GROUPBY MECHANICS

In [1]:

```
import numpy as np
import pandas as pd
```

In [6]:

```
#Groupby Mechanics

# The term split-apply-combine is used for describing group operations.
    # In the first stage of the process, data contained in a pandas object, whether a Seri
es, 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).
    # Once this is done, a function is applied to each group, producing a new value.
    # Finally, the results of all those function applications are combined into a result o
bject.
    # The form of the resulting object will usually depend on what’s being done to the dat
a.


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[6]:

	key1	key2	data1	data2
0	a	one	-0.699511	-0.388558
1	a	two	-0.345863	0.535094
2	b	one	-0.749546	-1.071714
3	b	two	-2.196750	0.524351
4	a	one	0.264189	2.621417

In [7]:

```
# Suppose you wanted to compute the mean of the data1 column using the labels from key1.
    #There are a number of ways to do this.
    # One is to access data1 and call groupby with the column (a Series) at key1:

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

Out[7]:

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x00000039EB6D3A48>
```

In [16]:

```
# 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 op
eration to
    # each of the groups.
    # For example, to compute group size we can call the GroupBy’s size method:

grouped.size()
```

Out[16]:

```
key1
a    3
b    2
Name: data1, dtype: int64
```

In [17]:

```
# To compute group summary statistics we can call the GroupBy’s describe method:

grouped.describe()
```

Out[17]:

	count	mean	std	min	25%	50%	75%	max
key1								
a	3.0	-0.260395	0.487502	-0.699511	-0.522687	-0.345863	-0.040837	0.264189
b	2.0	-1.473148	1.023328	-2.196750	-1.834949	-1.473148	-1.111347	-0.749546

In [19]:

```
# To compute group means we can call the GroupBy’s mean method:

grouped.mean()
```

Out[19]:

```
key1
a   -0.260395
b   -1.473148
Name: data1, dtype: float64
```

In [20]:

```
# Thus above, the data (a Series) has been aggregated according to the group key,
    # producing a new Series that is now indexed by the unique values in the key1 column.
    # The result index has the name 'key1' because the DataFrame column df['key1'] did.
    # If instead we had passed multiple arrays as a list, we’d get something different:

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

Out[20]:

```
key1  key2
a     one     -0.217661
      two     -0.345863
b     one     -0.749546
      two     -2.196750
Name: data1, dtype: float64
```

In [21]:

```
df.stack()
```

Out[21]:

```
0  key1           a
   key2         one
   data1  -0.699511
   data2  -0.388558
1  key1           a
   key2         two
   data1  -0.345863
   data2   0.535094
2  key1           b
   key2         one
   data1  -0.749546
   data2   -1.07171
3  key1           b
   key2         two
   data1   -2.19675
   data2   0.524351
4  key1           a
   key2         one
   data1   0.264189
   data2    2.62142
dtype: object
```

In [22]:

```
df.T
```

Out[22]:

	0	1	2	3	4
key1	a	a	b	b	a
key2	one	two	one	two	one
data1	-0.699511	-0.345863	-0.749546	-2.19675	0.264189
data2	-0.388558	0.535094	-1.07171	0.524351	2.62142

In [24]:

```
# Here we grouped the data using two keys, and the resulting Series now has a hierarchical
    # index consisting of the unique pairs of keys observed:

means.unstack()
```

Out[24]:

key2	one	two
key1		
a	-0.217661	-0.345863
b	-0.749546	-2.196750

In [25]:

```
# In this example, the group keys are all Series, though they could be any arrays of the r
ight length:

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

Out[25]:

```
California  2005   -0.345863
            2006   -0.749546
Ohio        2005   -1.448130
            2006    0.264189
Name: data1, dtype: float64
```

In [26]:

```
# Frequently the grouping information is found in the same DataFrame as the data you want
 to work on.
    # In that case, you can pass column names (whether those are strings, numbers, or othe
r Python objects)
    # as the group keys:

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

Out[26]:

	data1	data2
key1		
a	-0.260395	0.922651
b	-1.473148	-0.273681

In [27]:

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

Out[27]:

		data1	data2
key1	**key2**		
a	**one**	-0.217661	1.116430
	two	-0.345863	0.535094
b	**one**	-0.749546	-1.071714
	two	-2.196750	0.524351

In [28]:

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

Out[28]:

		data1	data2
key1	**key2**		
a	**one**	-0.217661	1.116430
	two	-0.345863	0.535094
b	**one**	-0.749546	-1.071714
	two	-2.196750	0.524351

In [29]:

```
# You may have noticed in the first case df.groupby('key1').mean() that there is no key2 c
olumn in the result.
    # Because df['key2'] is not numeric data, it is said to be a nuisance column,
    # which is therefore excluded from the result.
    # By default, all of the numeric columns are aggregated,
    # though it is possible to filter down to a subset, as you’ll see soon.

# Regardless of the objective in using groupby, a generally useful GroupBy method is size,
    # which returns a Series containing group sizes:


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

Out[29]:

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

In [ ]:

```
# Take note that any missing values in a group key will be excluded from the result.
```

In [30]:

```
# Iterating over the groups

# The GroupBy object supports iteration, generating a sequence of 2-tuples containing
    # the group name along with the chunk of data.

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

```
a
  key1 key2     data1     data2
0    a  one -0.699511 -0.388558
1    a  two -0.345863  0.535094
4    a  one  0.264189  2.621417
b
  key1 key2     data1     data2
2    b  one -0.749546 -1.071714
3    b  two -2.196750  0.524351
```

In [31]:

```
# for multiple keys

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

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

```
('a', 'one')
  key1 key2     data1     data2
0    a  one -0.699511 -0.388558
4    a  one  0.264189  2.621417
('a', 'two')
  key1 key2     data1     data2
1    a  two -0.345863  0.535094
('b', 'one')
  key1 key2     data1     data2
2    b  one -0.749546 -1.071714
('b', 'two')
  key1 key2    data1     data2
3    b  two -2.19675  0.524351
```

In [32]:

```
# computing a dict of the data pieces as a one-liner

# Of course, you can choose to do whatever you want with the pieces of data.
    # A recipe you may find useful is computing a dict of the data pieces as a one-liner:

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

Out[32]:

```
{'a':   key1 key2     data1     data2
 0    a  one -0.699511 -0.388558
 1    a  two -0.345863  0.535094
 4    a  one  0.264189  2.621417,
 'b':   key1 key2     data1     data2
 2    b  one -0.749546 -1.071714
 3    b  two -2.196750  0.524351}
```

In [33]:

```
pieces['a']
```

Out[33]:

	key1	key2	data1	data2
0	a	one	-0.699511	-0.388558
1	a	two	-0.345863	0.535094
4	a	one	0.264189	2.621417

In [34]:

```
pieces['b']
```

Out[34]:

	key1	key2	data1	data2
2	b	one	-0.749546	-1.071714
3	b	two	-2.196750	0.524351

In [35]:

```
# 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:

df.dtypes
```

Out[35]:

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

In [38]:

```
grouped1 = df.groupby(df.dtypes, axis=1)
grouped1
```

Out[38]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000039F1B05448>
```

In [39]:

```
grouped1.size()
```

Out[39]:

```
float64    2
object     2
dtype: int64
```

In [40]:

```
# We can print out the groups like so:

for dtype, group in grouped:
    print(dtype)
    print(group)
```

```
a
0   -0.699511
1   -0.345863
4    0.264189
Name: data1, dtype: float64
b
2   -0.749546
3   -2.196750
Name: data1, dtype: float64
```

In [41]:

```
for dtype, group in grouped1:
    print(dtype)
    print(group)
```

```
float64
      data1     data2
0 -0.699511 -0.388558
1 -0.345863  0.535094
2 -0.749546 -1.071714
3 -2.196750  0.524351
4  0.264189  2.621417
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

In [42]:

```
# 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.

# This means that:

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


# are syntactic sugar for:


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

Out[42]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000039F1862808>
```

In [43]:

```
# Dataframes

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

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

Out[43]:

		data2
key1	**key2**	
a	**one**	1.116430
	two	0.535094
b	**one**	-1.071714
	two	0.524351

In [45]:

```
# Series

# The object returned by this indexing operation is a grouped DataFrame if a list or
    # array is passed or a grouped Series if only a single column name is passed as a scal
ar:

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

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x00000039F1B0DB48>
```

Out[45]:

```
key1  key2
a     one      1.116430
      two      0.535094
b     one     -1.071714
      two      0.524351
Name: data2, dtype: float64
```

In [48]:

```
# Grouping with Dictionaries and Series

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

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

people
```

Out[48]:

	a	b	c	d	e
Joe	0.061253	-0.219182	0.258375	-1.033663	-0.642642
Steve	0.818724	-0.025076	1.151003	0.089388	0.024214
Wes	1.197966	-0.412310	-1.040145	-1.106289	-0.742792
Jim	-0.044602	0.463223	1.254485	0.765563	0.906828
Travis	-0.681018	-0.420505	0.081755	0.707100	0.900215

In [49]:

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

Out[49]:

	a	b	c	d	e
Joe	0.061253	-0.219182	0.258375	-1.033663	-0.642642
Steve	0.818724	-0.025076	1.151003	0.089388	0.024214
Wes	1.197966	NaN	NaN	-1.106289	-0.742792
Jim	-0.044602	0.463223	1.254485	0.765563	0.906828
Travis	-0.681018	-0.420505	0.081755	0.707100	0.900215

In [52]:

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

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




# Now, you could construct an array from this dict to pass to groupby,
    # but instead we can just pass the dict (I included the key 'f' to highlight
    # that unused grouping keys are OK):

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

print(people)
print()
print(by_column)
```

```
               a         b         c         d         e
Joe     0.061253 -0.219182  0.258375 -1.033663 -0.642642
Steve   0.818724 -0.025076  1.151003  0.089388  0.024214
Wes     1.197966       NaN       NaN -1.106289 -0.742792
Jim    -0.044602  0.463223  1.254485  0.765563  0.906828
Travis -0.681018 -0.420505  0.081755  0.707100  0.900215

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000039F1B6C348>
```

In [56]:

```
print(by_column.sum())

print()

print(by_column.mean())
```

```
            blue       red
Joe    -0.775288 -0.800571
Steve   1.240391  0.817862
Wes    -1.106289  0.455174
Jim     2.020048  1.325449
Travis  0.788855 -0.201309

            blue       red
Joe    -0.387644 -0.266857
Steve   0.620196  0.272621
Wes    -1.106289  0.227587
Jim     1.010024  0.441816
Travis  0.394428 -0.067103
```

In [58]:

```
# The same functionality holds for Series, which can be viewed as a fixed-size mapping:

map_series = pd.Series(mapping)
map_series
```

Out[58]:

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

In [59]:

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

Out[59]:

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

In [60]:

```
# 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.
    # More concretely, consider the example 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:

people.groupby(len).sum()  # grouping by the length of the names
```

Out[60]:

	a	b	c	d	e
3	1.214617	0.244041	1.512859	-1.374389	-0.478606
5	0.818724	-0.025076	1.151003	0.089388	0.024214
6	-0.681018	-0.420505	0.081755	0.707100	0.900215

In [61]:

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

key_list = ['one', 'one', 'one', 'two', 'two']
people.groupby([len, key_list]).min()  # Mixing functions with arrays, dicts, or Series
```

Out[61]:

		a	b	c	d	e
3	**one**	0.061253	-0.219182	0.258375	-1.106289	-0.742792
	two	-0.044602	0.463223	1.254485	0.765563	0.906828
5	**one**	0.818724	-0.025076	1.151003	0.089388	0.024214
6	**two**	-0.681018	-0.420505	0.081755	0.707100	0.900215

In [67]:

```
# 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:


columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
    [1, 3, 5, 1, 3]],
    names=['cty', 'tenor'])
print(columns)

print()
print()

hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)
print(hier_df)
```

```
MultiIndex([('US', 1),
            ('US', 3),
            ('US', 5),
            ('JP', 1),
            ('JP', 3)],
           names=['cty', 'tenor'])


cty          US                            JP
tenor         1         3         5         1         3
0      1.173735 -0.159624  0.656455 -0.989766 -0.529011
1      0.923138  0.036186  1.547744 -0.285430 -0.267631
2      0.232222 -1.022996  0.478855 -1.479527  0.163846
3     -0.574041  0.017466 -1.255832 -0.247150  0.242282
```

In [68]:

```
# To group by level, pass the level number or name using the level keyword:

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

Out[68]:

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

In [71]:

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

Out[71]:

tenor	1	3	5
0	2	2	1
1	2	2	1
2	2	2	1
3	2	2	1

In [82]:

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

Out[82]:

cty	JP	US
0	-0.759389	0.556855
1	-0.276530	0.835690
2	-0.657841	-0.103973
3	-0.002434	-0.604136

In [83]:

```
hier_df.groupby(level='tenor', axis=1).mean()
```

Out[83]:

tenor	1	3	5
0	0.091984	-0.344317	0.656455
1	0.318854	-0.115722	1.547744
2	-0.623652	-0.429575	0.478855
3	-0.410595	0.129874	-1.255832

In [86]:

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

Out[86]:

			count	mean	std	min	25%	50%	75%	max
cty	**cty**	**tenor**								
JP	**JP**	**1**	4.0	-0.750468	0.593964	-1.479527	-1.112206	-0.637598	-0.275860	-0.247150
		3	4.0	-0.097629	0.364646	-0.529011	-0.332976	-0.051892	0.183455	0.242282
US	**US**	**1**	4.0	0.438763	0.783844	-0.574041	0.030656	0.577680	0.985787	1.173735
		3	4.0	-0.282242	0.501655	-1.022996	-0.375467	-0.071079	0.022146	0.036186
		5	4.0	0.356805	1.172408	-1.255832	0.045183	0.567655	0.879277	1.547744

In [87]:

```
hier_df.groupby(level='tenor', axis=1).describe()
```

Out[87]:

			count	mean	std	min	25%	50%	75%	m
tenor	**cty**	**tenor**								
1	**US**	**1**	4.0	0.438763	0.783844	-0.574041	0.030656	0.577680	0.985787	1.1737
	JP	**1**	4.0	-0.750468	0.593964	-1.479527	-1.112206	-0.637598	-0.275860	-0.2471
3	**US**	**3**	4.0	-0.282242	0.501655	-1.022996	-0.375467	-0.071079	0.022146	0.0361
	JP	**3**	4.0	-0.097629	0.364646	-0.529011	-0.332976	-0.051892	0.183455	0.2422
5	**US**	**5**	4.0	0.356805	1.172408	-1.255832	0.045183	0.567655	0.879277	1.5477

In [ ]:

```
# Data Aggregation

# Aggregations refer to any data transformation that produces scalar values from arrays.
    # The preceding examples have used several of them, including mean, count, min, and su
m.
    # You may wonder what is going on when you invoke mean() on a GroupBy object.
    # Many common aggregations, such as those found in Table 10-1, have optimized implemen
tations.
    # However, you are not limited to only this set of methods.

# 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 va
riance
      #      min, max                Minimum and maximum of non-NA values prod Product of n
on-NA values
      #      first, last             First and last non-NA values

# You can use aggregations of your own devising and additionally call any method that
    # is also defined on the grouped object. For example, you might recall that quantile
    # computes sample quantiles of a Series or a DataFrame’s columns.

    # 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,
    # and then assembles those results together into the result object.
```

In [88]:

```
df
```

Out[88]:

	key1	key2	data1	data2
0	a	one	-0.699511	-0.388558
1	a	two	-0.345863	0.535094
2	b	one	-0.749546	-1.071714
3	b	two	-2.196750	0.524351
4	a	one	0.264189	2.621417

In [91]:

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

Out[91]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000039F2156C08>
```

In [92]:

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

Out[92]:

```
key1
a    0.142179
b   -0.894266
Name: data1, dtype: float64
```

In [98]:

```
# 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 makes it convenient to perform bucket
    # or quantile analysis on a dataset.

# Consider a simple random dataset and an equal-length bucket categorization using cut:


frame = pd.DataFrame({'data1': np.random.randn(1000),
    'data2': np.random.randn(1000)})
frame
```

Out[98]:

	data1	data2
0	-0.715645	0.823377
1	0.715335	-1.178312
2	-1.191247	0.205116
3	0.059871	1.524632
4	0.473645	0.742339
...	...	...
995	0.003551	0.153862
996	-0.001380	0.987662
997	-0.193659	-0.668325
998	-1.016595	-0.978162
999	-2.840384	-0.171102

1000 rows × 2 columns

In [99]:

```
print(frame)
```

```
        data1     data2
0   -0.715645  0.823377
1    0.715335 -1.178312
2   -1.191247  0.205116
3    0.059871  1.524632
4    0.473645  0.742339
..        ...       ...
995  0.003551  0.153862
996 -0.001380  0.987662
997 -0.193659 -0.668325
998 -1.016595 -0.978162
999 -2.840384 -0.171102

[1000 rows x 2 columns]
```

In [100]:

```
frame.unstack()
```

Out[100]:

```
data1  0      -0.715645
       1       0.715335
       2      -1.191247
       3       0.059871
       4       0.473645
                 ...
data2  995     0.153862
       996     0.987662
       997    -0.668325
       998    -0.978162
       999    -0.171102
Length: 2000, dtype: float64
```

In [102]:

```
frame.T
```

Out[102]:

	0	1	2	3	4	5	6	7	
data1	-0.715645	0.715335	-1.191247	0.059871	0.473645	-0.776308	-0.048923	-0.714903	-2.33
data2	0.823377	-1.178312	0.205116	1.524632	0.742339	1.582948	-0.755365	-0.086841	0.49

2 rows × 1000 columns

In [104]:

```
quartiles = pd.cut(frame.data1, 4)
quartiles
```

Out[104]:

```
0       (-1.294, 0.253]
1          (0.253, 1.8]
2       (-1.294, 0.253]
3       (-1.294, 0.253]
4          (0.253, 1.8]
             ...
995     (-1.294, 0.253]
996     (-1.294, 0.253]
997     (-1.294, 0.253]
998     (-1.294, 0.253]
999    (-2.847, -1.294]
Name: data1, Length: 1000, dtype: category
Categories (4, interval[float64]): [(-2.847, -1.294] < (-1.294, 0.253] < (0.2
53, 1.8] < (1.8, 3.347]]
```

In [106]:

```
quartiles.head(10)
```

Out[106]:

```
0     (-1.294, 0.253]
1        (0.253, 1.8]
2     (-1.294, 0.253]
3     (-1.294, 0.253]
4        (0.253, 1.8]
5     (-1.294, 0.253]
6     (-1.294, 0.253]
7     (-1.294, 0.253]
8    (-2.847, -1.294]
9        (0.253, 1.8]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.847, -1.294] < (-1.294, 0.253] < (0.2
53, 1.8] < (1.8, 3.347]]
```

In [108]:

```
quartiles[:10]
```

Out[108]:

```
0     (-1.294, 0.253]
1        (0.253, 1.8]
2     (-1.294, 0.253]
3     (-1.294, 0.253]
4        (0.253, 1.8]
5     (-1.294, 0.253]
6     (-1.294, 0.253]
7     (-1.294, 0.253]
8    (-2.847, -1.294]
9        (0.253, 1.8]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.847, -1.294] < (-1.294, 0.253] < (0.2
53, 1.8] < (1.8, 3.347]]
```

In [110]:

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

grouped = frame.data2.groupby(quartiles)
grouped
```

Out[110]:

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x00000039F21CCAC8>
```

In [112]:

```
# These were equal-length buckets; to compute equal-size buckets based on sample quantile
s, use qcut.
    # I’ll pass labels=False to just get quantile numbers:

# Return quantile numbers

grouping = pd.qcut(frame.data1, 10, labels=False)
grouping
```

Out[112]:

```
0      2
1      7
2      1
3      5
4      6
      ..
995    5
996    5
997    4
998    1
999    0
Name: data1, Length: 1000, dtype: int64
```

In [118]:

```
# Filling Missing Values with Group-Specific Values

# When cleaning up missing data, in some cases you will replace data observations using dr
opna,
    # but in others you may want to impute (fill in) the null (NA) values using a fixed va
lue
    # or some value derived from the data. fillna is the right tool to use;
    # for example, here I fill in NA values with the mean:

s = pd.Series(np.random.randn(6))
s
```

Out[118]:

```
0   -0.562639
1   -1.130607
2   -0.069880
3    1.280894
4    0.224934
5   -0.700833
dtype: float64
```

In [119]:

```
s[::2] = np.nan  # filling s with Nan values at intervals of 2
s
```

Out[119]:

```
0         NaN
1   -1.130607
2         NaN
3    1.280894
4         NaN
5   -0.700833
dtype: float64
```

In [120]:

```
s.fillna(s.mean())
```

Out[120]:

```
0   -0.183515
1   -1.130607
2   -0.183515
3    1.280894
4   -0.183515
5   -0.700833
dtype: float64
```

In [127]:

```
# Making the fill value to vary by group

# Suppose you need the fill value to vary by group. One way to do this is to group the
    # data and use apply with a function that calls fillna on each data chunk.

# Here is some sample data on US states divided into eastern and western regions:


states = ['Ohio', 'New York', 'Vermont', 'Florida','Oregon', 'Nevada', 'California', 'Idah
o']

group_key = ['East'] * 4 + ['West'] * 4

data = pd.Series(np.random.randn(8), index=states)

print(data)
print(group_key)
```

```
Ohio          0.105849
New York     -1.428066
Vermont      -0.453576
Florida      -0.124419
Oregon        0.493002
Nevada        0.361317
California   -1.791414
Idaho         2.516672
dtype: float64
['East', 'East', 'East', 'East', 'West', 'West', 'West', 'West']
```

In [123]:

```
# Let’s set some values in the data to be missing:

data[['Vermont', 'Nevada', 'Idaho']] = np.nan
data
```

Out[123]:

```
Ohio          0.304550
New York      0.872126
Vermont            NaN
Florida      -0.528893
Oregon        0.760108
Nevada             NaN
California    0.069828
Idaho              NaN
dtype: float64
```

In [124]:

```
data.groupby(group_key).mean()
```

Out[124]:

```
East    0.215928
West    0.414968
dtype: float64
```

In [129]:

```
# We can fill the NA values using the group means like so:

fill_mean = lambda g: g.fillna(g.mean())
data.groupby(group_key).apply(fill_mean)
```

Out[129]:

```
Ohio          0.105849
New York     -1.428066
Vermont      -0.453576
Florida      -0.124419
Oregon        0.493002
Nevada        0.361317
California   -1.791414
Idaho         2.516672
dtype: float64
```

In [130]:

```
# Using predefined fill values that vary by group

# In another case, you might have predefined fill values in your code that vary by group.
    # Since the groups have a name attribute set internally, we can use that:

fill_values = {'East': 0.5, 'West': -1}
fill_func = lambda g: g.fillna(fill_values[g.name])
data.groupby(group_key).apply(fill_func)
```

Out[130]:

```
Ohio          0.105849
New York     -1.428066
Vermont      -0.453576
Florida      -0.124419
Oregon        0.493002
Nevada        0.361317
California   -1.791414
Idaho         2.516672
dtype: float64
```

In [132]:

```
# Random Sampling and Permutation

# Drawing a random sample (with or without replacement) from alarge dataset,
    # eg:constructing a deck of English-style playing cards

# Suppose you wanted to draw a random sample (with or without replacement) from a large da
taset
    # for Monte Carlo simulation purposes or some other application.
    # There are a number of ways to perform the “draws”; here we use the sample method for
Series.

# Hearts, Spades, Clubs, Diamonds
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(range(2, 11)) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
deck
```

Out[132]:

```
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
AS      1
2S      2
3S      3
4S      4
5S      5
6S      6
7S      7
8S      8
9S      9
10S    10
JS     10
KS     10
QS     10
AC      1
2C      2
3C      3
4C      4
5C      5
6C      6
7C      7
8C      8
9C      9
10C    10
JC     10
KC     10
QC     10
AD      1
2D      2
3D      3
4D      4
5D      5
6D      6
7D      7
8D      8
9D      9
10D    10
JD     10
KD     10
QD     10
dtype: int64
```

In [133]:

```
# So now we have a Series of length 52 whose index contains card names and values are the
 ones used
    # in Blackjack and other games (to keep things simple, I just let the ace 'A' be 1):

deck[:13]
```

Out[133]:

```
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
dtype: int64
```

In [134]:

```
# Drawing a hand of five cards from the deck could be written as:

def draw(deck, n=5):
    return deck.sample(n)


draw(deck)
```

Out[134]:

```
3C     3
JS    10
JC    10
JD    10
8H     8
dtype: int64
```

In [136]:

```
# Drawing two random cards from each suit

# Suppose you wanted two random cards from each suit.
    # Because the suit is the last character of each card name, we can group based on this
and use apply:

get_suit = lambda card: card[-1] # last letter is suit
deck.groupby(get_suit).apply(draw, n=2)
```

Out[136]:

```
C  7C      7
   5C      5
D  10D    10
   9D      9
H  7H      7
   5H      5
S  AS      1
   KS     10
dtype: int64
```

In [137]:

```
# alternatively,we could write:

deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
```

Out[137]:

```
8C      8
7C      7
JD     10
2D      2
JH     10
10H    10
8S      8
6S      6
dtype: int64
```

In [144]:

```
# Group Weighted Average and Correlation

# Under the split-apply-combine paradigm of groupby, operations between columns in a DataF
rame or two Series,
    # such as a group weighted average, are possible.

# As an example, take this dataset containing group keys, values, and some weights:


df = pd.DataFrame({'category': ['a', 'a', 'a', 'a','b', 'b', 'b', 'b'],
                   'data': np.random.randn(8),
                   'weights': np.random.rand(8)})

df
```

Out[144]:

	category	data	weights
0	a	0.108480	0.278910
1	a	-0.924634	0.059491
2	a	-0.639827	0.341055
3	a	-0.271886	0.732100
4	b	0.906096	0.881823
5	b	-0.313899	0.018645
6	b	1.566793	0.745821
7	b	0.003243	0.425661

In [145]:

```
# The group weighted average by category would then be:

grouped = df.groupby('category')
get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
grouped.apply(get_wavg)
```

Out[145]:

```
category
a   -0.313141
b    0.947461
dtype: float64
```

In [156]:

```
data
```

Out[156]:

```
Ohio          0.105849
New York     -1.428066
Vermont      -0.453576
Florida      -0.124419
Oregon        0.493002
Nevada        0.361317
California   -1.791414
Idaho         2.516672
dtype: float64
```

In [164]:

```
# Cross-Tabulations

# A cross-tabulation (or crosstab for short) is a special case of a pivot table
    # that computes group frequencies.

# Here is an example:

data2 = pd.DataFrame({'Sample': ['1', '2', '3', '4','5', '6', '7', '8', '9','10'],
                   'Nationality': ['USA', 'Japan', 'USA', 'Japan', 'Japan', 'Japan','USA', 'USA','Japan','USA'],
                   'Handedness': ['Right-handed', 'Left-handed', 'Right-handed', 'Right-handed', 'Left-handed',
                                  'Right-handed', 'Right-handed', 'Left-handed', 'Right-handed', 'Right-handed']})
data2
```

Out[164]:

	Sample	Nationality	Handedness
0	1	USA	Right-handed
1	2	Japan	Left-handed
2	3	USA	Right-handed
3	4	Japan	Right-handed
4	5	Japan	Left-handed
5	6	Japan	Right-handed
6	7	USA	Right-handed
7	8	USA	Left-handed
8	9	Japan	Right-handed
9	10	USA	Right-handed

In [165]:

```
# As part of some survey analysis, we might want to summarize this data by nationality and handedness.
    # You could use pivot_table to do this, but the pandas.crosstab function can be more convenient:

pd.crosstab(data2.Nationality, data2.Handedness, margins=True)  #cross_tab
```

Out[165]:

Handedness	Left-handed	Right-handed	All
Nationality			
Japan	2	3	5
USA	1	4	5
All	3	7	10

The End