More on grouping and aggregation

You’ve discovered that pandas is a Python library that facilitates reviewing and manipulating tabular data. In addition, **`groupby()`** and **`agg()`** are essential **`DataFrame`** methods that data professionals use to group, aggregate, summarize, and better understand data. In this reading, you’ll review how these functions work, as well as when and how to apply them.

groupby()

The groupby() function is a method that belongs to the **`DataFrame`** class. It works by splitting data into groups based on specified criteria, applying a function to each group independently, then combining the results into a data structure. When applied to a dataframe, the function returns a groupby object. This groupby object serves as the foundation for different data manipulation operations, including:

- Aggregation: Computing summary statistics for each group
- Transformation: Applying functions to each group and returning modified data
- Filtration: Selecting specific groups based on certain conditions
- Iteration: Iterating over groups or values

Here are some examples that use the **`groupby()`** function on a dataframe consisting of different articles of clothing:

```
clothes = pd.DataFrame({'type': ['pants', 'shirt', 'shirt', 'pants', 'shirt', 'pants'],
                        'color': ['red', 'blue', 'green', 'blue', 'green', 'red'],
                        'price_usd': [20, 35, 50, 40, 100, 75],
                        'mass_g': [125, 440, 680, 200, 395, 485]})


clothes
```

RunReset

```
   color  mass_g  price_usd   type
0    red     125         20  pants
1   blue     440         35  shirt
2  green     680         50  shirt
3   blue     200         40  pants
4  green     395        100  shirt
5    red     485         75  pants
```

Grouping the dataframe by **`type`** results in a **`DataFrameGroupBy`** object:

```
grouped = clothes.groupby('type')
print(grouped)
print(type(grouped))
```

RunReset

```
<pandas.core.groupby.DataFrameGroupBy object at 0x7fc9d6323160>
<class 'pandas.core.groupby.DataFrameGroupBy'>
```

However, an aggregation function can be applied to the groupby object:

```
grouped = clothes.groupby('type')
grouped.mean()
```

RunReset

```
       mass_g  price_usd
type
pants   270.0  45.000000
shirt   505.0  61.666667
```

In the preceding example, **`groupby()`** combined all the items into groups based on their type and returned a **`DataFrame`** object containing the mean of each group for each numeric column in the dataframe. Note: In future versions of pandas it will be necessary to specify a **`numeric_only`** parameter when applying certain aggregation functions—like mean—to a groupby object. **`numeric_only`** refers to the datatype of each column. In earlier versions of pandas (like the version on this platform) it isn't necessary to specify **`numeric_only=True`**, but in future versions this must be done. Otherwise, it will be necessary to indicate the specific columns to be captured.)

In addition, groups may be created based on multiple columns:

```
clothes.groupby(['type', 'color']).min()
```

RunReset

```
             mass_g  price_usd
type  color
pants blue      200         40
      red       125         20
shirt blue      440         35
      green     395         50
```

In the preceding example, **`groupby()`** was called directly on the clothes dataframe. The data was grouped first by **`type`**, then by **`color`**. This resulted in four groups—the number of different existing combinations of values for type and color. Then, the **`min()`** function was applied to the result to filter each group by its minimum value.

To simply return the number of observations there are in each group, use the **`size()`** method. This will result in a **`Series`** object with the relevant information:

```
clothes.groupby(['type', 'color']).size()
```

RunReset

```
type   color
pants  blue      1
       red       2
shirt  blue      1
       green     2
dtype: int64
```

Built-in aggregation functions

The previous examples demonstrated the **`mean()`**, **`min()`**, and **`size()`** aggregation functions applied to groupby objects. There are many available built-in aggregation functions. Some of the more commonly used include:

- **`count()`**: The number of non-null values in each group
- **`sum()`**: The sum of values in each group
- **`mean()`**: The mean of values in each group
- **`median()`**: The median of values in each group
- **`min()`**: The minimum value in each group
- **`max()`**: The maximum value in each group
- **`std()`**: The standard deviation of values in each group
- **`var()`**: The variance of values in each group

agg()

The agg() function is useful when you want to apply multiple functions to a dataframe at the same time. **`agg()`** is a method that belongs to the **`DataFrame`** class. It stands for “aggregate.” Its most important parameters are:

- **`func`**: The function to be applied
- **`axis`**: The axis over which to apply the function (default= 0).

Following are some examples of how **`agg()`** can be used. Note that they demonstrate how this function can be used by itself (without **`groupby()`**). Note also that, due to platform limitations, some of the following code blocks are not executable. In these cases, output is provided as an image. Here is the original **`clothes`** dataframe again as a reminder:

```
clothes
```

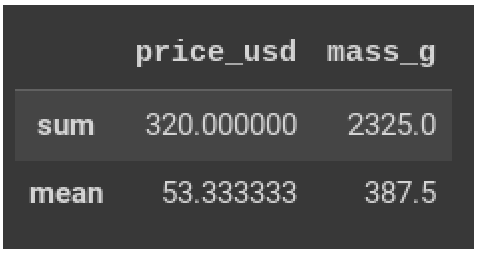
RunReset

```
   color  mass_g  price_usd   type
0    red     125         20  pants
1   blue     440         35  shirt
2  green     680         50  shirt
3   blue     200         40  pants
4  green     395        100  shirt
5    red     485         75  pants
```

The following example applies the **`sum()`** and **`mean()`** functions to the **`price`** and **`mass_g`** columns of the **`clothes`** dataframe.

```
clothes[['price_usd', 'mass_g']].agg(['sum', 'mean'])
```

Output:



	price_usd	mass_g
sum	320.000000	2325.0
mean	53.333333	387.5

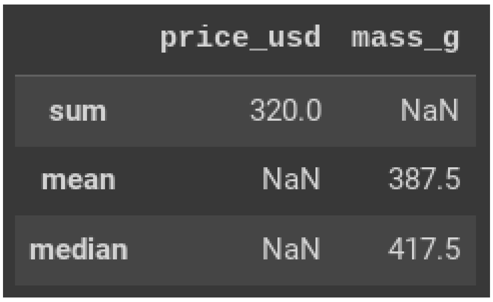
Notice the following:

- The two columns are subset from the dataframe before applying the **`agg()`** method. If you don't subset the relevant columns first, **`agg()`** will attempt to apply **`sum()`** and **`mean()`** to *all* of the columns, which wouldn't work because some columns contain strings. (Technically, **`sum()`** would work, but it would return something useless because it would just combine all the strings into one long string.)
- The **`sum()`** and **`mean()`** functions are entered as strings in a list, without their parentheses. This will work for any built-in aggregation function.

In this next example, different functions are applied to different columns.

```
             })
clothes.agg({'price_usd': 'sum',
             'mass_g': ['mean', 'median']
```

Output:



	price_usd	mass_g
sum	320.0	NaN
mean	NaN	387.5
median	NaN	417.5

Notice the following:

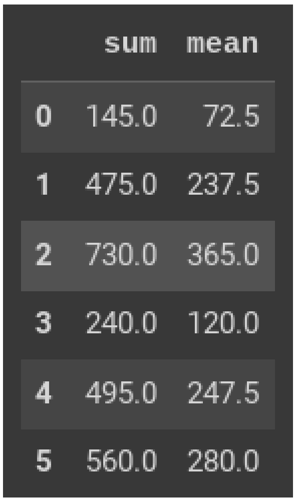
- Columns are not subset from the dataframe before applying the **`agg()`** function. This is unnecessary because the columns are specified within the **`agg()`** function itself.
- The argument to the **`agg()`** function is a dictionary whose keys are columns and whose values are the functions to be applied to those columns. If multiple functions are applied to a column, they are entered as a list. Again, each built-in function is entered as a string without parentheses.

- The resulting dataframe contains **NaN** values where a given function was not designated to be used.

The following example applies the **sum()** and **mean()** functions across axis 1. In other words, instead of applying the functions down each column, they're applied over each row.

```
clothes[['price_usd', 'mass_g']].agg(['sum', 'mean'], axis=1)
```

Output:



	sum	mean
0	145.0	72.5
1	475.0	237.5
2	730.0	365.0
3	240.0	120.0
4	495.0	247.5
5	560.0	280.0

groupby() with agg()

The **groupby()** and **agg()** functions are often used together. In such cases, first apply the **groupby()** function to a dataframe, then apply the **agg()** function to the result of the groupby. For reference, here is the **clothes** dataframe once again.

```
clothes
```

RunReset

```
   color  mass_g  price_usd   type
0    red     125         20  pants
1   blue     440         35  shirt
2  green     680         50  shirt
3   blue     200         40  pants
4  green     395        100  shirt
5    red     485         75  pants
```

In the following example, the items in **clothes** are grouped by **color**, then each of those groups has the **mean()** and **max()** functions applied to them at the **price_usd** and **mass_g** columns.

```
clothes.groupby('color').agg({'price_usd': ['mean', 'max'],
                              'mass_g': ['mean', 'max']})
```

RunReset

```
      price_usd      mass_g     
           mean  max   mean  max
color                           
blue       37.5   40  320.0  440
green      75.0  100  537.5  680
red        47.5   75  305.0  485
```

MultiIndex

You might have noticed that, when functions are applied to a groupby object, the resulting dataframe has tiered indices. This is an example of **MultiIndex**. MultiIndex is a hierarchical system of dataframe indexing. It enables you to store and manipulate data with any number of dimensions in lower dimensional data structures such as series and dataframes. This facilitates complex data manipulation.

This course will not require any deep knowledge of hierarchical indexing, but it's helpful to be familiar with it. Consider the following example:

```
grouped = clothes.groupby(['color', 'type']).agg(['mean', 'min'])
grouped
```

RunReset

```
            mass_g      price_usd    
              mean  min      mean min
color type                           
blue  pants  200.0  200      40.0  40
      shirt  440.0  440      35.0  35
green shirt  537.5  395      75.0  50
red   pants  305.0  125      47.5  20
```

Notice that **`color`** and **`type`** are positioned lower than the column names in the output. This indicates that **`color`** and **`type`** are no longer columns, but named row indices. Similarly, notice that **`price_usd`** and **`mass_g`** are positioned above **`mean`** and **`min`** in the output of column names, indicating a hierarchical column index.

If you inspect the row index, you'll get a **`MultiIndex`** object containing information about the row indices:

```
grouped.index
```

RunReset

```
MultiIndex(levels=[['blue', 'green', 'red'], ['pants', 'shirt']],
           labels=[[0, 0, 1, 2], [0, 1, 1, 0]],
           names=['color', 'type'])
```

The column index shows a **`MultiIndex`** object containing information about the column indices:

```
grouped.columns
```

RunReset

```
MultiIndex(levels=[['mass_g', 'price_usd'], ['mean', 'min']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

To perform selection on a dataframe with a MultiIndex, use **`loc[]`** selection and put indices in parentheses. Here are some examples on **`grouped`**, which is a dataframe with a two-level row index and a two-level column index. For reference, here is the **`grouped`** dataframe:

```
grouped
```

RunReset

```
            mass_g      price_usd
              mean  min      mean min
color type
blue  pants  200.0  200      40.0  40
      shirt  440.0  440      35.0  35
green shirt  537.5  395      75.0  50
red   pants  305.0  125      47.5  20
```

To select a first-level (top) column:

```
grouped.loc[:, 'price_usd']
```

RunReset

```
             mean  min
color type
blue  pants  40.0   40
      shirt  35.0   35
green shirt  75.0   50
red   pants  47.5   20
```

To select a second-level (bottom) column:

```
grouped.loc[:, ('price_usd', 'min')]
```

RunReset

```
color  type
blue   pants    40
       shirt    35
green  shirt    50
red    pants    20
Name: (price_usd, min), dtype: int64
```

To select first-level (left-most) row:

```
grouped.loc['blue', :]
```

RunReset

```
      mass_g       price_usd    
        mean  min       mean min
type                            
pants  200.0  200       40.0  40
shirt  440.0  440       35.0  35
```

To select a bottom-level (right-most) row:

```
grouped.loc[('green', 'shirt'), :]
```

RunReset

```
mass_g     mean    537.5
           min     395.0
price_usd  mean     75.0
           min      50.0
Name: (green, shirt), dtype: float64
```

And you can even select individual values:

```
grouped.loc[('blue', 'shirt'), ('mass_g', 'mean')]
```

RunReset

```
440.0
```

If you want to remove the row MultiIndex from a groupby result, include **`as_index=False`** as a parameter to your **`groupby()`** statement:

```
clothes.groupby(['color', 'type'], as_index=False).mean()
```

RunReset

```
   color   type  mass_g  price_usd
0   blue  pants   200.0       40.0
1   blue  shirt   440.0       35.0
2  green  shirt   537.5       75.0
3    red  pants   305.0       47.5
```

Notice how **`color`** and **`type`** are no longer row indices, but named columns. The row indices are the standard enumeration beginning from zero.

Again, you will not be expected to do any complex manipulations of hierarchically indexed data in this course, but it’s helpful to have a basic understanding of how MultIndex works, especially because `groupby()` manipulations typically result in a MultiIndex dataframe by default.

Key takeaways

`groupby()` will be an essential function in your work as a data professional, as it enables efficient combining and analysis of data. Similarly, `agg()` will help you apply multiple functions dynamically across a specified axis of a dataframe. Either on their own or when used together, these tools give data professionals deep access to data and help bring about successful projects.