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-core, the split-apply-combine paradigm is also a common way to process large datasets. In these situations we may want to split the data set into groups and do something with those groups. In the apply step, we might want to do one of the following:

- **Aggregation**: computing a summary statistic (or statistics) for each group. Some examples:
  - Compute group sums or means
  - Compute group sums or 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.

Many of these operations are defined in GroupBy objects. These operations are similar to the aggregating API, window API, and resample API.

It is possible that a given operation does not fall into one of these categories or is some combination of two. In such cases, it may be possible to compute the operation using GroupBy `apply`. Without this method will examine the results of the apply step and try to "glue" together the appropriate output. Some operations in the examples fall into this category.

Similar to those of the aggregating

Note

An operation that is split into multiple steps using built-in GroupBy operations will be more efficient than using the `apply` method with a user-defined Python function.

use cases

Since the set of object instance methods on pandas data structures is generally rich and expressive, we often simply want to ask, *I have a DataFrame/Series and want to do X*. Every Series GroupBy method is quite familiar to those who have used a SQL dialect or `DataFrame`, in which you can write code like:

```
select sum(foo), count(*), mean(bar), blah(bar) from df
group by baz
WHERE GP(foo, bar, baz)
```

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 groupby cookbook for some more advanced use-cases.

Similar to those of the aggregating

any of their sows? nothough groupby on sales? was going to the depressed

Splitting an object into groups

pandas objects can be split on any of their axes. The syntax for using GroupBy to produce a grouping of data to apply to group ranges. To create a GroupBy object (more on what the GroupBy object is later), you do the following:

```
In [501]: grouped = df.groupby(
          ...: ...:     lambda x: x[0],
          ...: ...:     axis=1,
          ...: ...:     level=None,
          ...: ...:     by=None,
          ...: ...:     sort=True,
          ...: ...:     group_keys=True,
          ...: ...:     squeeze=False,
          ...: ...:     observed=False,
          ...: ...:     dropna=True
          ...: ...: )

In [502]: grouped
Out[502]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f1ea831417650>

In [503]: grouped.groups()*100
Out[503]: {'a': [0, 2, 4], 'b': [1, 3, 5], 'c': [6]}
```

The mapping can be specified many different ways:

- A 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 by which to split along rows or columns.
- `pandas.Grouper` A named variable-type for `pandas.DataFrame.groupby`.
- 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

It is Important to `groupby` may wants 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 [504]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
          ...: ...:                        'foo', 'bar', 'foo', 'foo'],
          ...: ...:                        'B': ['one', 'one', 'two', 'three',
          ...: ...:                        'two', 'two', one', 'three'],
          ...: ...:                        'C': randn(8),
          ...: ...:                        'D': randn(8)})

In [505]: df
Out[505]:
     A    B         C         D
0  foo  one  0.895717  0.805244
1  bar  one  0.805244  -1.206412
2  foo  two  -1.206412  2.565646
3  bar  three  2.565646  1.431745
4  foo  two  1.431745  1.340923
5  bar  two  1.340923  -0.929808
6  foo  one  -0.929808  -2.691627
7  foo  three  -2.691627  -443728

In [506]: grouped = df.groupby('A')*100
In [507]: grouped = df.groupby('B', axis=1)*100
In [508]: grouped = df.groupby(['A', 'B'])*100
```

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 [509]: grouped = df.groupby('A')*100
In [510]: grouped = df.groupby(['A', 'B'])*100
```

If we also have a MultiIndex on columns `C` and `D`, we can group by all but the specified columns

```
In [511]: df2 = df.set_index('C', 'D')*100
In [512]: grouped = df2.groupby(level=df2.index.names[1])*100
In [513]: grouped = df.groupby('A')*100
In [514]: grouped = df.groupby('A')*100
In [515]: grouped = df.groupby('A')*100
```

These gift with the DataFrame are on index (rows). We could also split by the columns:

```
In [516]: def get_letter_type(letter):
          ...: ...:     if letter.lower() in 'aeiou':
          ...: ...:         return 'vowel'
          ...: ...:     else:
          ...: ...:         return 'consonant'
          ...: ...: grouped = df.groupby(get_letter_type, axis=1)*100
```

pandas `Series` 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 are considered to be in one group and thus the output of aggregation functions only sees a single non-duplicate value:

```
In [517]: lst = [1, 2, 3, 4, 5, 6]
In [518]: ser.Series(lst, 3, 3, 4, 5, 6)
In [519]: grouped = s.groupby(level=0)
In [520]: grouped.first()
Out[520]:
3  1
4  2
5  3
dtype: int64
In [521]: grouped.last()
Out[521]:
3  3
4  4
5  5
dtype: int64
In [522]: grouped.nth(2)
Out[522]:
3  3
5  5
dtype: int64
```

Note that the `nothing groups` with this 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 not 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 [523]: df = pd.DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})
In [524]: df.groupby('X')*100
Out[524]:
     Y
X
A  300
B  300
In [525]: df.groupby('X', sort=False)*100
Out[525]:
     Y
X
B  300
A  700
```

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 [526]: df = pd.DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})
In [527]: df.groupby('X', sort=False)*100
Out[527]:
     Y
X
B  300
A  700
In [528]: df.groupby('X')*100
Out[528]:
     Y
X
A  300
B  300
```

New in version 0.18.

Groupby dropna

By default NA values are excluded from group keys during the `groupby` operation. However, in case you want to include NA values in group keys, you could pass `dropna=False` to achieve it.

```
In [529]: df = DataFrame({'A': [1, 2, NA, NA], 'B': [3, 4, 5, 6]})
In [530]: df.groupby(groupby=False, sort=False)*100
Out[530]:
     B
A
1  300
2  700
In [531]: df.groupby('A')*100
Out[531]:
     B
A
1  300
2  700
In [532]: df.groupby('A', dropna=False)*100
Out[532]:
     B
A
1  300
2  700
NaN  1100
```

The default setting of `dropna` argument is `True` which means `NA` are not included in group keys.

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:

corresponding values are

```
In [533]: df.groupby('A')*100
Out[533]: {'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}
In [534]: df.groupby('A', sort=False)*100
Out[534]: {'foo': [0, 2, 4, 6, 7], 'bar': [1, 3, 5]}
```

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 [535]: grouped = df.groupby('A', 'B')
In [536]: grouped.groups()*100
Out[536]: {('bar', 'one'): [1], ('bar', 'three'): [3], ('bar', 'two'): [5], ('foo', 'one'): [0, 6], ('foo'
In [537]: len(grouped)
Out[537]: 4
```

`groups` will not compute column names (and other attributes).

```
In [538]: df
Out[538]:
     A    B         C         D
0  foo  one  0.895717  0.805244
1  bar  one  0.805244  -1.206412
2  foo  two  -1.206412  2.565646
3  bar  three  2.565646  1.431745
4  foo  two  1.431745  1.340923
5  bar  two  1.340923  -0.929808
6  foo  one  -0.929808  -2.691627
7  foo  three  -2.691627  -443728

In [539]: df.groupby('A')*100
Out[539]:
     B         C         D
0  300  0.895717  0.805244
1  500  0.805244  -1.206412
2  700  -1.206412  2.565646
3  900  2.565646  1.431745
4  700  1.431745  1.340923
5  500  1.340923  -0.929808
6  300  -0.929808  -2.691627
7  700  -2.691627  -443728
```

Groupby with MultiIndex

With hierarchically ordered data, it is quite common to group by one of the levels of the hierarchy.

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

```
In [540]: arrays = [['bar', 'bar', 'baz', 'baz',
          ...: ...:         'foo', 'foo', 'qux', 'qux'],
          ...: ...:         ['one', 'two', 'one', 'two',
          ...: ...:         'one', 'two', 'one', 'two']]
In [541]: index = zip(tuples.from_arrays(arrays, names=['first', 'second']))
In [542]: s = pd.Series(np.random.randn(8), index=index)
In [543]: s*100
Out[543]:
first  second
bar    one    -0.919854
       two    -0.042379
baz    one    1.381715
       two    0.274992
foo    one    1.799775
       two    -0.608772
qux    one    0.544074
       two    0.812772
dtype: float64
In [544]: grouped = s.groupby(level=0)*100
In [545]: grouped.first()*100
Out[545]:
first
bar  -0.962854
baz  0.828354
foo  0.595501
qux  0.678423
dtype: float64
In [546]: grouped.last()*100
Out[546]:
first
bar  -0.042379
baz  0.274992
foo  -0.608772
qux  0.812772
dtype: float64
In [547]: grouped.sum()*100
Out[547]:
first
bar  -1.005233
baz  1.656707
foo  -0.013271
qux  1.356846
dtype: float64
```

Index levels may also be specified by name.

```
In [548]: s.groupby(level='second')*100
Out[548]:
second
one  0.980950
two  1.991337
dtype: float64
```

Index levels may also be specified by name.

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

```
In [549]: df.groupby(['second', 'A'])*100
Out[549]:
           B         C         D
second A
one    foo  300  1.239447  -0.044329
       bar  400  0.744926  0.914154
two    foo  600  0.907105  -0.467716
       bar  500  -0.784318  -1.349742
```

Then group group by

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 [550]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
          ...: ...:                        'foo', 'bar', 'foo', 'foo'],
          ...: ...:                        'B': ['one', 'one', 'two', 'three',
          ...: ...:                        'two', 'two', one', 'three'],
          ...: ...:                        'C': randn(8),
          ...: ...:                        'D': randn(8)})
In [551]: df
Out[551]:
     A    B         C         D
0  foo  one  0.895717  0.805244
1  bar  one  0.805244  -1.206412
2  foo  two  -1.206412  2.565646
3  bar  three  2.565646  1.431745
4  foo  two  1.431745  1.340923
5  bar  two  1.340923  -0.929808
6  foo  one  -0.929808  -2.691627
7  foo  three  -2.691627  -443728
In [552]: grouped = df.groupby('A')*100
In [553]: grouped['C']*100
Out[553]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7f1ea831417650>
```

Thus, by using `[]` on the Grouby object in a similar way as the one used to get a column from a DataFrame

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

Iterating through groups

With the exception of `pandas` and `numpy` groupby, iterating through groups is not a common or recommended operation, but it is quite possible to do so:

```
In [554]: grouped = df.groupby('A')*100
In [555]: for name, group in grouped:
          ...: ...:     print(name)
          ...: ...:     print(group)
          ...: ...:
bar
     A    B         C         D
1  bar  one  0.805244  -1.206412
3  bar  three  2.565646  1.431745
5  bar  two  1.340923  -0.929808
foo
     A    B         C         D
0  foo  one  0.895717  0.805244
2  foo  two  -1.206412  2.565646
4  foo  two  1.431745  1.340923
6  foo  one  -0.929808  -2.691627
7  foo  three  -2.691627  -443728
```

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

```
In [556]: for name, group in df.groupby('A', 'B'):
          ...: ...:     print(name)
          ...: ...:     print(group)
          ...: ...:
('bar', 'one')
     A    B         C         D
1  bar  one  0.805244  -1.206412
('bar', 'three')
     A    B         C         D
3  bar  three  2.565646  1.431745
('bar', 'two')
     A    B         C         D
5  bar  two  1.340923  -0.929808
('foo', 'one')
     A    B         C         D
0  foo  one  0.895717  0.805244
6  foo  one  -0.929808  -2.691627
('foo', 'three')
     A    B         C         D
7  foo  three  -2.691627  -443728
('foo', 'two')
     A    B         C         D
2  foo  two  -1.206412  2.565646
4  foo  two  1.431745  1.340923
```

See Iterating-through-groups.

Selecting a group

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

```
In [557]: grouped.get_group('bar')*100
Out[557]:
     A    B         C         D
1  bar  one  0.805244  -1.206412
3  bar  three  2.565646  1.431745
5  bar  two  1.340923  -0.929808
```

Or for an object grouped on multiple columns:

```
In [558]: df.groupby('A', 'B').get_group('foo', 'one')*100
Out[558]:
     A    B         C         D
0  foo  one  0.895717  0.805244
6  foo  one  -0.929808  -2.691627
```

Aggregation

Aggregation is a GroupBy operation that reduces the dimension of the grouping object. The result of an aggregation is, or at minimum is, a scalar value for each column in a group. For example, producing a sum of each column in one of cells.

```
In [559]: aggregate.__doc__.format(name)
Out[559]: 'Aggregate {name} using callable.\n\n    Parameters\n    ----------\
In [560]: grouped = df.groupby('A')*100
In [561]: grouped.aggregate(np.sum)*100
Out[561]:
     B         C         D
A
bar  900  3.102312  0.283504
foo  1500  -2.511267  -2.442149
In [562]: df.groupby('A', 'B').aggregate(np.sum)*100
Out[562]:
           C         D
A   B
bar one  0.805244  -1.206412
    three  2.565646  1.431745
    two  1.340923  -0.929808
foo one  -0.034112  -1.886904
    two  0.224973  1.873569
    three  -2.691627  -443728
```

Built-in aggregation methods

Many common aggregations are built-in to GroupBy aliases as methods. Of the methods listed below, those with a `*` do not have Cython-optimized implementation.

Method	Description
`any()`	Compute whether any of the values in the groups are truthy
`all()`	Compute whether all of the values in the groups are truthy
`count()`	Compute the number of non-NA values in the groups
`cov()` *	Compute the covariance of the groups
`first()`	Compute the first occurring value in each group
`idxmax()` *	Compute the index of the maximum value in each group
`idxmin()` *	Compute the index of the minimum value in each group
`last()`	Compute the last occurring value in each group
`mean()`	Compute the mean of each group
`median()`	Compute the median of each group
`min()`	Compute the minimum value in each group
`nunique()`	Compute the number of unique values in each group
`prod()` *	Compute the product of the values in each group
`quantile(q)`	Compute a given quantile of the values in each group
`sem()`	Compute the standard error of the mean of the values in each group
`std()`	Compute the number of values in each group
`sum()`	Compute the sum of the values in each group
`var()`	Compute the variance of the values in each group

Some examples:

```
In [563]: df.groupby('A')*100
Out[563]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f1ea831417650>
In [564]: grouped.count()*100
Out[564]:
     B  C  D
A
bar  3  3  3
foo  5  5  5
In [565]: grouped.sum()*100
Out[565]:
     B         C         D
A
bar  900  3.102312  0.283504
foo  1500  -2.511267  -2.442149
In [566]: grouped.prod()*100
Out[566]:
     B         C         D
A
bar  900  -0.966977  -0.399835
foo  1500  0.332299  7.931069
In [567]: grouped.min()*100
Out[567]:
     B         C         D
A
bar  400  0.744926  -1.349742
foo  600  -1.206412  -443728
In [568]: grouped.max()*100
Out[568]:
     B         C         D
A
bar  500  2.565646  1.431745
foo  700  1.431745  2.565646
In [569]: grouped.first()*100
Out[569]:
     B         C         D
A
bar  400  0.744926  -1.349742
foo  600  -1.206412  -443728
In [570]: grouped.last()*100
Out[570]:
     B         C         D
A
bar  500  1.340923  -0.929808
foo  700  -2.691627  1.340923
In [571]: grouped.mean()*100
Out[571]:
     B         C         D
A
bar  300  1.034104  0.094501
foo  300  -0.502253  -0.488430
In [572]: grouped.median()*100
Out[572]:
     B         C         D
A
bar  300  0.805244  0.283504
foo  300  -0.929808  -443728
In [573]: grouped.std()*100
Out[573]:
     B         C         D
A
bar  100  1.760941  1.360388
foo  100  1.167228  1.873569
In [574]: grouped.var()*100
Out[574]:
     B         C         D
A
bar  100  3.101038  1.850620
foo  100  1.362457  3.510792
In [575]: grouped.sem()*100
Out[575]:
     B         C         D
A
bar  100  0.816497  0.785371
foo  100  0.521987  1.252781
In [576]: grouped.count()*100
Out[576]:
     B  C  D
A
bar  3  3  3
foo  5  5  5
In [577]: grouped.cov()*100
Out[577]:
           C         D
A   B
bar one  NaN  NaN
    three  NaN  NaN
    two  NaN  NaN
foo one  0.000000  0.690275
    two  0.433965  0.099909
    three  NaN  NaN
In [578]: grouped.cov()*100
Out[578]:
           C         D
A   B
bar one  NaN  NaN
    three  NaN  NaN
    two  NaN  NaN
foo one  0.000000  0.690275
    two  0.433965  0.099909
    three  NaN  NaN
In [579]: grouped.idxmax()*100
Out[579]:
     C  D
A
bar  1  1
foo  4  4
In [580]: grouped.idxmin()*100
Out[580]:
     C  D
A
bar  3  3
foo  7  7
```

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the `size` method. It returns a dtype int-64 bit pandas Series and whose values are the sizes of each group.

```
In [581]: grouped = df.groupby('A')*100
In [582]: grouped.size()*100
Out[582]:
A
bar  3
foo  5
dtype: int64
```

While the `describe()` method is not itself a reduction, it can be used to conveniently produce a collection of summary statistics about each group.

```
In [583]: grouped.describe()*100
Out[583]:
     B count  B  mean  B  std  B  min  B  25%  B  50%  B  75%  B  max  C count  C  mean  C  std  C  min  C  25%  C  50%  C  75%  C  max  D count  D  mean  D  std  D  min  D  25%  D  50%  D  75%  D  max
A
bar  3.0  300.0  100.0  300.0  300.0  300.0  300.0  300.0  3.0  1.034104  1.760941  0.744926  0.775125  0.805244  1.685445  2.565646  3.0  0.094501  1.360388  -1.349742  -1.139775  -0.929808  0.255968  1.431745
foo  5.0  300.0  100.0  300.0  300.0  300.0  300.0  300.0  5.0  -0.502253  1.167228  -1.206412  -1.206412  -0.929808  0.240577  1.431745  5.0  -0.488430  1.873569  -2.691627  -1.691627  -443728  0.670461  2.565646
```

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the `size` method. It returns a dtype int-64 bit pandas Series and whose values are the sizes of each group.

```
In [584]: grouped = df.groupby('A')*100
In [585]: grouped.size()*100
Out[585]:
A
bar  3
foo  5
dtype: int64
```

```
In [20]: grouped.describe()
Out[20]:
                 data1       data2       data3       data4       data5
key1 key2
a    one   0.314815  0.837648  0.392928  0.854659  0.976901
     two   0.333333  0.772826  0.745655  0.075762  0.641908
b    one   0.083333  0.529229  0.799159  0.304603  0.665000
     two   0.580000  0.441274  0.305384  0.432265  0.878396
[8 rows x 15 columns]
```

Another aggregation example is to compute the number of unique values of each group. This is similar to the `value_counts` function, except that it only counts the number of unique values.

```
In [21]: df3.groupby('key1').nunique()
Out[21]:
      key2  key3  key4  key5  key6  key7  key8  key9  key10
key1
a         2     3     3     3     3     3     3     3      3
b         2     2     2     2     2     2     2     2      2
[2 rows x 10 columns]

In [22]: df3.groupby('key1').count()
Out[22]:
      key2  key3  key4  key5  key6  key7  key8  key9  key10
key1
a         3     3     3     3     3     3     3     3      3
b         2     2     2     2     2     2     2     2      2
[2 rows x 10 columns]
```

> **Note**
>
> Aggregation functions will not return groups that you are aggregating over if they are named categorical columns. One reason for this is that any group key not seen in the `values` of the returned column.
>
> Passing `dropna=False` will return the groups that you are aggregating over, if they are named `Series` or `DataFrame`.
>
> ```
> In [23]: df3.groupby('key1', dropna=False).count()
> Out[23]:
> ```
>
> ...

whether they are named indices or columns

The `aggregate()` method

> **Note**
>
> The `aggregate()` method can accept many different types of inputs. This section details using string aliases for various GroupBy methods; other inputs are detailed in the sections below.
>
> Any reduction method that pandas implements can be passed as a string to `aggregate()`. Users are encouraged to use the shorter aliases in place of the corresponding method was called.
>
> ```
> In [24]: grouped = df.groupby('A')
>
> In [25]: grouped[['C', 'D']].aggregate('sum')
>
> Out[25]:
>
> for a GROUP_FORMAT
> GROUP_SPAN_CHAR
> GROUP_SEP
> NAT_FORMAT
> NAN_FORMAT
> NEWLINE
> EscapeChar
> ESCAPE_CHAR
> SPECIAL_CHARS
> MAX_ROWS_FOR_TRUNCATION
> WEIGHT_COMMENT_CHAR
> ENTRY_DELIMITER
> TRUNCATE_ROWS
> PRECISION
> SMALL_REAL_NUMBER
> FLOAT_FORMAT
> STANDARD_MAX_COLWIDTH
> TIMEDELTA_FORMAT
> TIMEDELTA_NAT
> TIMEDELTA_NAN
> INDEX_NAME
> KeyValueSeparator
> ```
>
> ...

Aggregation with User-Defined Functions

Users can also provide their own User-Defined Functions (UDFs) for custom aggregations.

> **Warning**
>
> When aggregating with a UDF, the UDF should not mutate the provided `Series`, see gotchas section of the User Guide for more detail.

See gotchas

> **Note**
>
> Aggregating with a UDF is often less performant than using the pandas built-in methods on GroupBy. Consider breaking up a complex operation into a chain of operations that utilize the built-in methods.
>
> ```
> In [26]: df.dtypes
>
> Out[26]:
> ```
>
> ...

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 [27]: grouped = df.groupby('A')
In [28]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[28]:
     sum      mean       std
foo  NaN       NaN       NaN
bar  NaN       NaN       NaN
[2 rows x 3 columns]
```

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 [29]: grouped[['C', 'D']].agg([np.sum, np.mean, np.std])
Out[29]:
     C                   D
   sum      mean       std       sum      mean       std
foo  NaN       NaN       NaN  0.339133  0.113044  0.335096
bar  NaN       NaN       NaN  1.376682  0.688341  0.963671
[2 rows x 6 columns]
```

The resulting aggregation will have the function names as the second level of the column hierarchy. If a function is applied on a column, it will inherit the column name.

```
In [30]: grouped[['C', 'D']].agg([np.sum, np.std])
Out[30]:
     C                   D
   sum       std       sum       std
foo  NaN       NaN  0.339133  0.335096
bar  NaN       NaN  1.376682  0.963671
[2 rows x 4 columns]
```

The resulting dtype will reflect that of the aggregating function. If the results from different groups have different dtypes, then a common dtype will be determined in the same way as `DataFrame` construction.

```
In [31]: df3.groupby('key1').agg({'key2': np.ndim, 'key3': np.sum})
Out[31]:
     key2  key3
key1
a       2    12
b       2     5
[2 rows x 2 columns]
```

Keys

Values

Keys can also could be `Series`-like. `DataFrame` functions in `agg` will broadcast over the `Series` while iterating.

```
In [32]: df.groupby('A').agg({'B': np.sum, 'C': lambda s: s[mid]})
Out[32]:
     B         C
foo  NaN       NaN
bar  NaN       NaN
[2 rows x 2 columns]
```

Values can also could be `Series`-like. Series functions in `agg` will broadcast over the groups.

Named aggregation

To support column-specific aggregation with control over the output column names, pandas accepts the special syntax in `GroupBy.agg()`, `SeriesGroupBy.agg()`, and `DataFrameGroupBy.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 `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 [33]: df.dtypes
Out[33]:
A             object
B             float64
C             float64
D             float64
dtype: object

In [34]: grouped = df.groupby('A')
In [35]: grouped[['C', 'D']].agg([np.sum, np.std])
Out[35]:
     C                   D
   sum       std       sum       std
foo  NaN       NaN  0.339133  0.335096
bar  NaN       NaN  1.376682  0.963671
[2 rows x 4 columns]

In [36]: df.groupby('A').agg(b_sum=('B', np.sum),
         c_std=('C', np.std))
Out[36]:
     b_sum     c_std
foo    NaN       NaN
bar    NaN       NaN
[2 rows x 2 columns]
```

If a dictionary maps the same column to multiple functions, then pandas will attempt to apply the same function (or two functions with the same name) to the same column.

```
In [37]: df.groupby('A').agg({'B': sum, 'C': min})
Out[37]:
     B         C
foo  NaN       NaN
bar  NaN       NaN
[2 rows x 2 columns]
```

> **Caution**
>
> In general, the output column names should be unique, but pandas will allow you apply to the same function (or two functions with the same name) to the same column.
>
> ```
> In [38]: df.groupby('A').agg({'B': sum, 'C': min})
> Out[38]:
> ```
>
> ...
>
> pandas also allows 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 [39]: df.groupby('A').agg({'B': [lambda x: x.max(), lambda x: x.min()]})
> Out[39]:
>
> for a GROUP_FORMAT
> GROUP_SPAN_CHAR
> GROUP_SEP
> NAT_FORMAT
> NAN_FORMAT
> NEWLINE
> EscapeChar
> ESCAPE_CHAR
> SPECIAL_CHARS
> MAX_ROWS_FOR_TRUNCATION
> WEIGHT_COMMENT_CHAR
> ENTRY_DELIMITER
> TRUNCATE_ROWS
> PRECISION
> SMALL_REAL_NUMBER
> FLOAT_FORMAT
> STANDARD_MAX_COLWIDTH
> TIMEDELTA_FORMAT
> TIMEDELTA_NAT
> TIMEDELTA_NAN
> INDEX_NAME
> KeyValueSeparator
> ```
>
> ...

If the column names that you want are not Python keywords, you can

apply them by expanding a dictionary with keyword keys and tuple values. For the aggregation functions, the keywords are the output column names, and the values are tuples whose first elements are the column to select and the second elements are the aggregation to apply. Keywords are not restricted to be valid Python variable names.

```
In [40]: df.groupby('A').agg(b_min=('B', 'min'),
         c_sum=('C', lambda x: x.sum()))
Out[40]:
     b_min     c_sum
foo    NaN       NaN
bar    NaN       NaN
[2 rows x 2 columns]
```

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 [41]: df.groupby('A').agg(b_min=('B', 'min'),
         c_sum=('C', lambda x: x.sum()))
Out[41]:
     b_min     c_sum
foo    NaN       NaN
bar    NaN       NaN
[2 rows x 2 columns]
```

Applying different functions to DataFrame columns

By passing a dict to `aggregate`, we can apply a different aggregation to the columns of a DataFrame:

```
In [42]: grouped.agg({'C': np.sum, 'D': lambda x: np.std(x, ddof=1)})
Out[42]:
     C         D
foo  NaN       NaN
bar  NaN       NaN
[2 rows x 2 columns]
```

either ... or

also?

The function passed on the dictionary can be any valid input to either one function or two functions.

```
In [43]: grouped.agg({'C': [np.sum, np.mean, np.std],
         'D': [np.std, lambda x: np.std(x, ddof=1)]})
Out[43]:
     C                   D
   sum      mean       std       std  <lambda>
foo  NaN       NaN       NaN  0.339133  0.335096
bar  NaN       NaN       NaN  1.376682  0.963671
[2 rows x 6 columns]
```

Transformation

A transformation is a GroupBy operation whose result is indexed the same as the one being grouped. Common examples include `cumsum()` and `diff()`.

```
In [44]: speeds
Out[44]:
     class  order  max  min
Park  Park   NaNN   NaN  NaN
Park  Park   Park2  Park2  Park2
Park  Park   Park3  Park3  Park3
Park  Park   Park4  Park4  Park4
[4 rows x 4 columns]

In [45]: grouped = speeds.groupby('class')['max', 'min']
In [46]: speed = lambda x:
         (x['max'] - x['min'])
Out[46]:
Park    NaN
Park    NaN
Park    NaN
Park    NaN
dtype: float64

In [47]: speed_ratio = lambda x:
         x['max'] / x['min']
Out[47]:
Park    NaN
Park    NaN
Park    NaN
Park    NaN
dtype: float64

In [48]: grouped.transform('sum')
Out[48]:
     max  min
Park  NaN  NaN
Park  NaN  NaN
Park  NaN  NaN
Park  NaN  NaN
[4 rows x 2 columns]
```

Similar to the aggregation, the transformation method can accept many aliases to the built-in transformation functions in the previous section. In addition to the aforementioned aliases, there also exists one additional alias, `ngroupby`. Finally, the transformation method, also from the aggregation method is provided, the result will be broadcast across the group.

```
In [49]: speeds
Out[49]:
     class  order  max  min
Park  Park   NaNN   NaN  NaN
Park  Park   Park2  Park2  Park2
Park  Park   Park3  Park3  Park3
Park  Park   Park4  Park4  Park4
[4 rows x 4 columns]

In [50]: grouped = speeds.groupby('class')['max', 'min']
In [51]: speed = lambda x:
         (x['max'] - x['min'])
Out[51]:
Park    NaN
Park    NaN
Park    NaN
Park    NaN
dtype: float64

In [52]: grouped.transform('sum')
Out[52]:
     max  min
Park  NaN  NaN
Park  NaN  NaN
Park  NaN  NaN
Park  NaN  NaN
[4 rows x 2 columns]
```

In addition, groupby has a `transform` method. It takes a function to apply to each group and expects that the result across the group, producing a transformed result. If the aggregation method is a Cython optimized, this will be performed as well.

The `transform()` method

Similar to the `apply()` method, the `transform()` method can accept among aliases to the built-in transformation methods in the previous section. In addition to accepting aliases to the built-in aggregation methods, When an aggregation method is provided, the result will be broadcast across the group.

```
In [53]: speeds
Out[53]:
     class  order  max  min
Park  Park   NaNN   NaN  NaN
Park  Park   Park2  Park2  Park2
Park  Park   Park3  Park3  Park3
Park  Park   Park4  Park4  Park4
[4 rows x 4 columns]

In [54]: grouped = speeds.groupby('class')['max', 'min']
In [55]: speed = lambda x:
         (x['max'] - x['min'])
Out[55]:
Park    NaN
Park    NaN
Park    NaN
Park    NaN
dtype: float64

In [56]: grouped.transform('sum')
Out[56]:
     max  min
Park  NaN  NaN
Park  NaN  NaN
Park  NaN  NaN
Park  NaN  NaN
[4 rows x 2 columns]
```

Similarly to the `apply()` method, the `transform()` method can accept among aliases to the built-in transformation methods in the previous section. In addition to accepting aliases to the built-in aggregation methods, When an aggregation method is provided, the result will be broadcast across the group.

- Functions that mutate the passed object can be used (so long as pandas does not attempt to preserve the original object in other places).
- Transformed group values must be the same shape as the original group.
- Group transforms should return a result that is the same size as the group chunk.
- Operations should not mutate groups in any way.
- Groups should not be modified by the UDF (for similar reasons as to the above).
- Conditionally operates on all columns of the entire group chunk at once. If this is expected, a fast path is used starting from the second chunk.

> **Changed in version 2.0.0:** When using `.transform()` on a grouped DataFrame and the transformation function returns a DataFrame, pandas now aligns the result's index with the input's index. You can call `.to_numpy()` within the transformation function to avoid alignment.
>
> Similar to the `Series` example, the resulting dtype will reflect that of the transformation function. If the results from different groups have different dtypes, then a common dtype will be determined in the same way as `DataFrame` construction.

Suppose we wished to standardize the data within each group:

```
In [57]: index = pd.date_range('10/1999', periods=1100)
In [58]: ts = ts.rolling(window=365, 0, 3, CORR, True)
In [59]: ts = np.sin(np.random.randn(1100,0.5*np.pi*np.sin(np.pi*x/2)))
Out[59]:
2000-01-01  0.000000
2000-01-02  0.479425
2000-01-03  0.841471
2000-01-04  0.997495
2000-01-05  0.909297
2000-01-06  0.598472
2000-01-07  0.141120
2000-01-08  -0.350783
2000-01-09  -0.756802
2000-01-10  -0.977530
2000-01-11  -0.958924
2000-01-12  -0.705540
2000-01-13  -0.279415
2000-01-14  0.156434
2000-01-15  0.563855
2000-01-16  0.873958
2000-01-17  0.993712
2000-01-18  0.915773
2000-01-19  0.644218
2000-01-20  0.216185
2000-01-21  -0.291995
2000-01-22  -0.681998
2000-01-23  -0.942222
2000-01-24  -0.989992
2000-01-25  -0.820148
2000-01-26  -0.409708
2000-01-27  0.087455
2000-01-28  0.583462
2000-01-29  0.894857
2000-01-30  0.996166
2000-01-31  0.878124
2000-02-01  0.455775
2000-02-02  0.035412
2000-02-03  -0.382026
2000-02-04  -0.768254
2000-02-05  -0.953923
2000-02-06  -0.970031
2000-02-07  -0.832980
2000-02-08  -0.498337
2000-02-09  -0.052571
2000-02-10  0.383436
2000-02-11  0.746099
2000-02-12  0.951602
2000-02-13  0.968827
2000-02-14  0.837097
2000-02-15  0.463418
2000-02-16  0.019203
2000-02-17  -0.416980
2000-02-18  -0.804321
2000-02-19  -0.974927
2000-02-20  -0.951125
2000-02-21  -0.732020
2000-02-22  -0.291834
2000-02-23  0.216395
2000-02-24  0.663318
2000-02-25  0.938705
2000-02-26  0.987166
2000-02-27  0.789353
2000-02-28  0.319507
2000-02-29  -0.189201
2000-03-01  -0.633824
2000-03-02  -0.909297
2000-03-03  -0.997495
2000-03-04  -0.841471
2000-03-05  -0.479425
2000-03-06  0.000000
2000-03-07  0.479425
2000-03-08  0.841471
2000-03-09  0.997495
2000-03-10  0.909297
2000-03-11  0.598472
2000-03-12  0.141120
2000-03-13  -0.350783
2000-03-14  -0.756802
2000-03-15  -0.977530
2000-03-16  -0.958924
2000-03-17  -0.705540
2000-03-18  -0.279415
2000-03-19  0.156434
2000-03-20  0.563855
2000-03-21  0.873958
2000-03-22  0.993712
2000-03-23  0.915773
2000-03-24  0.644218
2000-03-25  0.216185
2000-03-26  -0.291995
2000-03-27  -0.681998
2000-03-28  -0.942222
2000-03-29  -0.989992
2000-03-30  -0.820148
2000-03-31  -0.409708
2000-04-01  0.087455
2000-04-02  0.583462
2000-04-03  0.894857
2000-04-04  0.996166
2000-04-05  0.878124
2000-04-06  0.455775
2000-04-07  0.035412
2000-04-08  -0.382026
2000-04-09  -0.768254
2000-04-10  -0.953923
2000-04-11  -0.970031
2000-04-12  -0.832980
2000-04-13  -0.498337
2000-04-14  -0.052571
2000-04-15  0.383436
2000-04-16  0.746099
2000-04-17  0.951602
2000-04-18  0.968827
2000-04-19  0.837097
2000-04-20  0.463418
2000-04-21  0.019203
2000-04-22  -0.416980
2000-04-23  -0.804321
2000-04-24  -0.974927
2000-04-25  -0.951125
2000-04-26  -0.732020
2000-04-27  -0.291834
2000-04-28  0.216395
2000-04-29  0.663318
2000-04-30  0.938705
2000-05-01  0.987166
2000-05-02  0.789353
2000-05-03  0.319507
2000-05-04  -0.189201
2000-05-05  -0.633824
2000-05-06  -0.909297
2000-05-07  -0.997495
2000-05-08  -0.841471
2000-05-09  -0.479425
2000-05-10  0.000000
2000-05-11  0.479425
2000-05-12  0.841471
2000-05-13  0.997495
2000-05-14  0.909297
2000-05-15  0.598472
2000-05-16  0.141120
2000-05-17  -0.350783
2000-05-18  -0.756802
2000-05-19  -0.977530
2000-05-20  -0.958924
2000-05-21  -0.705540
2000-05-22  -0.279415
2000-05-23  0.156434
2000-05-24  0.563855
2000-05-25  0.873958
2000-05-26  0.993712
2000-05-27  0.915773
2000-05-28  0.644218
2000-05-29  0.216185
2000-05-30  -0.291995
2000-05-31  -0.681998
2000-06-01  -0.942222
2000-06-02  -0.989992
2000-06-03  -0.820148
2000-06-04  -0.409708
2000-06-05  0.087455
2000-06-06  0.583462
2000-06-07  0.894857
2000-06-08  0.996166
2000-06-09  0.878124
2000-06-10  0.455775
2000-06-11  0.035412
2000-06-12  -0.382026
2000-06-13  -0.768254
2000-06-14  -0.953923
2000-06-15  -0.970031
2000-06-16  -0.832980
2000-06-17  -0.498337
2000-06-18  -0.052571
2000-06-19  0.383436
2000-06-20  0.746099
2000-06-21  0.951602
2000-06-22  0.968827
2000-06-23  0.837097
2000-06-24  0.463418
2000-06-25  0.019203
2000-06-26  -0.416980
2000-06-27  -0.804321
2000-06-28  -0.974927
2000-06-29  -0.951125
2000-06-30  -0.732020
2000-07-01  -0.291834
2000-07-02  0.216395
2000-07-03  0.663318
2000-07-04  0.938705
2000-07-05  0.987166
2000-07-06  0.789353
2000-07-07  0.319507
2000-07-08  -0.189201
2000-07-09  -0.633824
2000-07-10  -0.909297
2000-07-11  -0.997495
2000-07-12  -0.841471
2000-07-13  -0.479425
2000-07-14  0.000000
2000-07-15  0.479425
2000-07-16  0.841471
2000-07-17  0.997495
2000-07-18  0.909297
2000-07-19  0.598472
2000-07-20  0.141120
2000-07-21  -0.350783
2000-07-22  -0.756802
2000-07-23  -0.977530
2000-07-24  -0.958924
2000-07-25  -0.705540
2000-07-26  -0.279415
2000-07-27  0.156434
2000-07-28  0.563855
2000-07-29  0.873958
2000-07-30  0.993712
2000-07-31  0.915773
2000-08-01  0.644218
2000-08-02  0.216185
2000-08-03  -0.291995
2000-08-04  -0.681998
2000-08-05  -0.942222
2000-08-06  -0.989992
2000-08-07  -0.820148
2000-08-08  -0.409708
2000-08-09  0.087455
2000-08-10  0.583462
2000-08-11  0.894857
2000-08-12  0.996166
2000-08-13  0.878124
2000-08-14  0.455775
2000-08-15  0.035412
2000-08-16  -0.382026
2000-08-17  -0.768254
2000-08-18  -0.953923
2000-08-19  -0.970031
2000-08-20  -0.832980
2000-08-21  -0.498337
2000-08-22  -0.052571
2000-08-23  0.383436
2000-08-24  0.746099
2000-08-25  0.951602
2000-08-26  0.968827
2000-08-27  0.837097
2000-08-28  0.463418
2000-08-29  0.019203
2000-08-30  -0.416980
2000-08-31  -0.804321
2000-09-01  -0.974927
2000-09-02  -0.951125
2000-09-03  -0.732020
2000-09-04  -0.291834
2000-09-05  0.216395
2000-09-06  0.663318
2000-09-07  0.938705
2000-09-08  0.987166
2000-09-09  0.789353
2000-09-10  0.319507
2000-09-11  -0.189201
2000-09-12  -0.633824
2000-09-13  -0.909297
2000-09-14  -0.997495
2000-09-15  -0.841471
2000-09-16  -0.479425
2000-09-17  0.000000
2000-09-18  0.479425
2000-09-19  0.841471
2000-09-20  0.997495
2000-09-21  0.909297
2000-09-22  0.598472
2000-09-23  0.141120
2000-09-24  -0.350783
2000-09-25  -0.756802
2000-09-26  -0.977530
2000-09-27  -0.958924
2000-09-28  -0.705540
2000-09-29  -0.279415
2000-09-30  0.156434
2000-10-01  0.563855
2000-10-02  0.873958
2000-10-03  0.993712
2000-10-04  0.915773
2000-10-05  0.644218
2000-10-06  0.216185
2000-10-07  -0.291995
2000-10-08  -0.681998
2000-10-09  -0.942222
2000-10-10  -0.989992
2000-10-11  -0.820148
2000-10-12  -0.409708
2000-10-13  0.087455
2000-10-14  0.583462
2000-10-15  0.894857
2000-10-16  0.996166
2000-10-17  0.878124
2000-10-18  0.455775
2000-10-19  0.035412
2000-10-20  -0.382026
2000-10-21  -0.768254
2000-10-22  -0.953923
2000-10-23  -0.970031
2000-10-24  -0.832980
2000-10-25  -0.498337
2000-10-26  -0.052571
2000-10-27  0.383436
2000-10-28  0.746099
2000-10-29  0.951602
2000-10-30  0.968827
2000-10-31  0.837097
2000-11-01  0.463418
2000-11-02  0.019203
2000-11-03  -0.416980
2000-11-04  -0.804321
2000-11-05  -0.974927
2000-11-06  -0.951125
2000-11-07  -0.732020
2000-11-08  -0.291834
2000-11-09  0.216395
2000-11-10  0.663318
2000-11-11  0.938705
2000-11-12  0.987166
2000-11-13  0.789353
2000-11-14  0.319507
2000-11-15  -0.189201
2000-11-16  -0.633824
2000-11-17  -0.909297
2000-11-18  -0.997495
2000-11-19  -0.841471
2000-11-20  -0.479425
2000-11-21  0.000000
2000-11-22  0.479425
2000-11-23  0.841471
2000-11-24  0.997495
2000-11-25  0.909297
2000-11-26  0.598472
2000-11-27  0.141120
2000-11-28  -0.350783
2000-11-29  -0.756802
2000-11-30  -0.977530
Freq: D, dtype: float64
```

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

```
In [60]: ts.groupby(lambda x: y.year).transform(lambda x: x.max) - x.min()
Out[60]:
2000-01-01  0.000000
2000-01-02  0.479425
2000-01-03  0.841471
2000-01-04  0.997495
2000-01-05  0.909297
2000-01-06  0.598472
2000-01-07  0.141120
2000-01-08  -0.350783
2000-01-09  -0.756802
2000-01-10  -0.977530
2000-01-11  -0.958924
2000-01-12  -0.705540
2000-01-13  -0.279415
2000-01-14  0.156434
2000-01-15  0.563855
2000-01-16  0.873958
2000-01-17  0.993712
2000-01-18  0.915773
2000-01-19  0.644218
2000-01-20  0.216185
2000-01-21  -0.291995
2000-01-22  -0.681998
2000-01-23  -0.942222
2000-01-24  -0.989992
2000-01-25  -0.820148
2000-01-26  -0.409708
2000-01-27  0.087455
2000-01-28  0.583462
2000-01-29  0.894857
2000-01-30  0.996166
2000-01-31  0.878124
2000-02-01  0.455775
2000-02-02  0.035412
2000-02-03  -0.382026
2000-02-04  -0.768254
2000-02-05  -0.953923
2000-02-06  -0.970031
2000-02-07  -0.832980
2000-02-08  -0.498337
2000-02-09  -0.052571
2000-02-10  0.383436
2000-02-11  0.746099
2000-02-12  0.951602
2000-02-13  0.968827
2000-02-14  0.837097
2000-02-15  0.463418
2000-02-16  0.019203
2000-02-17  -0.416980
2000-02-18  -0.804321
2000-02-19  -0.974927
2000-02-20  -0.951125
2000-02-21  -0.732020
2000-02-22  -0.291834
2000-02-23  0.216395
2000-02-24  0.663318
2000-02-25  0.938705
2000-02-26  0.987166
2000-02-27  0.789353
2000-02-28  0.319507
2000-02-29  -0.189201
2000-03-01  -0.633824
2000-03-02  -0.909297
2000-03-03  -0.997495
2000-03-04  -0.841471
2000-03-05  -0.479425
2000-03-06  0.000000
2000-03-07  0.479425
2000-03-08  0.841471
2000-03-09  0.997495
2000-03-10  0.909297
2000-03-11  0.598472
2000-03-12  0.141120
2000-03-13  -0.350783
2000-03-14  -0.756802
2000-03-15  -0.977530
2000-03-16  -0.958924
2000-03-17  -0.705540
2000-03-18  -0.279415
2000-03-19  0.156434
2000-03-20  0.563855
2000-03-21  0.873958
2000-03-22  0.993712
2000-03-23  0.915773
2000-03-24  0.644218
2000-03-25  0.216185
2000-03-26  -0.291995
2000-03-27  -0.681998
2000-03-28  -0.942222
2000-03-29  -0.989992
2000-03-30  -0.820148
2000-03-31  -0.409708
2000-04-01  0.087455
2000-04-02  0.583462
2000-04-03  0.894857
2000-04-04  0.996166
2000-04-05  0.878124
2000-04-06  0.455775
2000-04-07  0.035412
2000-04-08  -0.382026
2000-04-09  -0.768254
2000-04-10  -0.953923
2000-04-11  -0.970031
2000-04-12  -0.832980
2000-04-13  -0.498337
2000-04-14  -0.052571
2000-04-15  0.383436
2000-04-16  0.746099
2000-04-17  0.951602
2000-04-18  0.968827
2000-04-19  0.837097
2000-04-20  0.463418
2000-04-21  0.019203
2000-04-22  -0.416980
2000-04-23  -0.804321
2000-04-24  -0.974927
2000-04-25  -0.951125
2000-04-26  -0.732020
2000-04-27  -0.291834
2000-04-28  0.216395
2000-04-29  0.663318
2000-04-30  0.938705
2000-05-01  0.987166
2000-05-02  0.789353
2000-05-03  0.319507
2000-05-04  -0.189201
2000-05-05  -0.633824
2000-05-06  -0.909297
2000-05-07  -0.997495
2000-05-08  -0.841471
2000-05-09  -0.479425
2000-05-10  0.000000
2000-05-11  0.479425
2000-05-12  0.841471
2000-05-13  0.997495
2000-05-14  0.909297
2000-05-15  0.598472
2000-05-16  0.141120
2000-05-17  -0.350783
2000-05-18  -0.756802
2000-05-19  -0.977530
2000-05-20  -0.958924
2000-05-21  -0.705540
2000-05-22  -0.279415
2000-05-23  0.156434
2000-05-24  0.563855
2000-05-25  0.873958
2000-05-26  0.993712
2000-05-27  0.915773
2000-05-28  0.644218
2000-05-29  0.216185
2000-05-30  -0.291995
2000-05-31  -0.681998
2000-06-01  -0.942222
2000-06-02  -0.989992
2000-06-03  -0.820148
2000-06-04  -0.409708
2000-06-05  0.087455
2000-06-06  0.583462
2000-06-07  0.894857
2000-06-08  0.996166
2000-06-09  0.878124
2000-06-10  0.455775
2000-06-11  0.035412
2000-06-12  -0.382026
2000-06-13  -0.768254
2000-06-14  -0.953923
2000-06-15  -0.970031
2000-06-16  -0.832980
2000-06-17  -0.498337
2000-06-18  -0.052571
2000-06-19  0.383436
2000-06-20  0.746099
2000-06-21  0.951602
2000-06-22  0.968827
2000-06-23  0.837097
2000-06-24  0.463418
2000-06-25  0.019203
2000-06-26  -0.416980
2000-06-27  -0.804321
2000-06-28  -0.974927
2000-06-29  -0.951125
2000-06-30  -0.732020
2000-07-01  -0.291834
2000-07-02  0.216395
2000-07-03  0.663318
2000-07-04  0.938705
2000-07-05  0.987166
2000-07-06  0.789353
2000-07-07  0.319507
2000-07-08  -0.189201
2000-07-09  -0.633824
2000-07-10  -0.909297
2000-07-11  -0.997495
2000-07-12  -0.841471
2000-07-13  -0.479425
2000-07-14  0.000000
2000-07-15  0.479425
2000-07-16  0.841471
2000-07-17  0.997495
2000-07-18  0.909297
2000-07-19  0.598472
2000-07-20  0.141120
2000-07-21  -0.350783
2000-07-22  -0.756802
2000-07-23  -0.977530
2000-07-24  -0.958924
2000-07-25  -0.705540
2000-07-26  -0.279415
2000-07-27  0.156434
2000-07-28  0.563855
2000-07-29  0.873958
2000-07-30  0.993712
2000-07-31  0.915773
2000-08-01  0.644218
2000-08-02  0.216185
2000-08-03  -0.291995
2000-08-04  -0.681998
2000-08-05  -0.942222
2000-08-06  -0.989992
2000-08-07  -0.820148
2000-08-08  -0.409708
2000-08-09  0.087455
2000-08-10  0.583462
2000-08-11  0.894857
2000-08-12  0.996166
2000-08-13  0.878124
2000-08-14  0.455775
2000-08-15  0.035412
2000-08-16  -0.382026
2000-08-17  -0.768254
2000-08-18  -0.953923
2000-08-19  -0.970031
2000-08-20  -0.832980
2000-08-21  -0.498337
2000-08-22  -0.052571
2000-08-23  0.383436
2000-08-24  0.746099
2000-08-25  0.951602
2000-08-26  0.968827
2000-08-27  0.837097
2000-08-28  0.463418
2000-08-29  0.019203
2000-08-30  -0.416980
2000-08-31  -0.804321
2000-09-01  -0.974927
2000-09-02  -0.951125
2000-09-03  -0.732020
2000-09-04  -0.291834
2000-09-05  0.216395
2000-09-06  0.663318
2000-09-07  0.938705
2000-09-08  0.987166
2000-09-09  0.789353
2000-09-10  0.319507
2000-09-11  -0.189201
2000-09-12  -0.633824
2000-09-13  -0.909297
2000-09-14  -0.997495
2000-09-15  -0.841471
2000-09-16  -0.479425
2000-09-17  0.000000
2000-09-18  0.479425
2000-09-19  0.841471
2000-09-20  0.997495
2000-09-21  0.909297
2000-09-22  0.598472
2000-09-23  0.141120
2000-09-24  -0.350783
2000-09-25  -0.756802
2000-09-26  -0.977530
2000-09-27  -0.958924
2000-09-28  -0.705540
2000-09-29  -0.279415
2000-09-30  0.156434
2000-10-01  0.563855
2000-10-02  0.873958
2000-10-03  0.993712
2000-10-04  0.915773
2000-10-05  0.644218
2000-10-06  0.216185
2000-10-07  -0.291995
2000-10-08  -0.681998
2000-10-09  -0.942222
2000-10-10  -0.989992
2000-10-11  -0.820148
2000-10-12  -0.409708
2000-10-13  0.087455
2000-10-14  0.583462
2000-10-15  0.894857
2000-10-16  0.996166
2000-10-17  0.878124
2000-10-18  0.455775
2000-10-19  0.035412
2000-10-20  -0.382026
2000-10-21  -0.768254
2000-10-22  -0.953923
2000-10-23  -0.970031
2000-10-24  -0.832980
2000-10-25  -0.498337
2000-10-26  -0.052571
2000-10-27  0.383436
2000-10-28  0.746099
2000-10-29  0.951602
2000-10-30  0.968827
2000-10-31  0.837097
2000-11-01  0.463418
2000-11-02  0.019203
2000-11-03  -0.416980
2000-11-04  -0.804321
2000-11-05  -0.974927
2000-11-06  -0.951125
2000-11-07  -0.732020
2000-11-08  -0.291834
2000-11-09  0.216395
2000-11-10  0.663318
2000-11-11  0.938705
2000-11-12  0.987166
2000-11-13  0.789353
2000-11-14  0.319507
2000-11-15  -0.189201
2000-11-16  -0.633824
2000-11-17  -0.909297
2000-11-18  -0.997495
2000-11-19  -0.841471
2000-11-20  -0.479425
2000-11-21  0.000000
2000-11-22  0.479425
2000-11-23  0.841471
2000-11-24  0.997495
2000-11-25  0.909297
2000-11-26  0.598472
2000-11-27  0.141120
2000-11-28  -0.350783
2000-11-29  -0.756802
2000-11-30  -0.977530
Freq: D, dtype: float64
```

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

```
In [61]: grouped_ts = ts.groupby(lambda x: y.year)
In [62]: ts = grouped_ts.max() - ts
Out[62]:
2000-01-01  0.000000
2000-01-02  0.479425
2000-01-03  0.841471
2000-01-04  0.997495
2000-01-05  0.909297
2000-01-06  0.598472
2000-01-07  0.141120
2000-01-08  -0.350783
2000-01-09  -0.756802
2000-01-10  -0.977530
2000-01-11  -0.958924
2000-01-12  -0.705540
2000-01-13  -0.279415
2000-01-14  0.156434
2000-01-15  0.563855
2000-01-16  0.873958
2000-01-17  0.993712
2000-01-18  0.915773
2000-01-19  0.644218
2000-01-20  0.216185
2000-01-21  -0.291995
2000-01-22  -0.681998
2000-01-23  -0.942222
2000-01-24  -0.989992
2000-01-25  -0.820148
2000-01-26  -0.409708
2000-01-27  0.087455
2000-01-28  0.583462
2000-01-29  0.894857
2000-01-30  0.996166
2000-01-31  0.878124
2000-02-01  0.455775
2000-02-02  0.035412
2000-02-03  -0.382026
2000-02-04  -0.768254
2000-02-05  -0.953923
2000-02-06  -0.970031
2000-02-07  -0.832980
2000-02-08  -0.498337
2000-02-09  -0.052571
2000-02-10  0.383436
2000-02-11  0.746099
2000-02-12  0.951602
2000-02-13  0.968827
2000-02-14  0.837097
2000-02-15  0.463418
2000-02-16  0.019203
2000-02-17  -0.416980
2000-02-18  -0.804321
2000-02-19  -0.974927
2000-02-20  -0.951125
2000-02-21  -0.732020
2000-02-22  -0.291834
2000-02-23  0.216395
2000-02-24  0.663318
2000-02-25  0.938705
2000-02-26  0.987166
2000-02-27  0.789353
2000-02-28  0.319507
2000-02-29  -0.189201
2000-03-01  -0.633824
2000-03-02  -0.909297
2000-03-03  -0.997495
2000-03-04  -0.841471
2000-03-05  -0.479425
2000-03-06  0.000000
2000-03-07  0.479425
2000-03-08  0.841471
2000-03-09  0.997495
2000-03-10  0.909297
2000-03-11  0.598472
2000-03-12  0.141120
2000-03-13  -0.350783
2000-03-14  -0.756802
2000-03-15  -0.977530
2000-03-16  -0.958924
2000-03-17  -0.705540
2000-03-18  -0.279415
2000-03-19  0.156434
2000-03-20  0.563855
2000-03-21  0.873958
2000-03-22  0.993712
2000-03-23  0.915773
2000-03-24  0.644218
2000-03-25  0.216185
2000-03-26  -0.291995
2000-03-27  -0.681998
2000-03-28  -0.942222
2000-03-29  -0.989992
2000-03-30  -0.820148
2000-03-31  -0.409708
2000-04-01  0.087455
2000-04-02  0.583462
2000-04-03  0.894857
2000-04-04  0.996166
2000-04-05  0.878124
2000-04-06  0.455775
2000-04-07  0.035412
2000-04-08  -0.382026
2000-04-09  -0.768254
2000-04-10  -0.953923
2000-04-11  -0.970031
2000-04-12  -0.832980
2000-04-13  -0.498337
2000-04-14  -0.052571
2000-04-15  0.383436
2000-04-16  0.746099
2000-04-17  0.951602
2000-04-18  0.968827
2000-04-19  0.837097
2000-04-20  0.463418
2000-04-21  0.019203
2000-04-22  -0.416980
2000-04-23  -0.804321
2000-04-24  -0.974927
2000-04-25  -0.951125
2000-04-26  -0.732020
2000-04-27  -0.291834
2000-04-28  0.216395
2000-04-29  0.663318
2000-04-30  0.938705
2000-05-01  0.987166
2000-05-02  0.789353
2000-05-03  0.319507
2000-05-04  -0.189201
2000-05-05  -0.633824
2000-05-06
```

The `groupby()` method will accumulate a pandas `GroupBy` object. It the example for all the members of each unique group:

```
In [1]: df1 = pd.read_csv('3-expenses.csv')
In [2]: df1.groupby('Type').count()
Out[2]:
       Date  Amount  Description
Type
A      2  2       2
B      3  3       3
C      1  1       1
D      2  2       2
E      1  1       1
F      1  1       1
G      1  1       1
[7 rows x 3 columns]
```

Suppose you want to see the `Amount` values in each Type. Frequency in each group of your dataframe and add to the resulting values with the `.sum()` method:

```
In [3]: df1.groupby('Type').sum()
Out[3]:
       Amount  Unnamed: 0
Type
A      -2.00           1
B      -3.00           3
C      -1.00           4
D      -2.00           5
E      -1.00           6
F      -1.00           7
G      -1.00           8
[7 rows x 2 columns]
```

Suppose you want to see the `Amount` values in each Type. Frequency in each group of your dataframe and add to the resulting values with the `.mean()` method:

```
In [4]: df1.groupby('Type').mean()
Out[4]:
       Amount  Unnamed: 0
Type
A      -1.00           1
B      -1.00           3
C      -1.00           4
D      -1.00           5
E      -1.00           6
F      -1.00           7
G      -1.00           8
[7 rows x 2 columns]
```

Filtration

A filtration is a broadly operative the contents from original group objects. It may either filter out entire groups, part of groups, or both. Filtration then returns a filtered version of the other object, including the grouping columns when provided. In the following example, `.filter()` is included in the result.

```
In [5]: df2 = pd.read_csv('4-cars.csv')
In [6]: df2.groupby('Cylinders').count()
Out[6]:
           Unnamed: 0  Year  Make  Model  Mileage  Price  Cost
Cylinders
4                  2     2     2      2        2      2     2
6                  3     3     3      3        3      3     3
8                  2     2     2      2        2      2     2
In [7]: df2.groupby('Cylinders').filter(lambda x: len(x) > 2)
Out[7]:
   Unnamed: 0  Year  Make  Model  Mileage  Price  Cost
0           0  2008  Ford   Focus    73915   8000  2000
1           1  2009  Ford   Focus    80610   7000  2000
4           4  2008   Dog  Wagon    80000   4000  1000
5           5  2009   Dog  Wagon    77743   7000  1000
7           7  2009   Dog   Sport    62043   8000  1000
In [8]: df2[df2.groupby('Cylinders').count() > 2].dropna()
Out[8]:
   Unnamed: 0  Year  Make  Model  Mileage  Price  Cost
0           0  2008  Ford   Focus    73915   8000  2000
1           1  2009  Ford   Focus    80610   7000  2000
4           4  2008   Dog  Wagon    80000   4000  1000
5           5  2009   Dog  Wagon    77743   7000  1000
7           7  2009   Dog   Sport    62043   8000  1000
```

> **Note**
> Unlike aggregations, filtrations do not split the group keys to the index of the result. Because of this, passing `.groupby(..., dropna=False)` will not affect these transformations.

Filtrations will respect excluding NA values of the GroupBy object.

```
In [9]: df3 = pd.read_csv('7-pets.csv', sep=';')
In [10]: df3.groupby('Weight', dropna=False).filter(lambda x: len(x) > 1)
Out[10]:
   Name  Breed  Weight
0  Bomi  Scott     NaN
2  Dash   Lab     NaN
```

Built-in filtrations

The following filtrations are broadly set as filtrations. All these methods have a Cython-optimized implementation.

Method	Description
`head()`	Select the top row(s) of each group
`last()`	Select the last row(s) of each group
`nth()`	Select the bottom row(s) of each group

Users can also use transformations along with Boolean indexing to construct complex filtrations within groups. For example, suppose we are interested in values within a group that are greater than the mean of a group. We take to find the largest products reporting no more than 90% of the mean volume within each group.

```
In [11]: df4 = pd.read_csv('8-product_orders.csv')
In [12]: df4.groupby('product').head()
Out[12]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
2        1              NaN  ...            NaN           NaN
3        1              NaN  ...            NaN           NaN
4        2              NaN  ...            NaN           NaN
[5 rows x 6 columns]
In [13]: df4.groupby('product').head(2)
Out[13]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
4        2              NaN  ...            NaN           NaN
5        2              NaN  ...            NaN           NaN
6        3              NaN  ...            NaN           NaN
7        3              NaN  ...            NaN           NaN
[6 rows x 6 columns]
In [14]: df4.groupby('product').nth(1)
Out[14]:
   product  product_ordered  ...  product_price  product_cost
1        1              NaN  ...            NaN           NaN
5        2              NaN  ...            NaN           NaN
7        3              NaN  ...            NaN           NaN
8        4              NaN  ...            NaN           NaN
9        5              NaN  ...            NaN           NaN
[5 rows x 6 columns]
In [15]: df4.groupby('product').last()
Out[15]:
         product_ordered  product_price  product_cost
product
1                    NaN            NaN           NaN
2                    NaN            NaN           NaN
3                    NaN            NaN           NaN
4                    NaN            NaN           NaN
5                    NaN            NaN           NaN
In [16]: df4[df4.groupby('product').transform('count') > 1].dropna()
Out[16]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
2        1              NaN  ...            NaN           NaN
3        1              NaN  ...            NaN           NaN
4        2              NaN  ...            NaN           NaN
5        2              NaN  ...            NaN           NaN
6        3              NaN  ...            NaN           NaN
7        3              NaN  ...            NaN           NaN
[8 rows x 6 columns]
In [17]: df4[df4.groupby('product').transform('mean') < 0.9].dropna()
Out[17]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
2        1              NaN  ...            NaN           NaN
3        1              NaN  ...            NaN           NaN
[4 rows x 6 columns]
```

The `filter` method

> **Note**
> Filtering by supplying `filter` with a User-Defined Function (UDF) is often less performant than using the built-in methods on GroupBy. Consider rewriting an complex operation into a chain of operations that utilize the built-in methods.

The `filter` method takes a User-Defined Function (UDF) that, when applied to an entire group, returns either `True` or `False`. The result of the `filter` method is then the subset of groups for which the UDF returned `True`.

Suppose we want to take only those elements that belong to groups with a group sum greater than 2.

```
In [18]: df = pd.Series(n, 1, 2, 3, 3)
Out[18]:
0    1
1    2
2    3
3    3
dtype: int64
```

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

```
In [19]: df = pd.DataFrame({'A': [a,b,a,b,c], 'B': [1,2,3,4,5]})
Out[19]:
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
4  c  5
In [20]: df.groupby('A').filter(lambda x: len(x) > 2)
Out[20]:
   A  B
0  a  1
2  a  3
4  c  5
```

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 [21]: df.groupby('A').filter(lambda x: len(x) > 2, dropna=False)
Out[21]:
   A  B
0  a  1
1  NaN
2  a  3
3  NaN
4  c  5
```

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

```
In [22]: df2 = df[['A']]
In [23]: df2.groupby('A').filter(lambda x: len(x) > 2)
Out[23]:
   A
0  a
2  a
4  c
```

Flexible `apply`

Some operations on the grouped data might not fit into the aggregations, transformations, or filtration categories. For these, you can use the `apply` function.

> **Warning**
> `apply` has to be in the order for each whether it should act as a reduce, transformer, or filter, depending on exactly what is passed to it. So for groups with the same performance (as defined in the reduce or rule. While it tries to intelligently guess how to behave, it can sometimes guess wrong.

> **Note**
> All of the examples in this section can be more reliably, and more efficiently, computed using other pandas functionality.

```
In [24]: df = pd.DataFrame({'A': [a,b,a,b,c],
                            'B': [1,2,3,4,5]})
In [25]: df
Out[25]:
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
4  c  5
In [26]: grouped = df.groupby('A')
In [27]: def f(x):
         ...:     return pd.Series([x, x ** 2], index=['x', 'x_squared'])
         ...:
In [28]: grouped.apply(f)
Out[28]:
   x  x_squared
A
a  1          1
a  3          9
b  2          4
b  4         16
c  5         25
In [29]: df = pd.DataFrame({'A': [a,b,a,b,c],
                            'B': [1,2,3,4,5]})
In [30]: df
Out[30]:
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
4  c  5
In [31]: grouped = df.groupby('A')
In [32]: def f(x):
         ...:     return pd.Series([x, x ** 2], index=['x', 'x_squared'])
         ...:
In [33]: grouped.apply(f)
Out[33]:
   x  x_squared
A
a  1          1
a  3          9
b  2          4
b  4         16
c  5         25
```

The `groupby()` method

The `groupby()` method will accumulate a pandas `GroupBy` object. It the example for all the members of each unique group:

```
In [34]: df1 = pd.read_csv('3-expenses.csv')
In [35]: df1.groupby('Type').count()
Out[35]:
       Date  Amount  Description
Type
A      2  2       2
B      3  3       3
C      1  1       1
D      2  2       2
E      1  1       1
F      1  1       1
G      1  1       1
[7 rows x 3 columns]
```

Suppose you want to see the `Amount` values in each Type. Frequency in each group of your dataframe and add to the resulting values with the `.sum()` method:

```
In [36]: df1.groupby('Type').sum()
Out[36]:
       Amount  Unnamed: 0
Type
A      -2.00           1
B      -3.00           3
C      -1.00           4
D      -2.00           5
E      -1.00           6
F      -1.00           7
G      -1.00           8
[7 rows x 2 columns]
```

Suppose you want to see the `Amount` values in each Type. Frequency in each group of your dataframe and add to the resulting values with the `.mean()` method:

```
In [37]: df1.groupby('Type').mean()
Out[37]:
       Amount  Unnamed: 0
Type
A      -1.00           1
B      -1.00           3
C      -1.00           4
D      -1.00           5
E      -1.00           6
F      -1.00           7
G      -1.00           8
[7 rows x 2 columns]
```

Filtration

A filtration is a broadly operative the contents from original group objects. It may either filter out entire groups, part of groups, or both. Filtration then returns a filtered version of the other object, including the grouping columns when provided. In the following example, `.filter()` is included in the result.

```
In [38]: df2 = pd.read_csv('4-cars.csv')
In [39]: df2.groupby('Cylinders').count()
Out[39]:
           Unnamed: 0  Year  Make  Model  Mileage  Price  Cost
Cylinders
4                  2     2     2      2        2      2     2
6                  3     3     3      3        3      3     3
8                  2     2     2      2        2      2     2
In [40]: df2.groupby('Cylinders').filter(lambda x: len(x) > 2)
Out[40]:
   Unnamed: 0  Year  Make  Model  Mileage  Price  Cost
0           0  2008  Ford   Focus    73915   8000  2000
1           1  2009  Ford   Focus    80610   7000  2000
4           4  2008   Dog  Wagon    80000   4000  1000
5           5  2009   Dog  Wagon    77743   7000  1000
7           7  2009   Dog   Sport    62043   8000  1000
In [41]: df2[df2.groupby('Cylinders').count() > 2].dropna()
Out[41]:
   Unnamed: 0  Year  Make  Model  Mileage  Price  Cost
0           0  2008  Ford   Focus    73915   8000  2000
1           1  2009  Ford   Focus    80610   7000  2000
4           4  2008   Dog  Wagon    80000   4000  1000
5           5  2009   Dog  Wagon    77743   7000  1000
7           7  2009   Dog   Sport    62043   8000  1000
```

Built-in filtrations

The following filtrations are broadly set as filtrations. All these methods have a Cython-optimized implementation.

Method	Description
`head()`	Select the top row(s) of each group
`last()`	Select the last row(s) of each group
`nth()`	Select the bottom row(s) of each group

Users can also use transformations along with Boolean indexing to construct complex filtrations within groups. For example, suppose we are interested in values within a group that are greater than the mean of a group. We take to find the largest products reporting no more than 90% of the mean volume within each group.

```
In [42]: df4 = pd.read_csv('8-product_orders.csv')
In [43]: df4.groupby('product').head()
Out[43]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
2        1              NaN  ...            NaN           NaN
3        1              NaN  ...            NaN           NaN
4        2              NaN  ...            NaN           NaN
[5 rows x 6 columns]
In [44]: df4.groupby('product').head(2)
Out[44]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
4        2              NaN  ...            NaN           NaN
5        2              NaN  ...            NaN           NaN
6        3              NaN  ...            NaN           NaN
7        3              NaN  ...            NaN           NaN
[6 rows x 6 columns]
In [45]: df4.groupby('product').nth(1)
Out[45]:
   product  product_ordered  ...  product_price  product_cost
1        1              NaN  ...            NaN           NaN
5        2              NaN  ...            NaN           NaN
7        3              NaN  ...            NaN           NaN
8        4              NaN  ...            NaN           NaN
9        5              NaN  ...            NaN           NaN
[5 rows x 6 columns]
In [46]: df4.groupby('product').last()
Out[46]:
         product_ordered  product_price  product_cost
product
1                    NaN            NaN           NaN
2                    NaN            NaN           NaN
3                    NaN            NaN           NaN
4                    NaN            NaN           NaN
5                    NaN            NaN           NaN
In [47]: df4[df4.groupby('product').transform('count') > 1].dropna()
Out[47]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
2        1              NaN  ...            NaN           NaN
3        1              NaN  ...            NaN           NaN
4        2              NaN  ...            NaN           NaN
5        2              NaN  ...            NaN           NaN
6        3              NaN  ...            NaN           NaN
7        3              NaN  ...            NaN           NaN
[8 rows x 6 columns]
In [48]: df4[df4.groupby('product').transform('mean') < 0.9].dropna()
Out[48]:
   product  product_ordered  ...  product_price  product_cost
0        1              NaN  ...            NaN           NaN
1        1              NaN  ...            NaN           NaN
2        1              NaN  ...            NaN           NaN
3        1              NaN  ...            NaN           NaN
[4 rows x 6 columns]
```

The `filter` method

> **Note**
> Filtering by supplying `filter` with a User-Defined Function (UDF) is often less performant than using the built-in methods on GroupBy. Consider rewriting an complex operation into a chain of operations that utilize the built-in methods.

The `filter` method takes a User-Defined Function (UDF) that, when applied to an entire group, returns either `True` or `False`. The result of the `filter` method is then the subset of groups for which the UDF returned `True`.

Suppose we want to take only those elements that belong to groups with a group sum greater than 2.

```
In [49]: df = pd.Series(n, 1, 2, 3, 3)
Out[49]:
0    1
1    2
2    3
3    3
dtype: int64
```

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

```
In [50]: df = pd.DataFrame({'A': [a,b,a,b,c], 'B': [1,2,3,4,5]})
In [51]: df.groupby('A').filter(lambda x: len(x) > 2)
Out[51]:
   A  B
0  a  1
2  a  3
4  c  5
```

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 [52]: df2 = df[['A']]
In [53]: df2.groupby('A').filter(lambda x: len(x) > 2, dropna=False)
Out[53]:
   A
0  a
1  NaN
2  a
3  NaN
4  c
```

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

```
In [54]: df2 = df[['A']]
In [55]: df2.groupby('A').filter(lambda x: len(x) > 2)
Out[55]:
   A
0  a
2  a
4  c
```

Flexible `apply`

Some operations on the grouped data might not fit into the aggregations, transformations, or filtration categories. For these, you can use the `apply` function.

> **Warning**
> `apply` has to be in the order for each whether it should act as a reduce, transformer, or filter, depending on exactly what is passed to it. So for groups with the same performance (as defined in the reduce or rule. While it tries to intelligently guess how to behave, it can sometimes guess wrong.

> **Note**
> All of the examples in this section can be more reliably, and more efficiently, computed using other pandas functionality.

```
In [56]: df = pd.DataFrame({'A': [a,b,a,b,c],
                            'B': [1,2,3,4,5]})
In [57]: df
Out[57]:
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
4  c  5
In [58]: grouped = df.groupby('A')
In [59]: def f(x):
         ...:     return pd.Series([x, x ** 2], index=['x', 'x_squared'])
         ...:
In [60]: grouped.apply(f)
Out[60]:
   x  x_squared
A
a  1          1
a  3          9
b  2          4
b  4         16
c  5         25
In [61]: df = pd.DataFrame({'A': [a,b,a,b,c],
                            'B': [1,2,3,4,5]})
In [62]: df
Out[62]:
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
4  c  5
In [63]: grouped = df.groupby('A')
In [64]: def f(x):
         ...:     return pd.Series([x, x ** 2], index=['x', 'x_squared'])
         ...:
In [65]: grouped.apply(f)
Out[65]:
   x  x_squared
A
a  1          1
a  3          9
b  2          4
b  4         16
c  5         25
```

Numba Accelerated Routines

> **New in version 1.1.**

If Numba is installed as an optional dependency, the `transform` and `aggregate` methods support `engine='numba'` and `engine_kwargs` arguments. See enhancing performance with Numba for general usage of the arguments and performance considerations.

The function signature must have `def, values, index)` as the data ordering in each group will be passed into `values`, and optim group index will be passed into `index`.

> **Warning**
> When using `engine='numba'`, there will no "fast path" between calls. The group data and group index will be passed as NumPy arrays to the JITed user defined function, and no alternative execution attempts will be tried.

Other useful features

Exclusion of "nuisance" columns

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

```
In [66]: df
Out[66]:
     A      B         C         D
0  foo    one  0.326922  0.488777
1  bar    one  1.486072  1.603307
2  foo    two  1.390355  0.851010
3  bar  three  -0.348999  -0.843267
4  foo    two  0.133799  -1.000884
5  bar    two  -1.620073  0.333226
6  foo    one  -0.976483  -0.790351
7  foo  three  0.436274  0.737369
```

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. You can avoid nuisance columns by specifying `select_dtypes()` first:

```
In [67]: grouped = df.groupby('A')
In [68]: grouped.select_dtypes(np.float)
Out[68]:
     C         D
0  0.326922  0.488777
1  1.486072  1.603307
2  1.390355  0.851010
3  -0.348999  -0.843267
4  0.133799  -1.000884
5  -1.620073  0.333226
6  -0.976483  -0.790351
7  0.436274  0.737369
```

Note that `.sum()` and `.mean()` are more efficient than `grouped.select_dtypes(np.float).sum()`, or the code of an aggregation function is only interesting over one column (here `colname`), it may be filtered before applying the aggregation function.

> **Note**
> Any groupby operation involving a non-string type column will result in a UserWarning. In the future, this will raise a TypeError.

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 [69]: pd.Series([1, 1, 1], grouped,
                   ...: ['a', 'a', 'b'], categories=['a', 'b'], observed=False)
Out[69]:
a    1
a    1
b    1
dtype: int64
```

Show only the observed values:

```
In [70]: pd.Series([1, 1, 1], grouped,
                   ...: ['a', 'a', 'b'], categories=['a', 'b'], observed=True)
Out[70]:
a    2
b    1
dtype: int64
```

The aforesaid groups of the proposed will always included of the categories that were grouped.

```
In [71]: pd.Series([1, 1, 1], grouped,
                   ...: ['a', 'a', 'b'], categories=['a', 'b'], observed=True\n0
Out[71]:
a    2
b    1
dtype: int64
In [72]: s = df.groupby(df.A.astype('category'))
In [73]: s.count()
Out[73]:
A
foo  NaN
bar  NaN
dtype: float64
In [74]: s = df.groupby(df.A.astype('category'), observed=False)
In [75]: s.count()
Out[75]:
A
foo    4
bar    4
baz    0
dtype: int64
```

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 [76]: pd.Series([1, 1, 1], grouped,
                   ...: ['a', 'a', 'b'], categories=['a', 'b'], observed=False
Out[76]:
a    1
a    1
b    1
dtype: int64
```

Show only the observed values:

```
In [77]: pd.Series([1, 1, 1], grouped,
                   ...: ['a', 'a', 'b'], categories=['a', 'b'], observed=True
Out[77]:
a    2
b    1
dtype: int64
```

The aforesaid groups of the proposed will always included of the categories that were grouped.

```
In [78]: pd.Series([1, 1, 1], grouped,
                   ...: ['a', 'a', 'b'], categories=['a', 'b'], observed=True\n0
Out[78]:
a    2
b    1
dtype: int64
In [79]: s = df.groupby(df.A.astype('category'))
In [80]: s.count()
Out[80]:
A
foo  NaN
bar  NaN
dtype: float64
In [81]: s = df.groupby(df.A.astype('category'), observed=False)
In [82]: s.count()
Out[82]:
A
foo    4
bar    4
baz    0
dtype: int64
```

Grouping with ordered factors

Categorical variables represented in a sensical of operators. `Categorical` data can be used as group keys. If so, the order of the levels will be preserved:

```
In [83]: data = df.groupby(df.A.astype('category'))
In [84]: factor = pd.qcut(data, [0, 0.25, 0.5, 0.75, 1.0])
In [85]: data.size().groupby(factor).sum()
Out[85]:
A
(-1.621, -0.976]  2
(-0.976, 0.326]   2
(0.326, 0.436]    2
(0.436, 1.486]    2
dtype: int64
```

Grouping with a grouper specification

You may need to specify a bit more data to properly group. You can use the `pd.Grouper` to provide this local control.

```
In [86]: import numpy
In [87]: df = pd.DataFrame({
         ...:                'Branch' : A A A A A A A B',
         ...:                'Buyer' : Carl Mark Carl Carl Joe Joe',
         ...:                'Quantity' : 18, 3, 5, 1, 9, 3, 5,
         ...:                'date' : [ numpy.nan,
         ...:                          datetime.datetime(2013,11,1,13,0),
         ...:                          datetime.datetime(2013,10,1,20,0),
         ...:                          datetime.datetime(2013,10,2,10,0),
         ...:                          datetime.datetime(2013,12,2,12,0),
         ...:                          datetime.datetime(2013,12,2,14,0),
         ...:                          datetime.datetime(2013,12,3,10,0)],
         ...:                'type' : [ apple, pear, orange, apple,
         ...:                          apple, pear, orange]
         ...:                })
         ...:
In [88]: df
Out[88]:
   Branch Buyer  Quantity                date   type
0       A  Carl        18                 NaT  apple
1       A  Mark         3 2013-11-01 13:00:00   pear
2       A  Carl         5 2013-10-01 20:00:00 orange
3       A  Carl         1 2013-10-02 10:00:00  apple
4       A   Joe         9 2013-12-02 12:00:00   pear
5       A   Joe         3 2013-12-02 14:00:00 orange
6       B   Joe         5 2013-12-03 10:00:00  apple
[7 rows x 5 columns]
```

Groupby is specified by some sort this desired frequency. This is like resampling.

```
In [89]: df.groupby([pd.Grouper(freq='1M', key='date'), 'Buyer']).sum()
Out[89]:
            Quantity
date       Buyer
2013-10-31 Carl     5
           Mark     3
2013-11-30 Carl    18
2013-12-31 Joe     17
           Mark     1
2013-10-31 Joe     9
```

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

```
In [90]: df = df.set_index('Buyer')
In [91]: df.groupby([pd.Grouper(freq='1M', key='date'), 'Buyer').sum()
Out[91]:
            Quantity
date       Buyer
2013-10-31 Carl     5
           Mark     3
2013-11-30 Carl    18
2013-12-31 Joe     17
           Mark     1
2013-10-31 Joe     9
In [92]: df.groupby([pd.Grouper(freq='1M', level='date'), 'Buyer').sum()
Out[92]:
            Quantity
date       Buyer
2013-10-31 Carl     5
           Mark     3
2013-11-30 Carl    18
2013-12-31 Joe     17
           Mark     1
2013-10-31 Joe     9
```

Taking the first rows of each group

Just like for a DataFrame or Series you can call head and tail on a groupby:

```
In [93]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])
Out[93]:
   A  B
0  1  2
1  1  4
2  5  6
In [94]: g = df.groupby('A')
In [95]: g.head(1)
Out[95]:
   A  B
0  1  2
5  5  6
In [96]: g.tail(1)
Out[96]:
   A  B
1  1  4
5  5  6
```

This shows the first or last n rows from each group.

Taking the nth row of each group

To select the nth-item from each group, use `DataFrame.nth()` or `Series.nth()`. Arguments supplied can be any integer, list of integers, slice, or list of slices; see below for examples. When the nth element of a group does not exist an error is not raised but no corresponding rows are returned.

In addition to that non-grouped usual slicing, one may use `GroupBy.first()` which does non-skip groups matching it also is equivalent to `GroupBy.nth(0)`. Also `GroupBy.last()` is available, and is equivalent to `GroupBy.nth(-1)`.

```
In [97]: df = pd.DataFrame([[1, nan], [1, 4], [5, 6]], columns=['A', 'B'])
Out[97]:
   A  B
0  1  2
1  1  4
2  5  6
In [98]: g = df.groupby('A')
In [99]: g.nth(0)
Out[99]:
   A  B
0  1  2
5  5  6
In [100]: g.nth(-1)
Out[100]:
   A  B
1  1  4
5  5  6
In [101]: g.nth(1)
Out[101]:
   A  B
1  1  4
In [102]: g.nth(5)
Out[102]:
Empty DF
Columns: [A, B]
Index: [5]
In [103]: g.nth(2, dropna='any')
Out[103]:
   A  B
5  5  6
In [104]: g.nth(-3, dropna='any')
Out[104]:
   A  B
0  1  2
In [105]: business_days = pd.date_range('1/1/2000', freq='B', periods=5)
In [106]: df = pd.DataFrame(n, 1, 2, 3, 4, 5, columns=['a'], index=b
In [107]: df.groupby(lambda x: x.month).nth(0, dropna='any')
Out[107]:
   a
2  1
3  2
4  3
5  4
6  5
In [108]: df.groupby(df.index.map(lambda x: x.month)).nth(0, dropna='any')
Out[108]:
   a
2  1
3  2
4  3
5  4
6  5
In [109]: df.groupby(df.index.map(lambda x: x.month)).nth(-1, dropna='any')
Out[109]:
   a
2  1
3  2
4  3
5  4
6  5
```

You may also use a slice or lists of slices.

```
In [110]: df.groupby(df.index.map(lambda x: x.month)).nth(slice(1))
Out[110]:
   a
2  1
3  2
4  3
5  4
6  5
In [111]: df.groupby(df.index.map(lambda x: x.month)).nth([0, 1])
Out[111]:
   a
2  1
3  2
4  3
5  4
6  5
In [112]: df.groupby(df.index.map(lambda x: x.month)).nth([0, 1, -1])
Out[112]:
   a
2  1
3  2
4  3
5  4
6  5
```

```
2014-04-02  1  1
2014-04-03  1  1
2014-04-03  1  1
2014-04-04  1  1
2014-04-04  1  1
2014-04-05  1  1
            ..  ..
2014-06-24  1  1
2014-06-25  1  1
2014-06-26  1  1
2014-06-27  1  1
2014-06-30  1  1

[65 rows x 2 columns]
```

Enumerate group items

To see the order in which each row appears within its group, use the `cumcount` method:

```
In [2313]: dfg = pd.DataFrame(list("aaabba"), columns=["A"])

In [2312]: dfg
Out[2312]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [2313]: dfg.groupby("A").cumcount()
Out[2313]:
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64

In [2314]: dfg.groupby("A").cumcount(ascending=False)
Out[2314]:
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

Enumerate groups

To see the ordering of the groups (as opposed to the order of rows within a group given by `cumcount`) you can use `ngroup()`.

Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

```
In [2315]: dfg = pd.DataFrame(list("aaabba"), columns=["A"])

In [2316]: dfg
Out[2316]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [2317]: dfg.groupby("A").ngroup()
Out[2317]:
0    0
1    0
2    0
3    1
4    1
5    0
dtype: int64

In [2318]: dfg.groupby("A").ngroup(ascending=False)
Out[2318]:
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
```

Plotting

Groupby also works with some plotting methods. For example, suppose we suspect that some features in a DataFrame may differ by group, in this case, the values in column 1 where the group is "B" are 3 higher on average.

```
In [2319]: np.random.seed(1234)

In [2420]: df = pd.DataFrame(np.random.randn(50, 2))

In [2421]: df["g"] = np.random.choice(["A", "B"], size=50)

In [2422]: df.loc[df["g"] == "B", 1] += 3
```

We can easily visualize this with a boxplot:

```
In [2423]: df.groupby("g").boxplot()
Out[2423]:
A    AxesSubplot(0.1,0.15;0.363636w0.75)
B    AxesSubplot(0.536364,0.15;0.363636w0.75)
dtype: object
```

The result of calling `boxplot` is a dictionary whose keys are the values of our grouping column `g` ("A" and "B"). The values of the resulting dictionary can be controlled by the `return_type` keyword of `boxplot`. See the visualization documentation for more.

> **Warning**
>
> For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See here for an explanation.

Piping function calls

Similar to the functionality provided by `DataFrame` and `Series`, functions that take `GroupBy` objects can be chained together using a `pipe` method to allow for a cleaner, more readable syntax. To read about `.pipe` in general terms, see here.

Combining `.groupby` and `.pipe` is often useful when you need to reuse GroupBy objects.

As an example, imagine having a DataFrame with columns for stores, products, revenue and quantity sold. We'd like to do a groupwise calculation of prices (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable. First we set the data:

```
In [2444]: n = 1000

In [2445]: df = pd.DataFrame(
   .....:     {
   .....:         "Store": np.random.choice(["Store_1", "Store_2"], n),
   .....:         "Product": np.random.choice(["Product_1", "Product_2"], n),
   .....:         "Revenue": (np.random.random(n) * 50 + 10).round(2),
   .....:         "Quantity": np.random.randint(1, 10, size=n),
   .....:     }
   .....: )
   .....:

In [2446]: df.head(2)
Out[2446]:
     Store    Product  Revenue  Quantity
0  Store_2  Product_1    26.12         1
1  Store_2  Product_1    28.06         1
```

Now, to find prices per store/product, we can simply do:

```
In [2473]: (
   .....:     df.groupby(["Store", "Product"])
   .....:     .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
   .....:     .unstack()
   .....:     .round(2)
   .....: )
   .....:
Out[2473]:
Product  Product_1  Product_2
Store
Store_1       6.82       7.05
Store_2       6.30       6.64
```

Piping can also be expressive when you want to deliver a grouped object to some arbitrary function, for example:

```
In [2488]: def mean(groupby):
   .....:     return groupby.mean()
   .....:

In [2489]: df.groupby(["Store", "Product"]).pipe(mean)
Out[2489]:
                     Revenue  Quantity
Store   Product
Store_1 Product_1  34.622727  5.075758
        Product_2  35.482815  5.029630
Store_2 Product_1  32.972837  5.237589
        Product_2  34.684360  5.224000
```

where `mean` takes a GroupBy object and finds the mean of the Revenue and Quantity columns respectively for each Store-Product combination. The `mean` function can be any function that takes in a GroupBy object; the `.pipe` will pass the GroupBy object as a parameter into the function you specify.

Examples

Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.

```
In [250]: df = pd.DataFrame({"a": [1, 0, 0], "b": [0, 1, 0], "c": [1, 0, 0], "d": [2, 3, 4]})

In [251]: df
Out[251]:
   a  b  c  d
0  1  0  1  2
1  0  1  0  3
2  0  0  0  4

In [252]: df.groupby(df.sum(), axis=1).sum()
Out[252]:
   1  0
0  2  2
1  1  3
2  0  4
```

Multi-column factorization

By using `ngroup()`, we can extract information about the groups in a way similar to `factorize()` (as described further in the reshaping API) but which applies naturally to multiple columns of mixed type and different sources. This can be useful as an intermediate categorical-like step in processing, when the relationships between the group rows are more important than their content, or as input to an algorithm which only accepts the integer encoding. (For more information about support in pandas for full categorical data, see the Categorical introduction and the API documentation.)

```
In [253]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba")})

In [254]: dfg
Out[254]:
   A  B
0  1  a
1  1  a
2  2  a
3  3  b
4  2  a

In [255]: dfg.groupby(["A", "B"]).ngroup()
Out[255]:
0    0
1    0
2    1
3    2
4    1
dtype: int64

In [256]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
Out[256]:
0    0
1    0
2    1
3    3
4    2
dtype: int64
```

Groupby by indexer to 'resample' data

Resampling produces new hypothetical samples (resamples) from already existing observed data or from a model that generates data. These new samples are similar to the pre-existing samples.

In order to resample to work on indices that are non-datetimelike, the following procedure can be utilized.

In the following examples, **df.index // 5** returns a binary array which is used to determine what gets selected for the groupby operation.

> **Note**
>
> The below example shows how we can downsample by consolidation of samples into fewer samples. Here by using **df.index // 5**, we are aggregating the samples in bins. By applying **std()** function, we aggregate the information contained in many samples into a small subset of values which is their standard deviation thereby reducing the number of samples.

```
In [257]: df = pd.DataFrame(np.random.randn(10, 2))

In [258]: df
Out[258]:
          0         1
0 -0.793893  0.321153
1  0.342250  1.618906
2 -0.975807  1.918201
3 -0.810847 -1.405919
4 -1.977759  0.461659
5  0.730036 -1.316778
6 -0.751520  0.528308
7 -0.257750 -1.081009
8  0.505895 -1.701948
9 -1.006349  0.020280

In [259]: df.index // 5
Out[259]: Index([0, 0, 0, 0, 1, 1, 1, 1, 1, 1], dtype='int64')

In [260]: df.groupby(df.index // 5).std()
Out[260]:
          0         1
0  0.823647  1.312912
1  0.700280  0.942941
```

Returning a Series to propagate names

Group DataFrame columns, compute a set of metrics and return a named Series. The Series name is used as the name for the column index. This is especially useful in conjunction with reshaping operations such as stacking in which the column index name will be used as the name of the inserted column:

```
In [261]: df = pd.DataFrame(
   .....:     {
   .....:         "a": [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2],
   .....:         "b": [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
   .....:         "c": [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
   .....:         "d": [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1],
   .....:     }
   .....: )
   .....:

In [262]: def compute_metrics(x):
   .....:     result = {"sum": x["d"].sum(), "c_mean": x["c"].mean()}
   .....:     return pd.Series(result, name="metrics")
   .....:

In [263]: result = df.groupby(["a"].apply(compute_metrics)

In [264]: result
Out[264]:
metrics  d_sum  c_mean
a
0        2.0     0.5
1        2.0     0.5
2        2.0     0.5

In [265]: result.stack()
Out[265]:
a  metrics
0  d_sum    2.0
   c_mean   0.5
1  d_sum    2.0
   c_mean   0.5
2  d_sum    2.0
   c_mean   0.5
dtype: float64
```

Previous
pandas documentation

© 2023 pandas via NumFOCUS, Inc. Hosted by OVHcloud.
Created using Sphinx 5.3.0.