Lecture 7 - Data Aggregation and Group Operations

Data aggregation is the compiling of information from databases with intent to prepare combined datasets for data processing.Data aggregation essentially consists of 2 steps:

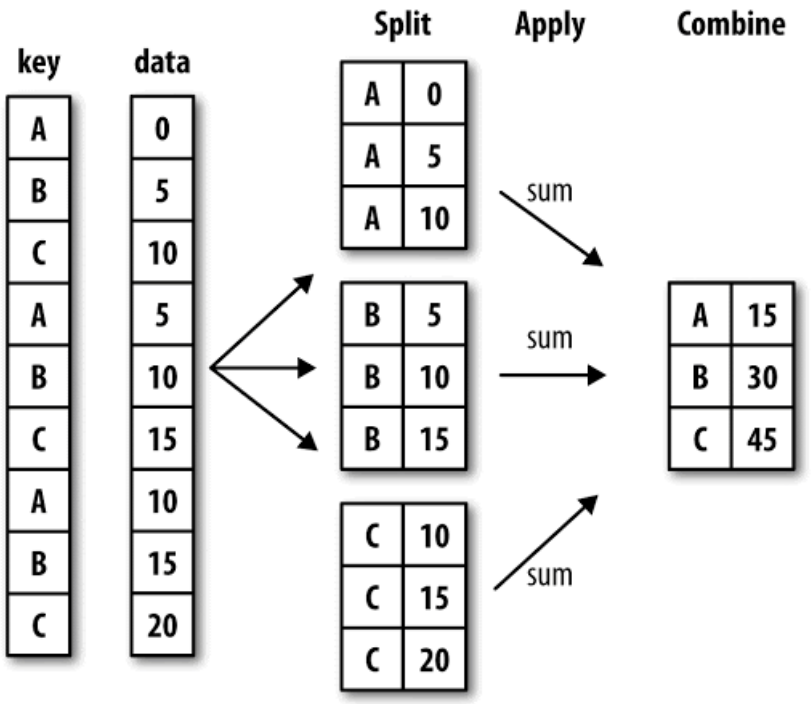
1. Data Grouping - identifies one or more data groups based on values in selected features.
2. Data Aggregation - puts together (aggregates) the values in one or more selected columns - for each group.

Grouping

GroupBy Mechanics

In the first stage of the process, datacontained in a pandas object, whether a Series, DataFrame, or otherwise, is split intogroups based on one or more keys that you provide. The splitting is performed on aparticular axis of an object. For example, a DataFrame can be grouped on its rows(axis=0) or its columns (axis=1). Once this is done, a function is applied to each group,producing a new value. Finally, the results of all those function applications are com-bined into a result object. The form of the resulting object will usually depend on what'sbeing done to the data.

Illustration of a group aggregation



Each grouping key can take many forms, and the keys do not have to be all of the same type:

• A list or array of values that is the same length as the axis being grouped

• A value indicating a column name in a DataFrame

• A dict or Series giving a correspondence between the values on the axis beinggrouped and the group names

• A function to be invoked on the axis index or the individual labels in the index

In [1]:

```
import pandas as pd
import numpy as np
from pandas import DataFrame, Series
df = DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                'key2' : ['one', 'two', 'one', 'two', 'one'],
                'data1' : np.random.randn(5),
                'data2' : np.random.randn(5)})
df
```

Out[1]:

	key1	key2	data1	data2

	key1	key2	data1	data2
0	a	one	-0.735640	0.800486
1	a	two	-1.736677	0.502906
2	b	one	0.510352	-1.104806
3	b	two	0.452037	-1.073484
4	a	one	-0.948772	-0.665086

In [2]:

```
grouped = df['data1'].groupby(df['key1'])
grouped
```

Out[2]:

<pandas.core.groupby.generic.SeriesGroupBy object at 0x7f984d70fa10>

This grouped variable is now a GroupBy object. It has not actually computed anything yet except for some intermediate data about the group key df['key1']. The idea is that this object has all of the information needed to then apply some operation to each of the groups. For example, to compute group means we can call the GroupBy's mean method:

In [3]:

```
grouped.mean()
```

Out[3]:

```
key1
a   -1.140363
b    0.481194
Name: data1, dtype: float64
```

In this case, we grouped the data using two keys, and the resulting Series now has ahierarchical index consisting of the unique pairs of keys observed

In [4]:

```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
```

Out[4]:

```
key1  key2
a     one    -0.842206
      two    -1.736677
b     one     0.510352
      two     0.452037
Name: data1, dtype: float64
```

In [5]:

```
means.unstack()
```

Out[5]:

key2	one	two
key1		
a	-0.842206	-1.736677
b	0.510352	0.452037

In these examples, the group keys are all Series, though they could be any arrays of theright length:

In [6]:

```
states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
years = np.array([2005, 2005, 2006, 2005, 2006])
df['data1'].groupby([states, years]).mean()
```

Out[6]:

```
California  2005   -1.736677
```

```
          2006    0.510352
Ohio      2005   -0.141802
          2006   -0.948772
Name: data1, dtype: float64
```

Frequently the grouping information to be found in the same DataFrame as the datayou want to work on. In that case, you can pass column names (whether those arestrings, numbers, or other Python objects) as the group keys:

In [7]:

```
df.groupby('key1').mean()
```

Out[7]:

key1	data1	data2
a	-1.140363	0.212769
b	0.481194	-1.089145

In [8]:

```
df.groupby(['key1', 'key2']).mean()
```

Out[8]:

key1	key2	data1	data2
a	one	-0.842206	0.067700
	two	-1.736677	0.502906
b	one	0.510352	-1.104806
	two	0.452037	-1.073484

Generally useful GroupBy method is size which return a Series containing group sizes:

In [9]:

```
df.groupby(['key1', 'key2']).size()
```

Out[9]:

```
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

Iterating Over Groups

The GroupBy object supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data. Consider the following small example data set:

In [10]:

```
for name, group in df.groupby('key1'):
  print (name)
  print (group)
```

```
a
  key1 key2     data1     data2
0    a  one -0.735640  0.800486
1    a  two -1.736677  0.502906
4    a  one -0.948772 -0.665086
b
  key1 key2     data1     data2
2    b  one  0.510352 -1.104806
3    b  two  0.452037 -1.073484
```

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

In [11]:

```
for (k1, k2), group in df.groupby(['key1', 'key2']):
  print (k1, k2)
  print (group)
```

```
a one
  key1 key2     data1     data2
0    a  one -0.735640  0.800486
4    a  one -0.948772 -0.665086
a two
  key1 key2     data1     data2
1    a  two -1.736677  0.502906
b one
  key1 key2     data1     data2
2    b  one  0.510352 -1.104806
b two
  key1 key2     data1     data2
3    b  two  0.452037 -1.073484
```

In [12]:

```
pieces = dict(list(df.groupby('key1')))
pieces['b']
```

Out[12]:

	key1	key2	data1	data2
2	b	one	0.510352	-1.104806
3	b	two	0.452037	-1.073484

By default groupby groups on axis=0, but you can group on any of the other axes. For example, we could group the columns of our example df here by dtype like so:

In [13]:

```
df.dtypes
```

Out[13]:

```
key1      object
key2      object
data1    float64
data2    float64
dtype: object
```

In [14]:

```
grouped = df.groupby(df.dtypes, axis=1)
```

In [15]:

```
dict(list(grouped))
```

Out[15]:

```
{dtype('float64'):       data1     data2
 0 -0.735640  0.800486
 1 -1.736677  0.502906
 2  0.510352 -1.104806
 3  0.452037 -1.073484
 4 -0.948772 -0.665086, dtype('O'):   key1 key2
 0    a  one
 1    a  two
 2    b  one
 3    b  two
 4    a  one}
```

Selecting a Column or Subset of Columns

Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the effect of selecting those columns for aggregation. This means that:

In [16]:

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

Out[16]:

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f984d629d90>

are syntactic sugar for:

In [17]:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Out[17]:

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f984d629590>

Especially for large data sets, it may be desirable to aggregate only a few columns. For example, in the above data set, to compute means for just the data2 column and get the result as a DataFrame, we could write:

In [18]:

```
df.groupby(['key1', 'key2'])[['data2']].mean()
```

Out[18]:

		data2
key1	**key2**	
a	**one**	0.067700
	two	0.502906
b	**one**	-1.104806
	two	-1.073484

The object returned by this indexing operation is a grouped DataFrame if a list or array is passed and a grouped Series is just a single column name that is passed as a scalar:

In [19]:

```
s_grouped = df.groupby(['key1', 'key2'])['data2']
s_grouped
```

Out[19]:

<pandas.core.groupby.generic.SeriesGroupBy object at 0x7f984d738150>

In [20]:

```
s_grouped.mean()
```

Out[20]:

```
key1  key2
a     one     0.067700
      two     0.502906
b     one    -1.104806
      two    -1.073484
Name: data2, dtype: float64
```

Grouping with Dicts and Series

Grouping information may exist in a form other than an array. Let's consider another example DataFrame:

In [22]:

```
people = DataFrame(np.random.randn(5, 5),
                   columns=['a', 'b', 'c', 'd', 'e'],
                   index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
people.isna[2:3, ['b', 'c']] = np.nan # Add a few NA valuesIn [40]: people
people
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-2bab2a55b36c> in <module>()
      2                   columns=['a', 'b', 'c', 'd', 'e'],
      3                   index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
----> 4 people.isna[2:3, ['b', 'c']] = np.nan # Add a few NA valuesIn [40]: people
      5 people

TypeError: 'method' object does not support item assignment
```

Now, suppose I have a group correspondence for the columns and want to sum together the columns by group:

In [ ]:

```
mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
           'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

Now, you could easily construct an array from this dict to pass to groupby, but instead we can just pass the dict:

In [ ]:

```
by_column = people.groupby(mapping, axis=1)
by_column.sum()
```

Out[ ]:

	blue	red
Joe	0.250949	0.356979
Steve	-0.748489	2.007008
Wes	3.237395	-3.506715
Jim	1.123620	3.523605
Travis	-1.089539	0.870992

The same functionality holds for Series, which can be viewed as a fixed size mapping. When I used Series as group keys in the above examples, pandas does, in fact, inspect each Series to ensure that its index is aligned with the axis it's grouping:

In [ ]:

```
map_series = Series(mapping)
map_series
```

Out[ ]:

```
a      red
b      red
c     blue
d     blue
e      red
f    orange
dtype: object
```

In [ ]:

```
people.groupby(map_series, axis=1).count()
```

Out[ ]:

	blue	red
Joe	2	3
Steve	2	3

	blue	red
Wes	2	3
Jim	2	3
Travis	2	3

Grouping with Functions

Using Python functions in what can be fairly creative ways is a more abstract way ofdefining a group mapping compared with a dict or Series. Any function passed as agroup key will be called once per index value, with the return values being used as the group names.

In [ ]:

```
people.groupby(len).sum()
```

Out[ ]:

	a	b	c	d	e
3	0.392898	1.723538	2.903377	1.708587	-1.742566
5	1.140920	1.358515	-0.510304	-0.238185	-0.492426
6	1.406850	-1.281476	0.133923	-1.223462	0.745618

Mixing functions with arrays, dicts, or Series is not a problem as everything gets con-verted to arrays internally:

In [ ]:

```
key_list = ['one', 'one', 'one', 'two', 'two']
people.groupby([len, key_list]).min()
```

Out[ ]:

		a	b	c	d	e
3	one	-1.523712	-1.297335	-0.006763	0.257712	-1.023906
	two	0.789746	2.766852	0.135074	0.988545	-0.032993
5	one	1.140920	1.358515	-0.510304	-0.238185	-0.492426
6	two	1.406850	-1.281476	0.133923	-1.223462	0.745618

Grouping by Index Levels

A final convenience for hierarchically-indexed data sets is the ability to aggregate using one of the levels of an axis index. To do this, pass the level number or name using the level keyword:

In [ ]:

```
columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
                                     [1, 3, 5, 1, 3]], names=['cty', 'tenor'])
hier_df = DataFrame(np.random.randn(4, 5), columns=columns)
hier_df
```

Out[ ]:

cty	US			JP	
tenor	1	3	5	1	3
0	-1.230101	-0.292889	0.865257	0.337283	-0.038832
1	1.171595	0.256918	-0.419728	0.589071	0.343562
2	-1.131444	-0.161085	-0.085181	-0.974848	1.132221
3	-0.434778	-1.089268	-0.354405	-2.703953	0.318307

In [ ]:

```
hier_df.groupby(level='cty', axis=1).count()
```

Out[ ]:

cty	JP	US

cty	JP	US
1	2	3
2	2	3
3	2	3

Data Aggregation

By aggregation, generally refers to any data transformation that produces scalarvalues from arrays.

In [ ]:

```
df
```

Out[ ]:

	key1	key2	data1	data2
0	a	one	-0.027700	-0.647615
1	a	two	-0.686131	1.237773
2	b	one	0.671435	0.728088
3	b	two	0.995261	0.783671
4	a	one	0.550162	-1.084850

In [ ]:

```
grouped = df.groupby('key1')
```

quantile computes sample quantiles of a Series or aDataFrame's columns

In [ ]:

```
grouped['data1'].quantile(0.9)
```

Out[ ]:

```
key1
a    0.434590
b    0.962878
Name: data1, dtype: float64
```

While quantile is not explicitly implemented for GroupBy, it is a Series method and thus available for use. Internally, GroupBy efficiently slices up the Series, calls *piece.quantile(0.9)* for each piece, then assembles those results together into the result object.To use your own aggregation functions, pass any function that aggregates an array to the *aggregate* or *agg* method:

In [ ]:

```
def peak_to_peak(arr):
  return arr.max() - arr.min()
grouped.agg(peak_to_peak)
```

Out[ ]:

	data1	data2
key1		
a	1.236293	2.322623
b	0.323825	0.055583

Some methods like describe also work, even though they are not aggregations

In [ ]:

```
grouped.describe()
```

Out[ ]:

	data1								data2						
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%
key1															
a	3.0	-0.054556	0.618584	-0.686131	-0.356915	-0.027700	0.261231	0.550162	3.0	-0.164897	1.234264	-1.084850	-0.866232	-0.647615	0.295079
b	2.0	0.833348	0.228979	0.671435	0.752391	0.833348	0.914304	0.995261	2.0	0.755880	0.039303	0.728088	0.741984	0.755880	0.769775

Optimized groupby methods

Function name	Description
count	Number of non-NA values in the group
sum	Sum of non-NA values
mean	Mean of non-NA values
median	Arithmetic median of non-NA values
std, var	Unbiased (n - 1 denominator) standard deviation and variance
min, max	Minimum and maximum of non-NA values
prod	Product of non-NA values
first, last	First and last non-NA values

In [ ]:

```
tips = pd.read_csv('sample_data/tips.csv')
tips['tip_pct'] = tips['tip'] / tips['total_bill']
tips[:6]
```

Out[ ]:

	total_bill	tip	sex	smoker	day	time	size	tip_pct
0	16.99	1.01	Female	No	Sun	Dinner	2	0.059447
1	10.34	1.66	Male	No	Sun	Dinner	3	0.160542
2	21.01	3.50	Male	No	Sun	Dinner	3	0.166587
3	23.68	3.31	Male	No	Sun	Dinner	2	0.139780
4	24.59	3.61	Female	No	Sun	Dinner	4	0.146808
5	25.29	4.71	Male	No	Sun	Dinner	4	0.186240

load *tips.csv* it with read_csv, and add a tipping percentage column *tip_pct*

Column-wise and Multiple Function Application

In [ ]:

```
grouped = tips.groupby(['sex', 'smoker'])
```

In [ ]:

```
grouped_pct = grouped['tip_pct']
grouped_pct.agg('mean')
```

Out[ ]:

```
sex     smoker
Female  No        0.156921
        Yes       0.182150
Male    No        0.160669
        Yes       0.152771
Name: tip_pct, dtype: float64
```

In [ ]:

```
grouped_pct.agg(['mean', 'std', peak_to_peak])
```

Out[ ]:

		mean	std	peak_to_peak
sex	**smoker**			
Female	**No**	0.156921	0.036421	0.195876
	Yes	0.182150	0.071595	0.360233
Male	**No**	0.160669	0.041849	0.220186
	Yes	0.152771	0.090588	0.674707

In [ ]:

```
grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
```

Out[ ]:

		foo	bar
sex	**smoker**		
Female	**No**	0.156921	0.036421
	Yes	0.182150	0.071595
Male	**No**	0.160669	0.041849
	Yes	0.152771	0.090588

With a DataFrame, you have more options as you can specify a list of functions to applyto all of the columns or different functions per column. To start, suppose we wanted to compute the same three statistics for the tip_pct and total_bill columns:

In [ ]:

```
functions = ['count', 'mean', 'max']
result = grouped['tip_pct', 'total_bill'].agg(functions)
result
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: FutureWarning: Indexing with multiple keys (implicitly converted to a tuple of keys) will be deprecated, use a list instead.
```

Out[ ]:

		tip_pct			total_bill		
		count	**mean**	**max**	**count**	**mean**	**max**
sex	**smoker**						
Female	**No**	54	0.156921	0.252672	54	18.105185	35.83
	Yes	33	0.182150	0.416667	33	17.977879	44.30
Male	**No**	97	0.160669	0.291990	97	19.791237	48.33
	Yes	60	0.152771	0.710345	60	22.284500	50.81

The resulting DataFrame has hierarchical columns, the same as you would get aggregating each column separately and using *concat* to glue the results together using the column names as the *keys* argument

In [ ]:

```
result['tip_pct']
```

Out[ ]:

		count	mean	max
sex	**smoker**			
Female	**No**	54	0.156921	0.252672
	Yes	33	0.182150	0.416667
Male	**No**	97	0.160669	0.291990
	Yes	60	0.152771	0.710345

In [ ]:

```
ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
grouped['tip_pct', 'total_bill'].agg(ftuples)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: FutureWarning: Indexing with multiple keys (implicitly converted to a tuple of keys) will be deprecated, use a list instead.
```

Out[ ]:

		tip_pct		total_bill	
		Durchschnitt	Abweichung	Durchschnitt	Abweichung
sex	smoker				
Female	No	0.156921	0.001327	18.105185	53.092422
	Yes	0.182150	0.005126	17.977879	84.451517
Male	No	0.160669	0.001751	19.791237	76.152961
	Yes	0.152771	0.008206	22.284500	98.244673

Suppose you wanted to apply potentially different functions to one or more of the columns. The trick is to pass a dict to *agg* that contains a mapping of column names to any of the function specifications listed so far:

In [ ]:

```
grouped.agg({'tip' : np.max, 'size' : 'sum'})
```

Out[ ]:

		tip	size
sex	smoker		
Female	No	5.2	140
	Yes	6.5	74
Male	No	9.0	263
	Yes	10.0	150

In [ ]:

```
grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
             'size' : 'sum'})
```

Out[ ]:

		tip_pct				size
		min	max	mean	std	sum
sex	smoker					
Female	No	0.056797	0.252672	0.156921	0.036421	140
	Yes	0.056433	0.416667	0.182150	0.071595	74
Male	No	0.071804	0.291990	0.160669	0.041849	263
	Yes	0.035638	0.710345	0.152771	0.090588	150

A DataFrame will have hierarchical columns only if multiple functions are applied to at least one column.

Returning Aggregated Data in "unindexed" Form

In all of the examples up until now, the aggregated data comes back with an index, potentially hierarchical, composed from the unique group key combinations observed. Since this isn't always desirable, you can disable this behavior in most cases by passing *as_index=False* to *groupby*:

In [ ]:

```
tips.groupby(['sex', 'smoker'], as_index=False).mean()
```

Out[ ]:

	sex	smoker	total_bill	tip	size	tip_pct
0	Female	No	18.105185	2.773519	2.592593	0.156921
1	Female	Yes	17.977879	2.931515	2.242424	0.182150
2	Male	No	19.791237	3.113402	2.711340	0.160669
3	Male	Yes	22.284500	3.051167	2.500000	0.152771

Of course, it's always possible to obtain the result in this format by calling *reset_index* on the result

Group-wise Operations and Transformations

Aggregation is only one kind of group operation. It is a special case in the more generalclass of data transformations; that is, it accepts functions that reduce a one-dimensional array to a scalar value. In examples below, we will use *transform* and apply methods, which will enable to do many other kinds of group operations

In [ ]:

```
df
```

Out[ ]:

	key1	key2	data1	data2
0	a	one	-0.027700	-0.647615
1	a	two	-0.686131	1.237773
2	b	one	0.671435	0.728088
3	b	two	0.995261	0.783671
4	a	one	0.550162	-1.084850

In [ ]:

```
k1_means = df.groupby('key1').mean().add_prefix('mean_')
k1_means
```

Out[ ]:

	mean_data1	mean_data2
key1		
a	-0.054556	-0.164897
b	0.833348	0.755880

In [ ]:

```
pd.merge(df, k1_means, left_on='key1', right_index=True)
```

Out[ ]:

	key1	key2	data1	data2	mean_data1	mean_data2
0	a	one	-0.027700	-0.647615	-0.054556	-0.164897
1	a	two	-0.686131	1.237773	-0.054556	-0.164897
4	a	one	0.550162	-1.084850	-0.054556	-0.164897
2	b	one	0.671435	0.728088	0.833348	0.755880
3	b	two	0.995261	0.783671	0.833348	0.755880

This works, but is somewhat inflexible. You can think of the operation as transformingthe two data columns using the *np.mean* function.

In [ ]:

```
key = ['one', 'two', 'one', 'two', 'one']
people.groupby(key).mean()
```

Out[ ]:

	a	b	c	d	e
one	0.336667	-0.774930	0.967408	-0.167807	-0.321319
two	0.965333	2.062683	-0.187615	0.375180	-0.262710

In [ ]:

```
people.groupby(key).transform(np.mean)
```

Out[ ]:

	a	b	c	d	e
Joe	0.336667	-0.774930	0.967408	-0.167807	-0.321319
Steve	0.965333	2.062683	-0.187615	0.375180	-0.262710
Wes	0.336667	-0.774930	0.967408	-0.167807	-0.321319
Jim	0.965333	2.062683	-0.187615	0.375180	-0.262710
Travis	0.336667	-0.774930	0.967408	-0.167807	-0.321319

transform applies a function to each group, then places the resultsin the appropriate locations.

In [ ]:

```
def demean(arr):
  return arr - arr.mean()
demeaned = people.groupby(key).transform(demean)
demeaned
```

Out[ ]:

	a	b	c	d	e
Joe	0.790197	1.028951	-0.974171	0.425518	-0.702587
Steve	0.175587	-0.704169	-0.322689	-0.613365	-0.229717
Wes	-1.860380	-0.522405	1.807657	0.630137	-0.364349
Jim	-0.175587	0.704169	0.322689	0.613365	0.229717
Travis	1.070183	-0.506546	-0.833486	-1.055655	1.066936

You can check that demeaned now has zero group means:

In [ ]:

```
demeaned.groupby(key).mean()
```

Out[ ]:

	a	b	c	d	e
one	7.401487e-17	7.401487e-17	7.401487e-17	0.0	0.0
two	5.551115e-17	-1.110223e-16	0.000000e+00	0.0	0.0

Apply: General split-apply-combine

Like aggregate, transform is a more specialized function having rigid requirements: the passed function must either produce a scalar value to be broadcasted (like np.mean) or a transformed array of the same size.

In [ ]:

```
def top(df, n=5, column='tip_pct'):
  return df.sort_index(axis=0)[-n:]
top(tips, n=6)
```

Out[ ]:

	total_bill	tip	sex	smoker	day	time	size	tip_pct

	total_bill	tip	sex	smoker	day	time	size	tip_pct
238	35.83	4.67	Female	No	Sat	Dinner	3	0.130338
239	29.03	5.92	Male	No	Sat	Dinner	3	0.203927
240	27.18	2.00	Female	Yes	Sat	Dinner	2	0.073584
241	22.67	2.00	Male	Yes	Sat	Dinner	2	0.088222
242	17.82	1.75	Male	No	Sat	Dinner	2	0.098204
243	18.78	3.00	Female	No	Thur	Dinner	2	0.159744

In [ ]:

```
tips.groupby('smoker').apply(top)
```

Out[ ]:

		total_bill	tip	sex	smoker	day	time	size	tip_pct
smoker									
No	**235**	10.07	1.25	Male	No	Sat	Dinner	2	0.124131
	238	35.83	4.67	Female	No	Sat	Dinner	3	0.130338
	239	29.03	5.92	Male	No	Sat	Dinner	3	0.203927
	242	17.82	1.75	Male	No	Sat	Dinner	2	0.098204
	243	18.78	3.00	Female	No	Thur	Dinner	2	0.159744
Yes	**234**	15.53	3.00	Male	Yes	Sat	Dinner	2	0.193175
	236	12.60	1.00	Male	Yes	Sat	Dinner	2	0.079365
	237	32.83	1.17	Male	Yes	Sat	Dinner	2	0.035638
	240	27.18	2.00	Female	Yes	Sat	Dinner	2	0.073584
	241	22.67	2.00	Male	Yes	Sat	Dinner	2	0.088222

In [ ]:

```
tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
```

Out[ ]:

			total_bill	tip	sex	smoker	day	time	size	tip_pct
smoker	**day**									
No	**Fri**	**223**	15.98	3.00	Female	No	Fri	Lunch	3	0.187735
	Sat	**242**	17.82	1.75	Male	No	Sat	Dinner	2	0.098204
	Sun	**185**	20.69	5.00	Male	No	Sun	Dinner	5	0.241663
	Thur	**243**	18.78	3.00	Female	No	Thur	Dinner	2	0.159744
Yes	**Fri**	**226**	10.09	2.00	Female	Yes	Fri	Lunch	2	0.198216
	Sat	**241**	22.67	2.00	Male	Yes	Sat	Dinner	2	0.088222
	Sun	**190**	15.69	1.50	Male	Yes	Sun	Dinner	2	0.095602
	Thur	**205**	16.47	3.23	Female	Yes	Thur	Lunch	3	0.196114

In [ ]:

```
result = tips.groupby('smoker')['tip_pct'].describe()
result
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
smoker								
No	151.0	0.159328	0.039910	0.056797	0.136906	0.155625	0.185014	0.291990
Yes	93.0	0.163196	0.085119	0.035638	0.106771	0.153846	0.195059	0.710345

In [ ]:

```
result.unstack('smoker')
```

Out[ ]:

```
       smoker
count  No       151.000000
       Yes       93.000000
mean   No         0.159328
       Yes        0.163196
std    No         0.039910
       Yes        0.085119
min    No         0.056797
       Yes        0.035638
25%    No         0.136906
       Yes        0.106771
50%    No         0.155625
       Yes        0.153846
75%    No         0.185014
       Yes        0.195059
max    No         0.291990
       Yes        0.710345
dtype: float64
```

In the examples above, you see that the resulting object has a hierarchical index formed from the group keys along with the indexes of each piece of the original object. This can be disabled by passing group_keys=False to groupby:

In [ ]:

```
tips.groupby('smoker', group_keys=False).apply(top)
```

Out[ ]:

	total_bill	tip	sex	smoker	day	time	size	tip_pct
235	10.07	1.25	Male	No	Sat	Dinner	2	0.124131
238	35.83	4.67	Female	No	Sat	Dinner	3	0.130338
239	29.03	5.92	Male	No	Sat	Dinner	3	0.203927
242	17.82	1.75	Male	No	Sat	Dinner	2	0.098204
243	18.78	3.00	Female	No	Thur	Dinner	2	0.159744
234	15.53	3.00	Male	Yes	Sat	Dinner	2	0.193175
236	12.60	1.00	Male	Yes	Sat	Dinner	2	0.079365
237	32.83	1.17	Male	Yes	Sat	Dinner	2	0.035638
240	27.18	2.00	Female	Yes	Sat	Dinner	2	0.073584
241	22.67	2.00	Male	Yes	Sat	Dinner	2	0.088222

Quantile and Bucket Analysis

Pandas has some tools, in particular cut and qcut, for slicing data up into buckets with bins of your choosing or by sample quantiles. Combining these functions with groupby, it becomes very simple to perform bucket or quantile analysis on a data set. Consider a simple random data set and an equal-length bucket categorization using cut:

In [ ]:

```
frame = DataFrame({'data1': np.random.randn(1000),
                   'data2': np.random.randn(1000)})
factor = pd.cut(frame.data1, 4)
factor[:10]
```

Out[ ]:

```
0    (-1.68, 0.127]
1    (-1.68, 0.127]
2    (0.127, 1.933]
3    (1.933, 3.739]
4    (0.127, 1.933]
5    (-1.68, 0.127]
6    (-1.68, 0.127]
7    (-1.68, 0.127]
8    (-1.68, 0.127]
9    (0.127, 1.933]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-3.494, -1.68] < (-1.68, 0.127] < (0.127, 1.933] < (1.933, 3.739]]
```

The Factor object returned by cut can be passed directly to groupby. So we could compute a set of statistics for the data2 column like so:

In [ ]:

```
def get_stats(group):
  return {'min': group.min(), 'max': group.max(),
          'count': group.count(), 'mean': group.mean()}
grouped = frame.data2.groupby(factor)
grouped.apply(get_stats).unstack()
```

Out[ ]:

	min	max	count	mean
data1				
(-3.494, -1.68]	-2.085330	2.592071	51.0	0.120350
(-1.68, 0.127]	-2.874071	2.942335	495.0	0.034777
(0.127, 1.933]	-2.486408	3.076796	420.0	0.015958
(1.933, 3.739]	-1.801146	2.038160	34.0	-0.161911

These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use qcut. I'll pass labels=False to just get quantile numbers.

In [ ]:

```
grouping = pd.qcut(frame.data1, 10, labels=False)
grouped = frame.data2.groupby(grouping)
grouped.apply(get_stats).unstack()
```

Out[ ]:

	min	max	count	mean
data1				
0	-2.329489	2.592071	100.0	0.095392
1	-2.874071	2.295804	100.0	0.057695
2	-2.606584	2.321878	100.0	0.086769
3	-2.284195	2.942335	100.0	0.059867
4	-2.115219	2.068028	100.0	-0.077452
5	-2.252734	2.740555	100.0	-0.028123
6	-2.486408	3.076796	100.0	-0.064732
7	-2.365190	2.832177	100.0	0.051960
8	-2.110240	2.217184	100.0	0.084444
9	-2.194715	2.913957	100.0	-0.020322

Pivot Tables and Cross-Tabulation

A pivot table is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible using the groupby facility described in this chapter combined with reshape operations utilizing hierarchical indexing. DataFrame has a pivot_table method, and additionally there is a top-level pandas.pivot_table function.