



Data Analysis with Python

Session-5&6





Table of Contents

- ▶ Basic Aggregation Methods
- ▶ Grouping Data (DataFrame.groupby())
- ▶ Useful Operations
 - Aggregate
 - Filter
 - Transform
 - Apply
 - Applymap
 - Map
 - Pivot & Pivot Table
 - Stack & Unstack

Basic Aggregation Methods

- ▶ count()
- ▶ mean()
- ▶ median()
- ▶ idxmin()
- ▶ idxmax()
- ▶ corr()
- ▶ min()
- ▶ max()
- ▶ std()
- ▶ var()
- ▶ sum()
- ▶ describe()

Grouping Data (DataFrame.groupby)

- ▶ DataFrame.groupby operation involves some combination of **splitting the object, applying a function, and combining the results.**
- ▶ This can be used to group large amounts of data and compute operations on these groups.

Grouping Data (DataFrame.groupby())

	species	sepal_length	sepal_width	petal_length	petal_width
0	setosa	5.1	3.5	1.4	0.2
1	setosa	4.9	3.0	1.4	0.2
2	setosa	4.7	3.2	1.3	0.2
3	setosa	4.6	3.1	1.5	0.2
4	setosa	5.0	3.6	1.4	0.2
50	versicolor	7.0	3.2	4.7	1.4
51	versicolor	6.4	3.2	4.5	1.5
52	versicolor	6.9	3.1	4.9	1.5
53	versicolor	5.5	2.3	4.0	1.3
54	versicolor	6.5	2.8	4.6	1.5
100	virginica	6.3	3.3	6.0	2.5
101	virginica	5.8	2.7	5.1	1.9
102	virginica	7.1	3.0	5.9	2.1
103	virginica	6.3	2.9	5.6	1.8
104	virginica	6.5	3.0	5.8	2.2

df.groupby('species').sum()

SUM

SUM

SUM

species	sepal_length	sepal_width	petal_length	petal_width
setosa	24.3	16.4	7.0	1.0
versicolor	32.3	14.6	22.7	7.2
virginica	32.0	14.9	28.4	10.5

Useful Operations

DataFrame.aggregate/agg

```
>>> df = pd.DataFrame([[1, 2, 3],  
...                      [4, 5, 6],  
...                      [7, 8, 9],  
...                      [np.nan, np.nan, np.nan]],  
...                      columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])  
      A    B    C  
sum  12.0  15.0  18.0  
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})  
      A    B  
sum  12.0  NaN  
min   1.0   2.0  
max   NaN   8.0
```

Aggregate different functions over the columns and rename the index

```
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))  
      A    B    C  
x  7.0  NaN  NaN  
y  NaN  2.0  NaN  
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")  
0    2.0  
1    5.0  
2    8.0  
3    NaN
```

Useful Operations



DataFrame.groupby.aggregate/agg

```
>>> df
   A   B       C
0  1   1  0.362838
1  1   2  0.227877
2  2   3  1.267767
3  2   4 -0.562860
```

The aggregation is for each column.

```
>>> df.groupby('A').agg('min')
      B       C
A
1  1  0.227877
2  3 -0.562860
```

Multiple aggregations

```
>>> df.groupby('A').agg(['min', 'max'])
      B       C
   min max     min     max
A
1   1   2  0.227877  0.362838
2   3   4 -0.562860  1.267767
```

Select a column for aggregation

```
>>> df.groupby('A').B.agg(['min', 'max'])
      min   max
A
1      1      2
2      3      4
```

Different aggregations per column

```
>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})
      B       C
   min max     sum
A
1   1   2  0.590715
2   3   4  0.704907
```

Useful Operations



DataFrame.filter

- ▶ Subset the dataframe rows or columns according to the specified index labels.

```
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),  
...                      index=['mouse', 'rabbit'],  
...                      columns=['one', 'two', 'three'])  
>>> df  
      one  two  three  
mouse   1    2    3  
rabbit  4    5    6
```

```
>>> # select columns by name  
>>> df.filter(items=['one', 'three'])  
          one  three  
mouse     1     3  
rabbit    4     6
```

```
>>> # select columns by regular expression  
>>> df.filter(regex='e$', axis=1)  
          one  three  
mouse     1     3  
rabbit    4     6
```

```
>>> # select rows containing 'bbi'  
>>> df.filter(like='bbi', axis=0)  
          one  two  three  
rabbit    4    5    6
```

Useful Operations

DataFrame.groupby.filter

- Return a copy of a DataFrame excluding filtered elements.

```
df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                           'foo', 'bar'],
                   'B' : [1, 2, 3, 4, 5, 6],
                   'C' : [2.0, 5., 8., 1., 2., 9.]})
```

```
df
```

	A	B	C
0	foo	1	2.0
1	bar	2	5.0
2	foo	3	8.0
3	bar	4	1.0
4	foo	5	2.0
5	bar	6	9.0

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

	B	C
A		
bar	4	5.0
foo	3	4.0

```
df.groupby('A').filter(lambda x: x['B'].mean() > 3.)
```

	A	B	C
1	bar	2	5.0
3	bar	4	1.0
5	bar	6	9.0

Useful Operations



DataFrame.transform

- ▶ Produced DataFrame will have same axis length as self.

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})  
>>> df  
      A    B  
0    0    1  
1    1    2  
2    2    3  
>>> df.transform(lambda x: x + 1)  
      A    B  
0    1    2  
1    2    3  
2    3    4
```

```
>>> s = pd.Series(range(3))  
>>> s  
0    0  
1    1  
2    2  
dtype: int64  
>>> s.transform([np.sqrt, np.exp])  
          sqrt        exp  
0  0.000000  1.000000  
1  1.000000  2.718282  
2  1.414214  7.389056
```

Useful Operations

DataFrame.groupby.transform

df2			
	groups	var1	var2
0	A	10	100
1	B	23	253
2	C	33	333
3	A	22	262
4	B	11	111
5	C	99	969
6	A	76	405
7	B	84	578
8	C	45	760

```
df2.groupby("groups")["var1"].mean()
```

```
groups
A    36.000000
B    39.333333
C    59.000000
```

```
Name: var1, dtype: float64
```

```
df2.groupby("groups")["var1"].transform("mean")
```

```
0    36.000000
1    39.333333
2    59.000000
3    36.000000
4    39.333333
5    59.000000
6    36.000000
7    39.333333
8    59.000000
```

Useful Operations

Types of “Apply”s

`Series.apply`

For applying more complex functions on a Series.

`DataFrame.apply`

Apply a function row-/column-wise.

`DataFrame.applymap`

Apply a function elementwise on a whole DataFrame.

Useful Operations



Series.apply

- ▶ Invoke function on values of Series.

```
>>> s = pd.Series([20, 21, 12],  
...                 index=['London', 'New York', 'Helsinki'])  
>>> s  
London    20  
New York  21  
Helsinki 12
```

```
>>> def square(x):  
...     return x ** 2  
>>> s.apply(square)  
London    400  
New York  441  
Helsinki 144
```

```
>>> s.apply(lambda x: x ** 2)  
London    400  
New York  441  
Helsinki 144
```

```
>>> s.apply(np.log)  
London    2.995732  
New York  3.044522  
Helsinki 2.484907
```

Useful Operations

DataFrame.apply

- ▶ Apply a function along an axis of the DataFrame.

```
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])  
>>> df  
   A    B  
0  4    9  
1  4    9  
2  4    9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)  
      A      B  
0  2.0  3.0  
1  2.0  3.0  
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)  
A    12  
B    27  
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)  
0    13  
1    13  
2    13  
dtype: int64
```

Useful Operations



DataFrame.applymap

- ▶ Apply a function elementwise on a whole DataFrame.

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])  
>>> df  
      0      1  
0  1.000  2.120  
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))  
      0      1  
0  3      4  
1  5      5
```

```
>>> df.applymap(lambda x: x**2)  
      0          1  
0  1.000000  4.494400  
1  11.262736 20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2  
      0      1  
0  1.000000  4.494400  
1  11.262736 20.857489
```

Useful Operations

Series.map

- Map values of Series according to input correspondence.

```
>>> s = pd.Series(['cat', 'dog', np.nan, 'rabbit'])
>>> s
0    cat
1    dog
2    NaN
3  rabbit
```

```
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})
0    kitten
1     puppy
2      NaN
3      NaN
```

```
>>> s.map('I am a {}'.format)
0      I am a cat
1      I am a dog
2      I am a nan
3  I am a rabbit
```

```
>>> s.map('I am a {}'.format, na_action='ignore')
0      I am a cat
1      I am a dog
2        NaN
3  I am a rabbit
```

Useful Operations

apply() vs applymap() vs map()

	DataFrame	Series
apply	✓	✓
map		✓
applymap	✓	

- ▶ **apply()** is used to apply a function **along an axis of the DataFrame or on values of Series**.
- ▶ **applymap()** is used to apply a function to a **DataFrame elementwise**.
- ▶ **map()** is used to **substitute each value in a Series with another value**.

Useful Operations



apply() vs transform()

```
df5.apply(lambda x: x+10)
```

	A	B
0	1	10
1	2	20
2	3	30

	A	B
0	1	10
1	2	20
2	3	30

```
df5.transform(lambda x: x+10)
```

	A	B
0	11	20
1	12	30
2	13	40

Useful Operations



apply() vs transform()

- ▶ **transform()** cannot produce aggregated results.
- ▶ **apply()** works with multiple Series at a time. But, **transform()** is only allowed to work with a single Series at a time.
- ▶ **transform()** returns a DataFrame that has the same length as the input

DataFrame/Series Operations

apply() vs transform()

- ▶ **transform()** cannot produce aggregated results.

	A	B
0	1	10
1	2	20
2	3	30

```
df5.apply(lambda x: x.sum())
# df5.transform(Lambda x:x.sum()) gives an error
```

```
A      6
B     60
dtype: int64
```

Useful Operations



apply() vs transform()

- ▶ **apply()** works with multiple Series at a time. But, **transform()** is only allowed to work with a single Series at a time.

	A	B
0	1	10
1	2	20
2	3	30

```
df5.apply(lambda x: x["B"]-x["A"], axis=1)
# df5.transform(Lambda x: x["B"]-x["A"], axis=1) gives an error

0      9
1     18
2     27
dtype: int64
```

Useful Operations



apply() vs transform()

- ▶ **apply()** works with multiple Series at a time. But, **transform()** is only allowed to work with a single Series at a time.

key	A	B	
0	a	0	1
1	b	1	2
2	c	2	3
3	a	3	1
4	b	4	2
5	c	5	3
6	a	6	1
7	b	7	2
8	c	8	3

```
df6.groupby('key').apply(lambda x: x["B"]-x["A"])
# df6.groupby('key').transform(Lambda x: x["B"]-x["A"]) gives an error
```

key		
a	0	1
	3	-2
	6	-5
b	1	1
	4	-2
	7	-5
c	2	1
	5	-2
	8	-5

Useful Operations



apply() vs transform()

- ▶ **transform()** returns a DataFrame that has the same length as the input

key	A	B
0	a	0 1
1	b	1 2
2	c	2 3
3	a	3 1
4	b	4 2
5	c	5 3
6	a	6 1
7	b	7 2
8	c	8 3

df6.groupby('key')['A'].sum()
key
a 9
b 12
c 15

df6.groupby('key')['A'].apply(lambda x: x.sum())
key
a 9
b 12
c 15

df6.groupby('key')['A'].transform(lambda x: x.sum())
0 9
1 12
2 15
3 9
4 12
5 15
6 9
7 12
8 15

Useful Operations

DataFrame.pivot

Row index in the new table
Columns in the new table
Cell values in the new table

<u>ix</u>	<u>Item</u>	<u>CType</u>	<u>USD</u>	<u>EU</u>
<u>0</u>	Item0	Gold	1\$	1€
<u>1</u>	Item0	Bronze	2\$	2€
<u>2</u>	Item1	Gold	3\$	3€
<u>3</u>	Item1	Silver	4\$	4€

<u>ix=Item</u>	<u>Bronze</u>	<u>Gold</u>	<u>Silver</u>
<u>Item0</u>	2\$	1\$	NaN
<u>Item1</u>	NaN	3\$	4\$

```
d.pivot(index='Item', columns='CType', values='USD')
```

Useful Operations



DataFrame.pivot_table

<u>ix</u>	<u>Item</u>	<u>CType</u>	<u>USD</u>	<u>EU</u>
<u>0</u>	Item0	Gold	1	1
<u>1</u>	Item0	Bronze	2	2
<u>2</u>	Item0	Gold	3	3
<u>3</u>	Item1	Silver	4	4

<u>ix=Item</u>	<u>Bronze</u>	<u>Gold</u>	<u>Silver</u>
<u>Item0</u>	2	<u>2 = mean(1,3)</u>	NaN
<u>Item1</u>	NaN	NaN	4

```
d.pivot_table(index='Item', columns='CType', values='USD', aggfunc=np.mean)
```

Useful Operations

.pivot() vs .pivot_table()

- ▶ **Pivot_table** is a generalization of **pivot** that can handle duplicate values for **one pivoted index/column pair**.
- ▶ **Pivot_table** will only allow **numeric types** as "values=", whereas **pivot** will take **string types** as "values=".

Useful Operations

.pivot() vs .pivot_table()

	gender	sport	age	height	weight
0	male	baseball	21	200	72
1	female	basketball	21	130	72
2	female	golf	22	150	73
3	male	basketball	20	175	75
4	female	archery	22	170	68

Pivot and pivot_table may only exhibit the same functionality if the data allows.

```
df.pivot_table(index='gender', columns='sport',  
               values=['age','height','weight'],  
               aggfunc='mean')
```

		age			height			weight					
	sport	archery	baseball	basketball	golf	archery	baseball	basketball	golf	archery	baseball	basketball	golf
gender													
female		22.0	NaN	21.0	22.0	68.0	NaN	72.0	73.0	170.0	NaN	130.0	150.0
male		NaN	21.0	20.0	NaN	NaN	72.0	75.0	NaN	NaN	200.0	175.0	NaN

```
df.pivot(index='gender', columns='sport',  
         values=['age','height','weight'])
```

		age			height			weight					
	sport	archery	baseball	basketball	golf	archery	baseball	basketball	golf	archery	baseball	basketball	golf
gender													
female		22.0	NaN	21.0	22.0	68.0	NaN	72.0	73.0	170.0	NaN	130.0	150.0
male		NaN	21.0	20.0	NaN	NaN	72.0	75.0	NaN	NaN	200.0	175.0	NaN

Useful Operations

.pivot() vs .pivot_table()

	gender	sport	age	height	weight
0	male	baseball	21	200	72
1	female	basketball	21	130	72
2	female	golf	22	150	73
3	male	basketball	20	175	75
4	female	basketball	22	170	68

```
df.pivot_table(index='gender', columns='sport',
               values=['age','height','weight'],
               aggfunc='mean')
```

		age		height		weight				
	sport	baseball	basketball	golf	baseball	basketball	golf	baseball	basketball	golf
gender										
female		NaN		21.5	22.0	NaN		70.0	73.0	NaN
male		21.0		20.0	NaN	72.0		75.0	NaN	200.0

```
df.pivot(columns='sport', values=['age','height','weight'])
```

		age		height		weight				
	sport	baseball	basketball	golf	baseball	basketball	golf	baseball	basketball	golf
0		21.0	NaN	NaN	72.0	NaN	NaN	200.0	NaN	NaN
1		NaN		21.0	NaN	NaN		72.0	NaN	NaN
2		NaN		NaN	22.0	NaN		NaN	73.0	NaN
3		NaN		20.0	NaN	NaN		75.0	NaN	NaN
4		NaN		22.0	NaN	NaN		68.0	NaN	NaN

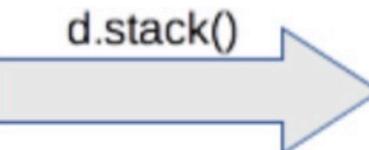
If there are duplicate entries possible from the index(es) of interest you will need to aggregate the data in pivot_table, not pivot.

Useful Operations



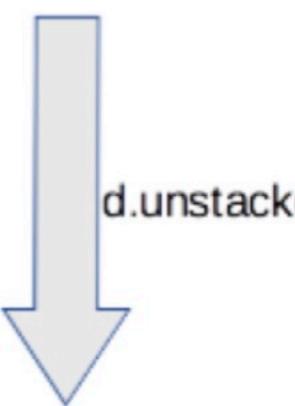
.stack() vs .unstack()

		c0		c1
		c00	c01	c10
r0	r00	(0,0)	(0,1)	(0,2)
	r01	(1,0)	(1,1)	(1,2)



		c0	c1
r0	r00	c00	(0,0) NaN
	r01	c01	(0,1) NaN
r0	c10	NaN	(0,2)
	c00	(1,0)	NaN
	c01	(1,1)	NaN
c10	NaN	(1,2)	

Stacked (i.e. taller)



		c0		c1	
		c00	c01	c10	
r0	r00	r00	r01	r00	r01
	r01	(0,0)	(1,0)	(0,1)	(1,1)

Unstacked (i.e. broader)