How to use the Split-Apply-Combine strategy in Pandas groupby

Master the Split-Apply-Combine pattern in Python with this visual guide to Pandas `groupby-apply`.

Filip Ciesielski [Follow]

Jul 16, 2019 · 8 min read ★

TL;DR

Pandas `groupby-apply` is an invaluable tool in a Python data scientist's toolkit. You can go pretty far with it without fully understanding all of its internal intricacies. However, sometimes that can manifest itself in unexpected behavior and errors. Ever had one of those? Or maybe you're struggling to figure out how to deal with more advanced data transformation problem? Then read this visual guide to Pandas `groupby-apply` paradigm to understand how it works, once and for all.

Source: Courtesy of my team at Sunscrapers.

Introduction

Solid understanding of the `groupby-apply` mechanism is often crucial when dealing with more advanced data transformations and pivot tables in Pandas.

That can be a steep learning curve for newcomers and a kind of 'gotcha' for intermediate Pandas users too. That's why I wanted to share a few visual guides with you that demonstrate what actually happens under the hood when we run the `groupby-apply` operations.

Here are a few things that I believe you should understand first to make working with more advanced Pandas pivot tables more straightforward:

Groupby — what does it do?

1. Apply — how does it work and what do we pass to it?

2. What happens during the combine stage?

3. What happens when the custom function returns a Series?

4. And what when it returns a DataFrame?

5. What happens when the custom function returns something else than the two above?

Read on to get answers to these questions and some extra insights about working with pivot tables in Pandas.
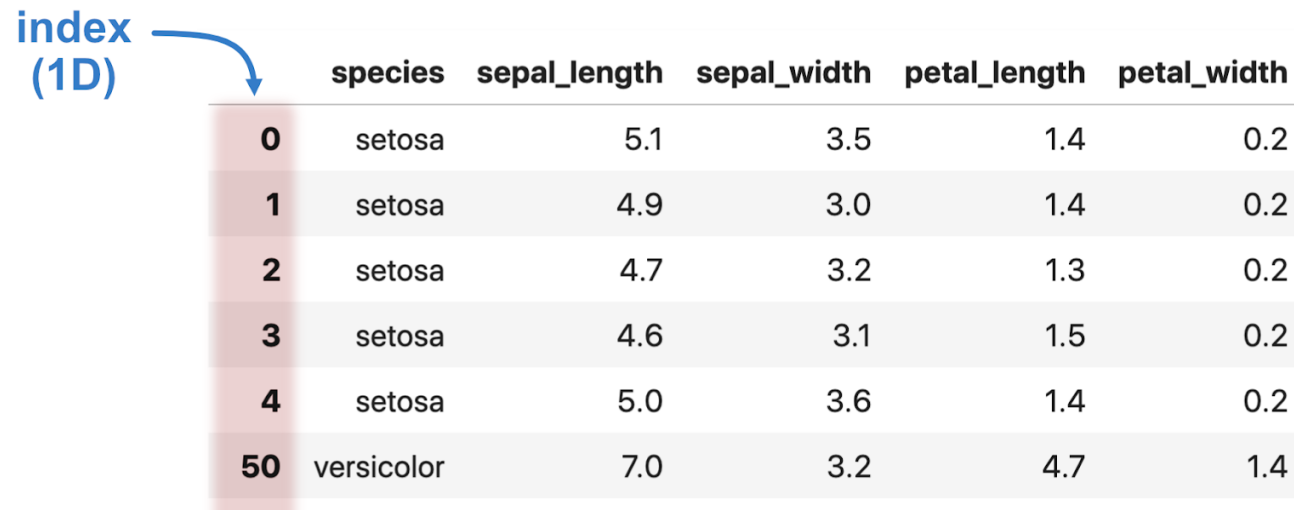
Our example table

Let's use the classic Iris dataset as an example (fetchable from the Seaborn plotting library) and limit it to just 15 rows for simplicity:

```
import seaborn as sns

iris_data = sns.load_dataset('iris')
df = iris_data.head(5).copy()
df = pd.concat([df, iris_data..iloc[50:55]])
df = pd.concat([df, iris_data.iloc[100:105]])
```

The table is quite small, but well sufficient enough for our needs and will suit us nicely for demonstration purposes in this article:

index (1D) →

	species	sepal_length	sepal_width	petal_length	petal_width
0	setosa	5.1	3.5	1.4	0.2
1	setosa	4.9	3.0	1.4	0.2
2	setosa	4.7	3.2	1.3	0.2
3	setosa	4.6	3.1	1.5	0.2
4	setosa	5.0	3.6	1.4	0.2
50	versicolor	7.0	3.2	4.7	1.4

51	versicolor	6.4	3.2	4.5	1.5
52	versicolor	6.9	3.1	4.9	1.5
53	versicolor	5.5	2.3	4.0	1.3
54	versicolor	6.5	2.8	4.6	1.5
100	virginica	6.3	3.3	6.0	2.5
101	virginica	5.8	2.7	5.1	1.9
102	virginica	7.1	3.0	5.9	2.1
103	virginica	6.3	2.9	5.6	1.8
104	virginica	6.5	3.0	5.8	2.2

Grouping with groupby()

Let's start with refreshing some basics about `groupby` and then build the complexity on top as we go along.

You can apply `groupby` method to a flat table with a simple 1D index column. That doesn't perform any operations on the table yet, but only returns a `DataFrameGroupBy` instance and so it needs to be chained to some kind of an aggregation function (for example, `sum`, `mean`, `min`, `max`, etc. see here for more) which will work on the grouped rows (we will discuss apply later on).

What's important to remember here is that it automatically concatenates the results of individual aggregated outcomes back into a single dataframe.

A very simple example can be grouping by a specific column value (eg. "species" in our table) and summing all the remaining applicable columns:

```
df.groupby('species').sum()
```


	species	sepal_length	sepal_width	petal_length	petal_width
0	setosa	5.1	3.5	1.4	0.2
1	setosa	4.9	3.0	1.4	0.2
2	setosa	4.7	3.2	1.3	0.2
3	setosa	4.6	3.1	1.5	0.2
4	setosa	5.0	3.6	1.4	0.2
50	versicolor	7.0	3.2	4.7	1.4
51	versicolor	6.4	3.2	4.5	1.5
52	versicolor	6.9	3.1	4.9	1.5
53	versicolor	5.5	2.3	4.0	1.3
54	versicolor	6.5	2.8	4.6	1.5
100	virginica	6.3	3.3	6.0	2.5
101	virginica	5.8	2.7	5.1	1.9
102	virginica	7.1	3.0	5.9	2.1
103	virginica	6.3	2.9	5.6	1.8
104	virginica	6.5	3.0	5.8	2.2

SUM

species	sepal_length	sepal_width	petal_length	petal_width
setosa	24.3	16.4	7.0	1.0
versicolor	32.3	14.6	22.7	7.2
virginica	32.0	14.9	28.4	10.5

SUM

SUM

Alternatively, we can specify which columns are to be summed up. A common mistake made by some is calculating the sum first and then sticking a column selector at the end like this:

```
df.groupby('species').sum()['sepal_width']  # ← BAD!
```

That means the summation is carried out first on every applicable column (numeric or string) and then a specified column is selected for output.

Here's a much better approach:

Specify the column before the aggregate function so only that one is summed up in the process, resulting in a SIGNIFICANT speed improvement (2.5x for this small table):

```
df.groupby('species')['sepal_width'].sum()  # ← BETTER & FASTER!
```

Note that since only a single column will be summed, the resulting output is a `pd.Series` object:

```
species
setosa 16.4
versicolor 14.6
virginica 14.9
Name: sepal_width, dtype: float64
```

However, if you want to automatically return a dataframe object directly (after all, it's much more readable), there's no need to cast it via `pd.Dataframe()` . Instead, provide the column name as a list to the column selection (essentially, use double brackets) like that:

```
df.groupby('species')[['sepal_width']].sum()
```

	sepal_width
species	
setosa	16.4
versicolor	14.6
virginica	14.9

Finally, `groupby` can take a list of column names and perform an aggregation function on all of the remaining applicable columns (that weren't mentioned before).

It's key to understand here that the resulting table will then have a `MultiIndex` object as the index and will behave slightly differently from a regular table.

```
multicol_sum = df.groupby(['species', 'petal_width']).sum()
```

species	petal_width	sepal_length	sepal_width	petal_length
setosa	0.2	24.3	16.4	7.0
versicolor	1.3	5.5	2.3	4.0
	1.4	7.0	3.2	4.7
	1.5	19.8	9.1	14.0
virginica	1.8	6.3	2.9	5.6

	1.9	5.8	2.7	5.1
	2.1	7.1	3.0	5.9
	2.2	6.5	3.0	5.8
	2.5	6.3	3.3	6.0

Useful tip: When working with `MultiIndex` tables, you can use `.xs` to select subsets of the dataframe by selecting a value and then a particular index level. The resulting output is usually also a dataframe object.

Here's an example:

```
multicol_sum.xs('virginica', level='species')
```

species	petal_width	sepal_length	sepal_width	petal_length
setosa	0.2	24.3	16.4	7.0
versicolor	1.3	5.5	2.3	4.0
	1.4	7.0	3.2	4.7
	1.5	19.8	9.1	14.0

.xs("virginica", level="species")

petal_width	sepal_length	sepal_width	petal_length

virginica	1.8	6.3	2.9	5.6
	1.9	5.8	2.7	5.1
	2.1	7.1	3.0	5.9
	2.2	6.5	3.0	5.8
	2.5	6.3	3.3	6.0

1.8	6.3	2.9	5.6
1.9	5.8	2.7	5.1
2.1	7.1	3.0	5.9
2.2	6.5	3.0	5.8
2.5	6.3	3.3	6.0

Also, don't forget that you can "flatten" the index into columns by running the `reset_idnex` method:

```
multi_sum.reset_index()
```

	species	petal_width	sepal_length	sepal_width	petal_length
0	setosa	0.2	24.3	16.4	7.0
1	versicolor	1.3	5.5	2.3	4.0
2	versicolor	1.4	7.0	3.2	4.7
3	versicolor	1.5	19.8	9.1	14.0
4	virginica	1.8	6.3	2.9	5.6
5	virginica	1.9	5.8	2.7	5.1
6	virginica	2.1	7.1	3.0	5.9
7	virginica	2.2	6.5	3.0	5.8

7	virginica	2.2	6.0	5.0	5.5
8	virginica	2.5	6.3	3.3	6.0

Additionally, if you pass a `drop=True` parameter to the `reset_index` function, your output dataframe will drop the columns that make up the `MultiIndex` and create a new index with incremental integer values.

The `apply()` method

Let's take it to the next level now.

Instead of using one of the stock functions provided by Pandas to operate on the groups we can define our own custom function and run it on the table via the `apply()` method.

To write a custom function well, you need to understand how the two methods work with each other in the so-called Groupby-Split-Apply-Combine chain mechanism (more on this here).

As I already mentioned, the first stage is creating a Pandas `groupby` object (`DataFrameGroupBy`) which provides an interface for the apply method to group rows together according to specified column(s) values.

We split the groups transiently and loop them over via an optimized Pandas inner code. We then pass each group to a specified function as either a `Series` or a `DataFrame` object.

The output of a function is stored temporarily until all groups have been processed. In the last stage all the results (from each function invocation) are finally combined into a single output.
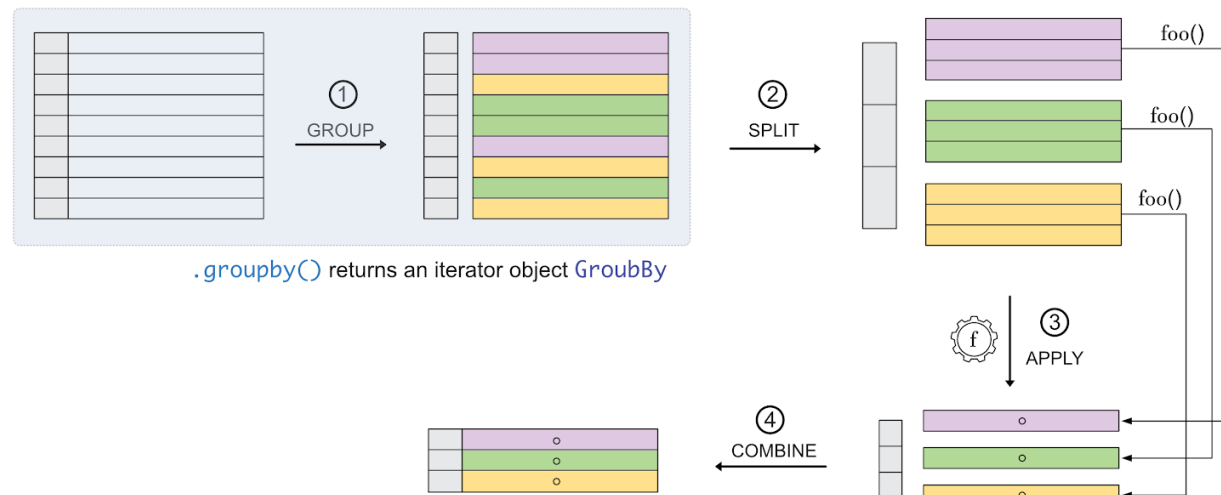
Here are a couple of points we need to keep in mind to avoid potential surprises that may arise when we work with the `groupby` and `apply` methods.

That's especially important on the rare occasions when we're forced to iterate over individual groups of rows with a *for-in* loop (usually, that's a sign of bad practice — however, it may be inevitable in certain situations).

1] Functions for apply

You can provide the logic for per-group operations in a couple of ways.

You can either define your separate function and pass it as an object to apply or pass a lambda expression directly. There's also the `.agg()` method which allows to pipe multiple aggregation functions by providing a name or a list of names of functions too (but that's outside of the scope of this article).

For example, the two commands

```
df.groupby('species').apply(lambda gr: gr.sum())
```

and

```
def my_sum(gr):
 return gr.sum()


df.groupby('species').apply(my_sum)
```

yield the exact same result:

species	species	sepal_length	sepal_width	petal_length	petal_width
setosa	setosasetosasetosasetosasetosa	24.3	16.4	7.0	1.0
versicolor	versicolorversicolorversicolorversicolorversic...	32.3	14.6	22.7	7.2
virginica	virginicavirginicavirginicavirginicavirginica	32.0	14.9	28.4	10.5

On a side note — yes, the columns with string values are also "summed," they are simply concatenated together.

2] Function input

The custom function should have one input parameter which will be either a `Series` or a `DataFrame` object, depending on whether a single or multiple columns are specified via the `groupby` method:

```
def foo(gr):
 print(type(gr))
 return None
```

```
df.groupby('species').apply(foo)
```

outputs:

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
```

, whereas (ignore the fact that there are 4 lines here — I'll explain it later)

```
df.groupby('species')['petal_length'].apply(foo)
```

returns:

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
```

3] Using the print() statement inside the custom function

When writing a complex table transformation you may sometimes want to walk yourself through the inner workings of the `apply`'ed function and add a `print()` statement to inspect what's going on during the operation (I do that from time to time to set my bearings when stuck).

It may come as quite a surprise to some of you, but when you run the following code, you will get one extra bit that you may not have anticipated:

```
def foo(gr):
 print(gr, '\n')

df.groupby('species').apply(func=foo)
```

and this is what's being printed out:

	species	sepal_length	sepal_width	petal_length	petal_width
0	setosa	5.1	3.5	1.4	0.2

???

```
   setosa              5.1         3.5            1.4          0.2
1  setosa              4.9         3.0            1.4          0.2       ←—— ¿¿¿
2  setosa              4.7         3.2            1.3          0.2
3  setosa              4.6         3.1            1.5          0.2
4  setosa              5.0         3.6            1.4          0.2

     species  sepal_length  sepal_width  petal_length  petal_width
0    setosa            5.1          3.5           1.4          0.2
1    setosa            4.9          3.0           1.4          0.2
2    setosa            4.7          3.2           1.3          0.2
3    setosa            4.6          3.1           1.5          0.2
4    setosa            5.0          3.6           1.4          0.2

         species  sepal_length  sepal_width  petal_length  petal_width
50   versicolor            7.0          3.2           4.7          1.4
51   versicolor            6.4          3.2           4.5          1.5
52   versicolor            6.9          3.1           4.9          1.5
53   versicolor            5.5          2.3           4.0          1.3
54   versicolor            6.5          2.8           4.6          1.5

          species  sepal_length  sepal_width  petal_length  petal_width
100   virginica            6.3          3.3           6.0          2.5
101   virginica            5.8          2.7           5.1          1.9
102   virginica            7.1          3.0           5.9          2.1
103   virginica            6.3          2.9           5.6          1.8
104   virginica            6.5          3.0           5.8          2.2
```

This mysterious behaviour is actually explained in the Pandas
documentation here, but it's easy to miss — I know I did and had to learn it
the hard way, so let me save you the trouble:

> "In the current implementation apply calls func twice on the first
> column/row to decide whether it can take a fast or slow code path. This can
> lead to unexpected behavior if func has side-effects, as they will take effect
> twice for the first column/row."

4] Function output

The final piece of the puzzle is the applied function's output and how it's handled in the *combine* stage. What happens there is that Pandas grabs all the outputs from each subsequent group operation and concatenates them with their corresponding labels, also provided by the `DataFrameGroupBy` object. The latter are then used to create a new index.

It implies that you can design a custom function to return anything, and it will be placed in a single row for a specific group under the group name label.

This example should illustrate that well:

```
def foo(gr):
  return pd.Series("This is a test")

df.groupby('species').apply(func=foo)
```

will create:

species	
setosa	This is a test
versicolor	This is a test
virginica	This is a test

What happens in most of the cases though (e.g., with the sum function) is that each iteration returns a Pandas `Series` object per row where the index values are used to assort the values to the right column name in the final dataframe.
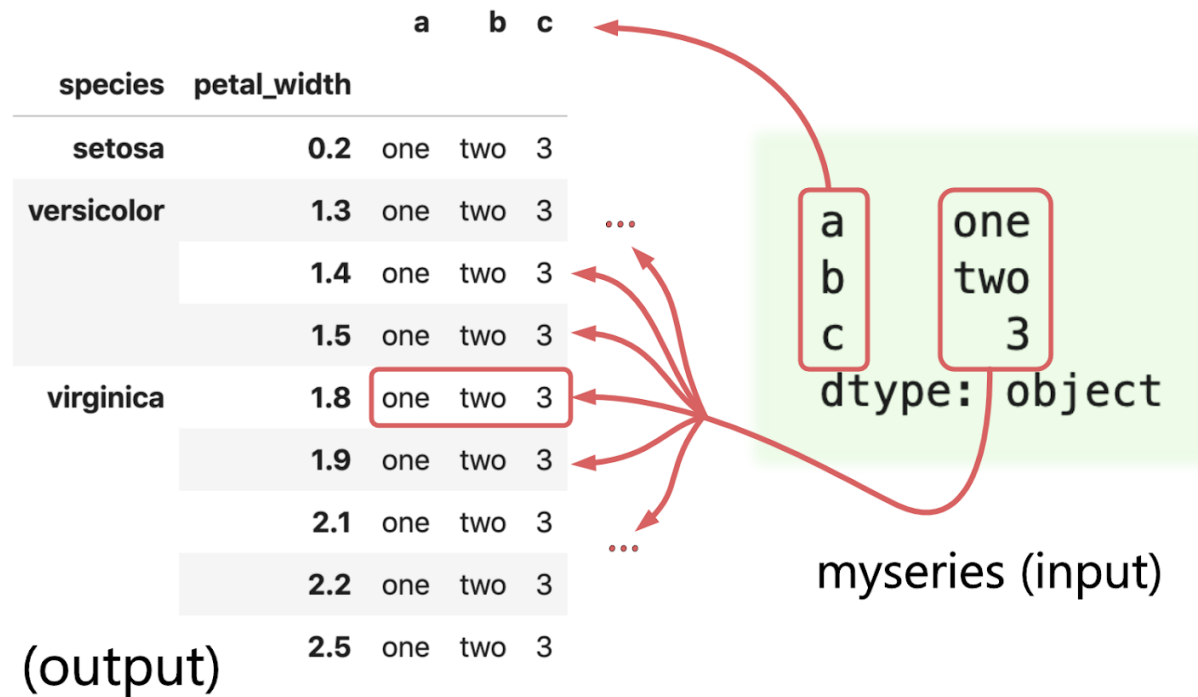
This example should illustrate that:

Let's create a custom `Series` object and return it on every group via the `apply` method:

```
myseries = pd.Series(
  data=['one', 'two', '3'],
  index=['a', 'b', 'c']
  )
```

```
def foo(gr):
 return myseries


df2.groupby(['species', 'petal_width']).apply(func=foo)
```

		a	b	c
species	**petal_width**			
setosa	0.2	one	two	3
versicolor	1.3	one	two	3
	1.4	one	two	3
	1.5	one	two	3
virginica	1.8	one	two	3
	1.9	one	two	3
	2.1	one	two	3
	2.2	one	two	3
	2.5	one	two	3

(output)

```
a      one
b      two
c        3
dtype: object
```

myseries (input)

What if we return a `DataFrame` per each iterated group? The result is somewhat interesting because it will create a table with a `MultiIndex`

structure. It will append the `DataFrame` into each row as is and its index will be integrated with the groups label value, for example:

```
def foo(gr):
  return pd.DataFrame(myseries)


df2.groupby(['species', 'petal_width']).apply(func=foo)
```


myseries
cast into
data frame
by the custom function

		c	3
virginica	1.8	a	one
		b	two
		c	3
	1.9	a	one
		b	two
		c	3
	2.1	a	one
		b	two
		c	3
	2.2	a	one
		b	two
		c	3
	2.5	a	one
		b	two
		c	3

And that's it for now.

Understanding these issues should make it easier for you to work with some of the more complex pivots and operations on Pandas tables. I revisit these basics from time to time or whenever I get stuck — and it always helps me to speed up the process a lot!

Data Science   Python   Pandas   Data Analysis   Tutorial

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

About   Help   Legal