Lecture 5-3

Pandas: Summaries with Pivot Tables and Group by

Week 5 Friday

Miles Chen, PhD

Adapted from Python Data Science Handbook by Jake VanderPlas and Python for Data Analysis by Wes McKinney

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Some important data transformation tools

Multi Index, Hierarchical Indexing

In [2]:

```
np.random.seed(1)
# manual multi-index creation:
# provide the index argument with a list of lists
data = pd.Series(np.random.randint(10, size = 9),
                 index = [['a','a','a','b','b','b','c','c','c'],
                          [ 1 , 2 , 3 , 1 , 2 , 3 , 1 , 2 , 3 ]])
```

In [3]:

```
data
```

Out[3]:

```
a  1    5
   2    8
   3    9
b  1    5
   2    0
   3    0
c  1    1
   2    7
   3    6
dtype: int32
```

In [4]:

```
data.index
```

Out[4]:

```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 2),
            ('b', 3),
            ('c', 1),
            ('c', 2),
```

```
 ('c', 3)],
)
```

In [5]:

```
# select via the outer index
data.loc['b']
```

Out[5]:

```
1    5
2    0
3    0
dtype: int32
```

In [6]:

```
# select via the inner index
data.loc[:,2]
```

Out[6]:

```
a    8
b    0
c    7
dtype: int32
```

In [7]:

```
type(data.loc[:,2])
```

Out[7]:

```
pandas.core.series.Series
```

In [8]:

```
data.loc[:,2].index
```

Out[8]:

```
Index(['a', 'b', 'c'], dtype='object')
```

In [9]:

```
# the unstack function returns a new DataFrame where the values have been unstacked
# similar to tidyr's spread()/pivot_wider function in R
data.unstack()
```

Out[9]:

	1	2	3
a	5	8	9
b	5	0	0
c	1	7	6

In [10]:

```
# after unstacking, the index is no longer a multi index
data.unstack().index
```

Out[10]:

```
Index(['a', 'b', 'c'], dtype='object')
```

In [11]:

```
data.unstack().shape
```

Out[11]:

```
(3, 3)
```

In [12]:

```
# the inverse operation of unstack() is stack()
# applying both of these functions will return the same series
data.unstack().stack()
```

Out[12]:

```
a  1    5
   2    8
   3    9
b  1    5
   2    0
   3    0
c  1    1
   2    7
   3    6
dtype: int32
```

In [13]:

```
# you can swap the levels of the multi index using swaplevel
data.swaplevel()
```

Out[13]:

```
1  a    5
2  a    8
3  a    9
1  b    5
2  b    0
3  b    0
1  c    1
2  c    7
3  c    6
dtype: int32
```

In [14]:

```
# the .loc accessors work as expected
data.swaplevel().loc[:,'a']
```

Out[14]:

```
1    5
2    8
3    9
dtype: int32
```

In [15]:

```
# swaplevel will keep the original order
# you may want to sort based on the new swapped index levels
# you must save the output as data remains unchanged
data.swaplevel().sort_index()
```

Out[15]:

```
1  a    5
   b    5
   c    1
2  a    8
   b    0
   c    7
3  a    9
   b    0
   c    6
dtype: int32
```

In [16]:

```
print(data)
```

```
a  1    5
   2    8
   3    9
b  1    5
   2    0
   3    0
c  1    1
   2    7
```

```
   3    6
dtype: int32
```

In [17]:

```
data.swaplevel().unstack()
```

Out[17]:

	a	b	c
1	5	5	1
2	8	0	7
3	9	0	6

In [18]:

```
# compare to:
data.unstack()
```

Out[18]:

	1	2	3
a	5	8	9
b	5	0	0
c	1	7	6

In [19]:

```
# summing and other aggregate functions can be performed on an index-based level
# calling sum() on a series, will sum the whole series
data.sum()
```

Out[19]:

```
41
```

In [20]:

```
# you can call sum on the level 0 (the first level of the index)
# we get sums for each value in the first level of the index
data.sum(level = 0)
```

Out[20]:

```
a    22
b     5
c    14
dtype: int32
```

In [21]:

```
data.sum(level = 1)
```

Out[21]:

```
1    11
2    15
3    15
dtype: int32
```

Reshaping and Pivoting Data

In [22]:

```
data = pd.DataFrame(np.arange(6).reshape((2, 3)),
                    index   = pd.Index(['alpha', 'beta'], name='letter'),
                    columns= pd.Index(['one', 'two', 'three'], name = 'number'))
data
```

Out[22]:

number	one	two	three
letter			
alpha	0	1	2
beta	3	4	5

In [23]:

```
data.stack()  # creates a multi-index
```

Out[23]:

```
letter  number
alpha   one       0
        two       1
        three     2
beta    one       3
        two       4
        three     5
dtype: int32
```

In [24]:

```
data.stack().unstack()  # unstack undoes the creation of the stacks
```

Out[24]:

number	one	two	three
letter			
alpha	0	1	2
beta	3	4	5

In [25]:

```
data.stack().unstack(0) # you can specify how the unstacking should be done
# here we specify that we should unstack the first level of the multi-index
```

Out[25]:

letter	alpha	beta
number		
one	0	3
two	1	4
three	2	5

In [26]:

```
data.stack().unstack('letter')
# you can specify the unstacking by the index level name
```

Out[26]:

number	letter alpha	beta
one	0	3
two	1	4
three	2	5

In [27]:

```
data.stack().unstack('number')
```

Out[27]:

letter	number one	two	three
alpha	0	1	2
beta	3	4	5

Unstacking can introduce missing values

In [28]:

```
s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
data2 = pd.concat([s1, s2], keys=['one', 'two'])
# using the argument keys when concat series will produce a multi-index
data2
```

Out[28]:

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

In [29]:

```
data2.unstack()
```

Out[29]:

	a	b	c	d	e
one	0.0	1.0	2.0	3.0	NaN
two	NaN	NaN	4.0	5.0	6.0

In [30]:

```
data2.unstack().stack() # stack() will filter out missing values
```

Out[30]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64
```

In [31]:

```
data2.unstack().stack(dropna = False) # you can force stack to keep the NaNs
```

Out[31]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
```

```
     e    6.0
dtype: float64
```

Small example data wrangling

In [32]:

```
data = pd.read_csv('macrodata.csv')
```

https://www.statsmodels.org/dev/datasets/generated/macrodata.html

In [33]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 203 entries, 0 to 202
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   year      203 non-null    float64
 1   quarter   203 non-null    float64
 2   realgdp   203 non-null    float64
 3   realcons  203 non-null    float64
 4   realinv   203 non-null    float64
 5   realgovt  203 non-null    float64
 6   realdpi   203 non-null    float64
 7   cpi       203 non-null    float64
 8   m1        203 non-null    float64
 9   tbilrate  203 non-null    float64
 10  unemp     203 non-null    float64
 11  pop       203 non-null    float64
```

```
 12  infl      203 non-null    float64
 13  realint   203 non-null    float64
dtypes: float64(14)
memory usage: 22.3 KB
```

In [34]:

```
data.head()
```

Out[34]:

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	m1	tbilrate	unemp	pop	infl	realint
0	1959.0	1.0	2710.349	1707.4	286.898	470.045	1886.9	28.98	139.7	2.82	5.8	177.146	0.00	0.00
1	1959.0	2.0	2778.801	1733.7	310.859	481.301	1919.7	29.15	141.7	3.08	5.1	177.830	2.34	0.74
2	1959.0	3.0	2775.488	1751.8	289.226	491.260	1916.4	29.35	140.5	3.82	5.3	178.657	2.74	1.09
3	1959.0	4.0	2785.204	1753.7	299.356	484.052	1931.3	29.37	140.0	4.33	5.6	179.386	0.27	4.06
4	1960.0	1.0	2847.699	1770.5	331.722	462.199	1955.5	29.54	139.6	3.50	5.2	180.007	2.31	1.19

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.PeriodIndex.html

In [35]:

```
# We can create a time based index of periods consisting of the year and quarter
periods = pd.PeriodIndex(year = data.year, quarter = data.quarter, name = 'date')
```

In [36]:

```
periods
```

Out[36]:

```
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '19
60Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '20
08Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', name='date', length=203, freq
='Q-DEC')
```

In [37]:

```
columns = pd.Index(['realgdp', 'infl', 'unemp'], name = 'item')
columns
```

Out[37]:

```
Index(['realgdp', 'infl', 'unemp'], dtype='object', name='item')
```

In [38]:

```
data = data.reindex(columns = columns) # forces columns to conform to the column index we specified
```

In [39]:

```
data.head(10)
```

Out[39]:

item	realgdp	infl	unemp
0	2710.349	0.00	5.8
1	2778.801	2.34	5.1
2	2775.488	2.74	5.3
3	2785.204	0.27	5.6
4	2847.699	2.31	5.2
5	2834.390	0.14	5.2
6	2839.022	2.70	5.6
7	2802.616	1.21	6.3
8	2819.264	-0.40	6.8
9	2872.005	1.47	7.0

In [40]:

```
periods.to_timestamp('D','start')  # changes 1959Q1 to a date: the start date of Q1 of 1959: 1959-01-01
```

Out[40]:

```
DatetimeIndex(['1959-01-01', '1959-04-01', '1959-07-01', '1959-10-
01',
               '1960-01-01', '1960-04-01', '1960-07-01', '1960-10-
01',
               '1961-01-01', '1961-04-01',
               ...
               '2007-04-01', '2007-07-01', '2007-10-01', '2008-01-
01',
               '2008-04-01', '2008-07-01', '2008-10-01', '2009-01-
01',
               '2009-04-01', '2009-07-01'],
              dtype='datetime64[ns]', name='date', length=203, fre
q='QS-OCT')
```

In [41]:

```
# the current index is just integers, and we want to replace it
data.index
```

Out[41]:

```
RangeIndex(start=0, stop=203, step=1)
```

In [42]:

```
# specify a new index directly
data.index = periods.to_timestamp('D','start')
```

In [43]:

```
data.head()
```

Out[43]:

item	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

In [44]:

```
data.stack().head(10)  # stack creates a series
```

Out[44]:

```
date        item
1959-01-01  realgdp    2710.349
            infl          0.000
            unemp         5.800
1959-04-01  realgdp    2778.801
            infl          2.340
            unemp         5.100
1959-07-01  realgdp    2775.488
            infl          2.740
            unemp         5.300
1959-10-01  realgdp    2785.204
dtype: float64
```

In [45]:

```
data.stack().reset_index().head()
# calling reset index turns the current index into a new column and creates a new index
```

Out[45]:

	date	item	0
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340

In [46]:

```
data.stack().reset_index().index
```

Out[46]:

```
RangeIndex(start=0, stop=609, step=1)
```

In [47]:

```
ldata = data.stack().reset_index().rename(columns = {0: 'value'})
# rename changes the column title '0' to 'value'
ldata.head(10)
```

Out[47]:

	date	item	value
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340
5	1959-04-01	unemp	5.100
6	1959-07-01	realgdp	2775.488
7	1959-07-01	infl	2.740
8	1959-07-01	unemp	5.300
9	1959-10-01	realgdp	2785.204

In [48]:

```
# unstack doesn't work, because the stacking and unstacking is powered by multi-index
ldata.unstack()
```

Out[48]:

```
date   0       1959-01-01 00:00:00
       1       1959-01-01 00:00:00
       2       1959-01-01 00:00:00
       3       1959-04-01 00:00:00
       4       1959-04-01 00:00:00
```

```
                    ...
value  604                  3.37
       605                   9.2
       606              12990.341
       607                  3.56
       608                   9.6
Length: 1827, dtype: object
```

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.pivot.html

In [49]:

```
# if the data is in 'long' form, you can change it to 'wide' form with pivot
ldata.pivot('date','item','value').head()
```

Out[49]:

item	infl	realgdp	unemp
date			
1959-01-01	0.00	2710.349	5.8
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2

In [50]:

```
# if the data is in 'long' form, you can change it to 'wide' form with pivot
ldata.pivot(index = 'item',columns = 'date',values = 'value').head()
```

Out[50]:

date	1959-01-01	1959-04-01	1959-07-01	1959-10-01	1960-01-01	1960-04-01	1960-07-01	1960-10-01	1961-01-01	1961-04-01
item										
infl	0.000	2.340	2.740	0.270	2.310	0.14	2.700	1.210	-0.400	1.470
realgdp	2710.349	2778.801	2775.488	2785.204	2847.699	2834.39	2839.022	2802.616	2819.264	2872.005
unemp	5.800	5.100	5.300	5.600	5.200	5.20	5.600	6.300	6.800	7.000

3 rows × 203 columns

In [51]:

```
data.head()
```

Out[51]:

item	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

Group By

In [52]:

```
np.random.seed(1)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randint(20, size = 5),
                   'data2' : np.random.randint(20, size = 5)})
df
```

Out[52]:

	key1	key2	data1	data2
0	a	one	5	11
1	a	two	11	5
2	b	one	12	15
3	b	two	8	0
4	a	one	9	16

In [53]:

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

Out[53]:

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x0000026BC48E8288>
```

In [54]:

```
grouped.mean()
```

Out[54]:

```
key1
a     8.333333
b    10.000000
Name: data1, dtype: float64
```

In [55]:

```
df
```

Out[55]:

	key1	key2	data1	data2
0	a	one	5	11
1	a	two	11	5
2	b	one	12	15
3	b	two	8	0
4	a	one	9	16

In [56]:

```
df.groupby(df['key1']).mean()
# if you don't specify the column, it'll apply the function to the entire dataframe
```

Out[56]:

key1	data1	data2
a	8.333333	10.666667
b	10.000000	7.500000

In [57]:

```
df
```

Out[57]:

	key1	key2	data1	data2
0	a	one	5	11
1	a	two	11	5
2	b	one	12	15
3	b	two	8	0
4	a	one	9	16

In [58]:

```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
# means has a multi-index
```

Out[58]:

```
key1  key2
a     one     7
      two    11
b     one    12
      two     8
Name: data1, dtype: int32
```

In [59]:

```
# with the multi-index, you can unstack
means.unstack()
```

Out[59]:

key2	one	two
key1		
a	7	11

key2	one	two
key1		
b	12	8

In [60]:

```
df
```

Out[60]:

	key1	key2	data1	data2
0	a	one	5	11
1	a	two	11	5
2	b	one	12	15
3	b	two	8	0
4	a	one	9	16

In [61]:

```
# you can perform group by on Series that are not in the dataframe, but are of the correct length
states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
years = np.array([2005, 2005, 2006, 2005, 2006])
df['data1'].groupby([states, years]).mean()
```

Out[61]:

```
California  2005    11.0
            2006    12.0
Ohio        2005     6.5
            2006     9.0
Name: data1, dtype: float64
```

In [62]:

```
df
```

Out[62]:

	key1	key2	data1	data2
0	a	one	5	11
1	a	two	11	5
2	b	one	12	15
3	b	two	8	0
4	a	one	9	16

In [63]:

```
df.groupby(['key1', 'key2']).size() # you don't always have to use mean, you can use other functions as well
```

Out[63]:

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

Iterating over groups

In [64]:

```
df
```

Out[64]:

	key1	key2	data1	data2
0	a	one	5	11
1	a	two	11	5
2	b	one	12	15
3	b	two	8	0
4	a	one	9	16

In [65]:

```
# the groupby creates a series of tuples that can be unpacked into name and group
for name, group in df.groupby('key1'):
    print(name)
    print(group)
    print(group.mean())
    print('----------------')
```

```
a
  key1 key2  data1  data2
0    a  one      5     11
1    a  two     11      5
4    a  one      9     16
data1     8.333333
data2    10.666667
dtype: float64
----------------
b
  key1 key2  data1  data2
2    b  one     12     15
3    b  two      8      0
data1    10.0
data2     7.5
dtype: float64
----------------
```

In [66]:

```
for name, group in df.groupby('key2'):
    print(name)
    print(group)
    print(group.sum())
    print('----------------')
```

```
one
  key1 key2  data1  data2
0    a  one      5     11
2    b  one     12     15
4    a  one      9     16
key1           aba
key2     oneoneone
data1           26
data2           42
dtype: object
----------------
two
  key1 key2  data1  data2
1    a  two     11      5
3    b  two      8      0
key1         ab
key2     twotwo
data1        19
data2         5
dtype: object
----------------
```