

Get started

Open in app



Follow

590K Followers



You have **2** free member-only stories left this month. [Sign up for Medium and get an extra one](#)

# All Pandas groupby() You Should Know for Grouping Data and Performing Operations

Pandas tips and tricks to help you get started with data analysis



B. Chen Mar 12 · 8 min read ★



Photo by [AbsolutVision](#) on [Unsplash](#)

In exploratory data analysis, we often would like to analyze data by some categories. In SQL, the `GROUP BY` statement groups row that has the same category values into summary rows. In Pandas, SQL's `GROUP BY` operation is performed using the similarly named `groupby()` method. Pandas' `groupby()` allows us to split data into separate groups to perform computations for better analysis.

In this article, you'll learn the “group by” process (split-apply-combine) and how to use Pandas's `groupby()` function to group data and perform operations. This article is structured as follows:

1. What is Pandas `groupby()` and how to access groups information?
2. The “group by” process: split-apply-combine
3. Aggregation
4. Transformation
5. Filtration
6. Grouping by multiple categories
7. Resetting index with `as_index`
8. Handling missing values

For demonstration, we will use the [Titanic dataset](#) available on Kaggle.

```
df = pd.read_csv('data/titanic/train.csv')
df.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	493	0	1	Molson, Mr. Harry Markland	male	55.0	0	0	113787	30.5000	C30	S
1	53	1	1	Harper, Mrs. Henry Sleeper (Myna Haxtun)	female	49.0	1	0	PC 17572	76.7292	D33	C
2	388	1	2	Buss, Miss. Kate	female	36.0	0	0	27849	13.0000	NaN	S
3	192	0	2	Carbines, Mr. William	male	19.0	0	0	28424	13.0000	NaN	S
4	687	0	3	Panula, Mr. Jaako Arnold	male	14.0	4	1	3101295	39.6875	NaN	S

Titanic dataset (image by author)

Please check out [Notebook](#) for the source code.

## 1. What is Pandas `groupby()` and how to access groups information?

The role of `groupby()` is anytime we want to analyze data by some categories. The simplest call must have a column name. In our example, let's use the **Sex** column.

```
df_groupby_sex = df.groupby('Sex')
```

The statement literally means we would like to analyze our data by different Sex values. By calling the `type()` function on the result, we can see that it returns a **DataFrameGroupBy** object.

```
>>> type(df_groupby_sex)
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

The `groupby()` function returns a **DataFrameGroupBy** object but essentially describes how the rows of the original dataset have been split. There are some attributes and methods available for us to access groups information

We can use `ngroups` attribute to get the number of groups

```
>>> df_groupby_sex.ngroups
```

```
2
```

Use `groups` attribute to get groups object. Those integer numbers in the list are the row number.

```
>>> df_groupby_sex.groups
```

```
{'female': [1, 2, 5, 8, 10, 21, 22, 25, 26, 27, 36, 41, 44, 47, 51, 58, 60, 65, 70, 71, 72, 76, 77, 78, 80, 87, 88, 93, 94, 95, 100,
```

```
102, 104, 105, 109, 113, 116, 119, 120, 121, 123, 129, 134, 138,
144, 146, 147, ...], 'male': [0, 3, 4, ...]}
```

We can use `size()` method to compute and display group sizes.

```
>>> df_groupby_sex.size()
```

```
Sex
female    256
male      456
dtype: int64
```

To preview groups, we can call `first()` or `last()` to preview the result with the first or last entry.

```
df_groupby_sex.first()
```

```
df_groupby_sex.last()
```

	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
<b>Sex</b>											
<b>female</b>	53	1	1	Harper, Mrs. Henry Sleeper (Myna Haxtun)	49.0	1	0	PC 17572	76.7292	D33	C
<b>male</b>	493	0	1	Molson, Mr. Harry Markland	55.0	0	0	113787	30.5000	C30	S

```
df_groupby_sex.first()
```

	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
<b>Sex</b>											
<b>female</b>	859	1	3	Baclini, Mrs. Solomon (Latifa Qurban)	24.0	0	3	2666	19.2583	B57 B59 B63 B66	C
<b>male</b>	476	0	1	Clifford, Mr. George Quincy	35.0	0	0	110465	52.0000	A14	S

```
df_groupby_sex.last()
```

(image by author)

We can use `get_group()` method to retrieve one of the created groups

```
df_female = df_groupby_sex.get_group('female')
df_female.head()
```

PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	
1	53	1	1	Harper, Mrs. Henry Sleeper (Myna Haxtun)	49.0	1	0	PC 17572	76.7292	D33	C
2	388	1	2	Buss, Miss. Kate	36.0	0	0	27849	13.0000	NaN	S
5	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	55.0	0	0	248706	16.0000	NaN	S
8	168	0	3	Skoog, Mrs. William (Anna Bernhardina Karlsson)	45.0	1	4	347088	27.9000	NaN	S
10	541	1	1	Crosby, Miss. Harriet R	36.0	0	2	WE/P 5735	71.0000	B22	S

```
df_female = df.groupby_sex.get_group('female')
df_female.head()
```

(image by author)

## 2. The “group by” process: split-apply-combine

Generally speaking, “group by” is referring to a process involving one or more of the following steps:

*(1) Splitting the data into groups. (2). Applying a function to each group independently, (3) Combining the results into a data structure.*

*Out of these, Pandas `groupby()` is widely used for the split step and it’s the most straightforward. In fact, in many situations, we may wish to do something with those groups. In the apply step, we might wish to do one of the following:*

**Aggregation:** *compute a summary statistic for each group. for example, sum, mean, or count.*

**Transformation:** *perform some group-specific computations and return a like-indexed object. For example, standardize data within a group or replacing missing values within groups.*

**Filtration:** *discard some groups, according to a group-wise computation that evaluates `True` or `False`. For example, discard data that belongs to groups with only a few members or filter out data based on the group sum or mean.*

*By Pandas Official Tutorial: [groupby: split-apply-combine \[1\]](#)*

In the following article, we will explore the real use cases of the “group by” process.

## 2. Aggregation

Once **DataFrameGroupBy** has been created, several methods are available to perform a computation on the grouped data. An obvious one is to perform aggregation - compute a summary statistic for each group.

## With column

To perform aggregation on a specific column

```
>>> df.groupby('Sex').Age.max()
```

```
Sex
female    63.0
male      80.0
Name: Age, dtype: float64
```

## With agg() method

There is a method called `agg()` and it allows us to specify multiple aggregation functions at once.

```
df.groupby('Sex').Age.agg(['max', 'min', 'count', 'median', 'mean'])
```

	max	min	count	median	mean
Sex					
female	63.0	0.75	211	27.0	28.062796
male	80.0	0.42	355	28.0	30.804000

(image by author)

Sometimes, you may prefer to use a custom column name:

```
df.groupby('Sex').Age.agg(
    sex_max=('max'),
    sex_min=('min'),
)
```

	sex_max	sex_min
Sex		
female	63.0	0.75
male	80.0	0.42

(image by author)

If you would like to use a custom aggregation function:

```
def categorize(x):  
    m = x.mean()  
    return True if m > 29 else False
```

```
df.groupby('Sex').Age.agg(['max', 'mean', categorize])
```

	max	mean	categorize
Sex			
female	63.0	28.062796	False
male	80.0	30.804000	True

(image by author)

We can also use a lambda expression

```
df.groupby('Sex').Age.agg(  
    ['max', 'mean', lambda x: True if x.mean() > 50 else False]  
)
```

	max	mean	<lambda_0>
Sex			
female	63.0	28.062796	False
male	80.0	30.804000	True

(image by author)

Without column

Turns out when writing a `groupby()` we don't actually have to specify a column like **Age**. Without a column, it will perform the aggregation across all of the numeric columns

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

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
Sex							
female	425.140625	0.753906	2.128906	28.062796	0.652344	0.652344	44.835532
male	449.791667	0.188596	2.390351	30.804000	0.418860	0.236842	26.233971

(image by author)

Similarly, we can call `agg()` without a column.

```
df.groupby('Sex').agg(['mean', 'median'])
```

	PassengerId		Survived		Pclass		Age		SibSp		Parch		Fare	
Sex	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median
female	425.140625	396.0	0.753906	1	2.128906	2	28.062796	27.0	0.652344	0	0.652344	0	44.835532	24.15
male	449.791667	454.5	0.188596	0	2.390351	3	30.804000	28.0	0.418860	0	0.236842	0	26.233971	10.50

(image by author)

### 3. Transforming data

Transformation is a process in which we perform some group-specific computations and return a like-indexed (same length) object. When looking for transforming data, `transform()` and `apply()` are the most commonly used functions.

Let's create a lambda expression for Standardization.

```
standardization = lambda x: (x - x.mean()) / x.std()
```



To perform the standardization on **Age** column with `transform()`

```
df.groupby('Sex').Age.transform(standardization)
```

```
0      1.630657
1      1.516751
2      0.574994
...
707    -0.294321
708         NaN
709     0.956720
710     0.282784
711         NaN
Name: Age, Length: 712, dtype: float64
```

To perform the standardization on **Age** column using `apply()`

```
df.groupby('Sex').Age.apply(standardization)
```

```
0      1.630657
1      1.516751
2      0.574994
...
707    -0.294321
708         NaN
709     0.956720
710     0.282784
711         NaN
Name: Age, Length: 712, dtype: float64
```

If you would like to learn more `transform()` and `apply()`, please check out:

### When to use Pandas transform() function

Some of the most useful Pandas tricks

[towardsdatascience.com](https://towardsdatascience.com)

### Introduction to Pandas apply, applymap and map

An intuitive Pandas tutorial for how to apply a function using `apply()` and `applymap()`, and how to substitute value...

[towardsdatascience.com](https://towardsdatascience.com)

## 4. Filtration

Filtration is a process in which we discard some groups, according to a group-wise computation that evaluates True or False.

Let's take a look at how to discard data that belongs to groups with only a few members.

First, we group the data by **Cabin** and take a quick look at the size for each group.

```
df.groupby('Cabin').size()
```

```
Cabin
A10    1
A14    1
A16    1
A19    1
..
F2     2
F33    3
F4     1
G6     2
T      1
Length: 128, dtype: int64
```

Now let's filter data to return all passengers that lived in a cabin has  $\geq 4$  people. To do that, we use `filter()` method with a lambda expression.

```
df.groupby('Cabin').filter(lambda x: len(x) >= 4)
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
77	1	1	Carter, Mrs. William Ernest (Lucile Polk)	female	36.0	1	2	113760	120.0	B96 B98	S
96	1	1	Carter, Mr. William Ernest	male	36.0	1	2	113760	120.0	B96 B98	S
105	1	1	Fortune, Miss. Alice Elizabeth	female	24.0	3	2	19950	263.0	C23 C25 C27	S
208	0	1	Fortune, Mr. Mark	male	64.0	1	4	19950	263.0	C23 C25 C27	S
237	1	1	Carter, Miss. Lucile Polk	female	14.0	1	2	113760	120.0	B96 B98	S
302	0	1	Fortune, Mr. Charles Alexander	male	19.0	3	2	19950	263.0	C23 C25 C27	S
450	1	1	Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0	B96 B98	S
629	1	1	Fortune, Miss. Mabel Helen	female	23.0	3	2	19950	263.0	C23 C25 C27	S

## 6. Grouping by multiple categories

So far, we have been passing a single label to `groupby()` to group data by one column. Instead of a label, we can also pass a list of labels to work with multiple grouping.

```
# Creating a subset
df_subset = df.loc[:, ['Sex', 'Pclass', 'Age', 'Fare']]

# Group by multiple categories
df_subset.groupby(['Sex', 'Pclass']).mean()
```

		Age	Fare
female	1	33.873239	104.311563
	2	28.647541	21.658730
	3	22.389241	15.650187
male	1	42.006329	70.611438
	2	29.946667	19.463516
	3	26.720995	12.488267

(image by author)

## 7. Resetting index with `as_index`

Grouping by multiple categories will result in a MultiIndex DataFrame. However, it is not practical to have **Sex** and **Pclass** columns as the index (See image above) when we need to perform some data analysis.

We can call the `reset_index()` method on the DataFrame to reset them and use the default 0-based integer index instead.

```
df_groupby_multi = subset.groupby(['Sex', 'Pclass']).mean()
```

```
# Resetting index  
df_groupby_multi.reset_index()
```

	Sex	Pclass	Age	Fare
	female	1	33.873239	104.311563
		2	28.647541	21.658730
		3	22.389241	15.650187
	male	1	42.006329	70.611438
		2	29.946667	19.463516
		3	26.720995	12.488267

`df_groupby_multi.reset_index()`

	Sex	Pclass	Age	Fare
0	female	1	33.873239	104.311563
1	female	2	28.647541	21.658730
2	female	3	22.389241	15.650187
3	male	1	42.006329	70.611438
4	male	2	29.946667	19.463516
5	male	3	26.720995	12.488267

(image by author)

But there is a more effective way using the `as_index` argument. The argument is to configure whether the index is group labels or not. If it is set to `False`, the group labels are represented as columns instead of index.

```
subset.groupby(['Sex', 'Pclass'], as_index=False).mean()
```

	Sex	Pclass	Age	Fare
0	female	1	33.873239	104.311563
1	female	2	28.647541	21.658730
2	female	3	22.389241	15.650187
3	male	1	42.006329	70.611438
4	male	2	29.946667	19.463516
5	male	3	26.720995	12.488267

(image by author)

## 8. Handling missing values

The `groupby()` function ignores the missing values by default. Let's first create some missing values in the `Sex` column.

```
# Creating missing value in the Sex column
subset.iloc[80:100, 0] = np.nan

# Validating the missing values
subset.isna().sum()

Sex          20
Pclass       0
Age         146
Fare         0
dtype: int64
```

When calculating the mean value for each category in the **Sex** column, we won't get any information about the missing values.

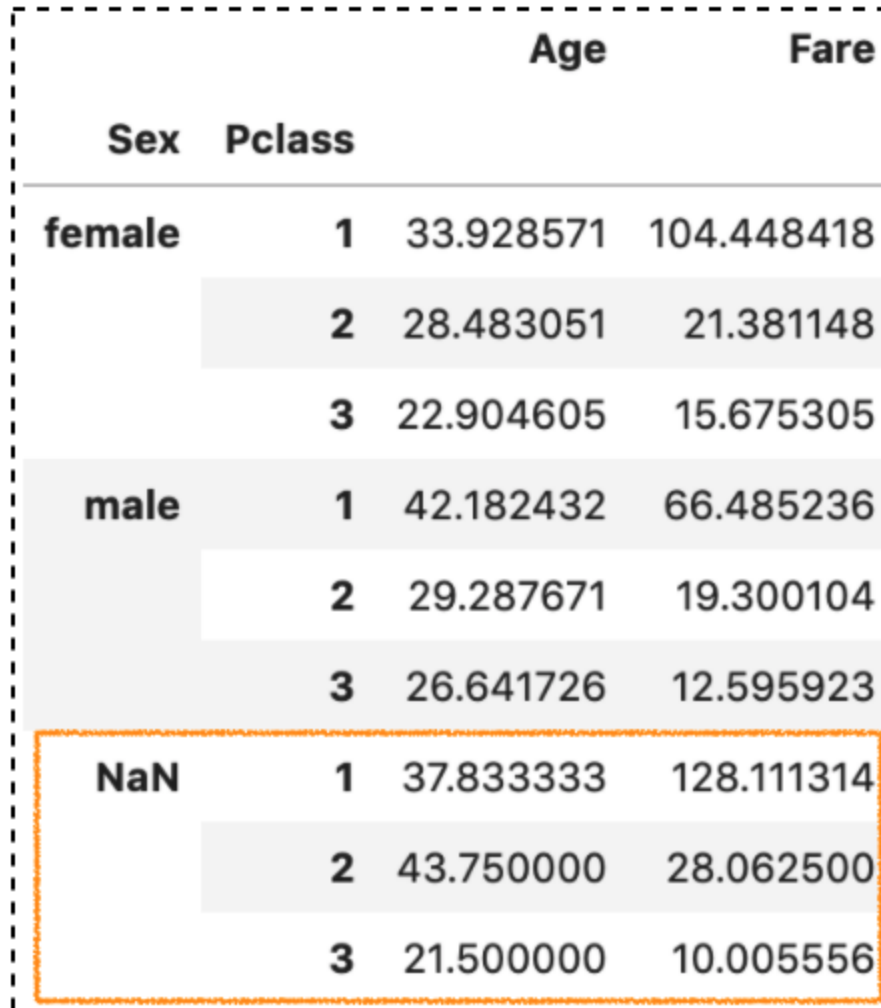
```
# The groupby function ignores the missing values by default.
subset.groupby(['Sex', 'Pclass']).mean()
```

			Age	Fare
	Sex	Pclass		
female		1	33.928571	104.448418
		2	28.483051	21.381148
		3	22.904605	15.675305
male		1	42.182432	66.485236
		2	29.287671	19.300104
		3	26.641726	12.595923

(image by author)

In some cases, we also need to get an overview of the missing values. We can set the `dropna` argument to `False` to include missing values.

```
subset.groupby(['Sex', 'Pclass'], dropna=False).mean()
```



		Age	Fare
female	1	33.928571	104.448418
	2	28.483051	21.381148
	3	22.904605	15.675305
male	1	42.182432	66.485236
	2	29.287671	19.300104
	3	26.641726	12.595923
NaN	1	37.833333	128.111314
	2	43.750000	28.062500
	3	21.500000	10.005556

(image by author)

## Conclusion

Pandas `groupby()` function is one of the most widely used functions in data analysis. It is really important because of its ability to aggregate, transform and filter data in each group.

I hope this article will help you to save time in learning Pandas. I recommend you to check out the [documentation](#) for the `groupby()` API and to know about other things you can do.

Thanks for reading. Please check out the [notebook](#) for the source code and stay tuned if you are interested in the practical aspect of machine learning.

## You may be interested in some of my other Pandas articles:

- [All Pandas json\\_normalize\(\)\\_you should know for flattening JSON](#)
- [Using Pandas method chaining to improve code readability](#)
- [How to do a Custom Sort on Pandas DataFrame](#)
- [All the Pandas shift\(\)\\_you should know for data analysis](#)
- [When to use Pandas transform\(\) function](#)
- [Pandas concat\(\) tricks you should know](#)
- [Difference between apply\(\) and transform\(\) in Pandas](#)
- [All the Pandas merge\(\)\\_you should know](#)
- [Working with datetime in Pandas DataFrame](#)
- [Pandas read\\_csv\(\) tricks you should know](#)
- [4 tricks you should know to parse date columns with Pandas read\\_csv\(\)](#)

More tutorials can be found on my [Github](#)

## References

- [1] Pandas Official Tutorial: [Group by: split-apply-combine](#)

---

## Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

Get this newsletter

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

