# Financial Data Analysis with Python

Instructor: Luping Yu

Mar 15, 2022

---

## Lecture 04. Data Aggregation and Group Operations

Categorizing a dataset and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow. After loading and preparing a dataset, you may need to compute group statistics or possibly pivot tables for reporting or visualization purposes. pandas provides a flexible groupby interface, enabling you to slice, dice, and summarize datasets in a natural way.

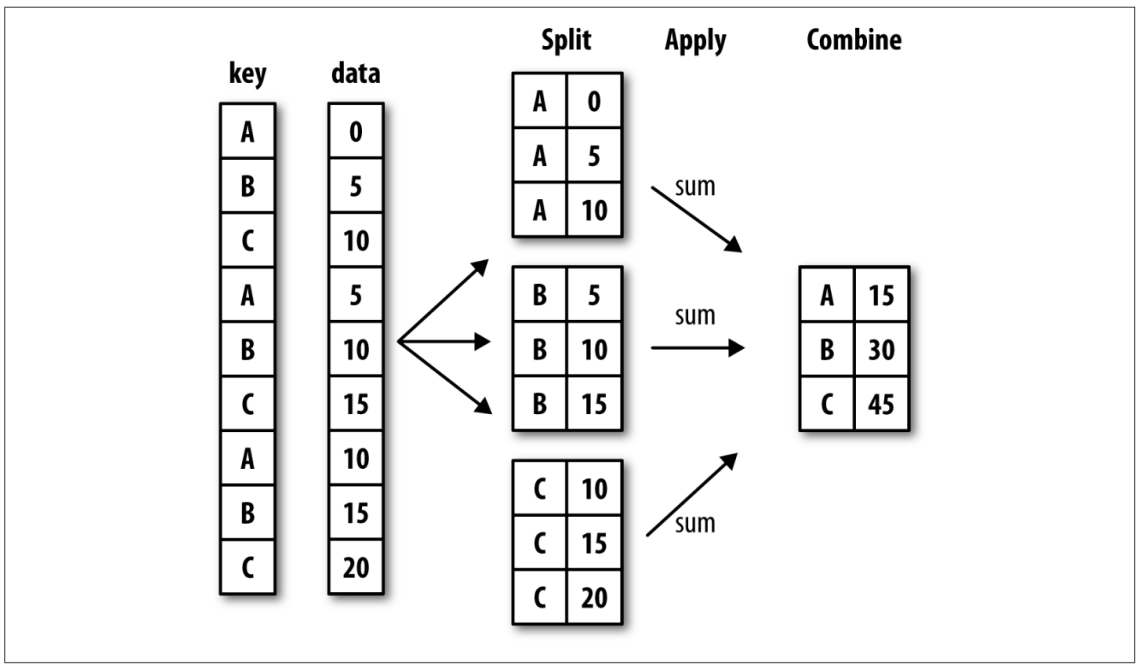
---

### GroupBy Mechanics

Punchline: **split-apply-combine （拆分-应用-合并）**

In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, 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 object.

The form of the resulting object will usually depend on what's being done to the data. See the following figure for a mockup of a simple group aggregation:




---

To get started, here is a small tabular dataset as a DataFrame:

In [87]:
```
import pandas as pd
import numpy as np
#np.random.randn: 生成随机数组

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[87]:

| | key1 | key2 | data1 | data2 |
|---|---|---|---|---|
| **0** | a | one | -0.039668 | -1.075274 |
| **1** | a | two | 0.874157 | 1.008029 |
| **2** | b | one | -1.137963 | -0.198576 |
| **3** | b | two | -0.108767 | 1.601767 |
| **4** | a | one | -0.761323 | -0.559282 |

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:

In [88]:
```
grouped = df['data1'].groupby(df['key1'])

grouped
```

Out[88]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x1210b9580>
```

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 operation to each of the groups.

For example, to compute group means we can call the GroupBy's mean method:

In [89]:
```
grouped.mean()
```

Out[89]:
```
key1
a    0.024389
b   -0.623365
Name: data1, dtype: float64
```

Later, I'll explain more about what happens when you call **.mean( )**. The important thing here is that 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.

If instead we had passed multiple arrays as a list, we'd get something different:

In [90]:
```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()

means
```

Out[90]:
```
key1  key2
a     one    -0.400496
      two     0.874157
b     one    -1.137963
      two    -0.108767
Name: data1, dtype: float64
```

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.

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

In [91]:
```
df.groupby(['key1', 'key2']).size()
```

Out[91]:
```
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

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

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:

In [92]:
```
df.groupby(['key1', 'key2'])[['data2']].mean()
```

Out[92]:

| | | data2 |
|---|---|---|
| **key1** | **key2** | |
| **a** | **one** | -0.817278 |
| | **two** | 1.008029 |
| **b** | **one** | -0.198576 |
| | **two** | 1.601767 |

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 scalar:

In [93]:
```
df.groupby(['key1', 'key2'])['data2'].mean()
```

Out[93]:
```
key1  key2
a     one    -0.817278
      two     1.008029
b     one    -0.198576
      two     1.601767
Name: data2, dtype: float64
```

---

## 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

sum.

| Function | 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 variance |
| min, max | Minimum and maximum of non-NA values |
| first, last | First and last non-NA values |

To use your own aggregation functions, pass any function that aggregates an array to the **aggregate** or **agg** method:

In [94]:
```
df = df[['key1','data1','data2']]

df
```

Out[94]:

| | key1 | data1 | data2 |
|---|---|---|---|
| **0** | a | -0.039668 | -1.075274 |
| **1** | a | 0.874157 | 1.008029 |
| **2** | b | -1.137963 | -0.198576 |
| **3** | b | -0.108767 | 1.601767 |
| **4** | a | -0.761323 | -0.559282 |

In [95]:
```
def peak_to_peak(arr):
    return arr.max() - arr.min()
```

In [96]:
```
grouped = df.groupby(df['key1'])

grouped.agg(peak_to_peak)
#grouped.apply(peak_to_peak)
```

Out[96]:

| | data1 | data2 |
|---|---|---|
| **key1** | | |
| **a** | 1.635480 | 2.083303 |
| **b** | 1.029195 | 1.800342 |

---

## Apply: General split-apply-combine

- Create analysis with .groupby( ) and built-in functions (mean, sum, count, etc.)
- Create analysis with .groupby( ) and user defined functions (agg)
- Use .transform( ) to join group stats to the original dataframe

Let's get started with the tipping dataset:

In [97]:
```
df = pd.read_csv('examples/tips.csv')

df = df[['day','size','total_bill','tip']]

df
```

Out[97]:

| | day | size | total_bill | tip |
|---|---|---|---|---|
| **0** | Sun | 2 | 16.99 | 1.01 |
| **1** | Sun | 3 | 10.34 | 1.66 |
| **2** | Sun | 3 | 21.01 | 3.50 |
| **3** | Sun | 2 | 23.68 | 3.31 |
| **4** | Sun | 4 | 24.59 | 3.61 |
| **...** | ... | ... | ... | ... |
| **239** | Sat | 3 | 29.03 | 5.92 |
| **240** | Sat | 2 | 27.18 | 2.00 |
| **241** | Sat | 2 | 22.67 | 2.00 |
| **242** | Sat | 2 | 17.82 | 1.75 |
| **243** | Thur | 2 | 18.78 | 3.00 |

244 rows × 4 columns

In [98]:
```
df.groupby('day').mean()
#df.groupby('day').agg('mean')
#df.groupby('day').apply('mean')
```

Out[98]:

| | size | total_bill | tip |
|---|---|---|---|
| **day** | | | |
| **Fri** | 2.105263 | 17.151579 | 2.734737 |
| **Sat** | 2.517241 | 20.441379 | 2.993103 |
| **Sun** | 2.842105 | 21.410000 | 3.255132 |
| **Thur** | 2.451613 | 17.682742 | 2.771452 |

In [99]:
```
df.groupby('day').mean().reset_index()
```

Out[99]:

| | day | size | total_bill | tip |
|---|---|---|---|---|
| **0** | Fri | 2.105263 | 17.151579 | 2.734737 |
| **1** | Sat | 2.517241 | 20.441379 | 2.993103 |
| **2** | Sun | 2.842105 | 21.410000 | 3.255132 |
| **3** | Thur | 2.451613 | 17.682742 | 2.771452 |

In [100…
```
df.groupby('day').transform('mean')
```

Out[100]:

| | size | total_bill | tip |
|---|---|---|---|
| **0** | 2.842105 | 21.410000 | 3.255132 |
| **1** | 2.842105 | 21.410000 | 3.255132 |
| **2** | 2.842105 | 21.410000 | 3.255132 |
| **3** | 2.842105 | 21.410000 | 3.255132 |
| **4** | 2.842105 | 21.410000 | 3.255132 |
| **...** | ... | ... | ... |
| **239** | 2.517241 | 20.441379 | 2.993103 |
| **240** | 2.517241 | 20.441379 | 2.993103 |
| **241** | 2.517241 | 20.441379 | 2.993103 |
| **242** | 2.517241 | 20.441379 | 2.993103 |
| **243** | 2.451613 | 17.682742 | 2.771452 |

244 rows × 3 columns

In [101…

```
df['day_avg_tip'] = df.groupby('day')['tip'].transform('mean')

df
```

Out[101]:

| | day | size | total_bill | tip | day_avg_tip |
|---|---|---|---|---|---|
| **0** | Sun | 2 | 16.99 | 1.01 | 3.255132 |
| **1** | Sun | 3 | 10.34 | 1.66 | 3.255132 |
| **2** | Sun | 3 | 21.01 | 3.50 | 3.255132 |
| **3** | Sun | 2 | 23.68 | 3.31 | 3.255132 |
| **4** | Sun | 4 | 24.59 | 3.61 | 3.255132 |
| **...** | ... | ... | ... | ... | ... |
| **239** | Sat | 3 | 29.03 | 5.92 | 2.993103 |
| **240** | Sat | 2 | 27.18 | 2.00 | 2.993103 |
| **241** | Sat | 2 | 22.67 | 2.00 | 2.993103 |
| **242** | Sat | 2 | 17.82 | 1.75 | 2.993103 |
| **243** | Thur | 2 | 18.78 | 3.00 | 2.771452 |

244 rows × 5 columns

---

## Column-Wise and Multiple Function Application

As you've already seen, aggregating a Series or all of the columns of a DataFrame is a matter of using aggregate with the desired function or calling a method like mean or std.

However, you may want to aggregate using a different function depending on the column, or multiple functions at once.

```
```

In [102…]
```
df = pd.read_csv('examples/tips.csv')

df['tip_pct'] = df['tip'] / df['total_bill']

df
```

Out[102]:

| | total_bill | tip | sex | smoker | day | time | size | tip_pct |
|---|---|---|---|---|---|---|---|---|
| **0** | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 | 0.059447 |
| **1** | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 | 0.160542 |
| **2** | 21.01 | 3.50 | Male | No | Sun | Dinner | 3 | 0.166587 |
| **3** | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 | 0.139780 |
| **4** | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 | 0.146808 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **239** | 29.03 | 5.92 | Male | No | Sat | Dinner | 3 | 0.203927 |
| **240** | 27.18 | 2.00 | Female | Yes | Sat | Dinner | 2 | 0.073584 |
| **241** | 22.67 | 2.00 | Male | Yes | Sat | Dinner | 2 | 0.088222 |
| **242** | 17.82 | 1.75 | Male | No | Sat | Dinner | 2 | 0.098204 |
| **243** | 18.78 | 3.00 | Female | No | Thur | Dinner | 2 | 0.159744 |

244 rows × 8 columns

In [103…]
```
df.groupby(['day','smoker'])['tip_pct'].agg('mean')
```

Out[103]:
```
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

If you pass a list of functions or function names instead, you get back a DataFrame with column names taken from the functions:

In [104…]
```
df.groupby(['day','smoker'])['tip_pct'].agg(['mean','median','std'])
```

Out[104]:

| | | mean | median | std |
|---|---|---|---|---|
| **day** | **smoker** | | | |
| **Fri** | **No** | 0.151650 | 0.149241 | 0.028123 |
| | **Yes** | 0.174783 | 0.173913 | 0.051293 |
| **Sat** | **No** | 0.158048 | 0.150152 | 0.039767 |
| | **Yes** | 0.147906 | 0.153624 | 0.061375 |
| **Sun** | **No** | 0.160113 | 0.161665 | 0.042347 |
| | **Yes** | 0.187250 | 0.138122 | 0.154134 |
| **Thur** | **No** | 0.160298 | 0.153492 | 0.038774 |
| | **Yes** | 0.163863 | 0.153846 | 0.039389 |

Here we passed a list of aggregation functions to agg to evaluate indepedently on the data groups.

The most general-purpose GroupBy method is **apply**.

Suppose you wanted to select the top five tip_pct values by group. First, write a function that selects the rows with the largest values in a particular column:

In [105…
```
def top(df, n=5, column='tip_pct'):
    return df.sort_values(by=column)[-n:]
```

In [106…
```
top(df)
```

Out[106]:

| | total_bill | tip | sex | smoker | day | time | size | tip_pct |
|---|---|---|---|---|---|---|---|---|
| **183** | 23.17 | 6.50 | Male | Yes | Sun | Dinner | 4 | 0.280535 |
| **232** | 11.61 | 3.39 | Male | No | Sat | Dinner | 2 | 0.291990 |
| **67** | 3.07 | 1.00 | Female | Yes | Sat | Dinner | 1 | 0.325733 |
| **178** | 9.60 | 4.00 | Female | Yes | Sun | Dinner | 2 | 0.416667 |
| **172** | 7.25 | 5.15 | Male | Yes | Sun | Dinner | 2 | 0.710345 |

Now, if we group by smoker, say, and call apply with this function, we get the following:

In [107…
```
df.groupby('smoker').apply(top)
```

Out[107]:

| smoker | | total_bill | tip | sex | smoker | day | time | size | tip_pct |
|---|---|---|---|---|---|---|---|---|---|
| **No** | **88** | 24.71 | 5.85 | Male | No | Thur | Lunch | 2 | 0.236746 |
| | **185** | 20.69 | 5.00 | Male | No | Sun | Dinner | 5 | 0.241663 |
| | **51** | 10.29 | 2.60 | Female | No | Sun | Dinner | 2 | 0.252672 |
| | **149** | 7.51 | 2.00 | Male | No | Thur | Lunch | 2 | 0.266312 |
| | **232** | 11.61 | 3.39 | Male | No | Sat | Dinner | 2 | 0.291990 |
| **Yes** | **109** | 14.31 | 4.00 | Female | Yes | Sat | Dinner | 2 | 0.279525 |
| | **183** | 23.17 | 6.50 | Male | Yes | Sun | Dinner | 4 | 0.280535 |
| | **67** | 3.07 | 1.00 | Female | Yes | Sat | Dinner | 1 | 0.325733 |
| | **178** | 9.60 | 4.00 | Female | Yes | Sun | Dinner | 2 | 0.416667 |
| | **172** | 7.25 | 5.15 | Male | Yes | Sun | Dinner | 2 | 0.710345 |

What has happened here? The top function is called on each row group from the DataFrame. The result therefore has a hierarchical index whose inner level contains index values from the original DataFrame.

If you pass a function to apply that takes other **arguments or keywords**, you can pass these after the function:

In [108…
```
df.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
```

Out[108]:

| smoker | day | | total_bill | tip | sex | smoker | day | time | size | tip_pct |
|---|---|---|---|---|---|---|---|---|---|---|
| **No** | **Fri** | **94** | 22.75 | 3.25 | Female | No | Fri | Dinner | 2 | 0.142857 |
| | **Sat** | **212** | 48.33 | 9.00 | Male | No | Sat | Dinner | 4 | 0.186220 |
| | **Sun** | **156** | 48.17 | 5.00 | Male | No | Sun | Dinner | 6 | 0.103799 |
| | **Thur** | **142** | 41.19 | 5.00 | Male | No | Thur | Lunch | 5 | 0.121389 |
| **Yes** | **Fri** | **95** | 40.17 | 4.73 | Male | Yes | Fri | Dinner | 4 | 0.117750 |
| | **Sat** | **170** | 50.81 | 10.00 | Male | Yes | Sat | Dinner | 3 | 0.196812 |
| | **Sun** | **182** | 45.35 | 3.50 | Male | Yes | Sun | Dinner | 3 | 0.077178 |
| | **Thur** | **197** | 43.11 | 5.00 | Female | Yes | Thur | Lunch | 4 | 0.115982 |

Beyond these basic usage mechanics, getting the most out of apply may require some creativity. What occurs inside the function passed is up to you; it only needs to return a **pandas object** or a **scalar value**.