

Data Aggregation and Group Operations

Prof. Gheith Abandah

Reference

- **Chapter 10: Data Aggregation and Group Operations**
- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Edition, 2018.
 - Material: <https://github.com/wesm/pypop-book>

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.
 - Split a pandas object **into pieces** using one or more keys
 - Calculate **group summary** statistics
 - Apply **within-group transformations** or other manipulations
 - Compute **pivot tables** and cross-tabulations

Outline

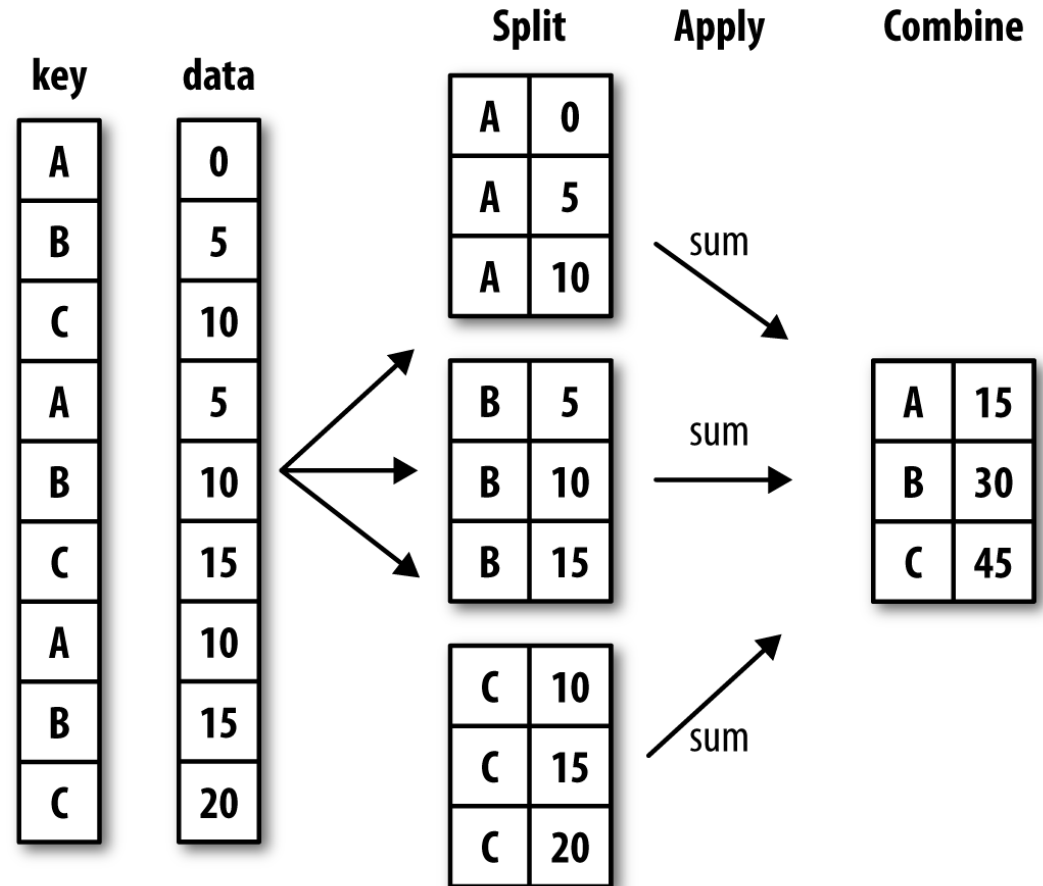
- 10.1 GroupBy Mechanics
- 10.2 Data Aggregation
- 10.3 Apply: General split-apply-combine
- 10.4 Pivot Tables and Cross-Tabulation

Outline

- 10.1 GroupBy Mechanics
 - 10.2 Data Aggregation
 - 10.3 Apply: General split-apply-combine
 - 10.4 Pivot Tables and Cross-Tabulation
- Iterating Over Groups
 - Selecting a Column or Subset of Columns
 - Grouping with Dicts and Series
 - Grouping with Functions

10.1 GroupBy Mechanics

- Group operations involve **split-apply-combine** sub-operations.
- **Grouping key**
 - A **list** or array of values of same length as the values grouped
 - A value indicating a **column name** in a DataFrame
 - A **dict or Series** giving a correspondence between the values on the axis being grouped and the group names
 - A **function**



10.1 GroupBy Mechanics

- **Example:** Group by a column
- Compute the **mean** of the **data1** using the labels from **key1**.
- The **groupby** method gives an object that can apply some operation to each of the groups.

	key1	key2	data1	data2
0	a	one	1.303569	1.411498
1	a	two	0.792029	-1.116429
2	b	one	-0.422705	0.589257
3	b	two	0.654579	-0.533492
4	a	one	0.567334	-1.029506

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

```
grouped  
<pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390>
```

```
grouped.mean()
```

```
key1  
a    0.746672  
b   -0.537585
```

```
grouped.size()
```

```
key1  
a    3  
b    2
```

The values and the keys can be default, one, or multiple columns.

Iterating Over Groups

- The **GroupBy** object **supports iteration**.
- Or the pieces can be put in a **dict**.

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

```
pieces['b']
   data1  data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```


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

```
('a', 'one')
   data1  data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
...
```


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.
- Can get a **Series** when a single column name is passed.

```
df.groupby(['key1', 'key2'])  
      [['data2']].mean()  
  
      data2  
key1 key2  
a     one  0.190996  
      two -1.116429  
b     one  0.589257  
      two -0.533492
```

 Hierarchical index

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

Grouping with Dicts and Series

- Grouping information may exist in a **dictionary** or a **series**.

- Example:** Group by mapping:

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

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

```
by_column.sum()  
           blue      red  
Joe      1.148819  2.478529  
Steve   -3.498286  0.783905  
Wes     -0.505604  0.442407  
Jim      1.265539 -2.598872  
Travis   0.691711  1.255747
```

```
people  
      a      b      c      d      e  
Joe  -1.403637  2.135849  1.131019  0.017800  1.746317  
Steve  0.313654  0.694288 -2.342447 -1.155839 -0.224036  
Wes    1.863238      NaN      NaN -0.505604 -1.420831  
Jim   -0.539711 -0.887612  1.136920  0.128619 -1.171549  
Travis -0.919159  1.603411 -0.469481  1.161192  0.571495
```

Grouping with Functions

- Any **function** passed as a group key will be **called once per index value**.
- **Example**: Group by length of index.

```
people.groupby(len).sum()
```

	a	b	c	d	e
3	-0.080110	1.248237	2.267939	-0.359185	-0.846063
5	0.313654	0.694288	-2.342447	-1.155839	-0.224036
6	-0.919159	1.603411	-0.469481	1.161192	0.571495

Outline

- 10.1 GroupBy Mechanics
- **10.2 Data Aggregation**
- 10.3 Apply: General split-apply-combine
- 10.4 Pivot Tables and Cross-Tabulation

10.2 Data Aggregation



- Aggregations refer to any data transformation that produces **scalar values from arrays**.
 - You can use your **own aggregation function**.
- ```
def peak_to_peak(arr):
 return arr.max() - arr.min()
grouped.agg(peak_to_peak)
```
- Some **other methods** also work.

Table 10-1. 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 variance |
| min, max      | Minimum and maximum of non-NA values                         |
| prod          | Product of non-NA values                                     |
| first, last   | First and last non-NA values                                 |

`grouped.describe()`

# Column-Wise and Multiple Function Application

```
Example: grouped.agg({'tip' : np.max,
 'size' : 'sum'})
```

- Pandas allows you to aggregate using a **different function** depending on the column, or **multiple functions** at once.

```
tips = pd.read_csv(
 'examples/tips.csv')
tips['tip_pct'] = tips['tip'] /
 tips['total_bill']
grouped = tips.groupby(['day',
 'smoker'])
functions = ['count', 'mean',
 'max']
result = grouped['tip_pct',
 'total_bill'].agg(functions)
```

```
result
```

|      |        | Tip_pct |          | total_bill |       |           |       |
|------|--------|---------|----------|------------|-------|-----------|-------|
|      |        | count   | mean     | max        | count | mean      | max   |
| day  | smoker |         |          |            |       |           |       |
| Fri  | No     | 4       | 0.151650 | 0.187735   | 4     | 18.420000 | 22.75 |
|      | Yes    | 15      | 0.174783 | 0.263480   | 15    | 16.813333 | 40.17 |
| Sat  | No     | 45      | 0.158048 | 0.291990   | 45    | 19.661778 | 48.33 |
|      | Yes    | 42      | 0.147906 | 0.325733   | 42    | 21.276667 | 50.81 |
| Sun  | No     | 57      | 0.160113 | 0.252672   | 57    | 20.506667 | 48.17 |
|      | Yes    | 19      | 0.187250 | 0.710345   | 19    | 24.120000 | 45.35 |
| Thur | No     | 45      | 0.160298 | 0.266312   | 45    | 17.113111 | 41.19 |
|      | Yes    | 17      | 0.163863 | 0.241255   | 17    | 19.190588 | 43.11 |

# Outline

- 10.1 GroupBy Mechanics
- 10.2 Data Aggregation
- **10.3 Apply: General split-apply-combine**
- 10.4 Pivot Tables and Cross-Tabulation

# 10.3 Apply: General split-apply-combine

- The **most general-purpose** GroupBy method is **apply**.
- **apply** applies the function to each group.
- **agg** aggregates each column for each group, so you end up with one value per column per group.
- **Example**: Filling missing values with group-specific values

data

Ohio 0.922264

New York -2.153545

Vermont NaN

Florida -0.375842

---

Oregon 0.329939

Nevada NaN

California 1.105913

Idaho NaN

East

West



# Example: Filling Missing Values with Group-Specific Values

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

data.groupby(group_key).mean()
East -0.535707
West 0.717926

fill_mean = lambda g:
 g.fillna(g.mean())

data.groupby(group_key).apply(
 fill_mean)

Ohio 0.922264
New York -2.153545
Vermont -0.535707
Florida -0.375842
Oregon 0.329939
Nevada 0.717926
California 1.105913
Idaho 0.717926
```

# Outline

- 10.1 GroupBy Mechanics
- 10.2 Data Aggregation
- 10.3 Apply: General split-apply-combine
- **10.4 Pivot Tables and Cross-Tabulation**

# Pivot Tables

- DataFrame has **pivot\_table** that performs groupby operations and adds partial totals (**margins**).

```
tips.pivot_table('tip_pct',
 index=['time', 'smoker'],
 columns='day', aggfunc='mean',
 margins=True)
```

| day    |        | Fri      | Sat      | Sun      | Thur     | All      |
|--------|--------|----------|----------|----------|----------|----------|
| time   | smoker |          |          |          |          |          |
| Dinner | No     | 0.139622 | 0.158048 | 0.160113 | 0.159744 | 0.158653 |
|        | Yes    | 0.165347 | 0.147906 | 0.187250 | NaN      | 0.160828 |
| Lunch  | No     | 0.187735 | NaN      | NaN      | 0.160311 | 0.160920 |
|        | Yes    | 0.188937 | NaN      | NaN      | 0.163863 | 0.170404 |
| All    |        | 0.169913 | 0.153152 | 0.166897 | 0.161276 | 0.160803 |

# Cross-Tabulation

- A **cross-tabulation** (**crosstab**) is a special case of a pivot table that computes **group frequencies**.

```
pd.crosstab([tips.time, tips.day],
 tips.smoker,
 margins=True)
```

| smoker |      | No  | Yes | All |
|--------|------|-----|-----|-----|
| time   | day  |     |     |     |
| Dinner | Fri  | 3   | 9   | 12  |
|        | Sat  | 45  | 42  | 87  |
|        | Sun  | 57  | 19  | 76  |
|        | Thur | 1   | 0   | 1   |
| Lunch  | Fri  | 1   | 6   | 7   |
|        | Thur | 44  | 17  | 61  |
| All    |      | 151 | 93  | 244 |

# Summary

- 10.1 GroupBy Mechanics
- 10.2 Data Aggregation
- 10.3 Apply: General split-apply-combine
- 10.4 Pivot Tables and Cross-Tabulation