Name: Prof. Dhanashree S.
Expt. No. 5 : Python Pandas - Aggregations, Missing Data, GroupBy

Python Pandas - Aggregations

Once the rolling, expanding and **ewm** objects are created, several methods are available to perform aggregations on data.

Applying Aggregations on DataFrame

Let us create a DataFrame and apply aggregations on it.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])

print df
r = df.rolling(window=3,min_periods=1)
print r
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   0.790670   -0.387854   -0.668132    0.267283
2000-01-03  -0.575523   -0.965025    0.060427   -2.179780
2000-01-04   1.669653    1.211759   -0.254695    1.429166
2000-01-05   0.100568   -0.236184    0.491646   -0.466081
2000-01-06   0.155172    0.992975   -1.205134    0.320958
2000-01-07   0.309468   -0.724053   -1.412446    0.627919
2000-01-08   0.099489   -1.028040    0.163206   -1.274331
2000-01-09   1.639500   -0.068443    0.714008   -0.565969
2000-01-10   0.326761    1.479841    0.664282   -1.361169

Rolling [window=3,min_periods=1,center=False,axis=0]
```

We can aggregate by passing a function to the entire DataFrame, or select a column via the standard **get item** method.

Apply Aggregation on a Whole Dataframe

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
```

```
print r.aggregate(np.sum)
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469

                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
```

Apply Aggregation on a Single Column of a Dataframe

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r['A'].aggregate(np.sum)
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
2000-01-01   1.088512
2000-01-02   1.879182
2000-01-03   1.303660
```

```
2000-01-04   1.884801
2000-01-05   1.194699
2000-01-06   1.925393
2000-01-07   0.565208
2000-01-08   0.564129
2000-01-09   2.048458
2000-01-10   2.065750
Freq: D, Name: A, dtype: float64
```

Apply Aggregation on Multiple Columns of a DataFrame

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r[['A','B']].aggregate(np.sum)
```

Its **output** is as follows −

```
                   A          B          C          D
2000-01-01   1.088512  -0.650942  -2.547450  -0.566858
2000-01-02   1.879182  -1.038796  -3.215581  -0.299575
2000-01-03   1.303660  -2.003821  -3.155154  -2.479355
2000-01-04   1.884801  -0.141119  -0.862400  -0.483331
2000-01-05   1.194699   0.010551   0.297378  -1.216695
2000-01-06   1.925393   1.968551  -0.968183   1.284044
2000-01-07   0.565208   0.032738  -2.125934   0.482797
2000-01-08   0.564129  -0.759118  -2.454374  -0.325454
2000-01-09   2.048458  -1.820537  -0.535232  -1.212381
2000-01-10   2.065750   0.383357   1.541496  -3.201469
                   A          B
2000-01-01   1.088512  -0.650942
2000-01-02   1.879182  -1.038796
2000-01-03   1.303660  -2.003821
2000-01-04   1.884801  -0.141119
2000-01-05   1.194699   0.010551
2000-01-06   1.925393   1.968551
2000-01-07   0.565208   0.032738
2000-01-08   0.564129  -0.759118
2000-01-09   2.048458  -1.820537
2000-01-10   2.065750   0.383357
```

Apply Multiple Functions on a Single Column of a DataFrame

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
```

```
print r['A'].aggregate([np.sum,np.mean])
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
                   sum        mean
2000-01-01   1.088512   1.088512
2000-01-02   1.879182   0.939591
2000-01-03   1.303660   0.434553
2000-01-04   1.884801   0.628267
2000-01-05   1.194699   0.398233
2000-01-06   1.925393   0.641798
2000-01-07   0.565208   0.188403
2000-01-08   0.564129   0.188043
2000-01-09   2.048458   0.682819
2000-01-10   2.065750   0.688583
```

Apply Multiple Functions on Multiple Columns of a DataFrame

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
   index = pd.date_range('1/1/2000', periods=10),
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r[['A','B']].aggregate([np.sum,np.mean])
```

Its **output** is as follows −

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   1.879182   -1.038796   -3.215581   -0.299575
2000-01-03   1.303660   -2.003821   -3.155154   -2.479355
2000-01-04   1.884801   -0.141119   -0.862400   -0.483331
2000-01-05   1.194699    0.010551    0.297378   -1.216695
2000-01-06   1.925393    1.968551   -0.968183    1.284044
2000-01-07   0.565208    0.032738   -2.125934    0.482797
2000-01-08   0.564129   -0.759118   -2.454374   -0.325454
2000-01-09   2.048458   -1.820537   -0.535232   -1.212381
2000-01-10   2.065750    0.383357    1.541496   -3.201469
                     A                       B
                   sum        mean         sum        mean
2000-01-01   1.088512   1.088512   -0.650942   -0.650942
2000-01-02   1.879182   0.939591   -1.038796   -0.519398
```

```
2000-01-03   1.303660   0.434553   -2.003821   -0.667940
2000-01-04   1.884801   0.628267   -0.141119   -0.047040
2000-01-05   1.194699   0.398233    0.010551    0.003517
2000-01-06   1.925393   0.641798    1.968551    0.656184
2000-01-07   0.565208   0.188403    0.032738    0.010913
2000-01-08   0.564129   0.188043   -0.759118   -0.253039
2000-01-09   2.048458   0.682819   -1.820537   -0.606846
2000-01-10   2.065750   0.688583    0.383357    0.127786
```

Apply Different Functions to Different Columns of a Dataframe

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(3, 4),
   index = pd.date_range('1/1/2000', periods=3),
   columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r.aggregate({'A' : np.sum,'B' : np.mean})
```

Its **output** is as follows −

```
                   A          B          C         D
2000-01-01  -1.575749  -1.018105   0.317797  0.545081
2000-01-02  -0.164917  -1.361068   0.258240  1.113091
2000-01-03   1.258111   1.037941  -0.047487  0.867371
                   A          B
2000-01-01  -1.575749  -1.018105
2000-01-02  -1.740666  -1.189587
2000-01-03  -0.482555  -0.447078
```

Python Pandas - Missing Data

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

When and Why Is Data Missed?

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

Let us now see how we can handle missing values (say NA or NaN) using Pandas.

```
# import the pandas library
```

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df
```

Its **output** is as follows −

```
          one         two       three
a    0.077988    0.476149    0.965836
b         NaN         NaN         NaN
c   -0.390208   -0.551605   -2.301950
d         NaN         NaN         NaN
e   -2.000303   -0.788201    1.510072
f   -0.930230   -0.670473    1.146615
g         NaN         NaN         NaN
h    0.085100    0.532791    0.887415
```

Using reindexing, we have created a DataFrame with missing values. In the output, **NaN** means **Not a Number.**

Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects −

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].isnull()
```

Its **output** is as follows −

```
a  False
b  True
c  False
d  True
e  False
f  False
g  True
h  False
Name: one, dtype: bool
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].notnull()
```

Its **output** is as follows −

```
a  True
b  False
c  True
d  False
e  True
f  True
g  False
h  True
Name: one, dtype: bool
```

Calculations with Missing Data

- When summing data, NA will be treated as Zero
- If the data are all NA, then the result will be NA

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].sum()
```

Its **output** is as follows −

```
2.02357685917
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])
print df['one'].sum()
```

Its **output** is as follows −

```
nan
```

Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c',
'e'],columns=['one',
'two', 'three'])

df = df.reindex(['a', 'b', 'c'])

print df
print ("NaN replaced with '0':")
print df.fillna(0)
```

Its **output** is as follows −

```
          one        two      three
a  -0.576991  -0.741695  0.553172
b        NaN        NaN       NaN
c   0.744328  -1.735166  1.749580

NaN replaced with '0':
          one        two      three
a  -0.576991  -0.741695  0.553172
b   0.000000   0.000000  0.000000
c   0.744328  -1.735166  1.749580
```

Here, we are filling with value zero; instead we can also fill with any other value.

Fill NA Forward and Backward

Using the concepts of filling discussed in the ReIndexing Chapter we will fill the missing values.

Sr.No	Method & Action
1	**pad/fill** Fill methods Forward
2	**bfill/backfill** Fill methods Backward

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='pad')
```

Its **output** is as follows −

```
          one        two      three
a   0.077988   0.476149   0.965836
b   0.077988   0.476149   0.965836
c  -0.390208  -0.551605  -2.301950
d  -0.390208  -0.551605  -2.301950
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g  -0.930230  -0.670473   1.146615
h   0.085100   0.532791   0.887415
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='backfill')
```

Its **output** is as follows −

```
          one        two      three
a   0.077988   0.476149   0.965836
b  -0.390208  -0.551605  -2.301950
c  -0.390208  -0.551605  -2.301950
d  -2.000303  -0.788201   1.510072
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g   0.085100   0.532791   0.887415
h   0.085100   0.532791   0.887415
```

Drop Missing Values

If you want to simply exclude the missing values, then use the **dropna** function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna()
```

Its **output** is as follows −

```
          one        two      three
a   0.077988   0.476149   0.965836
c  -0.390208  -0.551605  -2.301950
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
h   0.085100   0.532791   0.887415
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e',
'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna(axis=1)
```

Its **output** is as follows −

```
Empty DataFrame
Columns: [ ]
Index: [a, b, c, d, e, f, g, h]
```

Replace Missing (or) Generic Values

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the **fillna()** function.

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000],
'two':[1000,0,30,40,50,60]})

print df.replace({1000:10,2000:60})
```

Its **output** is as follows −

```
   one  two
0   10   10
```

```
1    20     0
2    30    30
3    40    40
4    50    50
5    60    60
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000],
'two':[1000,0,30,40,50,60]})
print df.replace({1000:10,2000:60})
```

Its **output** is as follows −

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

Python Pandas - GroupBy

Any **groupby** operation involves one of the following operations on the original object. They are −

- **Splitting** the Object
- **Applying** a function
- **Combining** the results

In many situations, we split the data into sets and we apply some functionality on each subset. In the apply functionality, we can perform the following operations −

- **Aggregation** − computing a summary statistic
- **Transformation** − perform some group-specific operation
- **Filtration** − discarding the data with some condition

Let us now create a DataFrame object and perform all the operations on it −

```
#import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
```

```
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print df
```

Its **output** is as follows −

```
    Points   Rank     Team   Year
0      876      1   Riders   2014
1      789      2   Riders   2015
2      863      2   Devils   2014
3      673      3   Devils   2015
4      741      3    Kings   2014
5      812      4    kings   2015
6      756      1    Kings   2016
7      788      1    Kings   2017
8      694      2   Riders   2016
9      701      4   Royals   2014
10     804      1   Royals   2015
11     690      2   Riders   2017
```

Split Data into Groups

Pandas object can be split into any of their objects. There are multiple ways to split an object like −

- obj.groupby('key')
- obj.groupby(['key1','key2'])
- obj.groupby(key,axis=1)

Let us now see how the grouping objects can be applied to the DataFrame object

Example

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print df.groupby('Team')
```

Its **output** is as follows −

```
<pandas.core.groupby.DataFrameGroupBy object at 0x7fa46a977e50>
```

View Groups

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print df.groupby('Team').groups
```

Its **output** is as follows −

```
{'Kings': Int64Index([4, 6, 7],       dtype='int64'),
'Devils': Int64Index([2, 3],          dtype='int64'),
'Riders': Int64Index([0, 1, 8, 11],   dtype='int64'),
'Royals': Int64Index([9, 10],         dtype='int64'),
'kings' : Int64Index([5],             dtype='int64')}
```

Example

Group by with multiple columns −

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print df.groupby(['Team','Year']).groups
```

Its **output** is as follows −

```
{('Kings', 2014): Int64Index([4], dtype='int64'),
 ('Royals', 2014): Int64Index([9], dtype='int64'),
 ('Riders', 2014): Int64Index([0], dtype='int64'),
 ('Riders', 2015): Int64Index([1], dtype='int64'),
 ('Kings', 2016): Int64Index([6], dtype='int64'),
 ('Riders', 2016): Int64Index([8], dtype='int64'),
 ('Riders', 2017): Int64Index([11], dtype='int64'),
 ('Devils', 2014): Int64Index([2], dtype='int64'),
 ('Devils', 2015): Int64Index([3], dtype='int64'),
 ('kings', 2015): Int64Index([5], dtype='int64'),
 ('Royals', 2015): Int64Index([10], dtype='int64'),
```

```
 ('Kings', 2017): Int64Index([7], dtype='int64')}
```

Iterating through Groups

With the **groupby** object in hand, we can iterate through the object similar to itertools.obj.

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

grouped = df.groupby('Year')

for name,group in grouped:
   print name
   print group
```

Its **output** is as follows −

```
2014
   Points  Rank     Team   Year
0     876     1   Riders   2014
2     863     2   Devils   2014
4     741     3   Kings    2014
9     701     4   Royals   2014

2015
   Points  Rank     Team   Year
1     789     2   Riders   2015
3     673     3   Devils   2015
5     812     4    kings   2015
10    804     1   Royals   2015

2016
   Points  Rank     Team   Year
6     756     1    Kings   2016
8     694     2   Riders   2016

2017
   Points  Rank    Team   Year
7     788     1   Kings   2017
11    690     2  Riders   2017
```

By default, the **groupby** object has the same label name as the group name.

Select a Group

Using the **get_group()** method, we can select a single group.

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

grouped = df.groupby('Year')
print grouped.get_group(2014)
```

Its **output** is as follows −

```
   Points  Rank     Team    Year
0     876     1   Riders    2014
2     863     2   Devils    2014
4     741     3   Kings     2014
9     701     4   Royals    2014
```

Aggregations

An aggregated function returns a single aggregated value for each group. Once the **group by** object is created, several aggregation operations can be performed on the grouped data.

An obvious one is aggregation via the aggregate or equivalent **agg** method −

```
# import the pandas library
import pandas as pd
import numpy as np

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

grouped = df.groupby('Year')
print grouped['Points'].agg(np.mean)
```

Its **output** is as follows −

```
Year
2014   795.25
2015   769.50
```

```
2016   725.00
2017   739.00
Name: Points, dtype: float64
```

Another way to see the size of each group is by applying the size() function −

```
import pandas as pd
import numpy as np

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

Attribute Access in Python Pandas
grouped = df.groupby('Team')
print grouped.agg(np.size)
```

Its **output** is as follows −

```
         Points   Rank   Year
Team
Devils        2      2      2
Kings         3      3      3
Riders        4      4      4
Royals        2      2      2
kings         1      1      1
```

Applying Multiple Aggregation Functions at Once

With grouped Series, you can also pass a **list** or **dict of functions** to do aggregation with, and generate DataFrame as output −

```
# import the pandas library
import pandas as pd
import numpy as np

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

grouped = df.groupby('Team')
print grouped['Points'].agg([np.sum, np.mean, np.std])
```

Its **output** is as follows −

```
Team      sum      mean          std
Devils   1536   768.000000   134.350288
Kings    2285   761.666667    24.006943
Riders   3049   762.250000    88.567771
Royals   1505   752.500000    72.831998
kings     812   812.000000          NaN
```

Transformations

Transformation on a group or a column returns an object that is indexed the same size of that is being grouped. Thus, the transform should return a result that is the same size as that of a group chunk.

```
# import the pandas library
import pandas as pd
import numpy as np

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

grouped = df.groupby('Team')
score = lambda x: (x - x.mean()) / x.std()*10
print grouped.transform(score)
```

Its **output** is as follows −

```
       Points        Rank        Year
0   12.843272  -15.000000  -11.618950
1    3.020286    5.000000   -3.872983
2    7.071068   -7.071068   -7.071068
3   -7.071068    7.071068    7.071068
4   -8.608621   11.547005  -10.910895
5         NaN         NaN         NaN
6   -2.360428   -5.773503    2.182179
7   10.969049   -5.773503    8.728716
8   -7.705963    5.000000    3.872983
9   -7.071068    7.071068   -7.071068
10   7.071068   -7.071068    7.071068
11  -8.157595    5.000000   11.618950
```

Filtration

Filtration filters the data on a defined criteria and returns the subset of data. The **filter()** function is used to filter the data.

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

```
ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print df.groupby('Team').filter(lambda x: len(x) >= 3)
```

Its **output** is as follows −

```
    Points  Rank     Team   Year
0      876     1   Riders   2014
1      789     2   Riders   2015
4      741     3   Kings    2014
6      756     1   Kings    2016
7      788     1   Kings    2017
8      694     2   Riders   2016
11     690     2   Riders   2017
```

In the above filter condition, we are asking to return the teams which have participated three or more times in IPL.