which is much more readable.

Parameters

func [callable or tuple of (callable, string)] Function to apply to this Resampler object or, alternatively, a *(callable, data_keyword)* tuple where *data_keyword* is a string indicating the keyword of *callable* that expects the Resampler object.

args [iterable, optional] Positional arguments passed into *func*.

kwargs [dict, optional] A dictionary of keyword arguments passed into *func*.

Returns

object [the return type of *func*.]

See also:

`Series.pipe` Apply a function with arguments to a series.
`DataFrame.pipe` Apply a function with arguments to a dataframe.
`apply` Apply function to each group instead of to the full Resampler object.

Notes

See more here

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4]},
...                   index=pd.date_range('2012-08-02', periods=4))
>>> df
            A
2012-08-02  1
2012-08-03  2
2012-08-04  3
2012-08-05  4
```

To get the difference between each 2-day period's maximum and minimum value in one pass, you can do

```
>>> df.resample('2D').pipe(lambda x: x.max() - x.min())
            A
2012-08-02  1
2012-08-04  1
```

3.12.3 Upsampling

Resampler.ffill(self[, limit])	Forward fill the values.
Resampler.backfill(self[, limit])	Backward fill the new missing values in the resampled data.
Resampler.bfill(self[, limit])	Backward fill the new missing values in the resampled data.
Resampler.pad(self[, limit])	Forward fill the values.
Resampler.nearest(self[, limit])	Resample by using the nearest value.
Resampler.fillna(self, method[, limit])	Fill missing values introduced by upsampling.
Resampler.asfreq(self[, fill_value])	Return the values at the new freq, essentially a reindex.

continues on next page

Table 408 – continued from previous page

Resampler.interpolate(self[, method, axis, ...])	Interpolate values according to different methods.

pandas.core.resample.Resampler.ffill

Resampler.**ffill**(*self*, *limit=None*)

Forward fill the values.

Parameters

limit [int, optional] Limit of how many values to fill.

Returns

An upsampled Series.

See also:

Series.fillna
DataFrame.fillna

pandas.core.resample.Resampler.backfill

Resampler.**backfill**(*self*, *limit=None*)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [1]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

Parameters

limit [int, optional] Limit of how many values to fill.

Returns

Series, DataFrame An upsampled Series or DataFrame with backward filled NaN values.

See also:

bfill Alias of backfill.
fillna Fill NaN values using the specified method, which can be 'backfill'.
nearest Fill NaN values with nearest neighbor starting from center.
pad Forward fill NaN values.
Series.fillna Fill NaN values in the Series using the specified method, which can be 'backfill'.
DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be 'backfill'.

References

[1]

Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

```
>>> s.resample('30min').backfill()
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').backfill(limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

Resampling a DataFrame that has missing values:

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                   index=pd.date_range('20180101', periods=3,
...                                       freq='h'))
>>> df
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').backfill()
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('15min').backfill(limit=2)
                       a    b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:15:00  NaN  NaN
2018-01-01 00:30:00  NaN  3.0
```

(continues on next page)

(continued from previous page)

```
2018-01-01 00:45:00  NaN  3.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 01:15:00  NaN  NaN
2018-01-01 01:30:00  6.0  5.0
2018-01-01 01:45:00  6.0  5.0
2018-01-01 02:00:00  6.0  5.0
```

pandas.core.resample.Resampler.bfill

Resampler.**bfill**(*self*, *limit=None*)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [1]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

Parameters

limit [int, optional] Limit of how many values to fill.

Returns

Series, DataFrame An upsampled Series or DataFrame with backward filled NaN values.

See also:

bfill Alias of backfill.
fillna Fill NaN values using the specified method, which can be 'backfill'.
nearest Fill NaN values with nearest neighbor starting from center.
pad Forward fill NaN values.
Series.fillna Fill NaN values in the Series using the specified method, which can be 'backfill'.
DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be 'backfill'.

References

[1]

Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

```
>>> s.resample('30min').backfill()
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
```

(continues on next page)

(continued from previous page)

```
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').backfill(limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

Resampling a DataFrame that has missing values:

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                   index=pd.date_range('20180101', periods=3,
...                                       freq='h'))
>>> df
                     a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').backfill()
                     a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('15min').backfill(limit=2)
                     a    b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:15:00  NaN  NaN
2018-01-01 00:30:00  NaN  3.0
2018-01-01 00:45:00  NaN  3.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 01:15:00  NaN  NaN
2018-01-01 01:30:00  6.0  5.0
2018-01-01 01:45:00  6.0  5.0
2018-01-01 02:00:00  6.0  5.0
```

pandas.core.resample.Resampler.pad

Resampler.**pad**(*self*, *limit=None*)

Forward fill the values.

Parameters

limit [int, optional] Limit of how many values to fill.

Returns

An upsampled Series.

See also:

Series.fillna
DataFrame.fillna

pandas.core.resample.Resampler.nearest

Resampler.**nearest**(*self*, *limit=None*)

Resample by using the nearest value.

When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The *nearest* method will replace NaN values that appeared in the resampled data with the value from the nearest member of the sequence, based on the index value. Missing values that existed in the original data will not be modified. If *limit* is given, fill only this many values in each direction for each of the original values.

Parameters

limit [int, optional] Limit of how many values to fill.

New in version 0.21.0.

Returns

Series or DataFrame An upsampled Series or DataFrame with NaN values filled with their nearest value.

See also:

backfill Backward fill the new missing values in the resampled data.
pad Forward fill NaN values.

Examples

```
>>> s = pd.Series([1, 2],
...               index=pd.date_range('20180101',
...                                   periods=2,
...                                   freq='1h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
Freq: H, dtype: int64
```

```
>>> s.resample('15min').nearest()
2018-01-01 00:00:00    1
2018-01-01 00:15:00    1
2018-01-01 00:30:00    2
2018-01-01 00:45:00    2
```

(continues on next page)

(continued from previous page)

```
2018-01-01 01:00:00    2
Freq: 15T, dtype: int64
```

Limit the number of upsampled values imputed by the nearest:

```
>>> s.resample('15min').nearest(limit=1)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
Freq: 15T, dtype: float64
```

pandas.core.resample.Resampler.fillna

Resampler.**fillna**(*self*, *method*, *limit=None*)

Fill missing values introduced by upsampling.

In statistics, imputation is the process of replacing missing data with substituted values [1]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency).

Missing values that existed in the original data will not be modified.

Parameters

method [{'pad', 'backfill', 'ffill', 'bfill', 'nearest'}] Method to use for filling holes in resampled data

- 'pad' or 'ffill': use previous valid observation to fill gap (forward fill).
- 'backfill' or 'bfill': use next valid observation to fill gap.
- 'nearest': use nearest valid observation to fill gap.

limit [int, optional] Limit of how many consecutive missing values to fill.

Returns

Series or DataFrame An upsampled Series or DataFrame with missing values filled.

See also:

backfill Backward fill NaN values in the resampled data.
pad Forward fill NaN values in the resampled data.
nearest Fill NaN values in the resampled data with nearest neighbor starting from center.
interpolate Fill NaN values using interpolation.
Series.fillna Fill NaN values in the Series using the specified method, which can be 'bfill' and 'ffill'.
DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be 'bfill' and 'ffill'.

References

[1]

Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

Without filling the missing values you get:

```
>>> s.resample("30min").asfreq()
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    2.0
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> s.resample('30min').fillna("backfill")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').fillna("backfill", limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

```
>>> s.resample('30min').fillna("pad")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    1
2018-01-01 01:00:00    2
2018-01-01 01:30:00    2
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('30min').fillna("nearest")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

Missing values present before the upsampling are not affected.

```
>>> sm = pd.Series([1, None, 3],
...                index=pd.date_range('20180101', periods=3, freq='h'))
>>> sm
2018-01-01 00:00:00    1.0
2018-01-01 01:00:00    NaN
2018-01-01 02:00:00    3.0
Freq: H, dtype: float64
```

```
>>> sm.resample('30min').fillna('backfill')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> sm.resample('30min').fillna('pad')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    1.0
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> sm.resample('30min').fillna('nearest')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

DataFrame resampling is done column-wise. All the same options are available.

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                   index=pd.date_range('20180101', periods=3,
...                                       freq='h'))
>>> df
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').fillna("bfill")
                       a  b
2018-01-01 00:00:00  2.0  1
```

(continues on next page)

(continued from previous page)

```
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

pandas.core.resample.Resampler.asfreq

Resampler.**asfreq**(*self*, *fill_value=None*)

Return the values at the new freq, essentially a reindex.

Parameters

fill_value [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns

DataFrame or Series Values at the specified freq.

See also:

Series.asfreq
DataFrame.asfreq

pandas.core.resample.Resampler.interpolate

Resampler.**interpolate**(*self*, *method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*. These methods use the numerical values of the index. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=5)`.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives* which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{'forward', 'backward', 'both'}, default 'forward'] If limit is specified, consecutive NaNs will be filled in this direction.

limit_area [{*None*, 'inside', 'outside'}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

New in version 0.23.0.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

****kwargs** Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame Returns the same object type as the caller, interpolated at some or all `NaN` values.

See also:

fillna Fill missing values using different methods.
scipy.interpolate.Akima1DInterpolator Piecewise cubic polynomials (Akima interpolator).
scipy.interpolate.BPoly.from_derivatives Piecewise polynomial in the Bernstein basis.
scipy.interpolate.interp1d Interpolate a 1-D function.
scipy.interpolate.KroghInterpolator Interpolate polynomial (Krogh interpolator).
scipy.interpolate.PchipInterpolator PCHIP 1-d monotonic cubic interpolation.
scipy.interpolate.CubicSpline Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in `NaN` in a *`Series`* via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series by padding, but filling at most two consecutive `NaN` at a time.

```
>>> s = pd.Series([np.nan, "single_one", np.nan,
...                "fill_two_more", np.nan, np.nan, np.nan,
...                4.71, np.nan])
>>> s
0              NaN
1       single_one
2              NaN
3    fill_two_more
4              NaN
5              NaN
6              NaN
7             4.71
8              NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0              NaN
1       single_one
2       single_one
3    fill_two_more
4    fill_two_more
5    fill_two_more
6              NaN
7             4.71
8             4.71
dtype: object
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
```

(continues on next page)

(continued from previous page)

```
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

3.12.4 Computations / descriptive stats

Resampler.count(self)	Compute count of group, excluding missing values.
Resampler.nunique(self[, _method])	Return number of unique elements in the group.
Resampler.first(self[, _method])	Compute first of group values.
Resampler.last(self[, _method])	Compute last of group values.
Resampler.max(self[, _method])	Compute max of group values.
Resampler.mean(self[, _method])	Compute mean of groups, excluding missing values.
Resampler.median(self[, _method])	Compute median of groups, excluding missing values.
Resampler.min(self[, _method])	Compute min of group values.
Resampler.ohlc(self[, _method])	Compute sum of values, excluding missing values.
Resampler.prod(self[, _method, min_count])	Compute prod of group values.
Resampler.size(self)	Compute group sizes.
Resampler.sem(self[, _method])	Compute standard error of the mean of groups, excluding missing values.
Resampler.std(self[, ddof])	Compute standard deviation of groups, excluding missing values.
Resampler.sum(self[, _method, min_count])	Compute sum of group values.
Resampler.var(self[, ddof])	Compute variance of groups, excluding missing values.
Resampler.quantile(self[, q])	Return value at the given quantile.

pandas.core.resample.Resampler.count

Resampler.**count**(*self*)

Compute count of group, excluding missing values.

Returns

Series or DataFrame Count of values within each group.

See also:

Series.groupby
DataFrame.groupby

pandas.core.resample.Resampler.nunique

Resampler.**nunique**(*self*, *_method='nunique'*)

Return number of unique elements in the group.

Returns

Series Number of unique values within each group.

pandas.core.resample.Resampler.first

Resampler.**first**(*self*, *_method='first'*, **args*, ***kwargs*)

Compute first of group values.

Returns

Series or DataFrame Computed first of values within each group.

pandas.core.resample.Resampler.last

Resampler.**last**(*self*, *_method='last'*, **args*, ***kwargs*)

Compute last of group values.

Returns

Series or DataFrame Computed last of values within each group.

pandas.core.resample.Resampler.max

Resampler.**max**(*self*, *_method='max'*, **args*, ***kwargs*)

Compute max of group values.

Returns

Series or DataFrame Computed max of values within each group.

pandas.core.resample.Resampler.mean

Resampler.**mean**(*self*, *_method='mean'*, **args*, ***kwargs*)

Compute mean of groups, excluding missing values.

Returns

pandas.Series or pandas.DataFrame

See also:

Series.groupby
DataFrame.groupby

Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5],
...                    'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```
>>> df.groupby('A').mean()
     B         C
A
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```
>>> df.groupby(['A', 'B']).mean()
       C
A B
1 2.0  2
  4.0  1
2 3.0  1
  5.0  2
```

Groupby one column and return the mean of only particular column in the group.

```
>>> df.groupby('A')['B'].mean()
A
1    3.0
2    4.0
Name: B, dtype: float64
```

pandas.core.resample.Resampler.median

Resampler.**median**(*self*, *_method='median'*, **args*, ***kwargs*)

Compute median of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

Returns

Series or DataFrame Median of values within each group.

See also:

Series.groupby
DataFrame.groupby

pandas.core.resample.Resampler.min

Resampler.**min**(*self*, *_method='min'*, **args*, ***kwargs*)

Compute min of group values.

Returns

Series or DataFrame Computed min of values within each group.

pandas.core.resample.Resampler.ohlc

Resampler.**ohlc**(*self*, *_method='ohlc'*, **args*, ***kwargs*)

Compute sum of values, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

Returns

DataFrame Open, high, low and close values within each group.

See also:

Series.groupby
DataFrame.groupby

pandas.core.resample.Resampler.prod

Resampler.**prod**(*self*, *_method='prod'*, *min_count=0*, **args*, ***kwargs*)

Compute prod of group values.

Returns

Series or DataFrame Computed prod of values within each group.

pandas.core.resample.Resampler.size

Resampler.**size**(*self*)

Compute group sizes.

Returns

Series Number of rows in each group.

See also:

Series.groupby
DataFrame.groupby

pandas.core.resample.Resampler.sem

Resampler.**sem**(*self*, *_method='sem'*, **args*, ***kwargs*)

Compute standard error of the mean of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Parameters

ddof [int, default 1] Degrees of freedom.

Returns

Series or DataFrame Standard error of the mean of values within each group.

See also:

Series.groupby
DataFrame.groupby

pandas.core.resample.Resampler.std

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

Compute standard deviation of groups, excluding missing values.

Parameters

ddof [int, default 1] Degrees of freedom.

Returns

DataFrame or Series Standard deviation of values within each group.

pandas.core.resample.Resampler.sum

Resampler.**sum**(*self*, *_method='sum'*, *min_count=0*, **args*, ***kwargs*)

Compute sum of group values.

Returns

Series or DataFrame Computed sum of values within each group.

pandas.core.resample.Resampler.var

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

Compute variance of groups, excluding missing values.

Parameters

ddof [int, default 1] Degrees of freedom.

Returns

DataFrame or Series Variance of values within each group.

pandas.core.resample.Resampler.quantile

Resampler.**quantile**(*self*, *q=0.5*, ***kwargs*)

Return value at the given quantile.

New in version 0.24.0.

Parameters

q [float or array-like, default 0.5 (50% quantile)]

Returns

DataFrame or Series Quantile of values within each group.

See also:

Series.quantile
DataFrame.quantile
DataFrameGroupBy.quantile

3.13 Style

`Styler` objects are returned by *pandas.DataFrame.style*.

3.13.1 Styler constructor

Styler(data[, precision, table_styles, ...])	Helps style a DataFrame or Series according to the data with HTML and CSS.
Styler.from_custom_template(searchpath, name)	Factory function for creating a subclass of `Styler`.

pandas.io.formats.style.Styler

class pandas.io.formats.style.**Styler**(*data*, *precision=None*, *table_styles=None*, *uuid=None*, *caption=None*, *table_attributes=None*, *cell_ids=True*, *na_rep: Optional[str] = None*)

Helps style a DataFrame or Series according to the data with HTML and CSS.

Parameters

data [Series or DataFrame] Data to be styled - either a Series or DataFrame.

precision [int] Precision to round floats to, defaults to pd.options.display.precision.

table_styles [list-like, default None] List of {selector: (attr, value)} dicts; see Notes.

uuid [str, default None] A unique identifier to avoid CSS collisions; generated automatically.

caption [str, default None] Caption to attach to the table.

table_attributes [str, default None] Items that show up in the opening `<table>` tag in addition to automatic (by default) id.

cell_ids [bool, default True] If True, each cell will have an `id` attribute in their HTML tag. The `id` takes the form `T_<uuid>_row<num_row>_col<num_col>` where `<uuid>` is the unique identifier, `<num_row>` is the row number and `<num_col>` is the column number.

na_rep [str, optional] Representation for missing values. If `na_rep` is None, no special formatting is applied

New in version 1.0.0.

See also:

DataFrame.style Return a Styler object containing methods for building a styled HTML representation for the DataFrame.

Notes

Most styling will be done by passing style functions into `Styler.apply` or `Styler.applymap`. Style functions should return values with strings containing CSS `'attr: value'` that will be applied to the indicated cells.

If using in the Jupyter notebook, Styler has defined a `_repr_html_` to automatically render itself. Otherwise call Styler.render to get the generated HTML.

CSS classes are attached to the generated HTML

- Index and Column names include `index_name` and `level<k>` where *k* is its level in a MultiIndex
- Index label cells include
  - `row_heading`
  - `row<n>` where *n* is the numeric position of the row
  - `level<k>` where *k* is the level in a MultiIndex
- Column label cells include * `col_heading` * `col<n>` where *n* is the numeric position of the column * `evel<k>` where *k* is the level in a MultiIndex
- Blank cells include `blank`
- Data cells include `data`

Attributes

env	(Jinja2 jinja2.Environment)
template	(Jinja2 Template)
loader	(Jinja2 Loader)

Methods

apply(self, func[, axis, subset])	Apply a function column-wise, row-wise, or table-wise.
applymap(self, func[, subset])	Apply a function elementwise.
background_gradient(self[, cmap, low, high, ...])	Color the background in a gradient style.
bar(self[, subset, axis, color, width, ...])	Draw bar chart in the cell backgrounds.
clear(self)	Reset the styler, removing any previously applied styles.
export(self)	Export the styles to applied to the current Styler.
format(self, formatter[, subset])	Format the text display value of cells.
from_custom_template(searchpath, name)	Factory function for creating a subclass of `Styler`.
hide_columns(self, subset)	Hide columns from rendering.
hide_index(self)	Hide any indices from rendering.
highlight_max(self[, subset, color, axis])	Highlight the maximum by shading the background.
highlight_min(self[, subset, color, axis])	Highlight the minimum by shading the background.
highlight_null(self[, null_color])	Shade the background `null_color` for missing values.
pipe(self, func, *args, **kwargs)	Apply `func(self, *args, **kwargs)`, and return the result.
render(self, **kwargs)	Render the built up styles to HTML.
set_caption(self, caption)	Set the caption on a Styler.

continues on next page

Table 411 – continued from previous page

set_na_rep(self, na_rep)	Set the missing data representation on a Styler.
set_precision(self, precision)	Set the precision used to render.
set_properties(self[, subset])	Method to set one or more non-data dependent properties or each cell.
set_table_attributes(self, attributes)	Set the table attributes.
set_table_styles(self, table_styles)	Set the table styles on a Styler.
set_uuid(self, uuid)	Set the uuid for a Styler.
to_excel(self, excel_writer[, sheet_name, ...])	Write Styler to an Excel sheet.
use(self, styles)	Set the styles on the current Styler.
where(self, cond, value[, other, subset])	Apply a function elementwise.

pandas.io.formats.style.Styler.apply

`Styler.`**`apply`**(*self*, *func*, *axis=0*, *subset=None*, ***kwargs*)

Apply a function column-wise, row-wise, or table-wise.

Updates the HTML representation with the result.

Parameters

func [function] `func` should take a Series or DataFrame (depending on `axis`), and return an object with the same shape. Must return a DataFrame with identical index and column labels when `axis=None`.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (`axis=0` or `'index'`), to each row (`axis=1` or `'columns'`), or to the entire DataFrame at once with `axis=None`.

subset [IndexSlice] A valid indexer to limit `data` to *before* applying the function. Consider using a pandas.IndexSlice.

****kwargs** [dict] Pass along to `func`.

Returns

self [Styler]

Notes

The output shape of `func` should match the input, i.e. if `x` is the input row, column, or table (depending on `axis`), then `func(x).shape == x.shape` should be true.

This is similar to `DataFrame.apply`, except that `axis=None` applies the function to the entire DataFrame at once, rather than column-wise or row-wise.

Examples

```
>>> def highlight_max(x):
...     return ['background-color: yellow' if v == x.max() else ''
                for v in x]
...
>>> df = pd.DataFrame(np.random.randn(5, 2))
>>> df.style.apply(highlight_max)
```

pandas.io.formats.style.Styler.applymap

Styler.**applymap**(*self*, *func*, *subset=None*, ***kwargs*)

Apply a function elementwise.

Updates the HTML representation with the result.

Parameters

func [function] `func` should take a scalar and return a scalar.

subset [IndexSlice] A valid indexer to limit `data` to *before* applying the function. Consider using a pandas.IndexSlice.

****kwargs** [dict] Pass along to `func`.

Returns

self [Styler]

See also:

Styler.where

pandas.io.formats.style.Styler.background_gradient

Styler.**background_gradient**(*self*, *cmap='PuBu'*, *low=0*, *high=0*, *axis=0*, *subset=None*, *text_color_threshold=0.408*, *vmin: Union[float, NoneType] = None*, *vmax: Union[float, NoneType] = None*)

Color the background in a gradient style.

The background color is determined according to the data in each column (optionally row). Requires matplotlib.

Parameters

cmap [str or colormap] Matplotlib colormap.

low [float] Compress the range by the low.

high [float] Compress the range by the high.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (`axis=0` or `'index'`), to each row (`axis=1` or `'columns'`), or to the entire DataFrame at once with `axis=None`.

subset [IndexSlice] A valid slice for `data` to limit the style application to.

text_color_threshold [float or int] Luminance threshold for determining text color. Facilitates text visibility across varying background colors. From 0 to 1. 0 = all text is dark colored, 1 = all text is light colored.

New in version 0.24.0.

vmin [float, optional] Minimum data value that corresponds to colormap minimum value. When None (default): the minimum value of the data will be used.

New in version 1.0.0.

vmax [float, optional] Maximum data value that corresponds to colormap maximum value. When None (default): the maximum value of the data will be used.

New in version 1.0.0.

Returns

self [Styler]

Raises

ValueError If `text_color_threshold` is not a value from 0 to 1.

Notes

Set `text_color_threshold` or tune `low` and `high` to keep the text legible by not using the entire range of the color map. The range of the data is extended by `low * (x.max() - x.min())` and `high * (x.max() - x.min())` before normalizing.

pandas.io.formats.style.Styler.bar

`Styler.`**`bar`**(*self*, *subset=None*, *axis=0*, *color='#d65f5f'*, *width=100*, *align='left'*, *vmin=None*, *vmax=None*)

Draw bar chart in the cell backgrounds.

Parameters

subset [IndexSlice, optional] A valid slice for *data* to limit the style application to.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (`axis=0` or `'index'`), to each row (`axis=1` or `'columns'`), or to the entire DataFrame at once with `axis=None`.

color [str or 2-tuple/list] If a str is passed, the color is the same for both negative and positive numbers. If 2-tuple/list is used, the first element is the color_negative and the second is the color_positive (eg: ['#d65f5f', '#5fba7d']).

width [float, default 100] A number between 0 or 100. The largest value will cover *width* percent of the cell's width.

align [{'left', 'zero',' mid'}, default 'left'] How to align the bars with the cells.

- 'left' : the min value starts at the left of the cell.
- 'zero' : a value of zero is located at the center of the cell.
- 'mid' : the center of the cell is at (max-min)/2, or if values are all negative (positive) the zero is aligned at the right (left) of the cell.

vmin [float, optional] Minimum bar value, defining the left hand limit of the bar drawing range, lower values are clipped to *vmin*. When None (default): the minimum value of the data will be used.

New in version 0.24.0.

vmax [float, optional] Maximum bar value, defining the right hand limit of the bar drawing range, higher values are clipped to *vmax*. When None (default): the maximum value of the data will be used.

New in version 0.24.0.

Returns

self [Styler]

pandas.io.formats.style.Styler.clear

`Styler.`**`clear`**(*self*)

Reset the styler, removing any previously applied styles.

Returns None.

pandas.io.formats.style.Styler.export

`Styler.`**`export`**(*self*)

Export the styles to applied to the current Styler.

Can be applied to a second style with `Styler.use`.

Returns

styles [list]

See also:

`Styler.use`

pandas.io.formats.style.Styler.format

`Styler.`**`format`**(*self*, *formatter*, *subset=None*, *na_rep: Union[str, NoneType] = None*)

Format the text display value of cells.

Parameters

formatter [str, callable, dict or None] If `formatter` is None, the default formatter is used

subset [IndexSlice] An argument to `DataFrame.loc` that restricts which elements `formatter` is applied to.

na_rep [str, optional] Representation for missing values. If `na_rep` is None, no special formatting is applied

New in version 1.0.0.

Returns

self [Styler]

Notes

`formatter` is either an `a` or a dict `{column name: a}` where `a` is one of

- str: this will be wrapped in: `a.format(x)`
- callable: called with the value of an individual cell

The default display value for numeric values is the "general" (`g`) format with `pd.options.display.precision` precision.

Examples

```
>>> df = pd.DataFrame(np.random.randn(4, 2), columns=['a', 'b'])
>>> df.style.format("{:.2%}")
>>> df['c'] = ['a', 'b', 'c', 'd']
>>> df.style.format({'c': str.upper})
```

pandas.io.formats.style.Styler.from_custom_template

classmethod `Styler.`**from_custom_template**(*searchpath*, *name*)

Factory function for creating a subclass of `Styler`.

Uses a custom template and Jinja environment.

Parameters

searchpath [str or list] Path or paths of directories containing the templates.

name [str] Name of your custom template to use for rendering.

Returns

MyStyler [subclass of Styler] Has the correct `env` and `template` class attributes set.

pandas.io.formats.style.Styler.hide_columns

`Styler.`**hide_columns**(*self*, *subset*)

Hide columns from rendering.

New in version 0.23.0.

Parameters

subset [IndexSlice] An argument to `DataFrame.loc` that identifies which columns are hidden.

Returns

self [Styler]

pandas.io.formats.style.Styler.hide_index

Styler.**hide_index**(*self*)

Hide any indices from rendering.

New in version 0.23.0.

Returns

self [Styler]

pandas.io.formats.style.Styler.highlight_max

Styler.**highlight_max**(*self*, *subset=None*, *color='yellow'*, *axis=0*)

Highlight the maximum by shading the background.

Parameters

subset [IndexSlice, default None] A valid slice for `data` to limit the style application to.

color [str, default 'yellow']

axis [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (`axis=0` or `'index'`), to each row (`axis=1` or `'columns'`), or to the entire DataFrame at once with `axis=None`.

Returns

self [Styler]

pandas.io.formats.style.Styler.highlight_min

Styler.**highlight_min**(*self*, *subset=None*, *color='yellow'*, *axis=0*)

Highlight the minimum by shading the background.

Parameters

subset [IndexSlice, default None] A valid slice for `data` to limit the style application to.

color [str, default 'yellow']

axis [{0 or 'index', 1 or 'columns', None}, default 0] Apply to each column (`axis=0` or `'index'`), to each row (`axis=1` or `'columns'`), or to the entire DataFrame at once with `axis=None`.

Returns

self [Styler]