Lecture 04. Data Aggregation and Group Operations

Instructor: Luping Yu

Mar 19, 2024

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 for reporting or visualization purposes.

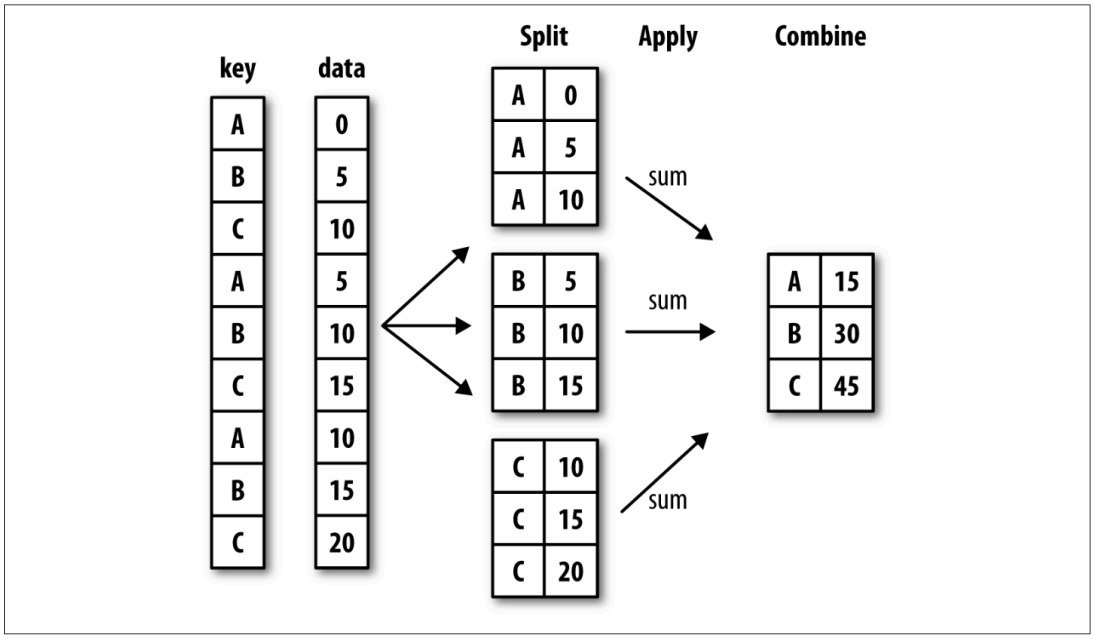
`pandas` provides a flexible `groupby()` interface, enabling you to slice, dice, and summarize datasets in a natural way.

GroupBy Mechanics

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

- In the first stage of the process, data is **split** into groups based on one or more keys that you provide.
- 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.

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 [ ]:
```
import pandas as pd
import numpy as np
#np.random.randn: generate random numbers

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
```

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 at `key1`:

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

grouped
```

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

The important thing here is that the data 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 [ ]:
```
df['data1'].groupby([df['key1'], df['key2']]).mean()
```

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.

A generally useful GroupBy method is `size()`, which returns a Series containing group sizes:

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

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, we could write:

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

Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. The preceding examples have used several of them, including `mean` and `size`. Built-in functions can be invoked using `agg()`.

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

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

df
```

In [ ]:
```
df.groupby('key1').max()
```

In [ ]:
```
df.groupby('key1').agg('min')
```

To use your own aggregation functions, pass any function that aggregates an array to the `apply` method:

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

In [ ]:
```
df.groupby(df['key1']).apply(peak_to_peak)
```

General split-apply-combine

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

Let's get started with the tipping dataset:

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

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

df
```

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

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

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

df
```

Column-Wise and Multiple Function Application

As you've already seen, aggregating data 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 [ ]:
```
df = pd.read_csv('examples/tips.csv')

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

df
```

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

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

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

The most general-purpose GroupBy method is `apply()`. Suppose you wanted to select the top five `tip_pct` values by group.

In [ ]:
```
df
```

First, write a function that selects the rows with the largest values in a particular column:

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

In [ ]:
```
top(df)
```

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

In [ ]:
```
df.groupby('sex').apply(top)
```

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 [ ]:
```
df.groupby(['sex', 'day']).apply(top, n=1, column='total_bill')
```

Beyond these basic usage mechanics, getting the most out of apply may require some creativity.