(continued from previous page)

```
2000-01-09  76.435631  174.094104  female
2000-01-10  45.306120  177.540920    male

In [33]: gb = df.groupby('gender')
```

```
In [34]: gb.<TAB>  # noqa: E225, E999
gb.agg        gb.boxplot    gb.cummin     gb.describe   gb.filter     gb.get_group ␣
↪gb.height     gb.last       gb.median     gb.ngroups    gb.plot       gb.rank      ␣
↪gb.std        gb.transform
gb.aggregate  gb.count      gb.cumprod    gb.dtype      gb.first      gb.groups    ␣
↪gb.hist       gb.max        gb.min        gb.nth        gb.prod       gb.resample  ␣
↪gb.sum        gb.var
gb.apply      gb.cummax     gb.cumsum     gb.fillna     gb.gender     gb.head      ␣
↪gb.indices    gb.mean       gb.name       gb.ohlc       gb.quantile   gb.size      ␣
↪gb.tail       gb.weight
```

GroupBy with MultiIndex

With *hierarchically-indexed data*, it's quite natural to group by one of the levels of the hierarchy.

Let's create a Series with a two-level `MultiIndex`.

```
In [35]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....:

In [36]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [37]: s = pd.Series(np.random.randn(8), index=index)

In [38]: s
Out[38]:
first  second
bar    one      -0.919854
       two      -0.042379
baz    one       1.247642
       two      -0.009920
foo    one       0.290213
       two       0.495767
qux    one       0.362949
       two       1.548106
dtype: float64
```

We can then group by one of the levels in `s`.

```
In [39]: grouped = s.groupby(level=0)

In [40]: grouped.sum()
Out[40]:
first
bar   -0.962232
baz    1.237723
foo    0.785980
qux    1.911055
dtype: float64
```

If the MultiIndex has names specified, these can be passed instead of the level number:

```
In [41]: s.groupby(level='second').sum()
Out[41]:
second
one    0.980950
two    1.991575
dtype: float64
```

The aggregation functions such as `sum` will take the level parameter directly. Additionally, the resulting index will be named according to the chosen level:

```
In [42]: s.sum(level='second')
Out[42]:
second
one    0.980950
two    1.991575
dtype: float64
```

Grouping with multiple levels is supported.

```
In [43]: s
Out[43]:
first  second  third
bar    doo     one     -1.131345
               two     -0.089329
baz    bee     one      0.337863
               two     -0.945867
foo    bop     one     -0.932132
               two      1.956030
qux    bop     one      0.017587
               two     -0.016692
dtype: float64

In [44]: s.groupby(level=['first', 'second']).sum()
Out[44]:
first  second
bar    doo      -1.220674
baz    bee      -0.608004
foo    bop       1.023898
qux    bop       0.000895
dtype: float64
```

Index level names may be supplied as keys.

```
In [45]: s.groupby(['first', 'second']).sum()
Out[45]:
first  second
bar    doo      -1.220674
baz    bee      -0.608004
foo    bop       1.023898
qux    bop       0.000895
dtype: float64
```

More on the `sum` function and aggregation later.

Grouping DataFrame with Index levels and columns

A DataFrame may be grouped by a combination of columns and index levels by specifying the column names as strings and the index levels as `pd.Grouper` objects.

```
In [46]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....:

In [47]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [48]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
   ....:                    'B': np.arange(8)},
   ....:                   index=index)
   ....:

In [49]: df
Out[49]:
              A  B
first second
bar   one     1  0
      two     1  1
baz   one     1  2
      two     1  3
foo   one     2  4
      two     2  5
qux   one     3  6
      two     3  7
```

The following example groups `df` by the `second` index level and the `A` column.

```
In [50]: df.groupby([pd.Grouper(level=1), 'A']).sum()
Out[50]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

Index levels may also be specified by name.

```
In [51]: df.groupby([pd.Grouper(level='second'), 'A']).sum()
Out[51]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

Index level names may be specified as keys directly to `groupby`.

```
In [52]: df.groupby(['second', 'A']).sum()
Out[52]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

DataFrame column selection in GroupBy

Once you have created the GroupBy object from a DataFrame, you might want to do something different for each of the columns. Thus, using `[]` similar to getting a column from a DataFrame, you can do:

```
In [53]: grouped = df.groupby(['A'])

In [54]: grouped_C = grouped['C']

In [55]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```
In [56]: df['C'].groupby(df['A'])
Out[56]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7f533ded0c50>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

2.13.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```
In [57]: grouped = df.groupby('A')

In [58]: for name, group in grouped:
   ....:     print(name)
   ....:     print(group)
   ....:
bar
     A      B         C         D
1  bar    one  0.254161  1.511763
3  bar  three  0.215897 -0.990582
5  bar    two -0.077118  1.211526
foo
     A      B         C         D
0  foo    one -0.575247  1.346061
2  foo    two -1.143704  1.627081
4  foo    two  1.193555 -0.441652
6  foo    one -0.408530  0.268520
7  foo  three -0.862495  0.024580
```

In the case of grouping by multiple keys, the group name will be a tuple:

```
In [59]: for name, group in df.groupby(['A', 'B']):
   ....:     print(name)
   ....:     print(group)
   ....:
('bar', 'one')
     A    B         C         D
1  bar  one  0.254161  1.511763
('bar', 'three')
     A      B         C         D
3  bar  three  0.215897 -0.990582
('bar', 'two')
     A    B         C         D
5  bar  two -0.077118  1.211526
('foo', 'one')
     A    B         C         D
0  foo  one -0.575247  1.346061
6  foo  one -0.408530  0.268520
('foo', 'three')
     A      B         C        D
7  foo  three -0.862495  0.02458
('foo', 'two')
     A    B         C         D
2  foo  two -1.143704  1.627081
4  foo  two  1.193555 -0.441652
```

See *Iterating through groups*.

2.13.3 Selecting a group

A single group can be selected using `get_group()`:

```
In [60]: grouped.get_group('bar')
Out[60]:
     A      B         C         D
1  bar    one  0.254161  1.511763
3  bar  three  0.215897 -0.990582
5  bar    two -0.077118  1.211526
```

Or for an object grouped on multiple columns:

```
In [61]: df.groupby(['A', 'B']).get_group(('bar', 'one'))
Out[61]:
     A    B         C         D
1  bar  one  0.254161  1.511763
```

2.13.4 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data. These operations are similar to the *aggregating API*, *window functions API*, and *resample API*.

An obvious one is aggregation via the `aggregate()` or equivalently `agg()` method:

```
In [62]: grouped = df.groupby('A')

In [63]: grouped.aggregate(np.sum)
Out[63]:
            C         D
A
bar  0.392940  1.732707
foo -1.796421  2.824590

In [64]: grouped = df.groupby(['A', 'B'])

In [65]: grouped.aggregate(np.sum)
Out[65]:
                  C         D
A   B
bar one    0.254161  1.511763
    three  0.215897 -0.990582
    two   -0.077118  1.211526
foo one   -0.983776  1.614581
    three -0.862495  0.024580
    two    0.049851  1.185429
```

As you can see, the result of the aggregation will have the group names as the new index along the grouped axis. In the case of multiple keys, the result is a *MultiIndex* by default, though this can be changed by using the `as_index` option:

```
In [66]: grouped = df.groupby(['A', 'B'], as_index=False)

In [67]: grouped.aggregate(np.sum)
Out[67]:
     A      B         C         D
0  bar    one  0.254161  1.511763
1  bar  three  0.215897 -0.990582
2  bar    two -0.077118  1.211526
3  foo    one -0.983776  1.614581
4  foo  three -0.862495  0.024580
5  foo    two  0.049851  1.185429

In [68]: df.groupby('A', as_index=False).sum()
Out[68]:
     A         C         D
0  bar  0.392940  1.732707
1  foo -1.796421  2.824590
```

Note that you could use the `reset_index` DataFrame function to achieve the same result as the column names are stored in the resulting `MultiIndex`:

```
In [69]: df.groupby(['A', 'B']).sum().reset_index()
Out[69]:
     A      B         C         D
0  bar    one  0.254161  1.511763
```

(continues on next page)

(continued from previous page)

```
1  bar  three  0.215897 -0.990582
2  bar    two -0.077118  1.211526
3  foo    one -0.983776  1.614581
4  foo  three -0.862495  0.024580
5  foo    two  0.049851  1.185429
```

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the `size` method. It returns a Series whose index are the group names and whose values are the sizes of each group.

```
In [70]: grouped.size()
Out[70]:
A    B
bar  one      1
     three    1
     two      1
foo  one      2
     three    1
     two      2
dtype: int64
```

```
In [71]: grouped.describe()
Out[71]:
      C                                                                    ...         D    ␣
↪
  count      mean       std       min       25%       50%       75%  ...      mean    ␣
↪   std       min       25%       50%       75%       max
0   1.0  0.254161       NaN  0.254161  0.254161  0.254161  0.254161  ...  1.511763    ␣
↪   NaN  1.511763  1.511763  1.511763  1.511763  1.511763
1   1.0  0.215897       NaN  0.215897  0.215897  0.215897  0.215897  ... -0.990582    ␣
↪   NaN -0.990582 -0.990582 -0.990582 -0.990582 -0.990582
2   1.0 -0.077118       NaN -0.077118 -0.077118 -0.077118 -0.077118  ...  1.211526    ␣
↪   NaN  1.211526  1.211526  1.211526  1.211526  1.211526
3   2.0 -0.491888  0.117887 -0.575247 -0.533567 -0.491888 -0.450209  ...  0.807291  0.
↪761937  0.268520  0.537905  0.807291  1.076676  1.346061
4   1.0 -0.862495       NaN -0.862495 -0.862495 -0.862495 -0.862495  ...  0.024580    ␣
↪   NaN  0.024580  0.024580  0.024580  0.024580  0.024580
5   2.0  0.024925  1.652692 -1.143704 -0.559389  0.024925  0.609240  ...  0.592714  1.
↪462816 -0.441652  0.075531  0.592714  1.109898  1.627081

[6 rows x 16 columns]
```

Note: Aggregation functions **will not** return the groups that you are aggregating over if they are named *columns*, when `as_index=True`, the default. The grouped columns will be the **indices** of the returned object.

Passing `as_index=False` **will** return the groups that you are aggregating over, if they are named *columns*.

Aggregating functions are the ones that reduce the dimension of the returned objects. Some common aggregating functions are tabulated below:

Function	Description
mean()	Compute mean of groups
sum()	Compute sum of group values
size()	Compute group sizes
count()	Compute count of group
std()	Standard deviation of groups
var()	Compute variance of groups
sem()	Standard error of the mean of groups
describe()	Generates descriptive statistics
first()	Compute first of group values
last()	Compute last of group values
nth()	Take nth value, or a subset if n is a list
min()	Compute min of group values
max()	Compute max of group values

The aggregating functions above will exclude NA values. Any function which reduces a `Series` to a scalar value is an aggregation function and will work, a trivial example is `df.groupby('A').agg(lambda ser: 1)`. Note that `nth()` can act as a reducer *or* a filter, see *here*.

Applying multiple functions at once

With grouped `Series` you can also pass a list or dict of functions to do aggregation with, outputting a DataFrame:

```
In [72]: grouped = df.groupby('A')

In [73]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[73]:
          sum      mean       std
A
bar  0.392940  0.130980  0.181231
foo -1.796421 -0.359284  0.912265
```

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

```
In [74]: grouped.agg([np.sum, np.mean, np.std])
Out[74]:
            C                             D
          sum      mean       std       sum      mean       std
A
bar  0.392940  0.130980  0.181231  1.732707  0.577569  1.366330
foo -1.796421 -0.359284  0.912265  2.824590  0.564918  0.884785
```

The resulting aggregations are named for the functions themselves. If you need to rename, then you can add in a chained operation for a `Series` like this:

```
In [75]: (grouped['C'].agg([np.sum, np.mean, np.std])
   ....:              .rename(columns={'sum': 'foo',
   ....:                               'mean': 'bar',
   ....:                               'std': 'baz'}))
   ....:
Out[75]:
          foo       bar       baz
A
```

(continues on next page)

(continued from previous page)

```
bar  0.392940  0.130980  0.181231
foo -1.796421 -0.359284  0.912265
```

For a grouped DataFrame, you can rename in a similar manner:

```
In [76]: (grouped.agg([np.sum, np.mean, np.std])
   ....:         .rename(columns={'sum': 'foo',
   ....:                          'mean': 'bar',
   ....:                          'std': 'baz'}))
   ....: 
Out[76]: 
            C                             D                    
          foo       bar       baz       foo       bar       baz
A                                                              
bar  0.392940  0.130980  0.181231  1.732707  0.577569  1.366330
foo -1.796421 -0.359284  0.912265  2.824590  0.564918  0.884785
```

Note: In general, the output column names should be unique. You can't apply the same function (or two functions with the same name) to the same column.

```
In [77]: grouped['C'].agg(['sum', 'sum'])
Out[77]: 
          sum       sum
A                      
bar  0.392940  0.392940
foo -1.796421 -1.796421
```

Pandas *does* allow you to provide multiple lambdas. In this case, pandas will mangle the name of the (nameless) lambda functions, appending _<i> to each subsequent lambda.

```
In [78]: grouped['C'].agg([lambda x: x.max() - x.min(),
   ....:                   lambda x: x.median() - x.mean()])
   ....: 
Out[78]: 
     <lambda_0>  <lambda_1>
A                          
bar    0.331279    0.084917
foo    2.337259   -0.215962
```

Named aggregation

New in version 0.25.0.

To support column-specific aggregation *with control over the output column names*, pandas accepts the special syntax in `GroupBy.agg()`, known as "named aggregation", where

- The keywords are the *output* column names
- The values are tuples whose first element is the column to select and the second element is the aggregation to apply to that column. Pandas provides the `pandas.NamedAgg` namedtuple with the fields `['column', 'aggfunc']` to make it clearer what the arguments are. As usual, the aggregation can be a callable or a string alias.

```
In [79]: animals = pd.DataFrame({'kind': ['cat', 'dog', 'cat', 'dog'],
   ....:                         'height': [9.1, 6.0, 9.5, 34.0],
   ....:                         'weight': [7.9, 7.5, 9.9, 198.0]})
   ....:

In [80]: animals
Out[80]:
  kind  height  weight
0  cat     9.1     7.9
1  dog     6.0     7.5
2  cat     9.5     9.9
3  dog    34.0   198.0

In [81]: animals.groupby("kind").agg(
   ....:     min_height=pd.NamedAgg(column='height', aggfunc='min'),
   ....:     max_height=pd.NamedAgg(column='height', aggfunc='max'),
   ....:     average_weight=pd.NamedAgg(column='weight', aggfunc=np.mean),
   ....: )
   ....:
Out[81]:
      min_height  max_height  average_weight
kind
cat          9.1         9.5            8.90
dog          6.0        34.0          102.75
```

`pandas.NamedAgg` is just a `namedtuple`. Plain tuples are allowed as well.

```
In [82]: animals.groupby("kind").agg(
   ....:     min_height=('height', 'min'),
   ....:     max_height=('height', 'max'),
   ....:     average_weight=('weight', np.mean),
   ....: )
   ....:
Out[82]:
      min_height  max_height  average_weight
kind
cat          9.1         9.5            8.90
dog          6.0        34.0          102.75
```

If your desired output column names are not valid python keywords, construct a dictionary and unpack the keyword arguments

```
In [83]: animals.groupby("kind").agg(**{
   ....:     'total weight': pd.NamedAgg(column='weight', aggfunc=sum),
   ....: })
   ....:
Out[83]:
      total weight
kind
cat           17.8
dog          205.5
```

Additional keyword arguments are not passed through to the aggregation functions. Only pairs of `(column, aggfunc)` should be passed as `**kwargs`. If your aggregation functions requires additional arguments, partially apply them with `functools.partial()`.

Note: For Python 3.5 and earlier, the order of `**kwargs` in a functions was not preserved. This means that the output column ordering would not be consistent. To ensure consistent ordering, the keys (and so output columns) will always be sorted for Python 3.5.

Named aggregation is also valid for Series groupby aggregations. In this case there's no column selection, so the values are just the functions.

```
In [84]: animals.groupby("kind").height.agg(
   ....:     min_height='min',
   ....:     max_height='max',
   ....: )
   ....: 
Out[84]: 
      min_height  max_height
kind                        
cat          9.1         9.5
dog          6.0        34.0
```

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 [85]: grouped.agg({'C': np.sum,
   ....:              'D': lambda x: np.std(x, ddof=1)})
   ....: 
Out[85]: 
            C         D
A                      
bar  0.392940  1.366330
foo -1.796421  0.884785
```

The function names can also be strings. In order for a string to be valid it must be either implemented on GroupBy or available via *dispatching*:

```
In [86]: grouped.agg({'C': 'sum', 'D': 'std'})
Out[86]: 
            C         D
A                      
bar  0.392940  1.366330
foo -1.796421  0.884785
```

Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, `std`, and `sem`, have optimized Cython implementations:

```
In [87]: df.groupby('A').sum()
Out[87]: 
            C         D
A                      
bar  0.392940  1.732707
foo -1.796421  2.824590

In [88]: df.groupby(['A', 'B']).mean()
Out[88]: 
                  C         D
```

(continues on next page)

(continued from previous page)

```
A   B
bar one     0.254161  1.511763
    three   0.215897 -0.990582
    two    -0.077118  1.211526
foo one    -0.491888  0.807291
    three  -0.862495  0.024580
    two     0.024925  0.592714
```

Of course `sum` and `mean` are implemented on pandas objects, so the above code would work even without the special versions via dispatching (see below).

2.13.5 Transformation

The `transform` method returns an object that is indexed the same (same size) as the one being grouped. The transform function must:

- Return a result that is either the same size as the group chunk or broadcastable to the size of the group chunk (e.g., a scalar, `grouped.transform(lambda x: x.iloc[-1])`).
- Operate column-by-column on the group chunk. The transform is applied to the first group chunk using chunk.apply.
- Not perform in-place operations on the group chunk. Group chunks should be treated as immutable, and changes to a group chunk may produce unexpected results. For example, when using `fillna`, `inplace` must be `False` (`grouped.transform(lambda x: x.fillna(inplace=False))`).
- (Optionally) operates on the entire group chunk. If this is supported, a fast path is used starting from the *second* chunk.

For example, suppose we wished to standardize the data within each group:

```
In [89]: index = pd.date_range('10/1/1999', periods=1100)

In [90]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)

In [91]: ts = ts.rolling(window=100, min_periods=100).mean().dropna()

In [92]: ts.head()
Out[92]:
2000-01-08    0.779333
2000-01-09    0.778852
2000-01-10    0.786476
2000-01-11    0.782797
2000-01-12    0.798110
Freq: D, dtype: float64

In [93]: ts.tail()
Out[93]:
2002-09-30    0.660294
2002-10-01    0.631095
2002-10-02    0.673601
2002-10-03    0.709213
2002-10-04    0.719369
Freq: D, dtype: float64

In [94]: transformed = (ts.groupby(lambda x: x.year)
```

(continues on next page)

(continued from previous page)

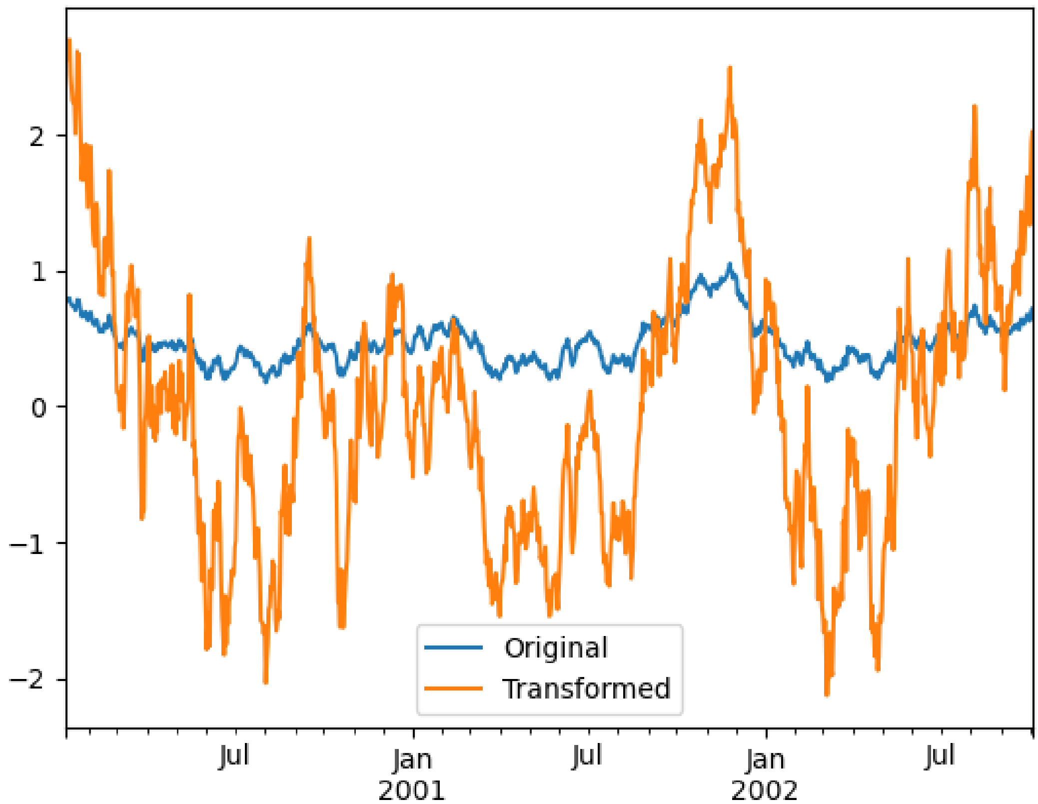
```
   ....:                   .transform(lambda x: (x - x.mean()) / x.std()))
   ....:
```

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```
# Original Data
In [95]: grouped = ts.groupby(lambda x: x.year)

In [96]: grouped.mean()
Out[96]:
2000    0.442441
2001    0.526246
2002    0.459365
dtype: float64

In [97]: grouped.std()
Out[97]:
2000    0.131752
2001    0.210945
2002    0.128753
dtype: float64

# Transformed Data
In [98]: grouped_trans = transformed.groupby(lambda x: x.year)

In [99]: grouped_trans.mean()
Out[99]:
2000    1.168208e-15
2001    1.454544e-15
2002    1.726657e-15
dtype: float64

In [100]: grouped_trans.std()
Out[100]:
2000    1.0
2001    1.0
2002    1.0
dtype: float64
```

We can also visually compare the original and transformed data sets.

```
In [101]: compare = pd.DataFrame({'Original': ts, 'Transformed': transformed})

In [102]: compare.plot()
Out[102]: <matplotlib.axes._subplots.AxesSubplot at 0x7f534055d7d0>
```



Transformation functions that have lower dimension outputs are broadcast to match the shape of the input array.

```
In [103]: ts.groupby(lambda x: x.year).transform(lambda x: x.max() - x.min())
Out[103]:
2000-01-08    0.623893
2000-01-09    0.623893
2000-01-10    0.623893
2000-01-11    0.623893
2000-01-12    0.623893
                ...
2002-09-30    0.558275
2002-10-01    0.558275
2002-10-02    0.558275
2002-10-03    0.558275
2002-10-04    0.558275
Freq: D, Length: 1001, dtype: float64
```

Alternatively, the built-in methods could be used to produce the same outputs.

```
In [104]: max = ts.groupby(lambda x: x.year).transform('max')

In [105]: min = ts.groupby(lambda x: x.year).transform('min')

In [106]: max - min
Out[106]:
```

(continues on next page)

(continued from previous page)

```
2000-01-08    0.623893
2000-01-09    0.623893
2000-01-10    0.623893
2000-01-11    0.623893
2000-01-12    0.623893
                ...
2002-09-30    0.558275
2002-10-01    0.558275
2002-10-02    0.558275
2002-10-03    0.558275
2002-10-04    0.558275
Freq: D, Length: 1001, dtype: float64
```

Another common data transform is to replace missing data with the group mean.

```
In [107]: data_df
Out[107]:
            A         B         C
0    1.539708 -1.166480  0.533026
1    1.302092 -0.505754       NaN
2   -0.371983  1.104803 -0.651520
3   -1.309622  1.118697 -1.161657
4   -1.924296  0.396437  0.812436
..        ...       ...       ...
995 -0.093110  0.683847 -0.774753
996 -0.185043  1.438572       NaN
997 -0.394469 -0.642343  0.011374
998 -1.174126  1.857148       NaN
999  0.234564  0.517098  0.393534

[1000 rows x 3 columns]

In [108]: countries = np.array(['US', 'UK', 'GR', 'JP'])

In [109]: key = countries[np.random.randint(0, 4, 1000)]

In [110]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [111]: grouped.count()
Out[111]:
      A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [112]: transformed = grouped.transform(lambda x: x.fillna(x.mean()))
```

We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.

```
In [113]: grouped_trans = transformed.groupby(key)

In [114]: grouped.mean()  # original group means
Out[114]:
```

(continues on next page)

(continued from previous page)

```
             A         B         C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603

In [115]: grouped_trans.mean()  # transformation did not change group means
Out[115]: 
             A         B         C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603

In [116]: grouped.count()  # original has some missing data points
Out[116]: 
      A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [117]: grouped_trans.count()  # counts after transformation
Out[117]: 
      A    B    C
GR  228  228  228
JP  267  267  267
UK  247  247  247
US  258  258  258

In [118]: grouped_trans.size()  # Verify non-NA count equals group size
Out[118]: 
GR    228
JP    267
UK    247
US    258
dtype: int64
```

Note: Some functions will automatically transform the input when applied to a GroupBy object, but returning an object of the same shape as the original. Passing `as_index=False` will not affect these transformation methods.

For example: `fillna, ffill, bfill, shift.`.

```
In [119]: grouped.ffill()
Out[119]: 
            A         B         C
0    1.539708 -1.166480  0.533026
1    1.302092 -0.505754  0.533026
2   -0.371983  1.104803 -0.651520
3   -1.309622  1.118697 -1.161657
4   -1.924296  0.396437  0.812436
..        ...       ...       ...
995 -0.093110  0.683847 -0.774753
996 -0.185043  1.438572 -0.774753
997 -0.394469 -0.642343  0.011374
998 -1.174126  1.857148 -0.774753
```

(continues on next page)

(continued from previous page)

```
999  0.234564  0.517098  0.393534

[1000 rows x 3 columns]
```

Window and resample operations

It is possible to use `resample()`, `expanding()` and `rolling()` as methods on groupbys.

The example below will apply the `rolling()` method on the samples of the column B based on the groups of column A.

```
In [120]: df_re = pd.DataFrame({'A': [1] * 10 + [5] * 10,
   .....:                       'B': np.arange(20)})
   .....:

In [121]: df_re
Out[121]:
    A   B
0   1   0
1   1   1
2   1   2
3   1   3
4   1   4
.. ..  ..
15  5  15
16  5  16
17  5  17
18  5  18
19  5  19

[20 rows x 2 columns]

In [122]: df_re.groupby('A').rolling(4).B.mean()
Out[122]:
A
1  0      NaN
   1      NaN
   2      NaN
   3      1.5
   4      2.5
         ...
5  15    13.5
   16    14.5
   17    15.5
   18    16.5
   19    17.5
Name: B, Length: 20, dtype: float64
```

The `expanding()` method will accumulate a given operation (`sum()` in the example) for all the members of each particular group.

```
In [123]: df_re.groupby('A').expanding().sum()
Out[123]:
         A      B
```

(continues on next page)

(continued from previous page)

```
A
1 0     1.0     0.0
  1     2.0     1.0
  2     3.0     3.0
  3     4.0     6.0
  4     5.0    10.0
...     ...     ...
5 15   30.0    75.0
  16   35.0    91.0
  17   40.0   108.0
  18   45.0   126.0
  19   50.0   145.0

[20 rows x 2 columns]
```

Suppose you want to use the `resample()` method to get a daily frequency in each group of your dataframe and wish to complete the missing values with the `ffill()` method.

```
In [124]: df_re = pd.DataFrame({'date': pd.date_range(start='2016-01-01', periods=4,
   .....:                                         freq='W'),
   .....:                       'group': [1, 1, 2, 2],
   .....:                       'val': [5, 6, 7, 8]}).set_index('date')
   .....:

In [125]: df_re
Out[125]:
            group  val
date
2016-01-03      1    5
2016-01-10      1    6
2016-01-17      2    7
2016-01-24      2    8

In [126]: df_re.groupby('group').resample('1D').ffill()
Out[126]:
                  group  val
group date
1     2016-01-03      1    5
      2016-01-04      1    5
      2016-01-05      1    5
      2016-01-06      1    5
      2016-01-07      1    5
...                 ...  ...
2     2016-01-20      2    7
      2016-01-21      2    7
      2016-01-22      2    7
      2016-01-23      2    7
      2016-01-24      2    8

[16 rows x 2 columns]
```

2.13.6 Filtration

The `filter` method returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```
In [127]: sf = pd.Series([1, 1, 2, 3, 3, 3])

In [128]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[128]:
3    3
4    3
5    3
dtype: int64
```

The argument of `filter` must be a function that, applied to the group as a whole, returns `True` or `False`.

Another useful operation is filtering out elements that belong to groups with only a couple members.

```
In [129]: dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [130]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[130]:
   A  B
2  2  b
3  3  b
4  4  b
5  5  b
```

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [131]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[131]:
     A    B
0  NaN  NaN
1  NaN  NaN
2  2.0    b
3  3.0    b
4  4.0    b
5  5.0    b
6  NaN  NaN
7  NaN  NaN
```

For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion.

```
In [132]: dff['C'] = np.arange(8)

In [133]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
Out[133]:
   A  B  C
2  2  b  2
3  3  b  3
4  4  b  4
5  5  b  5
```

Note: Some functions when applied to a groupby object will act as a **filter** on the input, returning a reduced shape of the original (and potentially eliminating groups), but with the index unchanged. Passing `as_index=False` will not affect these transformation methods.

For example: `head, tail`.

```
In [134]: dff.groupby('B').head(2)
Out[134]:
   A  B  C
0  0  a  0
1  1  a  1
2  2  b  2
3  3  b  3
6  6  c  6
7  7  c  7
```

2.13.7 Dispatching to instance methods

When doing an aggregation or transformation, you might just want to call an instance method on each data group. This is pretty easy to do by passing lambda functions:

```
In [135]: grouped = df.groupby('A')

In [136]: grouped.agg(lambda x: x.std())
Out[136]:
            C         D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

But, it's rather verbose and can be untidy if you need to pass additional arguments. Using a bit of metaprogramming cleverness, GroupBy now has the ability to "dispatch" method calls to the groups:

```
In [137]: grouped.std()
Out[137]:
            C         D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

What is actually happening here is that a function wrapper is being generated. When invoked, it takes any passed arguments and invokes the function with any arguments on each group (in the above example, the `std` function). The results are then combined together much in the style of `agg` and `transform` (it actually uses `apply` to infer the gluing, documented next). This enables some operations to be carried out rather succinctly:

```
In [138]: tsdf = pd.DataFrame(np.random.randn(1000, 3),
   .....:                     index=pd.date_range('1/1/2000', periods=1000),
   .....:                     columns=['A', 'B', 'C'])
   .....:

In [139]: tsdf.iloc[::2] = np.nan

In [140]: grouped = tsdf.groupby(lambda x: x.year)

In [141]: grouped.fillna(method='pad')
Out[141]:
                   A         B         C
```

(continues on next page)

(continued from previous page)

```
2000-01-01       NaN       NaN       NaN
2000-01-02 -0.353501 -0.080957 -0.876864
2000-01-03 -0.353501 -0.080957 -0.876864
2000-01-04  0.050976  0.044273 -0.559849
2000-01-05  0.050976  0.044273 -0.559849
...              ...       ...       ...
2002-09-22  0.005011  0.053897 -1.026922
2002-09-23  0.005011  0.053897 -1.026922
2002-09-24 -0.456542 -1.849051  1.559856
2002-09-25 -0.456542 -1.849051  1.559856
2002-09-26  1.123162  0.354660  1.128135

[1000 rows x 3 columns]
```

In this example, we chopped the collection of time series into yearly chunks then independently called *fillna* on the groups.

The `nlargest` and `nsmallest` methods work on `Series` style groupbys:

```
In [142]: s = pd.Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])

In [143]: g = pd.Series(list('abababab'))

In [144]: gb = s.groupby(g)

In [145]: gb.nlargest(3)
Out[145]: 
a  4    19.0
   0     9.0
   2     7.0
b  1     8.0
   3     5.0
   7     3.3
dtype: float64

In [146]: gb.nsmallest(3)
Out[146]: 
a  6    4.2
   2    7.0
   0    9.0
b  5    1.0
   7    3.3
   3    5.0
dtype: float64
```

2.13.8 Flexible `apply`

Some operations on the grouped data might not fit into either the aggregate or transform categories. Or, you may simply want GroupBy to infer how to combine the results. For these, use the `apply` function, which can be substituted for both `aggregate` and `transform` in many standard use cases. However, `apply` can handle some exceptional use cases, for example:

```
In [147]: df
Out[147]: 
     A      B         C         D
```

(continues on next page)

(continued from previous page)

```
0  foo    one -0.575247  1.346061
1  bar    one  0.254161  1.511763
2  foo    two -1.143704  1.627081
3  bar  three  0.215897 -0.990582
4  foo    two  1.193555 -0.441652
5  bar    two -0.077118  1.211526
6  foo    one -0.408530  0.268520
7  foo  three -0.862495  0.024580

In [148]: grouped = df.groupby('A')

# could also just call .describe()
In [149]: grouped['C'].apply(lambda x: x.describe())
Out[149]:
A
bar  count    3.000000
     mean     0.130980
     std      0.181231
     min     -0.077118
     25%      0.069390
                ...
foo  min     -1.143704
     25%     -0.862495
     50%     -0.575247
     75%     -0.408530
     max      1.193555
Name: C, Length: 16, dtype: float64
```

The dimension of the returned result can also change:

```
In [150]: grouped = df.groupby('A')['C']

In [151]: def f(group):
   .....:     return pd.DataFrame({'original': group,
   .....:                          'demeaned': group - group.mean()})
   .....:

In [152]: grouped.apply(f)
Out[152]:
   original  demeaned
0 -0.575247 -0.215962
1  0.254161  0.123181
2 -1.143704 -0.784420
3  0.215897  0.084917
4  1.193555  1.552839
5 -0.077118 -0.208098
6 -0.408530 -0.049245
7 -0.862495 -0.503211
```

`apply` on a Series can operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame:

```
In [153]: def f(x):
   .....:     return pd.Series([x, x ** 2], index=['x', 'x^2'])
   .....:
```

(continues on next page)

(continued from previous page)

```
In [154]: s = pd.Series(np.random.rand(5))

In [155]: s
Out[155]:
0    0.321438
1    0.493496
2    0.139505
3    0.910103
4    0.194158
dtype: float64

In [156]: s.apply(f)
Out[156]:
          x       x^2
0  0.321438  0.103323
1  0.493496  0.243538
2  0.139505  0.019462
3  0.910103  0.828287
4  0.194158  0.037697
```

Note: `apply` can act as a reducer, transformer, *or* filter function, depending on exactly what is passed to it. So depending on the path taken, and exactly what you are grouping. Thus the grouped columns(s) may be included in the output as well as set the indices.

2.13.9 Other useful features

Automatic exclusion of "nuisance" columns

Again consider the example DataFrame we've been looking at:

```
In [157]: df
Out[157]:
     A      B         C         D
0  foo    one -0.575247  1.346061
1  bar    one  0.254161  1.511763
2  foo    two -1.143704  1.627081
3  bar  three  0.215897 -0.990582
4  foo    two  1.193555 -0.441652
5  bar    two -0.077118  1.211526
6  foo    one -0.408530  0.268520
7  foo  three -0.862495  0.024580
```

Suppose we wish to compute the standard deviation grouped by the `A` column. There is a slight problem, namely that we don't care about the data in column `B`. We refer to this as a "nuisance" column. If the passed aggregation function can't be applied to some columns, the troublesome columns will be (silently) dropped. Thus, this does not pose any problems:

```
In [158]: df.groupby('A').std()
Out[158]:
            C         D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

Note that `df.groupby('A').colname.std().` is more efficient than `df.groupby('A').std().colname`, so if the result of an aggregation function is only interesting over one column (here `colname`), it may be filtered *before* applying the aggregation function.

Note: Any object column, also if it contains numerical values such as `Decimal` objects, is considered as a "nuisance" columns. They are excluded from aggregate functions automatically in groupby.

If you do wish to include decimal or object columns in an aggregation with other non-nuisance data types, you must do so explicitly.

```
In [159]: from decimal import Decimal

In [160]: df_dec = pd.DataFrame(
   .....:     {'id': [1, 2, 1, 2],
   .....:      'int_column': [1, 2, 3, 4],
   .....:      'dec_column': [Decimal('0.50'), Decimal('0.15'),
   .....:                     Decimal('0.25'), Decimal('0.40')]
   .....:      }
   .....: )
   .....:

# Decimal columns can be sum'd explicitly by themselves...
In [161]: df_dec.groupby(['id'])[['dec_column']].sum()
Out[161]:
   dec_column
id
1        0.75
2        0.55

# ...but cannot be combined with standard data types or they will be excluded
In [162]: df_dec.groupby(['id'])[['int_column', 'dec_column']].sum()
Out[162]:
    int_column
id
1            4
2            6

# Use .agg function to aggregate over standard and "nuisance" data types
# at the same time
In [163]: df_dec.groupby(['id']).agg({'int_column': 'sum', 'dec_column': 'sum'})
Out[163]:
    int_column dec_column
id
1            4       0.75
2            6       0.55
```

Handling of (un)observed Categorical values

When using a `Categorical` grouper (as a single grouper, or as part of multiple groupers), the `observed` keyword controls whether to return a cartesian product of all possible groupers values (`observed=False`) or only those that are observed groupers (`observed=True`).

Show all values:

```
In [164]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                              categories=['a', 'b']),
   .....:                               observed=False).count()
   .....:
Out[164]:
a    3
b    0
dtype: int64
```

Show only the observed values:

```
In [165]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                              categories=['a', 'b']),
   .....:                               observed=True).count()
   .....:
Out[165]:
a    3
dtype: int64
```

The returned dtype of the grouped will *always* include *all* of the categories that were grouped.

```
In [166]: s = pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                                  categories=['a', 'b']),
   .....:                                   observed=False).count()
   .....:

In [167]: s.index.dtype
Out[167]: CategoricalDtype(categories=['a', 'b'], ordered=False)
```

NA and NaT group handling

If there are any NaN or NaT values in the grouping key, these will be automatically excluded. In other words, there will never be an "NA group" or "NaT group". This was not the case in older versions of pandas, but users were generally discarding the NA group anyway (and supporting it was an implementation headache).

Grouping with ordered factors

Categorical variables represented as instance of pandas's `Categorical` class can be used as group keys. If so, the order of the levels will be preserved:

```
In [168]: data = pd.Series(np.random.randn(100))

In [169]: factor = pd.qcut(data, [0, .25, .5, .75, 1.])

In [170]: data.groupby(factor).mean()
Out[170]:
(-2.645, -0.523]   -1.362896
```

(continues on next page)