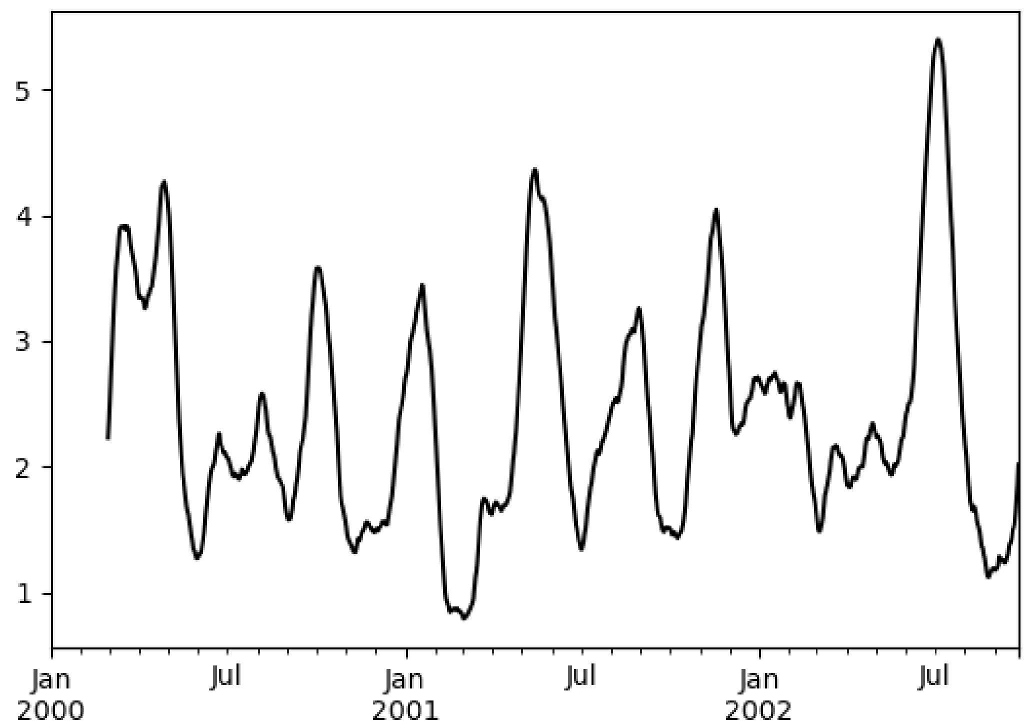
Rolling Apply

The `apply()` function takes an extra `func` argument and performs generic rolling computations. The `func` argument should be a single function that produces a single value from an ndarray input. Suppose we wanted to compute the mean absolute deviation on a rolling basis:

```
In [51]: def mad(x):
   ....:     return np.fabs(x - x.mean()).mean()
   ....:

In [52]: s.rolling(window=60).apply(mad, raw=True).plot(style='k')
Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d69d090>
```



New in version 1.0.

Additionally, `apply()` can leverage Numba if installed as an optional dependency. The apply aggregation can be executed using Numba by specifying `engine='numba'` and `engine_kwargs` arguments (`raw` must also be set to `True`). Numba will be applied in potentially two routines:

1. If `func` is a standard Python function, the engine will JIT the passed function. `func` can also be a JITed function in which case the engine will not JIT the function again. 2. The engine will JIT the for loop where the apply function is applied to each window.

The `engine_kwargs` argument is a dictionary of keyword arguments that will be passed into the numba.jit decorator. These keyword arguments will be applied to *both* the passed function (if a standard Python function) and the apply for loop over each window. Currently only `nogil`, `nopython`, and `parallel` are supported, and their default values are set to False, True and False respectively.

Note: In terms of performance, **the first time a function is run using the Numba engine will be slow** as Numba will have some function compilation overhead. However, rolling objects will cache the function and subsequent calls will be fast. In general, the Numba engine is performant with a larger amount of data points (e.g. 1+ million).

```
In [1]: data = pd.Series(range(1_000_000))

In [2]: roll = data.rolling(10)

In [3]: def f(x):
   ...:     return np.sum(x) + 5
# Run the first time, compilation time will affect performance
In [4]: %timeit -r 1 -n 1 roll.apply(f, engine='numba', raw=True)  # noqa: E225
1.23 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
# Function is cached and performance will improve
In [5]: %timeit roll.apply(f, engine='numba', raw=True)
188 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [6]: %timeit roll.apply(f, engine='cython', raw=True)
3.92 s ± 59 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Rolling windows

Passing win_type to .rolling generates a generic rolling window computation, that is weighted according the win_type. The following methods are available:

Method	Description
sum()	Sum of values
mean()	Mean of values

The weights used in the window are specified by the win_type keyword. The list of recognized types are the scipy.signal window functions:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)

- `slepian` (needs width)
- `exponential` (needs tau).

```
In [53]: ser = pd.Series(np.random.randn(10),
   ....:                 index=pd.date_range('1/1/2000', periods=10))
   ....:

In [54]: ser.rolling(window=5, win_type='triang').mean()
Out[54]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -1.037870
2000-01-06   -0.767705
2000-01-07   -0.383197
2000-01-08   -0.395513
2000-01-09   -0.558440
2000-01-10   -0.672416
Freq: D, dtype: float64
```

Note that the `boxcar` window is equivalent to `mean()`.

```
In [55]: ser.rolling(window=5, win_type='boxcar').mean()
Out[55]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64

In [56]: ser.rolling(window=5).mean()
Out[56]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

For some windowing functions, additional parameters must be specified:

```
In [57]: ser.rolling(window=5, win_type='gaussian').mean(std=0.1)
Out[57]:
2000-01-01         NaN
2000-01-02         NaN
```

(continues on next page)

(continued from previous page)

```
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -1.309989
2000-01-06   -1.153000
2000-01-07    0.606382
2000-01-08   -0.681101
2000-01-09   -0.289724
2000-01-10   -0.996632
Freq: D, dtype: float64
```

Note: For `.sum()` with a `win_type`, there is no normalization done to the weights for the window. Passing custom weights of `[1, 1, 1]` will yield a different result than passing weights of `[2, 2, 2]`, for example. When passing a `win_type` instead of explicitly specifying the weights, the weights are already normalized so that the largest weight is 1.

In contrast, the nature of the `.mean()` calculation is such that the weights are normalized with respect to each other. Weights of `[1, 1, 1]` and `[2, 2, 2]` yield the same result.

Time-aware rolling

It is possible to pass an offset (or convertible) to a `.rolling()` method and have it produce variable sized windows based on the passed time window. For each time point, this includes all preceding values occurring within the indicated time delta.

This can be particularly useful for a non-regular time frequency index.

```
In [58]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.date_range('20130101 09:00:00',
   ....:                                        periods=5,
   ....:                                        freq='s'))
   ....:

In [59]: dft
Out[59]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [60]: dft.rolling(2).sum()
Out[60]:
                       B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  NaN

In [61]: dft.rolling(2, min_periods=1).sum()
```

(continues on next page)

(continued from previous page)

```
Out[61]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0
```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```
In [62]: dft.rolling('2s').sum()
Out[62]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0
```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```
In [63]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.Index([pd.Timestamp('20130101 09:00:00'),
   ....:                                    pd.Timestamp('20130101 09:00:02'),
   ....:                                    pd.Timestamp('20130101 09:00:03'),
   ....:                                    pd.Timestamp('20130101 09:00:05'),
   ....:                                    pd.Timestamp('20130101 09:00:06')],
   ....:                                   name='foo'))
   ....:

In [64]: dft
Out[64]:
                       B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

In [65]: dft.rolling(2).sum()
Out[65]:
                       B
foo
2013-01-01 09:00:00  NaN
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  NaN
```

Using the time-specification generates variable windows for this sparse data.

```
In [66]: dft.rolling('2s').sum()
Out[66]:
                       B
foo
2013-01-01 09:00:00  0.0
```

(continues on next page)

(continued from previous page)

```
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a DataFrame.

```
In [67]: dft = dft.reset_index()

In [68]: dft
Out[68]:
                  foo    B
0 2013-01-01 09:00:00  0.0
1 2013-01-01 09:00:02  1.0
2 2013-01-01 09:00:03  2.0
3 2013-01-01 09:00:05  NaN
4 2013-01-01 09:00:06  4.0

In [69]: dft.rolling('2s', on='foo').sum()
Out[69]:
                  foo    B
0 2013-01-01 09:00:00  0.0
1 2013-01-01 09:00:02  1.0
2 2013-01-01 09:00:03  3.0
3 2013-01-01 09:00:05  NaN
4 2013-01-01 09:00:06  4.0
```

Custom window rolling

New in version 1.0.

In addition to accepting an integer or offset as a `window` argument, `rolling` also accepts a `BaseIndexer` subclass that allows a user to define a custom method for calculating window bounds. The `BaseIndexer` subclass will need to define a `get_window_bounds` method that returns a tuple of two arrays, the first being the starting indices of the windows and second being the ending indices of the windows. Additionally, `num_values`, `min_periods`, `center`, `closed` and will automatically be passed to `get_window_bounds` and the defined method must always accept these arguments.

For example, if we have the following `DataFrame`:

```
In [70]: use_expanding = [True, False, True, False, True]

In [71]: use_expanding
Out[71]: [True, False, True, False, True]

In [72]: df = pd.DataFrame({'values': range(5)})

In [73]: df
Out[73]:
   values
0       0
1       1
2       2
3       3
4       4
```

and we want to use an expanding window where `use_expanding` is `True` otherwise a window of size 1, we can create the following `BaseIndexer`:

```
In [2]: from pandas.api.indexers import BaseIndexer
...:
...: class CustomIndexer(BaseIndexer):
...:
...:    def get_window_bounds(self, num_values, min_periods, center, closed):
...:        start = np.empty(num_values, dtype=np.int64)
...:        end = np.empty(num_values, dtype=np.int64)
...:        for i in range(num_values):
...:            if self.use_expanding[i]:
...:                start[i] = 0
...:                end[i] = i + 1
...:            else:
...:                start[i] = i
...:                end[i] = i + self.window_size
...:        return start, end
...:

In [3]: indexer = CustomIndexer(window_size=1, use_expanding=use_expanding)

In [4]: df.rolling(indexer).sum()
Out[4]:
   values
0     0.0
1     1.0
2     3.0
3     3.0
4    10.0
```

Rolling window endpoints

The inclusion of the interval endpoints in rolling window calculations can be specified with the `closed` parameter:

`closed`	Description	Default for
`right`	close right endpoint	time-based windows
`left`	close left endpoint	
`both`	close both endpoints	fixed windows
`neither`	open endpoints	

For example, having the right endpoint open is useful in many problems that require that there is no contamination from present information back to past information. This allows the rolling window to compute statistics "up to that point in time", but not including that point in time.

```
In [74]: df = pd.DataFrame({'x': 1},
   ....:                   index=[pd.Timestamp('20130101 09:00:01'),
   ....:                          pd.Timestamp('20130101 09:00:02'),
   ....:                          pd.Timestamp('20130101 09:00:03'),
   ....:                          pd.Timestamp('20130101 09:00:04'),
   ....:                          pd.Timestamp('20130101 09:00:06')])
   ....:

In [75]: df["right"] = df.rolling('2s', closed='right').x.sum()  # default
```

(continues on next page)

(continued from previous page)

```
In [76]: df["both"] = df.rolling('2s', closed='both').x.sum()

In [77]: df["left"] = df.rolling('2s', closed='left').x.sum()

In [78]: df["neither"] = df.rolling('2s', closed='neither').x.sum()

In [79]: df
Out[79]:
                     x  right  both  left  neither
2013-01-01 09:00:01  1    1.0   1.0   NaN      NaN
2013-01-01 09:00:02  1    2.0   2.0   1.0      1.0
2013-01-01 09:00:03  1    2.0   3.0   2.0      1.0
2013-01-01 09:00:04  1    2.0   3.0   2.0      1.0
2013-01-01 09:00:06  1    1.0   2.0   1.0      NaN
```

Currently, this feature is only implemented for time-based windows. For fixed windows, the closed parameter cannot be set and the rolling window will always have both endpoints closed.

Time-aware rolling vs. resampling

Using `.rolling()` with a time-based index is quite similar to *resampling*. They both operate and perform reductive operations on time-indexed pandas objects.

When using `.rolling()` with an offset. The offset is a time-delta. Take a backwards-in-time looking window, and aggregate all of the values in that window (including the end-point, but not the start-point). This is the new value at that point in the result. These are variable sized windows in time-space for each point of the input. You will get a same sized result as the input.

When using `.resample()` with an offset. Construct a new index that is the frequency of the offset. For each frequency bin, aggregate points from the input within a backwards-in-time looking window that fall in that bin. The result of this aggregation is the output for that frequency point. The windows are fixed size in the frequency space. Your result will have the shape of a regular frequency between the min and the max of the original input object.

To summarize, `.rolling()` is a time-based window operation, while `.resample()` is a frequency-based window operation.

Centering windows

By default the labels are set to the right edge of the window, but a `center` keyword is available so the labels can be set at the center.

```
In [80]: ser.rolling(window=5).mean()
Out[80]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [81]: ser.rolling(window=5, center=True).mean()
Out[81]:
2000-01-01          NaN
2000-01-02          NaN
2000-01-03    -0.841164
2000-01-04    -0.779948
2000-01-05    -0.565487
2000-01-06    -0.502815
2000-01-07    -0.553755
2000-01-08    -0.472211
2000-01-09          NaN
2000-01-10          NaN
Freq: D, dtype: float64
```

Binary window functions

`cov()` and `corr()` can compute moving window statistics about two `Series` or any combination of `DataFrame/Series` or `DataFrame/DataFrame`. Here is the behavior in each case:

- two `Series`: compute the statistic for the pairing.
- `DataFrame/Series`: compute the statistics for each column of the DataFrame with the passed Series, thus returning a DataFrame.
- `DataFrame/DataFrame`: by default compute the statistic for matching column names, returning a DataFrame. If the keyword argument `pairwise=True` is passed then computes the statistic for each pair of columns, returning a `MultiIndexed DataFrame` whose `index` are the dates in question (see *the next section*).

For example:

```
In [82]: df = pd.DataFrame(np.random.randn(1000, 4),
   ....:                   index=pd.date_range('1/1/2000', periods=1000),
   ....:                   columns=['A', 'B', 'C', 'D'])
   ....:

In [83]: df = df.cumsum()

In [84]: df2 = df[:20]

In [85]: df2.rolling(window=5).corr(df2['B'])
Out[85]:
                   A    B         C         D
2000-01-01       NaN  NaN       NaN       NaN
2000-01-02       NaN  NaN       NaN       NaN
2000-01-03       NaN  NaN       NaN       NaN
2000-01-04       NaN  NaN       NaN       NaN
2000-01-05  0.768775  1.0 -0.977990  0.800252
...              ...  ...       ...       ...
2000-01-16  0.691078  1.0  0.807450 -0.939302
2000-01-17  0.274506  1.0  0.582601 -0.902954
2000-01-18  0.330459  1.0  0.515707 -0.545268
2000-01-19  0.046756  1.0 -0.104334 -0.419799
2000-01-20 -0.328241  1.0 -0.650974 -0.777777
```

(continues on next page)

(continued from previous page)

```
[20 rows x 4 columns]
```

Computing rolling pairwise covariances and correlations

In financial data analysis and other fields it's common to compute covariance and correlation matrices for a collection of time series. Often one is also interested in moving-window covariance and correlation matrices. This can be done by passing the `pairwise` keyword argument, which in the case of `DataFrame` inputs will yield a MultiIndexed `DataFrame` whose `index` are the dates in question. In the case of a single DataFrame argument the `pairwise` argument can even be omitted:

Note: Missing values are ignored and each entry is computed using the pairwise complete observations. Please see the *covariance section* for *caveats* associated with this method of calculating covariance and correlation matrices.

```
In [86]: covs = (df[['B', 'C', 'D']].rolling(window=50)
   ....:                              .cov(df[['A', 'B', 'C']], pairwise=True))
   ....:

In [87]: covs.loc['2002-09-22':]
Out[87]:
                     B         C         D
2002-09-22 A  1.367467  8.676734 -8.047366
           B  3.067315  0.865946 -1.052533
           C  0.865946  7.739761 -4.943924
2002-09-23 A  0.910343  8.669065 -8.443062
           B  2.625456  0.565152 -0.907654
           C  0.565152  7.825521 -5.367526
2002-09-24 A  0.463332  8.514509 -8.776514
           B  2.306695  0.267746 -0.732186
           C  0.267746  7.771425 -5.696962
2002-09-25 A  0.467976  8.198236 -9.162599
           B  2.307129  0.267287 -0.754080
           C  0.267287  7.466559 -5.822650
2002-09-26 A  0.545781  7.899084 -9.326238
           B  2.311058  0.322295 -0.844451
           C  0.322295  7.038237 -5.684445
```

```
In [88]: correls = df.rolling(window=50).corr()

In [89]: correls.loc['2002-09-22':]
Out[89]:
                     A         B         C         D
2002-09-22 A  1.000000  0.186397  0.744551 -0.769767
           B  0.186397  1.000000  0.177725 -0.240802
           C  0.744551  0.177725  1.000000 -0.712051
           D -0.769767 -0.240802 -0.712051  1.000000
2002-09-23 A  1.000000  0.134723  0.743113 -0.758758
...                ...       ...       ...       ...
2002-09-25 D -0.739160 -0.164179 -0.704686  1.000000
2002-09-26 A  1.000000  0.087756  0.727792 -0.736562
           B  0.087756  1.000000  0.079913 -0.179477
           C  0.727792  0.079913  1.000000 -0.692303
```

(continues on next page)

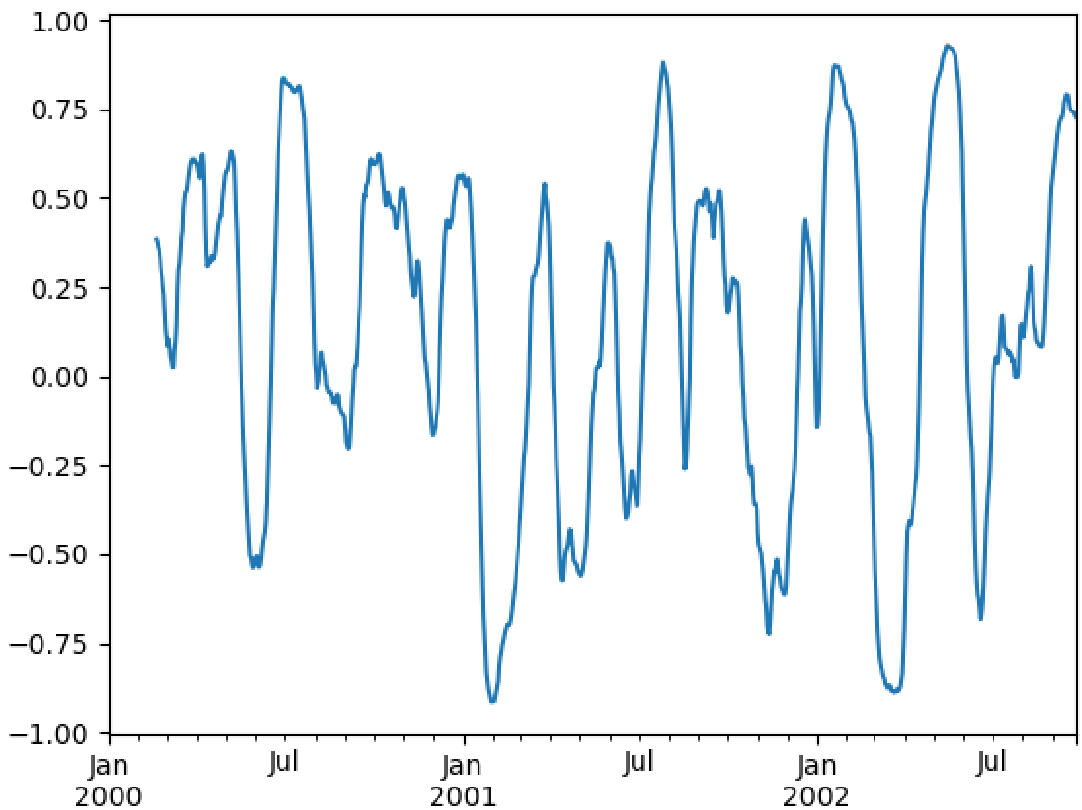
(continued from previous page)

```
          D -0.736562 -0.179477 -0.692303  1.000000

[20 rows x 4 columns]
```

You can efficiently retrieve the time series of correlations between two columns by reshaping and indexing:

```
In [90]: correls.unstack(1)[('A', 'C')].plot()
Out[90]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d45f510>
```



2.12.3 Aggregation

Once the `Rolling`, `Expanding` or `EWM` objects have been created, several methods are available to perform multiple computations on the data. These operations are similar to the *aggregating API*, *groupby API*, and *resample API*.

```
In [91]: dfa = pd.DataFrame(np.random.randn(1000, 3),
   ....:                    index=pd.date_range('1/1/2000', periods=1000),
   ....:                    columns=['A', 'B', 'C'])
   ....:

In [92]: r = dfa.rolling(window=60, min_periods=1)

In [93]: r
Out[93]: Rolling [window=60,min_periods=1,center=False,axis=0]
```

We can aggregate by passing a function to the entire DataFrame, or select a Series (or multiple Series) via standard `__getitem__`.

```
In [94]: r.aggregate(np.sum)
Out[94]:
                   A          B         C
2000-01-01 -0.289838  -0.370545 -1.284206
2000-01-02 -0.216612  -1.675528 -1.169415
2000-01-03  1.154661  -1.634017 -1.566620
2000-01-04  2.969393  -4.003274 -1.816179
2000-01-05  4.690630  -4.682017 -2.717209
...              ...        ...       ...
2002-09-22  2.860036  -9.270337  6.415245
2002-09-23  3.510163  -8.151439  5.177219
2002-09-24  6.524983 -10.168078  5.792639
2002-09-25  6.409626  -9.956226  5.704050
2002-09-26  5.093787  -7.074515  6.905823

[1000 rows x 3 columns]

In [95]: r['A'].aggregate(np.sum)
Out[95]:
2000-01-01   -0.289838
2000-01-02   -0.216612
2000-01-03    1.154661
2000-01-04    2.969393
2000-01-05    4.690630
                ...
2002-09-22    2.860036
2002-09-23    3.510163
2002-09-24    6.524983
2002-09-25    6.409626
2002-09-26    5.093787
Freq: D, Name: A, Length: 1000, dtype: float64

In [96]: r[['A', 'B']].aggregate(np.sum)
Out[96]:
                   A          B
2000-01-01 -0.289838  -0.370545
2000-01-02 -0.216612  -1.675528
2000-01-03  1.154661  -1.634017
2000-01-04  2.969393  -4.003274
2000-01-05  4.690630  -4.682017
...              ...        ...
2002-09-22  2.860036  -9.270337
2002-09-23  3.510163  -8.151439
2002-09-24  6.524983 -10.168078
2002-09-25  6.409626  -9.956226
2002-09-26  5.093787  -7.074515

[1000 rows x 2 columns]
```

As you can see, the result of the aggregation will have the selected columns, or all columns if none are selected.

Applying multiple functions

With windowed Series you can also pass a list of functions to do aggregation with, outputting a DataFrame:

```
In [97]: r['A'].agg([np.sum, np.mean, np.std])
Out[97]:
                 sum      mean       std
2000-01-01 -0.289838 -0.289838       NaN
2000-01-02 -0.216612 -0.108306  0.256725
2000-01-03  1.154661  0.384887  0.873311
2000-01-04  2.969393  0.742348  1.009734
2000-01-05  4.690630  0.938126  0.977914
...              ...       ...       ...
2002-09-22  2.860036  0.047667  1.132051
2002-09-23  3.510163  0.058503  1.134296
2002-09-24  6.524983  0.108750  1.144204
2002-09-25  6.409626  0.106827  1.142913
2002-09-26  5.093787  0.084896  1.151416

[1000 rows x 3 columns]
```

On a windowed DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [98]: r.agg([np.sum, np.mean])
Out[98]:
                   A                    B                   C
                 sum      mean        sum      mean       sum      mean
2000-01-01 -0.289838 -0.289838  -0.370545 -0.370545 -1.284206 -1.284206
2000-01-02 -0.216612 -0.108306  -1.675528 -0.837764 -1.169415 -0.584708
2000-01-03  1.154661  0.384887  -1.634017 -0.544672 -1.566620 -0.522207
2000-01-04  2.969393  0.742348  -4.003274 -1.000819 -1.816179 -0.454045
2000-01-05  4.690630  0.938126  -4.682017 -0.936403 -2.717209 -0.543442
...              ...       ...        ...       ...       ...       ...
2002-09-22  2.860036  0.047667  -9.270337 -0.154506  6.415245  0.106921
2002-09-23  3.510163  0.058503  -8.151439 -0.135857  5.177219  0.086287
2002-09-24  6.524983  0.108750 -10.168078 -0.169468  5.792639  0.096544
2002-09-25  6.409626  0.106827  -9.956226 -0.165937  5.704050  0.095068
2002-09-26  5.093787  0.084896  -7.074515 -0.117909  6.905823  0.115097

[1000 rows x 6 columns]
```

Passing a dict of functions has different behavior by default, see the next section.

Applying different functions to DataFrame columns

By passing a dict to aggregate you can apply a different aggregation to the columns of a DataFrame:

```
In [99]: r.agg({'A': np.sum, 'B': lambda x: np.std(x, ddof=1)})
Out[99]:
                   A         B
2000-01-01 -0.289838       NaN
2000-01-02 -0.216612  0.660747
2000-01-03  1.154661  0.689929
2000-01-04  2.969393  1.072199
2000-01-05  4.690630  0.939657
```

(continues on next page)

(continued from previous page)

```
...                 ...       ...
2002-09-22  2.860036  1.113208
2002-09-23  3.510163  1.132381
2002-09-24  6.524983  1.080963
2002-09-25  6.409626  1.082911
2002-09-26  5.093787  1.136199

[1000 rows x 2 columns]
```

The function names can also be strings. In order for a string to be valid it must be implemented on the windowed object

```
In [100]: r.agg({'A': 'sum', 'B': 'std'})
Out[100]:
                   A         B
2000-01-01 -0.289838       NaN
2000-01-02 -0.216612  0.660747
2000-01-03  1.154661  0.689929
2000-01-04  2.969393  1.072199
2000-01-05  4.690630  0.939657
...              ...       ...
2002-09-22  2.860036  1.113208
2002-09-23  3.510163  1.132381
2002-09-24  6.524983  1.080963
2002-09-25  6.409626  1.082911
2002-09-26  5.093787  1.136199

[1000 rows x 2 columns]
```

Furthermore you can pass a nested dict to indicate different aggregations on different columns.

```
In [101]: r.agg({'A': ['sum', 'std'], 'B': ['mean', 'std']})
Out[101]:
                   A                   B
                 sum       std      mean       std
2000-01-01 -0.289838       NaN -0.370545       NaN
2000-01-02 -0.216612  0.256725 -0.837764  0.660747
2000-01-03  1.154661  0.873311 -0.544672  0.689929
2000-01-04  2.969393  1.009734 -1.000819  1.072199
2000-01-05  4.690630  0.977914 -0.936403  0.939657
...              ...       ...       ...       ...
2002-09-22  2.860036  1.132051 -0.154506  1.113208
2002-09-23  3.510163  1.134296 -0.135857  1.132381
2002-09-24  6.524983  1.144204 -0.169468  1.080963
2002-09-25  6.409626  1.142913 -0.165937  1.082911
2002-09-26  5.093787  1.151416 -0.117909  1.136199

[1000 rows x 4 columns]
```

2.12.4 Expanding windows

A common alternative to rolling statistics is to use an *expanding* window, which yields the value of the statistic with all the data available up to that point in time.

These follow a similar interface to `.rolling`, with the `.expanding` method returning an `Expanding` object.

As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

```
In [102]: df.rolling(window=len(df), min_periods=1).mean()[:5]
Out[102]:
                   A         B         C         D
2000-01-01  0.314226 -0.001675  0.071823  0.892566
2000-01-02  0.654522 -0.171495  0.179278  0.853361
2000-01-03  0.708733 -0.064489 -0.238271  1.371111
2000-01-04  0.987613  0.163472 -0.919693  1.566485
2000-01-05  1.426971  0.288267 -1.358877  1.808650

In [103]: df.expanding(min_periods=1).mean()[:5]
Out[103]:
                   A         B         C         D
2000-01-01  0.314226 -0.001675  0.071823  0.892566
2000-01-02  0.654522 -0.171495  0.179278  0.853361
2000-01-03  0.708733 -0.064489 -0.238271  1.371111
2000-01-04  0.987613  0.163472 -0.919693  1.566485
2000-01-05  1.426971  0.288267 -1.358877  1.808650
```

These have a similar set of methods to `.rolling` methods.

Method summary

Function	Description
`count()`	Number of non-null observations
`sum()`	Sum of values
`mean()`	Mean of values
`median()`	Arithmetic median of values
`min()`	Minimum
`max()`	Maximum
`std()`	Unbiased standard deviation
`var()`	Unbiased variance
`skew()`	Unbiased skewness (3rd moment)
`kurt()`	Unbiased kurtosis (4th moment)
`quantile()`	Sample quantile (value at %)
`apply()`	Generic apply
`cov()`	Unbiased covariance (binary)
`corr()`	Correlation (binary)

Aside from not having a `window` parameter, these functions have the same interfaces as their `.rolling` counterparts. Like above, the parameters they all accept are:

- `min_periods`: threshold of non-null data points to require. Defaults to minimum needed to compute statistic. No `NaNs` will be output once `min_periods` non-null data points have been seen.
- `center`: boolean, whether to set the labels at the center (default is False).

Note: The output of the `.rolling` and `.expanding` methods do not return a `NaN` if there are at least `min_periods` non-null values in the current window. For example:

```
In [104]: sn = pd.Series([1, 2, np.nan, 3, np.nan, 4])

In [105]: sn
Out[105]:
0    1.0
1    2.0
2    NaN
3    3.0
4    NaN
5    4.0
dtype: float64

In [106]: sn.rolling(2).max()
Out[106]:
0    NaN
1    2.0
2    NaN
3    NaN
4    NaN
5    NaN
dtype: float64

In [107]: sn.rolling(2, min_periods=1).max()
Out[107]:
0    1.0
1    2.0
2    2.0
3    3.0
4    3.0
5    4.0
dtype: float64
```

In case of expanding functions, this differs from `cumsum()`, `cumprod()`, `cummax()`, and `cummin()`, which return `NaN` in the output wherever a `NaN` is encountered in the input. In order to match the output of `cumsum` with `expanding`, use `fillna()`:

```
In [108]: sn.expanding().sum()
Out[108]:
0     1.0
1     3.0
2     3.0
3     6.0
4     6.0
5    10.0
dtype: float64

In [109]: sn.cumsum()
Out[109]:
0     1.0
1     3.0
2     NaN
3     6.0
4     NaN
5    10.0
```

(continues on next page)

(continued from previous page)

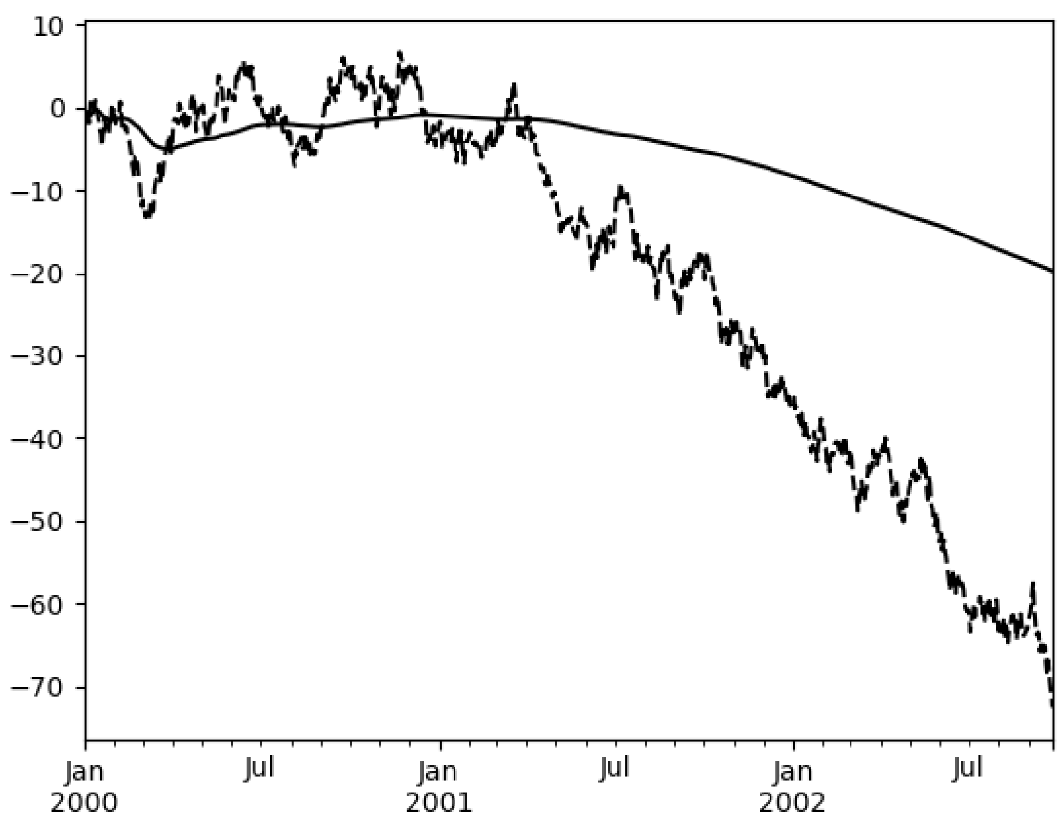
```
dtype: float64

In [110]: sn.cumsum().fillna(method='ffill')
Out[110]:
0     1.0
1     3.0
2     3.0
3     6.0
4     6.0
5    10.0
dtype: float64
```

An expanding window statistic will be more stable (and less responsive) than its rolling window counterpart as the increasing window size decreases the relative impact of an individual data point. As an example, here is the `mean()` output for the previous time series dataset:

```
In [111]: s.plot(style='k--')
Out[111]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d454890>

In [112]: s.expanding().mean().plot(style='k')
Out[112]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d454890>
```



2.12.5 Exponentially weighted windows

A related set of functions are exponentially weighted versions of several of the above statistics. A similar interface to `.rolling` and `.expanding` is accessed through the `.ewm` method to receive an `EWM` object. A number of expanding EW (exponentially weighted) methods are provided:

Function	Description
`mean()`	EW moving average
`var()`	EW moving variance
`std()`	EW moving standard deviation
`corr()`	EW moving correlation
`cov()`	EW moving covariance

In general, a weighted moving average is calculated as

$$y_t = \frac{\sum_{i=0}^t w_i x_{t-i}}{\sum_{i=0}^t w_i},$$

where x_t is the input, y_t is the result and the w_i are the weights.

The EW functions support two variants of exponential weights. The default, `adjust=True`, uses the weights $w_i = (1-\alpha)^i$ which gives

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ... + (1-\alpha)^t x_0}{1 + (1-\alpha) + (1-\alpha)^2 + ... + (1-\alpha)^t}$$

When `adjust=False` is specified, moving averages are calculated as

$$\begin{split}y_0 &= x_0\\ y_t &= (1-\alpha)y_{t-1} + \alpha x_t,\end{split}$$

which is equivalent to using weights

$$w_i = \begin{cases} \alpha(1-\alpha)^i & \text{if } i < t \\ (1-\alpha)^i & \text{if } i = t. \end{cases}$$

Note: These equations are sometimes written in terms of $\alpha' = 1 - \alpha$, e.g.

$$y_t = \alpha' y_{t-1} + (1-\alpha')x_t.$$

The difference between the above two variants arises because we are dealing with series which have finite history. Consider a series of infinite history, with `adjust=True`:

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...}{1 + (1-\alpha) + (1-\alpha)^2 + ...}$$

Noting that the denominator is a geometric series with initial term equal to 1 and a ratio of $1-\alpha$ we have

$$\begin{split}y_t &= \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...}{\frac{1}{1-(1-\alpha)}}\\ &= [x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...]\alpha\\ &= \alpha x_t + [(1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...]\alpha\\ &= \alpha x_t + (1-\alpha)[x_{t-1} + (1-\alpha)x_{t-2} + ...]\alpha\\ &= \alpha x_t + (1-\alpha)y_{t-1}\end{split}$$

which is the same expression as `adjust=False` above and therefore shows the equivalence of the two variants for infinite series. When `adjust=False`, we have $y_0 = x_0$ and $y_t = \alpha x_t + (1 - \alpha) y_{t-1}$. Therefore, there is an assumption that x_0 is not an ordinary value but rather an exponentially weighted moment of the infinite series up to that point.

One must have $0 < \alpha \leq 1$, and while it is possible to pass α directly, it's often easier to think about either the **span**, **center of mass (com)** or **half-life** of an EW moment:

$$\alpha = \begin{cases} \frac{2}{s + 1}, & \text{for span}\ s \geq 1\\ \frac{1}{1 + c}, & \text{for center of mass}\ c \geq 0\\ 1 - \exp^{\frac{\log 0.5}{h}}, & \text{for half-life}\ h > 0 \end{cases}$$

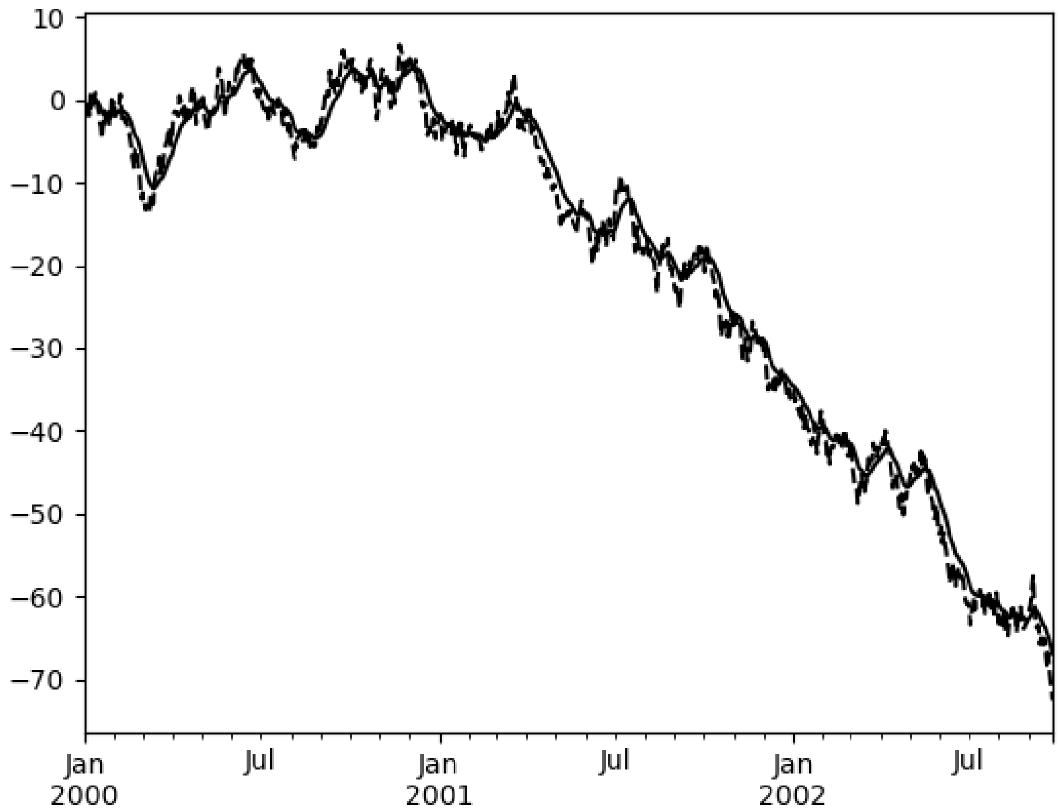
One must specify precisely one of **span**, **center of mass**, **half-life** and **alpha** to the EW functions:

- **Span** corresponds to what is commonly called an "N-day EW moving average".
- **Center of mass** has a more physical interpretation and can be thought of in terms of span: $c = (s - 1)/2$.
- **Half-life** is the period of time for the exponential weight to reduce to one half.
- **Alpha** specifies the smoothing factor directly.

Here is an example for a univariate time series:

```
In [113]: s.plot(style='k--')
Out[113]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d28a6d0>

In [114]: s.ewm(span=20).mean().plot(style='k')
Out[114]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d28a6d0>
```



EWM has a `min_periods` argument, which has the same meaning it does for all the `.expanding` and `.rolling` methods: no output values will be set until at least `min_periods` non-null values are encountered in the (expanding) window.

EWM also has an `ignore_na` argument, which determines how intermediate null values affect the calculation of the weights. When `ignore_na=False` (the default), weights are calculated based on absolute positions, so that intermediate null values affect the result. When `ignore_na=True`, weights are calculated by ignoring intermediate null values. For example, assuming `adjust=True`, if `ignore_na=False`, the weighted average of `3, NaN, 5` would be calculated as

$$\frac{(1-\alpha)^2 \cdot 3 + 1 \cdot 5}{(1-\alpha)^2 + 1}.$$

Whereas if `ignore_na=True`, the weighted average would be calculated as

$$\frac{(1-\alpha) \cdot 3 + 1 \cdot 5}{(1-\alpha) + 1}.$$

The `var()`, `std()`, and `cov()` functions have a `bias` argument, specifying whether the result should contain biased or unbiased statistics. For example, if `bias=True`, `ewmvar(x)` is calculated as `ewmvar(x) = ewma(x**2) - ewma(x)**2`; whereas if `bias=False` (the default), the biased variance statistics are scaled by debiasing factors

$$\frac{\left(\sum_{i=0}^t w_i\right)^2}{\left(\sum_{i=0}^t w_i\right)^2 - \sum_{i=0}^t w_i^2}.$$

(For $w_i = 1$, this reduces to the usual $N/(N-1)$ factor, with $N = t + 1$.) See Weighted Sample Variance on Wikipedia for further details.

2.13 Group By: split-apply-combine

By "group by" we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a data structure.

Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups. In the apply step, we might wish to do one of the following:

- **Aggregation**: compute a summary statistic (or statistics) for each group. Some examples:
  - Compute group sums or means.
  - Compute group sizes / counts.
- **Transformation**: perform some group-specific computations and return a like-indexed object. Some examples:
  - Standardize data (zscore) within a group.
  - Filling NAs within groups with a value derived from each group.
- **Filtration**: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:
  - Discard data that belongs to groups with only a few members.
  - Filter out data based on the group sum or mean.
- Some combination of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it doesn't fit into either of the above two categories.

Since the set of object instance methods on pandas data structures are generally rich and expressive, we often simply want to invoke, say, a DataFrame function on each group. The name GroupBy should be quite familiar to those who have used a SQL-based tool (or `itertools`), in which you can write code like:

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

We aim to make operations like this natural and easy to express using pandas. We'll address each area of GroupBy functionality then provide some non-trivial examples / use cases.

See the *cookbook* for some advanced strategies.

2.13.1 Splitting an object into groups

pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names. To create a GroupBy object (more on what the GroupBy object is later), you may do the following:

```
In [1]: df = pd.DataFrame([('bird', 'Falconiformes', 389.0),
   ...:                    ('bird', 'Psittaciformes', 24.0),
   ...:                    ('mammal', 'Carnivora', 80.2),
   ...:                    ('mammal', 'Primates', np.nan),
   ...:                    ('mammal', 'Carnivora', 58)],
   ...:                   index=['falcon', 'parrot', 'lion', 'monkey', 'leopard'],
   ...:                   columns=('class', 'order', 'max_speed'))
   ...: 

In [2]: df
Out[2]: 
          class           order  max_speed
falcon     bird   Falconiformes      389.0
parrot     bird  Psittaciformes       24.0
lion     mammal       Carnivora       80.2
monkey   mammal        Primates        NaN
leopard  mammal       Carnivora       58.0

# default is axis=0
In [3]: grouped = df.groupby('class')

In [4]: grouped = df.groupby('order', axis='columns')

In [5]: grouped = df.groupby(['class', 'order'])
```

The mapping can be specified many different ways:

- A Python function, to be called on each of the axis labels.
- A list or NumPy array of the same length as the selected axis.
- A dict or `Series`, providing a `label -> group name` mapping.
- For `DataFrame` objects, a string indicating a column to be used to group. Of course `df.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`, but it makes life simpler.
- For `DataFrame` objects, a string indicating an index level to be used to group.
- A list of any of the above things.

Collectively we refer to the grouping objects as the **keys**. For example, consider the following `DataFrame`:

Note: A string passed to `groupby` may refer to either a column or an index level. If a string matches both a column name and an index level name, a `ValueError` will be raised.

```
In [6]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
   ...:                          'foo', 'bar', 'foo', 'foo'],
   ...:                    'B': ['one', 'one', 'two', 'three',
   ...:                          'two', 'two', 'one', 'three'],
   ...:                    'C': np.random.randn(8),
   ...:                    'D': np.random.randn(8)})
   ...: 
```

(continues on next page)

(continued from previous page)

```
In [7]: df
Out[7]:
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

On a DataFrame, we obtain a GroupBy object by calling *groupby()*. We could naturally group by either the A or B columns, or both:

```
In [8]: grouped = df.groupby('A')

In [9]: grouped = df.groupby(['A', 'B'])
```

New in version 0.24.

If we also have a MultiIndex on columns A and B, we can group by all but the specified columns

```
In [10]: df2 = df.set_index(['A', 'B'])

In [11]: grouped = df2.groupby(level=df2.index.names.difference(['B']))

In [12]: grouped.sum()
Out[12]:
            C         D
A
bar -1.591710 -1.739537
foo -0.752861 -1.402938
```

These will split the DataFrame on its index (rows). We could also split by the columns:

```
In [13]: def get_letter_type(letter):
   ....:     if letter.lower() in 'aeiou':
   ....:         return 'vowel'
   ....:     else:
   ....:         return 'consonant'
   ....:

In [14]: grouped = df.groupby(get_letter_type, axis=1)
```

pandas *Index* objects support duplicate values. If a non-unique index is used as the group key in a groupby operation, all values for the same index value will be considered to be in one group and thus the output of aggregation functions will only contain unique index values:

```
In [15]: lst = [1, 2, 3, 1, 2, 3]

In [16]: s = pd.Series([1, 2, 3, 10, 20, 30], lst)

In [17]: grouped = s.groupby(level=0)

In [18]: grouped.first()
Out[18]:
```

(continues on next page)

(continued from previous page)

```
1    1
2    2
3    3
dtype: int64

In [19]: grouped.last()
Out[19]:
1    10
2    20
3    30
dtype: int64

In [20]: grouped.sum()
Out[20]:
1    11
2    22
3    33
dtype: int64
```

Note that **no splitting occurs** until it's needed. Creating the GroupBy object only verifies that you've passed a valid mapping.

Note: Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though can't be guaranteed to be the most efficient). You can get quite creative with the label mapping functions.

GroupBy sorting

By default the group keys are sorted during the `groupby` operation. You may however pass `sort=False` for potential speedups:

```
In [21]: df2 = pd.DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})

In [22]: df2.groupby(['X']).sum()
Out[22]:
   Y
X
A  7
B  3

In [23]: df2.groupby(['X'], sort=False).sum()
Out[23]:
   Y
X
B  3
A  7
```

Note that `groupby` will preserve the order in which *observations* are sorted *within* each group. For example, the groups created by `groupby()` below are in the order they appeared in the original `DataFrame`:

```
In [24]: df3 = pd.DataFrame({'X': ['A', 'B', 'A', 'B'], 'Y': [1, 4, 3, 2]})

In [25]: df3.groupby(['X']).get_group('A')
Out[25]:
```

(continues on next page)

(continued from previous page)

```
   X  Y
0  A  1
2  A  3

In [26]: df3.groupby(['X']).get_group('B')
Out[26]:
   X  Y
1  B  4
3  B  2
```

GroupBy object attributes

The `groups` attribute is a dict whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. In the above example we have:

```
In [27]: df.groupby('A').groups
Out[27]:
{'bar': Int64Index([1, 3, 5], dtype='int64'),
 'foo': Int64Index([0, 2, 4, 6, 7], dtype='int64')}

In [28]: df.groupby(get_letter_type, axis=1).groups
Out[28]:
{'consonant': Index(['B', 'C', 'D'], dtype='object'),
 'vowel': Index(['A'], dtype='object')}
```

Calling the standard Python `len` function on the GroupBy object just returns the length of the `groups` dict, so it is largely just a convenience:

```
In [29]: grouped = df.groupby(['A', 'B'])

In [30]: grouped.groups
Out[30]:
{('bar', 'one'): Int64Index([1], dtype='int64'),
 ('bar', 'three'): Int64Index([3], dtype='int64'),
 ('bar', 'two'): Int64Index([5], dtype='int64'),
 ('foo', 'one'): Int64Index([0, 6], dtype='int64'),
 ('foo', 'three'): Int64Index([7], dtype='int64'),
 ('foo', 'two'): Int64Index([2, 4], dtype='int64')}

In [31]: len(grouped)
Out[31]: 6
```

`GroupBy` will tab complete column names (and other attributes):

```
In [32]: df
Out[32]:
               height      weight  gender
2000-01-01  42.849980  157.500553    male
2000-01-02  49.607315  177.340407    male
2000-01-03  56.293531  171.524640    male
2000-01-04  48.421077  144.251986  female
2000-01-05  46.556882  152.526206    male
2000-01-06  68.448851  168.272968  female
2000-01-07  70.757698  136.431469    male
2000-01-08  58.909500  176.499753  female
```

(continues on next page)