFrame.sum Reducing sum for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0     NaN
1     NaN
2     6.0
3     9.0
4    12.0
dtype: float64
```

```
>>> s.expanding(3).sum()
0     NaN
1     NaN
2     6.0
3    10.0
4    15.0
dtype: float64
```

```
>>> s.rolling(3, center=True).sum()
0     NaN
1     6.0
2     9.0
```

(continues on next page)

(continued from previous page)

```
3    12.0
4     NaN
dtype: float64
```

For DataFrame, each expanding sum is computed column-wise.

```
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A   B
0  1   1
1  2   4
2  3   9
3  4  16
4  5  25
```

```
>>> df.rolling(3).sum()
      A     B
0   NaN   NaN
1   NaN   NaN
2   6.0  14.0
3   9.0  29.0
4  12.0  50.0
```

pandas.core.window.expanding.Expanding.mean

`Expanding.`**`mean`**(*self*, **args*, ***kwargs*)

Calculate the expanding mean of the values.

Parameters

***args** Under Review.

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

`Series.expanding` Calling object with Series data.
`DataFrame.expanding` Calling object with DataFrames.
`Series.mean` Equivalent method for Series.
`DataFrame.mean` Equivalent method for DataFrame.

Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0    NaN
1    NaN
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.expanding.Expanding.median

Expanding.**median**(*self*, ***kwargs*)

Calculate the expanding median.

Parameters

****kwargs** For compatibility with other expanding methods. Has no effect on the computed median.

Returns

Series or DataFrame Returned type is the same as the original object.

See also:

Series.expanding Calling object with Series data.
DataFrame.expanding Calling object with DataFrames.
Series.median Equivalent method for Series.
DataFrame.median Equivalent method for DataFrame.

Examples

Compute the rolling median of a series with a window size of 3.

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.rolling(3).median()
0    NaN
1    NaN
2    1.0
3    2.0
4    3.0
dtype: float64
```

pandas.core.window.expanding.Expanding.var

Expanding.**var**(*self*, *ddof=1*, **args*, ***kwargs*)

Calculate unbiased expanding variance.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.
DataFrame.expanding Calling object with DataFrames.
Series.var Equivalent method for Series.
DataFrame.var Equivalent method for DataFrame.
numpy.var Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in `Series.var()` is different than the default *ddof* of 0 in `numpy.var()`.

A minimum of 1 period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0         NaN
1         NaN
2    0.333333
3    1.000000
4    1.000000
5    1.333333
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).var()
0         NaN
1         NaN
2    0.333333
3    0.916667
4    0.800000
5    0.700000
6    0.619048
dtype: float64
```

pandas.core.window.expanding.Expanding.std

`Expanding.std(self, ddof=1, *args, **kwargs)`

Calculate expanding standard deviation.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.
DataFrame.expanding Calling object with DataFrames.
Series.std Equivalent method for Series.
DataFrame.std Equivalent method for DataFrame.
numpy.std Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in Series.std is different than the default *ddof* of 0 in numpy.std.

A minimum of one period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0         NaN
1         NaN
2    0.577350
3    1.000000
4    1.000000
5    1.154701
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).std()
0         NaN
1         NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

pandas.core.window.expanding.Expanding.min

Expanding.**min**(*self*, **args*, ***kwargs*)

Calculate the expanding minimum.

Parameters

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with a Series.
DataFrame.expanding Calling object with a DataFrame.
Series.min Similar method for Series.
DataFrame.min Similar method for DataFrame.

Examples

Performing a rolling minimum with a window size of 3.

```
>>> s = pd.Series([4, 3, 5, 2, 6])
>>> s.rolling(3).min()
0    NaN
1    NaN
2    3.0
3    2.0
4    2.0
dtype: float64
```

pandas.core.window.expanding.Expanding.max

Expanding.**max**(*self*, **args*, ***kwargs*)

Calculate the expanding maximum.

Parameters

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.
DataFrame.expanding DataFrame expanding.

pandas.core.window.expanding.Expanding.corr

Expanding.**corr**(*self*, *other=None*, *pairwise=None*, ***kwargs*)

Calculate expanding correlation.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self.

pairwise [bool, default None] Calculate pairwise combinations of columns within a DataFrame. If *other* is not specified, defaults to *True*, otherwise defaults to *False*. Not relevant for *Series*.

****kwargs** Unused.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.
DataFrame.expanding Calling object with DataFrames.
Series.corr Equivalent method for Series.
DataFrame.corr Equivalent method for DataFrame.
expanding.cov Similar method to calculate covariance.
numpy.corrcoef NumPy Pearson's correlation calculation.

Notes

This function uses Pearson's definition of correlation (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient).

When *other* is not specified, the output will be self correlation (e.g. all 1's), except for `DataFrame` inputs with *pairwise* set to *True*.

Function will return `NaN` for correlations of equal valued sequences; this is the result of a 0/0 division error.

When *pairwise* is set to *False*, only matching columns between *self* and *other* will be used.

When *pairwise* is set to *True*, the output will be a MultiIndex DataFrame with the original index on the first level, and the *other* DataFrame columns on the second level.

In the case of missing elements, only complete pairwise observations will be used.

Examples

The below example shows a rolling calculation with a window size of four matching the equivalent function call using `numpy.corrcoef()`.

```
>>> v1 = [3, 3, 3, 5, 8]
>>> v2 = [3, 4, 4, 4, 8]
>>> # numpy returns a 2X2 array, the correlation coefficient
>>> # is the number at entry [0][1]
>>> print(f"{np.corrcoef(v1[:-1], v2[:-1])[0][1]:.6f}")
0.333333
>>> print(f"{np.corrcoef(v1[1:], v2[1:])[0][1]:.6f}")
0.916949
>>> s1 = pd.Series(v1)
>>> s2 = pd.Series(v2)
>>> s1.rolling(4).corr(s2)
0         NaN
1         NaN
2         NaN
3    0.333333
4    0.916949
dtype: float64
```

The below example shows a similar rolling calculation on a DataFrame using the pairwise option.

```
>>> matrix = np.array([[51., 35.], [49., 30.], [47., 32.],    [46., 31.], [50.,␣
↪36.]])
>>> print(np.corrcoef(matrix[:-1,0], matrix[:-1,1]).round(7))
[[1.         0.6263001]
 [0.6263001  1.       ]]
>>> print(np.corrcoef(matrix[1:,0], matrix[1:,1]).round(7))
[[1.         0.5553681]
 [0.5553681  1.       ]]
>>> df = pd.DataFrame(matrix, columns=['X','Y'])
>>> df
      X     Y
0  51.0  35.0
1  49.0  30.0
2  47.0  32.0
3  46.0  31.0
4  50.0  36.0
```

(continues on next page)

(continued from previous page)

```
>>> df.rolling(4).corr(pairwise=True)
              X         Y
0 X         NaN       NaN
  Y         NaN       NaN
1 X         NaN       NaN
  Y         NaN       NaN
2 X         NaN       NaN
  Y         NaN       NaN
3 X    1.000000  0.626300
  Y    0.626300  1.000000
4 X    1.000000  0.555368
  Y    0.555368  1.000000
```

pandas.core.window.expanding.Expanding.cov

Expanding.**cov**(*self*, *other=None*, *pairwise=None*, *ddof=1*, ***kwargs*)

Calculate the expanding sample covariance.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.
DataFrame.expanding DataFrame expanding.

pandas.core.window.expanding.Expanding.skew

Expanding.**skew**(*self*, ***kwargs*)

Unbiased expanding skewness.

Parameters

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.
DataFrame.expanding DataFrame expanding.

pandas.core.window.expanding.Expanding.kurt

Expanding.**kurt**(*self*, ***kwargs*)

Calculate unbiased expanding kurtosis.

This function uses Fisher's definition of kurtosis without bias.

Parameters

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.
DataFrame.expanding Calling object with DataFrames.
Series.kurt Equivalent method for Series.
DataFrame.kurt Equivalent method for DataFrame.
scipy.stats.skew Third moment of a probability density.
scipy.stats.kurtosis Reference SciPy method.

Notes

A minimum of 4 periods is required for the expanding calculation.

Examples

The example below will show an expanding calculation with a window size of four matching the equivalent function call using *scipy.stats*.

```
>>> arr = [1, 2, 3, 4, 999]
>>> import scipy.stats
>>> print(f"{scipy.stats.kurtosis(arr[:-1], bias=False):.6f}")
-1.200000
>>> print(f"{scipy.stats.kurtosis(arr, bias=False):.6f}")
4.999874
>>> s = pd.Series(arr)
>>> s.expanding(4).kurt()
0         NaN
1         NaN
2         NaN
3   -1.200000
4    4.999874
dtype: float64
```

pandas.core.window.expanding.Expanding.apply

Expanding.**apply**(*self*, *func*, *raw=False*, *args=()*, *kwargs={}*)

The expanding function's apply function.

Parameters

func [function] Must produce a single value from an ndarray input if `raw=True` or a single value from a Series if `raw=False`. Can also accept a Numba JIT function with `engine='numba'` specified.

Changed in version 1.0.0.

raw [bool, default None]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

engine [str, default 'cython']

- `'cython'` : Runs rolling apply through C-extensions from cython.
- `'numba'` : Runs rolling apply through JIT compiled code from numba. Only available when `raw` is set to `True`.

  New in version 1.0.0.

engine_kwargs [dict, default None]

- For `'cython'` engine, there are no accepted `engine_kwargs`
- For `'numba'` engine, the engine can accept `nopython`, `nogil` and `parallel` dictionary keys. The values must either be `True` or `False`. The default `engine_kwargs` for the `'numba'` engine is `{'nopython': True, 'nogil': False, 'parallel': False}` and will be applied to both the `func` and the `apply` rolling aggregation.

  New in version 1.0.0.

args [tuple, default None] Positional arguments to be passed into func.

kwargs [dict, default None] Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.
DataFrame.expanding DataFrame expanding.

Notes

See *Rolling Apply* for extended documentation and performance considerations for the Numba engine.

pandas.core.window.expanding.Expanding.aggregate

Expanding.**aggregate**(*self*, *func*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series/Dataframe or when passed to Series/Dataframe.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See also:

DataFrame.expanding.aggregate
DataFrame.rolling.aggregate
DataFrame.aggregate

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'])
>>> df
          A         B         C
0 -2.385977 -0.102758  0.438822
1 -1.004295  0.905829 -0.954544
2  0.735167 -0.165272 -1.619346
3 -0.702657 -1.340923 -0.706334
4 -0.246845  0.211596 -0.901819
5  2.463718  3.157577 -1.380906
6 -1.142255  2.340594 -0.039875
7  1.396598 -1.647453  1.677227
8 -0.543425  1.761277 -0.220481
9 -0.640505  0.289374 -1.550670
```

```
>>> df.ewm(alpha=0.5).mean()
          A         B         C
0 -2.385977 -0.102758  0.438822
1 -1.464856  0.569633 -0.490089
2 -0.207700  0.149687 -1.135379
3 -0.471677 -0.645305 -0.906555
4 -0.355635 -0.203033 -0.904111
5  1.076417  1.503943 -1.146293
6 -0.041654  1.925562 -0.588728
7  0.680292  0.132049  0.548693
8  0.067236  0.948257  0.163353
9 -0.286980  0.618493 -0.694496
```

pandas.core.window.expanding.Expanding.quantile

Expanding.**quantile**(*self*, *quantile*, *interpolation='linear'*, ***kwargs*)

Calculate the expanding quantile.

Parameters

quantile [float] Quantile to compute. 0 <= quantile <= 1.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.23.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i* + *(j - i)* * *fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

****kwargs** For compatibility with other expanding methods. Has no effect on the result.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.quantile Computes value at the given quantile over all data in Series.
DataFrame.quantile Computes values at the given quantile over requested axis in DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
0    NaN
1    1.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

3.10.3 Exponentially-weighted moving window functions

EWM.mean(self, *args, **kwargs)	Exponential weighted moving average.
EWM.std(self[, bias])	Exponential weighted moving stddev.
EWM.var(self[, bias])	Exponential weighted moving variance.
EWM.corr(self[, other, pairwise])	Exponential weighted sample correlation.
EWM.cov(self[, other, pairwise, bias])	Exponential weighted sample covariance.

pandas.core.window.ewm.EWM.mean

EWM.**mean**(*self*, **args*, ***kwargs*)

Exponential weighted moving average.

Parameters

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.
DataFrame.ewm DataFrame ewm.

pandas.core.window.ewm.EWM.std

EWM.**std**(*self*, *bias=False*, **args*, ***kwargs*)

Exponential weighted moving stddev.

Parameters

bias [bool, default False] Use a standard estimation bias correction.

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.
DataFrame.ewm DataFrame ewm.

pandas.core.window.ewm.EWM.var

EWM.**var**(*self*, *bias=False*, **args*, ***kwargs*)

Exponential weighted moving variance.

Parameters

bias [bool, default False] Use a standard estimation bias correction.

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.
DataFrame.ewm DataFrame ewm.

pandas.core.window.ewm.EWM.corr

EWM.**corr**(*self*, *other=None*, *pairwise=None*, ***kwargs*)

Exponential weighted sample correlation.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.
DataFrame.ewm DataFrame ewm.

pandas.core.window.ewm.EWM.cov

EWM.**cov**(*self*, *other=None*, *pairwise=None*, *bias=False*, ***kwargs*)

Exponential weighted sample covariance.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

bias [bool, default False] Use a standard estimation bias correction.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

`Series.ewm` Series ewm.
`DataFrame.ewm` DataFrame ewm.

3.10.4 Window Indexer

Base class for defining custom window boundaries.

api.indexers.BaseIndexer(index_array, ...)	Base class for window bounds calculations

pandas.api.indexers.BaseIndexer

class pandas.api.indexers.**BaseIndexer**(*index_array: Optional[numpy.ndarray] = None*, *window_size: int = 0*, ***kwargs*)

Base class for window bounds calculations

Methods

get_window_bounds(self, num_values, ...)	Computes the bounds of a window.

pandas.api.indexers.BaseIndexer.get_window_bounds

BaseIndexer.**get_window_bounds**(*self*, *num_values: int = 0*, *min_periods: Union[int, NoneType] = None*, *center: Union[bool, NoneType] = None*, *closed: Union[str, NoneType] = None*) → Tuple[numpy.ndarray, numpy.ndarray]

Computes the bounds of a window.

Parameters

num_values [int, default 0] number of values that will be aggregated over

window_size [int, default 0] the number of rows in a window

min_periods [int, default None] min_periods passed from the top level rolling API

center [bool, default None] center passed from the top level rolling API

closed [str, default None] closed passed from the top level rolling API

win_type [str, default None] win_type passed from the top level rolling API

Returns

A tuple of ndarray[int64]s, indicating the boundaries of each

window

3.11 GroupBy

GroupBy objects are returned by groupby calls: *pandas.DataFrame.groupby()*, *pandas.Series.groupby()*, etc.

3.11.1 Indexing, iteration

GroupBy.__iter__(self)	Groupby iterator.
GroupBy.groups	Dict {group name -> group labels}.
GroupBy.indices	Dict {group name -> group indices}.
GroupBy.get_group(self, name[, obj])	Construct DataFrame from group with provided name.

pandas.core.groupby.GroupBy.__iter__

GroupBy.**__iter__**(*self*)

Groupby iterator.

Returns

Generator yielding sequence of (name, subsetted object)

for each group

pandas.core.groupby.GroupBy.groups

property GroupBy.**groups**

Dict {group name -> group labels}.

pandas.core.groupby.GroupBy.indices

property GroupBy.**indices**

Dict {group name -> group indices}.

pandas.core.groupby.GroupBy.get_group

GroupBy.**get_group**(*self*, *name*, *obj=None*)

Construct DataFrame from group with provided name.

Parameters

name [object] The name of the group to get as a DataFrame.

obj [DataFrame, default None] The DataFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used.

Returns

group [same type as obj]

Grouper(*args, **kwargs)	A Grouper allows the user to specify a groupby instruction for an object.