In [1]:
```
import numpy as np
import pandas as pd
```

Hadley Wickham, coined the term split-apply-combine for describing group operations.

STEP 1: data contained in a pandas object, whether a Series, Data-Frame, or otherwise, is split into groups based on one or more keys that you provide.

The splitting is performed on a particular axis of an object.
- For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1).

STEP 2: A function is applied to each group, producing a new value.

STEP 3: The results of all those function applications are combined into a result object.
- The form of the resulting object will usually depend on what's being done to the data.

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 being grouped and the group names
- A function to be invoked on the axis index or the individual labels in the index

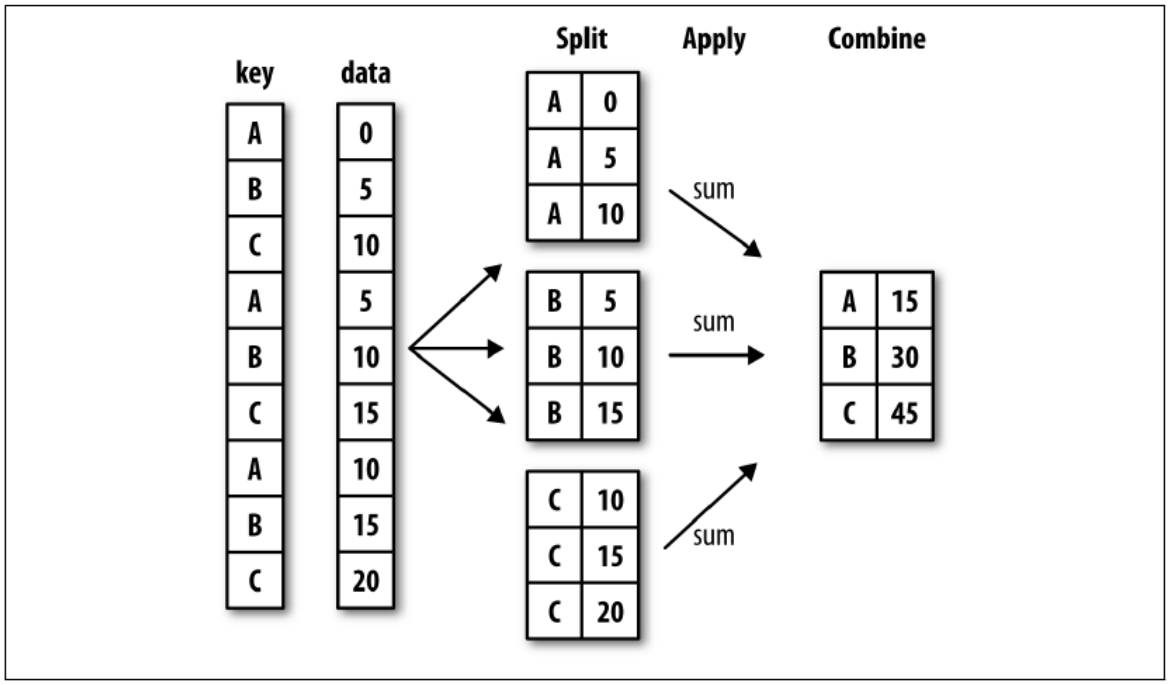

Figure 10-1. Illustration of a group aggregation

EXAMPLE 1a

In [2]:
```
df1 = pd.DataFrame({'key' : ['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C'],
                    'data' : [0,5,10,5,10,15,10,15,20]})
df1
```

Out[2]:

	key	data
0	A	0
1	B	5
2	C	10
3	A	5
4	B	10
5	C	15
6	A	10
7	B	15
8	C	20

In [3]:
```
group = df1['data'].groupby(df1['key'])
group
```

Out[3]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001F5076EDA00>
```

In [5]:
```
group.sum()
```

Out[5]:
```
key
A    15
B    30
C    45
Name: data, dtype: int64
```

In [6]:
```
# in a single statement
df1['data'].groupby(df1['key']).sum()
```

Out[6]:
```
key
A    15
B    30
C    45
Name: data, dtype: int64
```

NOTE : Missing values in a group key will be excluded from the result. *

NOTE: By default, all of the numeric columns are aggregated *

EXAMPLE 1b

In [7]:
```
df2 = pd.DataFrame({'key' : ['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C'],
                    'data' : [0,np.nan,10,5,np.nan,15,10,15,20]})
df2
```

Out[7]:

	key	data
0	A	0.0
1	B	NaN
2	C	10.0
3	A	5.0
4	B	NaN
5	C	15.0
6	A	10.0
7	B	15.0
8	C	20.0

In [8]:
```
df2['data'].groupby(df2['key']).mean()
```

Out[8]:
```
key
A     5.0
B    15.0
C    15.0
Name: data, dtype: float64
```

In [ ]:
```

```

In [9]:
```
df = pd.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[9]:

	key1	key2	data1	data2
0	a	one	-0.051378	-0.962742
1	a	two	-0.134297	1.509390
2	b	one	0.667821	-0.237095
3	b	two	-2.050491	-0.754728
4	a	one	-0.520796	1.331972

Type *Markdown* and LaTeX: α^2

EXAMPLE 2

```
Suppose you wanted to compute the mean of the data1 column using the labels from
key1.
- One way, is to access data1 and call groupby with the column (a Series) at key1:
```

```
Example: Aggregate the data (a Series) according to the group key (say column key1),
producing a new Series that is now indexed by the unique values in the key1 column.
```

In [10]:
```
grouped = df['data1'].groupby(df['key1'])
grouped
```

Out[10]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001F5076C1730>
```

```
This object has all of the information needed to apply some operation to each of the
groups.
For example, to compute group means we can call the GroupBy's mean method:
```

In [11]:
```
grouped.mean()
```

Out[11]:
```
key1
a   -0.235491
b   -0.691335
Name: data1, dtype: float64
```

In [ ]:

In [ ]:

EXAMPLE 3 - group the data using two keys

In [12]:
```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
# -> the resulting Series has a hierarchical index consisting of the unique pairs of ke
```

Out[12]:
```
key1  key2
a     one    -0.286087
      two    -0.134297
b     one     0.667821
      two    -2.050491
Name: data1, dtype: float64
```

In [13]:
```
means.unstack()
```

Out[13]:

key2	one	two
key1		
a	-0.286087	-0.134297
b	0.667821	-2.050491

EXAMPLE 4

In [14]:
```
df
```

Out[14]:

	key1	key2	data1	data2
0	a	one	-0.051378	-0.962742
1	a	two	-0.134297	1.509390
2	b	one	0.667821	-0.237095
3	b	two	-2.050491	-0.754728
4	a	one	-0.520796	1.331972

In [15]:
```
states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
years = np.array([2005, 2005, 2006, 2005, 2006])
df['data1'].groupby([states, years]).mean()
```

Out[15]:
```
California  2005   -0.134297
            2006    0.667821
Ohio        2005   -1.050934
            2006   -0.520796
Name: data1, dtype: float64
```

EXAMPLE 5

```
Frequently the grouping information is found in the same DataFrame as the data you
want to work on.
- Pass column names (whether those are strings,numbers, or other Python objects) as
the group keys:
```

In [16]:
```
df
```

Out[16]:

	key1	key2	data1	data2
0	a	one	-0.051378	-0.962742
1	a	two	-0.134297	1.509390
2	b	one	0.667821	-0.237095
3	b	two	-2.050491	-0.754728
4	a	one	-0.520796	1.331972

In [17]:
```
df.groupby('key1').mean()
```

Out[17]:

key1	data1	data2
a	-0.235491	0.626207
b	-0.691335	-0.495912

In [18]:
```
df.groupby(['key1', 'key2']).mean()
```

Out[18]:

		data1	data2
key1	**key2**		
a	**one**	-0.286087	0.184615
	two	-0.134297	1.509390
b	**one**	0.667821	-0.237095
	two	-2.050491	-0.754728

In [19]:
```
df.describe()
```

Out[19]:

	data1	data2
count	5.000000	5.000000
mean	-0.417828	0.177359
std	1.008650	1.167029
min	-2.050491	-0.962742
25%	-0.520796	-0.754728
50%	-0.134297	-0.237095
75%	-0.051378	1.331972
max	0.667821	1.509390

Iterating Over Groups

In [20]:
```
df
```

Out[20]:

	key1	key2	data1	data2
0	a	one	-0.051378	-0.962742
1	a	two	-0.134297	1.509390
2	b	one	0.667821	-0.237095
3	b	two	-2.050491	-0.754728
4	a	one	-0.520796	1.331972

In [21]:
```
for name, group in df.groupby('key1'):
    print(name)
    print(group)
```

```
a
  key1 key2     data1     data2
0    a  one -0.051378 -0.962742
1    a  two -0.134297  1.509390
4    a  one -0.520796  1.331972
b
  key1 key2     data1     data2
2    b  one  0.667821 -0.237095
3    b  two -2.050491 -0.754728
```

In [22]:
```
#In the case of multiple keys, the first element in the tuple will be a tuple of key v
for (k1, k2), group in df.groupby(['key1', 'key2']):
    print((k1, k2))
    print(group)
```

```
('a', 'one')
  key1 key2     data1     data2
0    a  one -0.051378 -0.962742
4    a  one -0.520796  1.331972
('a', 'two')
  key1 key2     data1    data2
1    a  two -0.134297  1.50939
('b', 'one')
  key1 key2     data1     data2
2    b  one  0.667821 -0.237095
('b', 'two')
  key1 key2     data1     data2
3    b  two -2.050491 -0.754728
```

In [23]:
```
#pieces = dict(list(df.groupby('key1')))
lst1 = list(df.groupby('key1'))
print(lst1)
pieces = dict(lst1)
pieces
```

```
[('a',   key1 key2     data1     data2
0    a  one -0.051378 -0.962742
1    a  two -0.134297  1.509390
4    a  one -0.520796  1.331972), ('b',   key1 key2     data1     data2
2    b  one  0.667821 -0.237095
3    b  two -2.050491 -0.754728)]
```

Out[23]:
```
{'a':   key1 key2     data1     data2
 0    a  one -0.051378 -0.962742
 1    a  two -0.134297  1.509390
 4    a  one -0.520796  1.331972,
 'b':   key1 key2     data1     data2
 2    b  one  0.667821 -0.237095
 3    b  two -2.050491 -0.754728}
```

In [24]:
```
pieces['b']
```

Out[24]:

	key1	key2	data1	data2
2	b	one	0.667821	-0.237095
3	b	two	-2.050491	-0.754728

```
......read from textbook
```

In [ ]:

In [ ]:

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 column subsetting for aggregation.
```

In [25]:
```
df
```

Out[25]:

	key1	key2	data1	data2
0	a	one	-0.051378	-0.962742
1	a	two	-0.134297	1.509390
2	b	one	0.667821	-0.237095
3	b	two	-2.050491	-0.754728
4	a	one	-0.520796	1.331972

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

similar to
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])

Especially for large datasets, it may be desirable to aggregate only a few columns.
```

In [26]:
```
#to compute means for just the data2 column and get the result as a DataFrame
df.groupby(['key1', 'key2'])[['data2']].mean()
# result is a grouped DataFrame
```

Out[26]:

		data2
key1	**key2**	
a	**one**	0.184615
	two	1.509390
b	**one**	-0.237095
	two	-0.754728

In [27]:
```
s_grouped = df.groupby(['key1', 'key2'])['data2']
s_grouped
```

Out[27]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001F507761BE0>
```

In [28]:
```
s_grouped.mean()
```

Out[28]:
```
key1  key2
a     one     0.184615
      two     1.509390
b     one    -0.237095
      two    -0.754728
Name: data2, dtype: float64
```

In [ ]:
```

```

Grouping with Dicts and Series

In [29]:
```
people = pd.DataFrame(np.random.randn(5, 5),
                      columns=['a', 'b', 'c', 'd', 'e'],
                      index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
people
```

Out[29]:

	a	b	c	d	e
Joe	-0.546033	0.629988	0.287592	-0.680961	-0.407830
Steve	-1.400705	-0.255094	0.088846	-0.696793	-0.344939
Wes	-2.617020	-1.157487	-0.980971	-0.791466	-0.475874
Jim	-2.029291	-0.516041	0.246031	0.928949	0.329846
Travis	-2.265158	0.278336	-0.229511	-0.563968	1.119181

In [30]:
```
people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values
people
```

Out[30]:

	a	b	c	d	e
Joe	-0.546033	0.629988	0.287592	-0.680961	-0.407830
Steve	-1.400705	-0.255094	0.088846	-0.696793	-0.344939
Wes	-2.617020	NaN	NaN	-0.791466	-0.475874
Jim	-2.029291	-0.516041	0.246031	0.928949	0.329846
Travis	-2.265158	0.278336	-0.229511	-0.563968	1.119181

In [31]:
```
mapping = {'a': 'red', 'b': 'red', 'c': 'blue','d': 'blue', 'e': 'red', 'f' : 'orange'
mapping
```

Out[31]:
```
{'a': 'red', 'b': 'red', 'c': 'blue', 'd': 'blue', 'e': 'red', 'f': 'orange'}
```

In [32]:
```
by_column = people.groupby(mapping, axis=1)
by_column.sum()
```

Out[32]:

	blue	red
Joe	-0.393369	-0.323875
Steve	-0.607948	-2.000737
Wes	-0.791466	-3.092894
Jim	1.174980	-2.215486
Travis	-0.793478	-0.867641

In [33]:
```
map_series = pd.Series(mapping)
map_series
```

Out[33]:
```
a       red
b       red
c      blue
d      blue
e       red
f    orange
dtype: object
```

In [34]:
```
people.groupby(map_series, axis=1).count()
```

Out[34]:

	blue	red
Joe	2	3
Steve	2	3
Wes	1	2
Jim	2	3
Travis	2	3

(Simple) Example

In [35]:
```
data1 = pd.DataFrame(np.arange(25).reshape(5, 5),
                     columns=['a', 'b', 'c', 'd', 'e'])
data1
```

Out[35]:

	a	b	c	d	e
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

In [36]:
```
mapping = {'a': 'red', 'b': 'red', 'c': 'blue','d': 'blue', 'e': 'green', 'f' : 'orang
mapping
```

Out[36]:
```
{'a': 'red', 'b': 'red', 'c': 'blue', 'd': 'blue', 'e': 'green', 'f': 'orange'}
```

In [37]:
```
by_column = data1.groupby(mapping, axis=1)
by_column.sum()
```

Out[37]:

	blue	green	red
0	5	4	1
1	15	9	11
2	25	14	21
3	35	19	31
4	45	24	41

In [38]:
```
data1.iloc[2:3, [1, 2]] = np.nan # Add a few NA values
data1
```

Out[38]:

	a	b	c	d	e
0	0	1.0	2.0	3	4
1	5	6.0	7.0	8	9
2	10	NaN	NaN	13	14
3	15	16.0	17.0	18	19
4	20	21.0	22.0	23	24

In [39]:
```
by_column = data1.groupby(mapping, axis=1)
by_column.sum()
```

Out[39]:

	blue	green	red
0	5.0	4.0	1.0
1	15.0	9.0	11.0
2	13.0	14.0	10.0
3	35.0	19.0	31.0
4	45.0	24.0	41.0

In [40]:
```
by_column.mean()    # *** Missing values in a group key will be excluded from the resu
```

Out[40]:

	blue	green	red
0	2.5	4.0	0.5
1	7.5	9.0	5.5
2	13.0	14.0	10.0
3	17.5	19.0	15.5
4	22.5	24.0	20.5

Grouping with Functions

```
Using Python functions is a more generic way of defining a group mapping compared
with a dict or Series.

Any function passed as a group key will be called once per index value, with the
return values being used as the group names.
Example:
Consider a DataFrame from the previous section, which has people's first names as
index values. Suppose you wanted to group by the length of the names;
while you could compute an array of string lengths, it's simpler to just pass the
len
function:
```

In [41]:
```
people
```

Out[41]:

	a	b	c	d	e
Joe	-0.546033	0.629988	0.287592	-0.680961	-0.407830
Steve	-1.400705	-0.255094	0.088846	-0.696793	-0.344939
Wes	-2.617020	NaN	NaN	-0.791466	-0.475874
Jim	-2.029291	-0.516041	0.246031	0.928949	0.329846
Travis	-2.265158	0.278336	-0.229511	-0.563968	1.119181

In [42]:
```
people.groupby(len).sum()
```

Out[42]:

	a	b	c	d	e
3	-5.192344	0.113946	0.533623	-0.543478	-0.553858
5	-1.400705	-0.255094	0.088846	-0.696793	-0.344939
6	-2.265158	0.278336	-0.229511	-0.563968	1.119181

```
Mixing functions with arrays, dicts, or Series is not a problem as everything gets
converted
to arrays internally:
```

In [43]:
```
key_list = ['one', 'one', 'one', 'two', 'two']
people.groupby([len, key_list]).min()
```

Out[43]:

		a	b	c	d	e
3	**one**	-2.617020	0.629988	0.287592	-0.791466	-0.475874
	two	-2.029291	-0.516041	0.246031	0.928949	0.329846
5	**one**	-1.400705	-0.255094	0.088846	-0.696793	-0.344939
6	**two**	-2.265158	0.278336	-0.229511	-0.563968	1.119181

In [ ]:
```

```

Grouping by Index Levels

```
A final convenience for hierarchically indexed datasets is the ability to aggregate
using one of the levels of an axis index. Let’s look at an example:
In [47]:
```

In [44]:
```
columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
                                     [1, 3, 5, 1, 3]],names=['cty', 'tenor'])
hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)
hier_df
```

Out[44]:

cty	US			JP	
tenor	1	3	5	1	3
0	-0.636608	0.964260	-0.772904	-1.064918	-1.426189
1	0.550701	1.985605	-0.165582	-1.234031	1.001684
2	-1.009413	1.449641	-0.092238	1.409683	0.331937
3	-0.050421	-0.210326	0.613293	0.838488	0.139383

In [45]:
```
#To group by level, pass the level number or name using the level keyword:
hier_df.groupby(level='cty', axis=1).count()
```

Out[45]:

cty	JP	US
0	2	3
1	2	3
2	2	3
3	2	3

In [ ]:
```

```