Group by: split-apply-combine

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

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

Out of these, the split step is the most straightforward. In the apply step, we might wish to do one of the following:

- **Aggregation**: compute a summary statistic (or statistics) for each group. Some examples:

    - Compute group sums or means.
    - Compute group sizes / counts.

- **Transformation**: perform some group-specific computations and return a like-indexed object. Some examples:

    - Standardize data (zscore) within a group.
    - Filling NAs within groups with a value derived from each group.

- **Filtration**: discard some groups, according to a group-wise computation that evaluates to True or False. Some examples:

    - Discard data that belong to groups with only a few members.
    - Filter out data based on the group sum or mean.

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

Skip to main content

It is possible that a given operation does not fall into one of these categories or is some combination of them. In such a case, it may be possible to compute the operation using GroupBy's `apply` method. This method will examine the results of the apply step and try to sensibly combine them into a single result if it doesn't fit into either of the above three categories.

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.

The name GroupBy should be quite familiar to those who have used a SQL-based tool (or `itertools`), in which you can write code like:

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

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

See the cookbook for some advanced strategies.

Splitting an object into groups

The abstract definition of grouping is to provide a mapping of labels to group names. To create a GroupBy object (more on what the GroupBy object is later), you may do the following:

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

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

In [3]: grouped = speeds.groupby("class")

In [4]: grouped = speeds.groupby(["class", "order"])
```

The mapping can be specified many different ways:

- A Python function, to be called on each of the index labels.
- A list or NumPy array of the same length as the index.
- A dict or `Series`, providing a `label -> group name` mapping.
- For `DataFrame` objects, a string indicating either a column name or an index level name to be used to group.
- A list of any of the above things.

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

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

```
In [5]: df = pd.DataFrame(
   ...:     {
   ...:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
   ...:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
   ...:         "C": np.random.randn(8),
   ...:         "D": np.random.randn(8),
   ...:     }
   ...: )
   ...:

In [6]: df
Out[6]:
     A      B         C         D
0  foo    one  0.469112 -0.861849
```

```
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

On a DataFrame, we obtain a GroupBy object by calling `groupby()`. This method returns a `pandas.api.typing.DataFrameGroupBy` instance. We could naturally group by either the `A` or `B` columns, or both:

```
In [7]: grouped = df.groupby("A")

In [8]: grouped = df.groupby("B")

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

Note

`df.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`.

If we also have a MultiIndex on columns `A` and `B`, we can group by all the columns except the one we specify:

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

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

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

The above GroupBy will split the DataFrame on its index (rows). To split by columns, first do a transpose:

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

Skip to main content

```
In [14]: grouped = df.T.groupby(get_letter_type)
```

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

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

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

In [17]: s
Out[17]:
1     1
2     2
3     3
1    10
2    20
3    30
dtype: int64

In [18]: grouped = s.groupby(level=0)

In [19]: grouped.first()
Out[19]:
1    1
2    2
3    3
dtype: int64

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

In [21]: grouped.sum()
Out[21]:
1    11
2    22
3    33
dtype: int64
```

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

Note

Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though it can't be guaranteed to be the most efficient implementation). 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. With `sort=False` the order among group-keys follows the order of appearance of the keys in the original dataframe:

```
In [22]: df2 = pd.DataFrame({"X": ["B", "B", "A", "A"], "Y": [1, 2, 3, 4]})

In [23]: df2.groupby(["X"]).sum()
Out[23]:
   Y
X
A  7
B  3

In [24]: df2.groupby(["X"], sort=False).sum()
Out[24]:
   Y
X
B  3
A  7
```

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

```
In [25]: df3 = pd.DataFrame({"X": ["A", "B", "A", "B"], "Y": [1, 4, 3, 2]})

In [26]: df3.groupby("X").get_group("A")
Out[26]:
   X  Y
0  A  1
2  A  3

In [27]: df3.groupby(["X"]).get_group(("B",))
Out[27]:
   X  Y
```

Skip to main content

```
1  B  4
3  B  2
```

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 [28]: df_list = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]

In [29]: df_dropna = pd.DataFrame(df_list, columns=["a", "b", "c"])

In [30]: df_dropna
Out[30]: 
   a    b  c
0  1  2.0  3
1  1  NaN  4
2  2  1.0  3
3  1  2.0  2
```

```
# Default ``dropna`` is set to True, which will exclude NaNs in keys
In [31]: df_dropna.groupby(by=["b"], dropna=True).sum()
Out[31]: 
     a  c
b        
1.0  2  3
2.0  2  5

# In order to allow NaN in keys, set ``dropna`` to False
In [32]: df_dropna.groupby(by=["b"], dropna=False).sum()
Out[32]: 
     a  c
b        
1.0  2  3
2.0  2  5
NaN  1  4
```

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 dictionary whose keys are the computed unique groups and corresponding values are the axis labels belonging to each group. In the above example we have:

```
In [33]: df.groupby("A").groups
Out[33]: {'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}

In [34]: df.T.groupby(get_letter_type).groups
Out[34]: {'consonant': ['B', 'C', 'D'], 'vowel': ['A']}
```

Calling the standard Python `len` function on the GroupBy object returns the number of groups, which is the same as the length of the `groups` dictionary:

```
In [35]: grouped = df.groupby(["A", "B"])

In [36]: grouped.groups
Out[36]: {('bar', 'one'): [1], ('bar', 'three'): [3], ('bar', 'two'): [5], ('foo',

In [37]: len(grouped)
Out[37]: 6
```

`GroupBy` will tab complete column names, GroupBy operations, and other attributes:

```
In [38]: n = 10

In [39]: weight = np.random.normal(166, 20, size=n)

In [40]: height = np.random.normal(60, 10, size=n)

In [41]: time = pd.date_range("1/1/2000", periods=n)

In [42]: gender = np.random.choice(["male", "female"], size=n)

In [43]: df = pd.DataFrame(
   ....:     {"height": height, "weight": weight, "gender": gender}, index=time
   ....: )
   ....:

In [44]: df
Out[44]:
               height      weight  gender
2000-01-01  42.849980  157.500553    male
2000-01-02  49.607315  177.340407    male
2000-01-03  56.293531  171.524640    male
2000-01-04  48.421077  144.251986  female
2000-01-05  46.556882  152.526206    male
2000-01-06  68.448851  168.272968  female
```

Skip to main content

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

In [45]: gb = df.groupby("gender")
```

```
In [46]: gb.<TAB>  # noqa: E225, E999
gb.agg        gb.boxplot    gb.cummin     gb.describe   gb.filter     gb.get_group
gb.aggregate  gb.count      gb.cumprod    gb.dtype      gb.first      gb.groups
gb.apply      gb.cummax     gb.cumsum     gb.fillna     gb.gender     gb.head
```

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 [47]: arrays = [
   ....:     ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ....:     ["one", "two", "one", "two", "one", "two", "one", "two"],
   ....: ]
   ....:

In [48]: index = pd.MultiIndex.from_arrays(arrays, names=["first", "second"])

In [49]: s = pd.Series(np.random.randn(8), index=index)

In [50]: s
Out[50]:
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 [51]: grouped = s.groupby(level=0)

In [52]: grouped.sum()
```

Skip to main content

```
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 [53]: s.groupby(level="second").sum()
Out[53]:
second
one    0.980950
two    1.991575
dtype: float64
```

Grouping with multiple levels is supported.

```
In [54]: arrays = [
   ....:     ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ....:     ["doo", "doo", "bee", "bee", "bop", "bop", "bop", "bop"],
   ....:     ["one", "two", "one", "two", "one", "two", "one", "two"],
   ....: ]
   ....:

In [55]: index = pd.MultiIndex.from_arrays(arrays, names=["first", "second", "thir

In [56]: s = pd.Series(np.random.randn(8), index=index)

In [57]: s
Out[57]:
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 [58]: s.groupby(level=["first", "second"]).sum()
Out[58]:
first  second
bar    doo      -1.220674
baz    bee      -0.608004
foo    bop       1.023898
qux    bop       0.000895
dtype: float64
```

Skip to main content

Index level names may be supplied as keys.

```
In [59]: s.groupby(["first", "second"]).sum()
Out[59]:
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. You can specify both column and index names, or use a `Grouper`.

Let's first create a DataFrame with a MultiIndex:

```
In [60]: arrays = [
   ....:     ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ....:     ["one", "two", "one", "two", "one", "two", "one", "two"],
   ....: ]
   ....:

In [61]: index = pd.MultiIndex.from_arrays(arrays, names=["first", "second"])

In [62]: df = pd.DataFrame({"A": [1, 1, 1, 1, 2, 2, 3, 3], "B": np.arange(8)}, ind

In [63]: df
Out[63]:
              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
```

Then we group `df` by the `second` index level and the `A` column.

Skip to main content

```
In [64]: df.groupby([pd.Grouper(level=1), "A"]).sum()
Out[64]: 
          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 [65]: df.groupby([pd.Grouper(level="second"), "A"]).sum()
Out[65]: 
          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 [66]: df.groupby(["second", "A"]).sum()
Out[66]: 
          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, by using `[]` on the GroupBy object in a similar way as the one used to get a column from a DataFrame, you can do:

```
In [67]: df = pd.DataFrame(
   ....:     {
   ....:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
   ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"]
   ....:         "C": np.random.randn(8),
   ....:         "D": np.random.randn(8),
   ....:     }
   ....: )
   ....:

In [68]: df
Out[68]:
     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

In [69]: grouped = df.groupby(["A"])

In [70]: grouped_C = grouped["C"]

In [71]: grouped_D = grouped["D"]
```

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

```
In [72]: df["C"].groupby(df["A"])
Out[72]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7f10570765f0>
```

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

You can also include the grouping columns if you want to operate on them.

```
In [73]: grouped[["A", "B"]].sum()
Out[73]:
                   A                  B
A
bar        barbarbar        onethreetwo
foo  foofoofoofoofoo  onetwotwoonethree
```

Skip to main content

Iterating through groups

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

```
In [74]: grouped = df.groupby('A')

In [75]: 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 [76]: 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
```

Skip to main content

```
2  foo  two -1.143704  1.627081
4  foo  two  1.193555 -0.441652
```

See Iterating through groups.

Selecting a group

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

```
In [77]: grouped.get_group("bar")
Out[77]:
     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 [78]: df.groupby(["A", "B"]).get_group(("bar", "one"))
Out[78]:
     A    B         C         D
1  bar  one  0.254161  1.511763
```

Aggregation

An aggregation is a GroupBy operation that reduces the dimension of the grouping object. The result of an aggregation is, or at least is treated as, a scalar value for each column in a group. For example, producing the sum of each column in a group of values.

```
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]:
```

Skip to main content

```
1  dog     6.0     7.5
2  cat     9.5     9.9
3  dog    34.0   198.0

In [81]: animals.groupby("kind").sum()
Out[81]:
      height  weight
kind
cat     18.6    17.8
dog     40.0   205.5
```

In the result, the keys of the groups appear in the index by default. They can be instead included in the columns by passing `as_index=False`.

```
In [82]: animals.groupby("kind", as_index=False).sum()
Out[82]:
  kind  height  weight
0  cat    18.6    17.8
1  dog    40.0   205.5
```

Built-in aggregation methods

Many common aggregations are built-in to GroupBy objects as methods. Of the methods listed below, those with a `*` do *not* have an efficient, GroupBy-specific, 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

Skip to main content

Method	Description
`max()`	Compute the maximum 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()`	Compute a given quantile of the values in each group
`sem()`	Compute the standard error of the mean of the values in each group
`size()`	Compute the number of values in each group
`skew()` *	Compute the skew of the values in each group
`std()`	Compute the standard deviation of the 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 [83]: df.groupby("A")[["C", "D"]].max()
Out[83]:
            C         D
A
bar  0.254161  1.511763
foo  1.193555  1.627081

In [84]: df.groupby(["A", "B"]).mean()
Out[84]:
                  C         D
A   B
bar one    0.254161  1.511763
    three  0.215897 -0.990582
    two   -0.077118  1.211526
foo one   -0.491888  0.807291
```

Skip to main content

```
    three -0.862495  0.024580
    two    0.024925  0.592714
```

Another 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 consists of the group names and the values are the sizes of each group.

```
In [85]: grouped = df.groupby(["A", "B"])

In [86]: grouped.size()
Out[86]:
A    B
bar  one      1
     three    1
     two      1
foo  one      2
     three    1
     two      2
dtype: int64
```

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

```
In [87]: grouped.describe()
Out[87]:
              C                         ...         D
          count      mean       std     ...       50%       75%       max
A   B                                   ...
bar one     1.0  0.254161       NaN     ...  1.511763  1.511763  1.511763
    three   1.0  0.215897       NaN     ... -0.990582 -0.990582 -0.990582
    two     1.0 -0.077118       NaN     ...  1.211526  1.211526  1.211526
foo one     2.0 -0.491888  0.117887     ...  0.807291  1.076676  1.346061
    three   1.0 -0.862495       NaN     ...  0.024580  0.024580  0.024580
    two     2.0  0.024925  1.652692     ...  0.592714  1.109898  1.627081

[6 rows x 16 columns]
```

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

```
In [88]: ll = [['foo', 1], ['foo', 2], ['foo', 2], ['bar', 1], ['bar', 1]]

In [89]: df4 = pd.DataFrame(ll, columns=["A", "B"])
```

Skip to main content

```
Out[90]:
     A  B
0  foo  1
1  foo  2
2  foo  2
3  bar  1
4  bar  1

In [91]: df4.groupby("A")["B"].nunique()
Out[91]:
A
bar    1
foo    2
Name: B, dtype: int64
```

Note

Aggregation functions **will not** return the groups that you are aggregating over as 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 as named columns, regardless if they are named **indices** or *columns* in the inputs.

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 shorthand, `agg`. It will operate as if the corresponding method was called.

```
In [92]: grouped = df.groupby("A")

In [93]: grouped[["C", "D"]].aggregate("sum")
Out[93]:
```

Skip to main content

```
bar  0.392940  1.732707
foo -1.796421  2.824590

In [94]: grouped = df.groupby(["A", "B"])

In [95]: grouped.agg("sum")
Out[95]: 
                  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
```

The result of the aggregation will have the group names as the new index. In the case of multiple keys, the result is a MultiIndex by default. As mentioned above, this can be changed by using the `as_index` option:

```
In [96]: grouped = df.groupby(["A", "B"], as_index=False)

In [97]: grouped.agg("sum")
Out[97]: 
     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 [98]: df.groupby("A", as_index=False)[["C", "D"]].agg("sum")
Out[98]: 
     A         C         D
0  bar  0.392940  1.732707
1  foo -1.796421  2.824590
```

Note that you could use the `DataFrame.reset_index()` DataFrame function to achieve the same result as the column names are stored in the resulting `MultiIndex`, although this will make an extra copy.

```
In [99]: df.groupby(["A", "B"]).agg("sum").reset_index()
Out[99]: 
     A      B         C         D
0  bar    one  0.254161  1.511763
1  bar  three  0.215897 -0.990582
```

Skip to main content

```
4  foo  three -0.862495  0.024580
5  foo    two  0.049851  1.185429
```

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 Mutating with User Defined Function (UDF) methods for more information.

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 [100]: animals
Out[100]:
  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 [101]: animals.groupby("kind")[["height"]].agg(lambda x: set(x))
Out[101]:
           height
kind
cat    {9.1, 9.5}
dog   {34.0, 6.0}
```

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 [102]: animals.groupby("kind")[["height"]].agg(lambda x: x.astype(int).sum())
Out[102]:
```

Skip to main content

```
cat        18
dog        40
```

Applying multiple functions at once

On a grouped `Series`, you can pass a list or dict of functions to **`SeriesGroupBy.agg()`**, outputting a DataFrame:

```
In [103]: grouped = df.groupby("A")

In [104]: grouped["C"].agg(["sum", "mean", "std"])
Out[104]: 
          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 **`DataFrameGroupBy.agg()`** to aggregate each column, which produces an aggregated result with a hierarchical column index:

```
In [105]: grouped[["C", "D"]].agg(["sum", "mean", "std"])
Out[105]: 
            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 after the functions themselves. If you need to rename, then you can add in a chained operation for a `Series` like this:

```
In [106]: (
   .....:     grouped["C"]
   .....:     .agg(["sum", "mean", "std"])
   .....:     .rename(columns={"sum": "foo", "mean": "bar", "std": "baz"})
   .....: )
   .....: 
Out[106]: 
          foo       bar       baz
A                                
bar  0.392940  0.130980  0.181231
foo -1.796421 -0.359284  0.912265
```

Skip to main content

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

```
In [107]: (
   .....:     grouped[["C", "D"]].agg(["sum", "mean", "std"]).rename(
   .....:         columns={"sum": "foo", "mean": "bar", "std": "baz"}
   .....:     )
   .....: )
   .....:
Out[107]:
            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, but pandas will allow you apply to the same function (or two functions with the same name) to the same column.

```
In [108]: grouped["C"].agg(["sum", "sum"])
Out[108]:
          sum       sum
A
bar  0.392940  0.392940
foo -1.796421 -1.796421
```

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 [109]: grouped["C"].agg([lambda x: x.max() - x.min(), lambda x: x.media
Out[109]:
     <lambda_0>  <lambda_1>
A
bar    0.331279    0.084917
foo    2.337259   -0.215962
```

Named aggregation

Skip to main content

"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 [110]: animals
Out[110]:
  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 [111]: 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="mean"),
   .....: )
   .....:
Out[111]:
      min_height  max_height  average_weight
kind
cat          9.1         9.5            8.90
dog          6.0        34.0          102.75
```

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

```
In [112]: animals.groupby("kind").agg(
   .....:     min_height=("height", "min"),
   .....:     max_height=("height", "max"),
   .....:     average_weight=("weight", "mean"),
   .....: )
   .....:
Out[112]:
      min_height  max_height  average_weight
kind
cat          9.1         9.5            8.90
dog          6.0        34.0          102.75
```

If the column names you want are not valid Python keywords, construct a dictionary and unpack the keyword arguments

```
In [113]: animals.groupby("kind").agg(
   .....:     **{
   .....:         "total weight": pd.NamedAgg(column="weight", aggfunc="sum")
   .....:     }
   .....: )
   .....:
Out[113]:
      total weight
kind
cat           17.8
dog          205.5
```

When using named aggregation, 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 require additional arguments, apply them partially with `functools.partial()`.

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 [114]: animals.groupby("kind").height.agg(
   .....:     min_height="min",
   .....:     max_height="max",
   .....: )
   .....:
Out[114]:
      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 [115]: grouped.agg({"C": "sum", "D": lambda x: np.std(x, ddof=1)})
Out[115]:
            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 implemented on GroupBy:

```
In [116]: grouped.agg({"C": "sum", "D": "std"})
Out[116]:
            C         D
A
bar  0.392940  1.366330
foo -1.796421  0.884785
```

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 [117]: speeds
Out[117]:
          class           order  max_speed
falcon     bird   Falconiformes      389.0
parrot     bird  Psittaciformes       24.0
lion     mammal       Carnivora       80.2
monkey   mammal        Primates        NaN
leopard  mammal       Carnivora       58.0

In [118]: grouped = speeds.groupby("class")["max_speed"]

In [119]: grouped.cumsum()
Out[119]:
falcon     389.0
parrot     413.0
lion        80.2
monkey       NaN
leopard    138.2
Name: max_speed, dtype: float64

In [120]: grouped.diff()
Out[120]:
falcon       NaN
parrot    -365.0
lion         NaN
monkey       NaN
leopard      NaN
Name: max_speed, dtype: float64
```

Unlike aggregations, the groupings that are used to split the original object are not included in the

Note

Since transformations do not include the groupings that are used to split the result, the arguments `as_index` and `sort` in `DataFrame.groupby()` and `Series.groupby()` have no effect.

A common use of a transformation is to add the result back into the original DataFrame.

```
In [121]: result = speeds.copy()

In [122]: result["cumsum"] = grouped.cumsum()

In [123]: result["diff"] = grouped.diff()

In [124]: result
Out[124]: 
          class           order  max_speed  cumsum   diff
falcon     bird   Falconiformes      389.0   389.0    NaN
parrot     bird  Psittaciformes       24.0   413.0 -365.0
lion     mammal       Carnivora       80.2    80.2    NaN
monkey   mammal        Primates        NaN     NaN    NaN
leopard  mammal       Carnivora       58.0   138.2    NaN
```

Built-in transformation methods

The following methods on GroupBy act as transformations.

Method	Description
`bfill()`	Back fill NA values within each group
`cumcount()`	Compute the cumulative count within each group
`cummax()`	Compute the cumulative max within each group
`cummin()`	Compute the cumulative min within each group
`cumprod()`	Compute the cumulative product within each group
`cumsum()`	Compute the cumulative sum within each group

Skip to main content

Method	Description
`ffill()`	Forward fill NA values within each group
`pct_change()`	Compute the percent change between adjacent values within each group
`rank()`	Compute the rank of each value within each group
`shift()`	Shift values up or down within each group

In addition, passing any built-in aggregation method as a string to `transform()` (see the next section) will broadcast the result across the group, producing a transformed result. If the aggregation method has an efficient implementation, this will be performant as well.

The `transform()` method

Similar to the aggregation method, the `transform()` method can accept string aliases to the built-in transformation methods in the previous section. It can *also* accept string aliases to the built-in aggregation methods. When an aggregation method is provided, the result will be broadcast across the group.

```
In [125]: speeds
Out[125]:
          class           order  max_speed
falcon     bird   Falconiformes      389.0
parrot     bird  Psittaciformes       24.0
lion     mammal       Carnivora       80.2
monkey   mammal        Primates        NaN
leopard  mammal       Carnivora       58.0

In [126]: grouped = speeds.groupby("class")[["max_speed"]]

In [127]: grouped.transform("cumsum")
Out[127]:
         max_speed
falcon       389.0
parrot       413.0
lion          80.2
monkey         NaN
leopard      138.2

In [128]: grouped.transform("sum")
Out[128]:
         max_speed
```

Skip to main content

```
parrot      413.0
lion        138.2
monkey      138.2
leopard     138.2
```

In addition to string aliases, the `transform()` method can also accept User-Defined Functions (UDFs). The UDF 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. See Mutating with User Defined Function (UDF) methods for more information.
- (Optionally) operates on all columns of the entire group chunk at once. If this is supported, a fast path is used starting from the *second* chunk.

Note

Transforming by supplying `transform` with a UDF is often less performant than using the built-in methods on GroupBy. Consider breaking up a complex operation into a chain of operations that utilize the built-in methods.

All of the examples in this section can be made more performant by calling built-in methods instead of using UDFs. See below for examples.

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 aggregate() method, 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 wish to standardize the data within each group:

Skip to main content

```
In [129]: index = pd.date_range("10/1/1999", periods=1100)

In [130]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)

In [131]: ts = ts.rolling(window=100, min_periods=100).mean().dropna()

In [132]: ts.head()
Out[132]: 
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 [133]: ts.tail()
Out[133]: 
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 [134]: transformed = ts.groupby(lambda x: x.year).transform(
   .....:     lambda x: (x - x.mean()) / x.std()
   .....: )
   .....: 
```

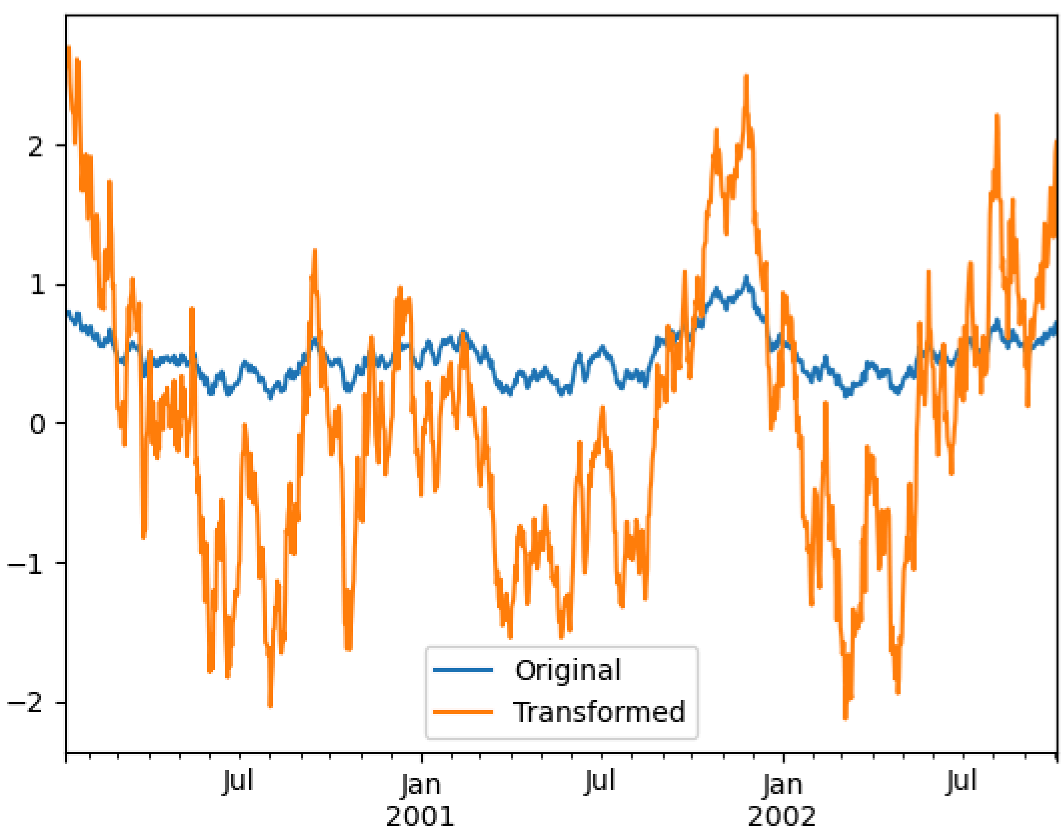
We would expect the result to now have mean 0 and standard deviation 1 within each group (up to floating-point error), which we can easily check:

```
# Original Data
In [135]: grouped = ts.groupby(lambda x: x.year)

In [136]: grouped.mean()
Out[136]: 
2000    0.442441
2001    0.526246
2002    0.459365
dtype: float64

In [137]: grouped.std()
Out[137]: 
2000    0.131752
2001    0.210945
2002    0.128753
dtype: float64

# Transformed Data
```

```
In [139]: grouped_trans.mean()
Out[139]:
2000   -4.870756e-16
2001   -1.545187e-16
2002    4.136282e-16
dtype: float64

In [140]: grouped_trans.std()
Out[140]:
2000    1.0
2001    1.0
2002    1.0
dtype: float64
```

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

```
In [141]: compare = pd.DataFrame({"Original": ts, "Transformed": transformed})

In [142]: compare.plot()
Out[142]: <Axes: >
```



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

Skip to main content

```
In [143]: ts.groupby(lambda x: x.year).transform(lambda x: x.max() - x.min())
Out[143]: 
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 [144]: cols = ["A", "B", "C"]

In [145]: values = np.random.randn(1000, 3)

In [146]: values[np.random.randint(0, 1000, 100), 0] = np.nan

In [147]: values[np.random.randint(0, 1000, 50), 1] = np.nan

In [148]: values[np.random.randint(0, 1000, 200), 2] = np.nan

In [149]: data_df = pd.DataFrame(values, columns=cols)

In [150]: data_df
Out[150]: 
            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 [151]: countries = np.array(["US", "UK", "GR", "JP"])

In [152]: key = countries[np.random.randint(0, 4, 1000)]
```

Skip to main content

```
# Non-NA count in each group
In [154]: grouped.count()
Out[154]:
      A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [155]: 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 [156]: grouped_trans = transformed.groupby(key)

In [157]: grouped.mean()  # original group means
Out[157]:
           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 [158]: grouped_trans.mean()  # transformation did not change group means
Out[158]:
           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 [159]: grouped.count()  # original has some missing data points
Out[159]:
      A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [160]: grouped_trans.count()  # counts after transformation
Out[160]:
      A    B    C
GR  228  228  228
JP  267  267  267
UK  247  247  247
US  258  258  258

In [161]: grouped_trans.size()  # Verify non-NA count equals group size
Out[161]:
GR    228
```

```
US    258
dtype: int64
```

As mentioned in the note above, each of the examples in this section can be computed more efficiently using built-in methods. In the code below, the inefficient way using a UDF is commented out and the faster alternative appears below.

```
# result = ts.groupby(lambda x: x.year).transform(
#     lambda x: (x - x.mean()) / x.std()
# )
In [162]: grouped = ts.groupby(lambda x: x.year)

In [163]: result = (ts - grouped.transform("mean")) / grouped.transform("std")

# result = ts.groupby(lambda x: x.year).transform(lambda x: x.max() - x.min())
In [164]: grouped = ts.groupby(lambda x: x.year)

In [165]: result = grouped.transform("max") - grouped.transform("min")

# grouped = data_df.groupby(key)
# result = grouped.transform(lambda x: x.fillna(x.mean()))
In [166]: grouped = data_df.groupby(key)

In [167]: result = data_df.fillna(grouped.transform("mean"))
```

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 [168]: df_re = pd.DataFrame({"A": [1] * 10 + [5] * 10, "B": np.arange(20)})

In [169]: df_re
Out[169]:
    A   B
0   1   0
1   1   1
2   1   2
3   1   3
4   1   4
.. ..  ..
15  5  15
```

Skip to main content

```
18  5  18
19  5  19

[20 rows x 2 columns]

In [170]: df_re.groupby("A").rolling(4).B.mean()
Out[170]:
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 [171]: df_re.groupby("A").expanding().sum()
Out[171]:
          B
A
1 0     0.0
  1     1.0
  2     3.0
  3     6.0
  4    10.0
...     ...
5 15   75.0
  16   91.0
  17  108.0
  18  126.0
  19  145.0

[20 rows x 1 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 [172]: df_re = pd.DataFrame(
   .....:     {
   .....:         "date": pd.date_range(start="2016-01-01", periods=4, freq="W"),
```

Skip to main content

```
   .....:     }
   .....: ).set_index("date")
   .....:

In [173]: df_re
Out[173]:
            group  val
date
2016-01-03      1    5
2016-01-10      1    6
2016-01-17      2    7
2016-01-24      2    8

In [174]: df_re.groupby("group").resample("1D", include_groups=False).ffill()
Out[174]:
                  val
group date
1     2016-01-03    5
      2016-01-04    5
      2016-01-05    5
      2016-01-06    5
      2016-01-07    5
...               ...
2     2016-01-20    7
      2016-01-21    7
      2016-01-22    7
      2016-01-23    7
      2016-01-24    8

[16 rows x 1 columns]
```

Filtration

A filtration is a GroupBy operation that subsets the original grouping object. It may either filter out entire groups, part of groups, or both. Filtrations return a filtered version of the calling object, including the grouping columns when provided. In the following example, `class` is included in the result.

```
In [175]: speeds
Out[175]:
          class           order  max_speed
falcon     bird   Falconiformes      389.0
parrot     bird  Psittaciformes       24.0
lion     mammal       Carnivora       80.2
monkey   mammal        Primates        NaN
leopard  mammal       Carnivora       58.0
```

```
         class           order  max_speed
parrot    bird  Psittaciformes       24.0
monkey  mammal        Primates        NaN
```

Note

Unlike aggregations, filtrations do not add the group keys to the index of the result. Because of this, passing `as_index=False` or `sort=True` will not affect these methods.

Filtrations will respect subsetting the columns of the GroupBy object.

```
In [177]: speeds.groupby("class")[["order", "max_speed"]].nth(1)
Out[177]:
                 order  max_speed
parrot  Psittaciformes       24.0
monkey        Primates        NaN
```

Built-in filtrations

The following methods on GroupBy act as filtrations. All these methods have an efficient, GroupBy-specific, implementation.

Method	Description
`head()`	Select the top row(s) of each group
`nth()`	Select the nth row(s) of each group
`tail()`	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 given groups of products and their volumes, and we wish to subset the data to only the largest products capturing no more than 90% of the total volume within each group.

```
In [178]: product_volumes = pd.DataFrame(
   .....:     {
   .....:         "group": list("xxxxyyy"),
```

Skip to main content

```
   .....:     }
   .....: )
   .....:

In [179]: product_volumes
Out[179]: 
  group product  volume
0     x       a      10
1     x       b      30
2     x       c      20
3     x       d      15
4     y       e      40
5     y       f      10
6     y       g      20

# Sort by volume to select the largest products first
In [180]: product_volumes = product_volumes.sort_values("volume", ascending=False)

In [181]: grouped = product_volumes.groupby("group")["volume"]

In [182]: cumpct = grouped.cumsum() / grouped.transform("sum")

In [183]: cumpct
Out[183]: 
4    0.571429
1    0.400000
2    0.666667
6    0.857143
3    0.866667
0    1.000000
5    1.000000
Name: volume, dtype: float64

In [184]: significant_products = product_volumes[cumpct <= 0.9]

In [185]: significant_products.sort_values(["group", "product"])
Out[185]: 
  group product  volume
1     x       b      30
2     x       c      20
3     x       d      15
4     y       e      40
6     y       g      20
```

The `filter` method

Skip to main content

Note

Filtering by supplying `filter` with a User-Defined Function (UDF) is often less performant than using the built-in methods on GroupBy. Consider breaking up a 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 elements that belong to groups with a group sum greater than 2.

```
In [186]: sf = pd.Series([1, 1, 2, 3, 3, 3])

In [187]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[187]:
3    3
4    3
5    3
dtype: int64
```

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

```
In [188]: dff = pd.DataFrame({"A": np.arange(8), "B": list("aabbbbcc")})

In [189]: dff.groupby("B").filter(lambda x: len(x) > 2)
Out[189]:
   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 [190]: dff.groupby("B").filter(lambda x: len(x) > 2, dropna=False)
Out[190]:
     A    B
0  NaN  NaN
1  NaN  NaN
```

Skip to main content

```
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 [191]: dff["C"] = np.arange(8)

In [192]: dff.groupby("B").filter(lambda x: len(x["C"]) > 2)
Out[192]: 
   A  B  C
2  2  b  2
3  3  b  3
4  4  b  4
5  5  b  5
```

Flexible `apply`

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

Warning

`apply` has to try to infer from the result whether it should act as a reducer, transformer, *or* filter, depending on exactly what is passed to it. Thus the grouped column(s) may be included in the output or not. 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 [193]: df
Out[193]: 
```

Skip to main content

```
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 [194]: grouped = df.groupby("A")

# could also just call .describe()
In [195]: grouped["C"].apply(lambda x: x.describe())
Out[195]:
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 [196]: grouped = df.groupby('A')['C']

In [197]: def f(group):
   .....:     return pd.DataFrame({'original': group,
   .....:                          'demeaned': group - group.mean()})
   .....:

In [198]: grouped.apply(f)
Out[198]:
       original  demeaned
A
bar 1  0.254161  0.123181
    3  0.215897  0.084917
    5 -0.077118 -0.208098
foo 0 -0.575247 -0.215962
    2 -1.143704 -0.784420
    4  1.193555  1.552839
    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,

```
In [199]: def f(x):
   .....:     return pd.Series([x, x ** 2], index=["x", "x^2"])
   .....:

In [200]: s = pd.Series(np.random.rand(5))

In [201]: s
Out[201]:
0    0.582898
1    0.098352
2    0.001438
3    0.009420
4    0.815826
dtype: float64

In [202]: s.apply(f)
Out[202]:
          x       x^2
0  0.582898  0.339770
1  0.098352  0.009673
2  0.001438  0.000002
3  0.009420  0.000089
4  0.815826  0.665572
```

Similar to The aggregate() method, the resulting dtype will reflect that of the apply function. If the results from different groups have different dtypes, then a common dtype will be determined in the same way as `DataFrame` construction.

Control grouped column(s) placement with `group_keys`

To control whether the grouped column(s) are included in the indices, you can use the argument `group_keys` which defaults to `True`. Compare

```
In [203]: df.groupby("A", group_keys=True).apply(lambda x: x, include_groups=False
Out[203]:
           B         C         D
A
bar 1    one  0.254161  1.511763
    3  three  0.215897 -0.990582
    5    two -0.077118  1.211526
foo 0    one -0.575247  1.346061
    2    two -1.143704  1.627081
    4    two  1.193555 -0.441652
    6    one -0.408530  0.268520
    7  three -0.862495  0.024580
```

Skip to main content

with

```
In [204]: df.groupby("A", group_keys=False).apply(lambda x: x, include_groups=Fals
Out[204]:
       B         C         D
0    one -0.575247  1.346061
1    one  0.254161  1.511763
2    two -1.143704  1.627081
3  three  0.215897 -0.990582
4    two  1.193555 -0.441652
5    two -0.077118  1.211526
6    one -0.408530  0.268520
7  three -0.862495  0.024580
```

Numba Accelerated Routines

Added 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 start with `values, index` **exactly** as the data belonging to each group will be passed into `values`, and the group index will be passed into `index`.

Warning

When using `engine='numba'`, there will be no “fall back” behavior internally. 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 non-numeric columns

Skip to main content

```
In [205]: df
Out[205]:
     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` because it is not numeric. You can avoid non-numeric columns by specifying `numeric_only=True`:

```
In [206]: df.groupby("A").std(numeric_only=True)
Out[206]:
            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 needed over one column (here `colname`), it may be filtered *before* applying the aggregation function.

```
In [207]: from decimal import Decimal

In [208]: 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"),
   .....:         ],
   .....:     }
   .....: )
   .....:

In [209]: df_dec.groupby(["id"])[["dec_column"]].sum()
Out[209]:
   dec_column
```

Skip to main content

```
1         0.75
2         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 [210]: pd.Series([1, 1, 1]).groupby(
   .....:     pd.Categorical(["a", "a", "a"], categories=["a", "b"]), observed=Fal
   .....: ).count()
   .....:
Out[210]:
a    3
b    0
dtype: int64
```

Show only the observed values:

```
In [211]: pd.Series([1, 1, 1]).groupby(
   .....:     pd.Categorical(["a", "a", "a"], categories=["a", "b"]), observed=Tru
   .....: ).count()
   .....:
Out[211]:
a    3
dtype: int64
```

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

```
In [212]: s = (
   .....:     pd.Series([1, 1, 1])
   .....:     .groupby(pd.Categorical(["a", "a", "a"], categories=["a", "b"]), obs
   .....:     .count()
   .....: )
   .....:

In [213]: s.index.dtype
Out[213]: CategoricalDtype(categories=['a', 'b'], ordered=False, categories_dtype=
```

Skip to main content

NA group handling

By `NA`, we are referring to any `NA` values, including `NA`, `NaN`, `NaT`, and `None`. If there are any `NA` values in the grouping key, by default these will be excluded. In other words, any "`NA` group" will be dropped. You can include NA groups by specifying `dropna=False`.

```
In [214]: df = pd.DataFrame({"key": [1.0, 1.0, np.nan, 2.0, np.nan], "A": [1, 2, 3

In [215]: df
Out[215]:
   key  A
0  1.0  1
1  1.0  2
2  NaN  3
3  2.0  4
4  NaN  5

In [216]: df.groupby("key", dropna=True).sum()
Out[216]:
     A
key
1.0  3
2.0  4

In [217]: df.groupby("key", dropna=False).sum()
Out[217]:
     A
key
1.0  3
2.0  4
NaN  8
```

Grouping with ordered factors

Categorical variables represented as instances of pandas's `Categorical` class can be used as group keys. If so, the order of the levels will be preserved. When `observed=False` and `sort=False`, any unobserved categories will be at the end of the result in order.

```
In [218]: days = pd.Categorical(
   .....:     values=["Wed", "Mon", "Thu", "Mon", "Wed", "Sat"],
   .....:     categories=["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"],
   .....: )
   .....:

In [219]: data = pd.DataFrame(
```

Skip to main content

```
   .....:         "day": days,
   .....:         "workers": [3, 4, 1, 4, 2, 2],
   .....:     }
   .....: )
   .....:

In [220]: data
Out[220]:
   day  workers
0  Wed        3
1  Mon        4
2  Thu        1
3  Mon        4
4  Wed        2
5  Sat        2

In [221]: data.groupby("day", observed=False, sort=True).sum()
Out[221]:
     workers
day
Mon        8
Tue        0
Wed        5
Thu        1
Fri        0
Sat        2
Sun        0

In [222]: data.groupby("day", observed=False, sort=False).sum()
Out[222]:
     workers
day
Wed        5
Mon        8
Thu        1
Sat        2
Tue        0
Fri        0
Sun        0
```

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 [223]: import datetime

In [224]: df = pd.DataFrame(
```

Skip to main content

```
   .....:         "Buyer": "Carl Mark Carl Carl Joe Joe Joe Carl".split(),
   .....:         "Quantity": [1, 3, 5, 1, 8, 1, 9, 3],
   .....:         "Date": [
   .....:             datetime.datetime(2013, 1, 1, 13, 0),
   .....:             datetime.datetime(2013, 1, 1, 13, 5),
   .....:             datetime.datetime(2013, 10, 1, 20, 0),
   .....:             datetime.datetime(2013, 10, 2, 10, 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),
   .....:         ],
   .....:     }
   .....: )
   .....:

In [225]: df
Out[225]:
  Branch Buyer  Quantity                Date
0      A  Carl         1 2013-01-01 13:00:00
1      A  Mark         3 2013-01-01 13:05:00
2      A  Carl         5 2013-10-01 20:00:00
3      A  Carl         1 2013-10-02 10:00:00
4      A   Joe         8 2013-10-01 20:00:00
5      A   Joe         1 2013-10-02 10:00:00
6      A   Joe         9 2013-12-02 12:00:00
7      B  Carl         3 2013-12-02 14:00:00
```

Groupby a specific column with the desired frequency. This is like resampling.

```
In [226]: df.groupby([pd.Grouper(freq="1ME", key="Date"), "Buyer"])[["Quantity"]].
Out[226]:
                  Quantity
Date       Buyer
2013-01-31 Carl          1
           Mark          3
2013-10-31 Carl          6
           Joe           9
2013-12-31 Carl          3
           Joe           9
```

When `freq` is specified, the object returned by `pd.Grouper` will be an instance of `pandas.api.typing.TimeGrouper`. When there is a column and index with the same name, you can use `key` to group by the column and `level` to group by the index.

```
In [227]: df = df.set_index("Date")

In [228]: df["Date"] = df.index + pd.offsets.MonthEnd(2)
```

Skip to main content

```
Out[229]:
                     Quantity
Date       Buyer
2013-02-28 Carl             1
           Mark             3
2014-02-28 Carl             9
           Joe             18

In [230]: df.groupby([pd.Grouper(freq="6ME", level="Date"), "Buyer"])[["Quantity"]
Out[230]:
                     Quantity
Date       Buyer
2013-01-31 Carl             1
           Mark             3
2014-01-31 Carl             9
           Joe             18
```

Taking the first rows of each group

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

```
In [231]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=["A", "B"])

In [232]: df
Out[232]:
   A  B
0  1  2
1  1  4
2  5  6

In [233]: g = df.groupby("A")

In [234]: g.head(1)
Out[234]:
   A  B
0  1  2
2  5  6

In [235]: g.tail(1)
Out[235]:
   A  B
1  1  4
2  5  6
```

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

Skip to main content

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

In general this operation acts as a filtration. In certain cases it will also return one row per group, making it also a reduction. However because in general it can return zero or multiple rows per group, pandas treats it as a filtration in all cases.

```
In [236]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=["A", "B"])

In [237]: g = df.groupby("A")

In [238]: g.nth(0)
Out[238]:
   A    B
0  1  NaN
2  5  6.0

In [239]: g.nth(-1)
Out[239]:
   A    B
1  1  4.0
2  5  6.0

In [240]: g.nth(1)
Out[240]:
   A    B
1  1  4.0
```

If the nth element of a group does not exist, then no corresponding row is included in the result. In particular, if the specified `n` is larger than any group, the result will be an empty DataFrame.

```
In [241]: g.nth(5)
Out[241]:
Empty DataFrame
Columns: [A, B]
Index: []
```

If you want to select the nth not-null item, use the `dropna` kwarg. For a DataFrame this should be either `'any'` or `'all'` just like you would pass to dropna:

```
# nth(0) is the same as g.first()
```

Skip to main content

```
   A    B
1  1  4.0
2  5  6.0

In [243]: g.first()
Out[243]:
     B
A
1  4.0
5  6.0

# nth(-1) is the same as g.last()
In [244]: g.nth(-1, dropna="any")
Out[244]:
   A    B
1  1  4.0
2  5  6.0

In [245]: g.last()
Out[245]:
     B
A
1  4.0
5  6.0

In [246]: g.B.nth(0, dropna="all")
Out[246]:
1    4.0
2    6.0
Name: B, dtype: float64
```

You can also select multiple rows from each group by specifying multiple nth values as a list of ints.

```
In [247]: business_dates = pd.date_range(start="4/1/2014", end="6/30/2014", freq="

In [248]: df = pd.DataFrame(1, index=business_dates, columns=["a", "b"])

# get the first, 4th, and last date index for each month
In [249]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[249]:
            a  b
2014-04-01  1  1
2014-04-04  1  1
2014-04-30  1  1
2014-05-01  1  1
2014-05-06  1  1
2014-05-30  1  1
2014-06-02  1  1
2014-06-05  1  1
2014-06-30  1  1
```

Skip to main content

You may also use slices or lists of slices.

```
In [250]: df.groupby([df.index.year, df.index.month]).nth[1:]
Out[250]:
            a  b
2014-04-02  1  1
2014-04-03  1  1
2014-04-04  1  1
2014-04-07  1  1
2014-04-08  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

[62 rows x 2 columns]

In [251]: df.groupby([df.index.year, df.index.month]).nth[1:, :-1]
Out[251]:
            a  b
2014-04-01  1  1
2014-04-02  1  1
2014-04-03  1  1
2014-04-04  1  1
2014-04-07  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 [252]: dfg = pd.DataFrame(list("aaabba"), columns=["A"])

In [253]: dfg
Out[253]:
   A
0  a
1  a
```

Skip to main content

```
4  b
5  a

In [254]: dfg.groupby("A").cumcount()
Out[254]:
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64

In [255]: dfg.groupby("A").cumcount(ascending=False)
Out[255]:
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 `DataFrameGroupBy.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 [256]: dfg = pd.DataFrame(list("aaabba"), columns=["A"])

In [257]: dfg
Out[257]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [258]: dfg.groupby("A").ngroup()
Out[258]:
0    0
1    0
```

```
4    1
5    0
dtype: int64

In [259]: dfg.groupby("A").ngroup(ascending=False)
Out[259]:
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
```

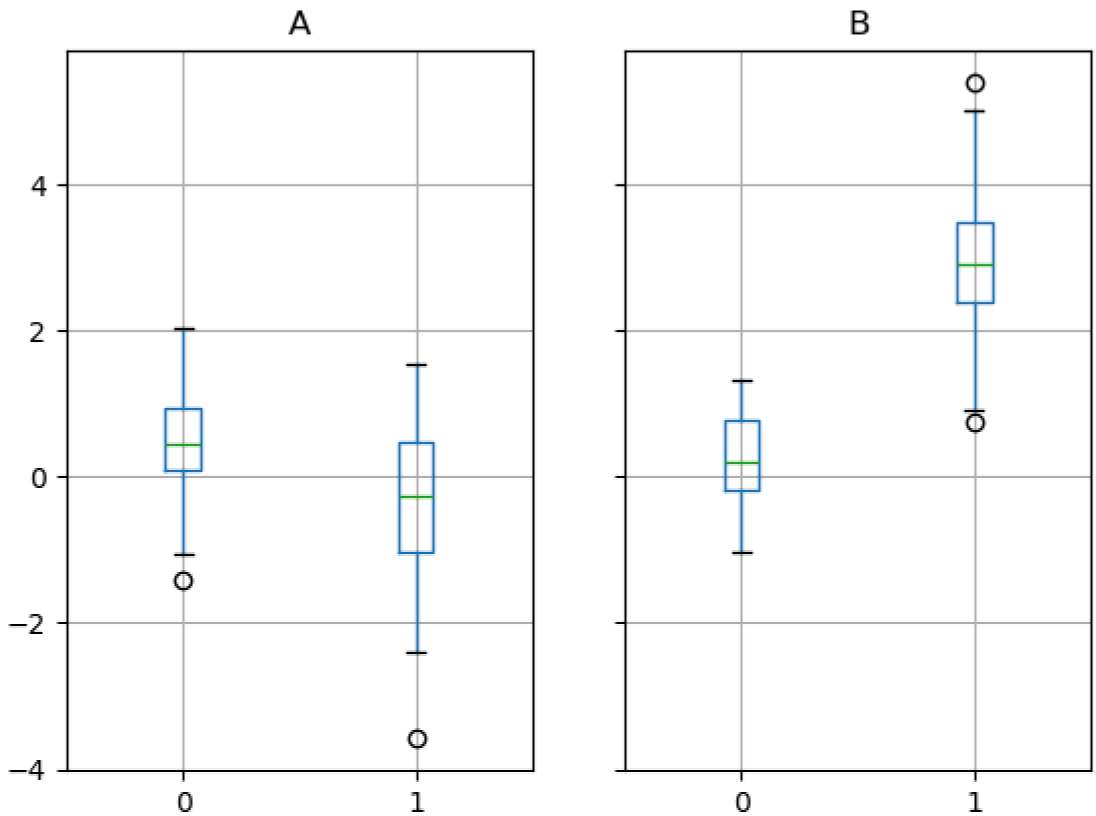
Plotting

Groupby also works with some plotting methods. In this case, suppose we suspect that the values in column 1 are 3 times higher on average in group "B".

```
In [260]: np.random.seed(1234)

In [261]: df = pd.DataFrame(np.random.randn(50, 2))

In [262]: df["g"] = np.random.choice(["A", "B"], size=50)

In [263]: df.loc[df["g"] == "B", 1] += 3
```

We can easily visualize this with a boxplot:

```
In [264]: df.groupby("g").boxplot()
Out[264]:
A         Axes(0.1,0.15;0.363636x0.75)
B    Axes(0.536364,0.15;0.363636x0.75)
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.

Skip to main content

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 [265]: n = 1000

In [266]: 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 [267]: df.head(2)
Out[267]:
     Store    Product  Revenue  Quantity
0  Store_2  Product_1    26.12         1
1  Store_2  Product_1    28.86         1
```

We now find the prices per store/product.

```
In [268]: (
   .....:     df.groupby(["Store", "Product"])
   .....:     .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
   .....:     .unstack()
   .....:     .round(2)
   .....: )
   .....:
Out[268]:
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 [269]: def mean(groupby):
   .....:     return groupby.mean()
   .....:
```

Skip to main content

```
                        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
```

Here `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

Multi-column factorization

By using `DataFrameGroupBy.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 [271]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba")})

In [272]: dfg
Out[272]:
   A  B
0  1  a
1  1  a
2  2  a
3  3  b
4  2  a

In [273]: dfg.groupby(["A", "B"]).ngroup()
Out[273]:
0    0
1    0
2    1
3    2
```

Skip to main content

```
In [274]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
Out[274]:
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 for resample to work on indices that are non-datetimelike, the following procedure can be utilized.

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

Note

The example below shows how we can downsample by consolidation of samples into fewer ones. 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 [275]: df = pd.DataFrame(np.random.randn(10, 2))

In [276]: df
Out[276]:
          0         1
```