Split-Apply-Combine Strategy for Data Mining

medium.com/analytics-vidhya/split-apply-combine-strategy-for-data-mining-4fd6e2a0cc99

7 oktober 2020

This is your free member-only story this month.

Top highlight

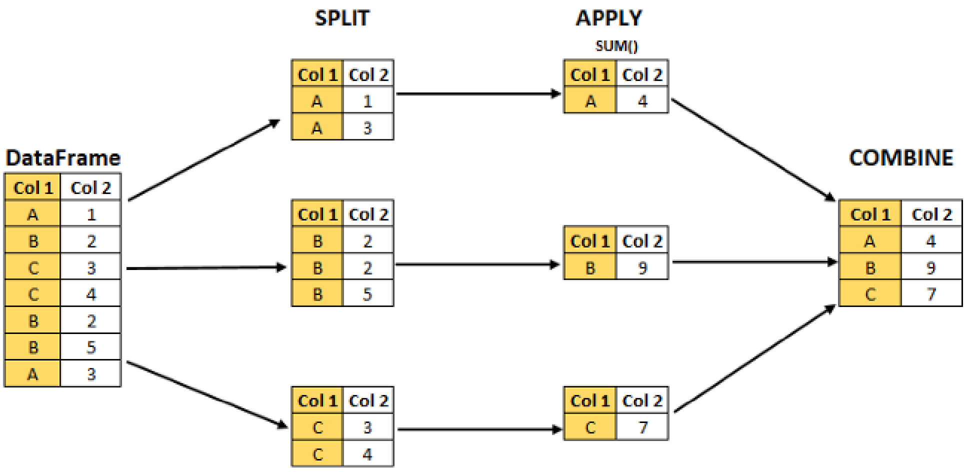
In a typical exploratory data analysis, we approach the problem by dividing the data set at some granular level and then aggregating the data at that granularity in order to understand the central tendency. Similarly, a famous (must read) paper by, Hadley Wickham, outlines split-apply-combine strategy as one of the most common strategies in data analysis. Be it Marketing Segmentation, or any Behavioral Research, we use this technique at some point during our analysis.

Introduction

This article attempts to illustrate split-apply-combine strategy in which we break up a big problem into small manageable pieces (Split), operate on each piece independently (Apply) and then put all the pieces back together (Combine). Split-Apply-Combine can be used by many existing tools by using GroupBy function in SQL and Python, LOD in Tableau, and by using plyr functions in R to name a few. In this article, we will not be discussing only the implementation of this strategy, but also we will see some relevant application of this strategy in Feature Engineering.

In Python we do this by using GroupBy and it involves one or more of the three steps of the Split-Apply-Combine strategy. Let us start by defining each of the three steps:

Figure A: Shows the Split-Apply-Combine using an aggregation function.



1. **Split:** Split the data into groups based on some criteria thereby creating a GroupBy object. (We can use the column or a combination of columns to split the data into groups)
2. **Apply:** Apply a function to each group independently. (Aggregate, Transform, or Filter the data in this step)
3. **Combine:** Combine the results into a data structure (Pandas Series, Pandas DataFrame)

Dataset

- To go a bit deeper , lets create a fictitious data to serve as an example. Have a thorough look at the dataframe(data_sales), given below, because it will be used throughout this article.
- To access the code used in this article please visit ***this link***.

```
Import Libraries and create a small Dataset to work on.
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Create an Example Data-set in the form of dictionary having key value pairs.

```
sales_dict={'colour':['Yellow','Black','Blue','Red','Yellow','Black','Blue',
                      'Red','Yellow','Black','Blue','Red','Yellow','Black','Blue','Red','Blue','Red'],
            'sales':[100000,150000,80000,90000,200000,145000,120000,
                     300000,250000,200000,160000,90000,90100,150000,142000,130000,400000,350000],
           'transactions':[100,150,820,920,230,120,70,250,250,110,130,860,980,300,150,170,230,280],
            'product':['type A','type A','type A','type A','type A','type A','type A',
                       'type A','type A','type B','type B','type B','type B','type B','type B','type B','type B','type B']}
```

```
data_sales=pd.DataFrame(sales_dict)
```

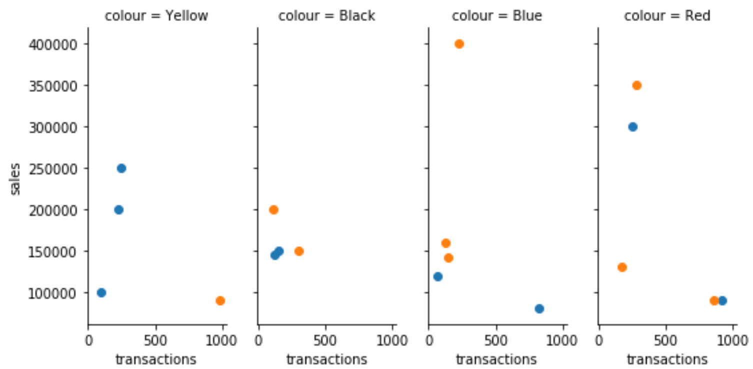
```
data_sales
```

To summarize the whole data, seaborn library have been used to create a visualization, which includes all the data graphically.

	colour	sales	transactions	product
0	Yellow	100000	100	type A
1	Black	150000	150	type A
2	Blue	80000	820	type A
3	Red	90000	920	type A
4	Yellow	200000	230	type A
5	Black	145000	120	type A
6	Blue	120000	70	type A
7	Red	300000	250	type A
8	Yellow	250000	250	type A
9	Black	200000	110	type B
10	Blue	160000	130	type B
11	Red	90000	860	type B
12	Yellow	90100	980	type B
13	Black	150000	300	type B
14	Blue	142000	150	type B
15	Red	130000	170	type B
16	Blue	400000	230	type B
17	Red	350000	280	type B

data_sales

```
graph = sns.FacetGrid(data_sales, col="colour", height=4, hue="product",aspect=.5)
graph.map(plt.scatter, "transactions", "sales");
```



Visual Data Summary

After creating and summarizing the data, as a first step lets move on to the first part of Split-Apply-Combine.

SPLIT : Create an Object.

In this step we will create the the groups from the dataframe ‘data_sales’ by grouping on the basis of the column ‘colour’.

```
# Split: Groupby the column 'colour'

data_gby = data_sales.groupby('colour')
print(type(data_gby))
```

```
<class 'pandas.core.groupby.groupby.DataFrameGroupBy'>
```

Once we apply the groupby() function on the dataframe, it creates the **GroupBy object** as a result. We can think of this object as a separate dataframe for each group. Each group has been created based on the categories in a grouped column (4 Groups will be created 'Black', 'Blue', 'Red', 'Yellow' from the column 'colour' of the dataframe in our case).

A GroupBy object stores the data of the individual groups in the form of key value pairs as in dictionary. To know the group names , we can either use attribute 'keys' or use the attribute 'groups' of the GroupBy object.

```
#  lets check the names of the groups

data_gby.groups
```

```
{'Black': Int64Index([1, 5, 9, 13], dtype='int64'),
 'Blue': Int64Index([2, 6, 10, 14, 16], dtype='int64'),
 'Red': Int64Index([3, 7, 11, 15, 17], dtype='int64'),
 'Yellow': Int64Index([0, 4, 8, 12], dtype='int64')}
```

For further clarity on Groups and its content, we can run a loop and print the key value pairs.

```
### 'key' is the name of the group and 'value' is the segmented rows from the original DataFrame.

for key, value in data_gby:
    print('GroupName: ',key)
    print(value)
    print('-----------------------------------------')
```

```
GroupName:  Black
   colour   sales  transactions product
1   Black  150000           150  type A
5   Black  145000           120  type A
9   Black  200000           110  type B
13  Black  150000           300  type B
-------------------------------------------
GroupName:  Blue
   colour   sales  transactions product
2    Blue   80000           820  type A
6    Blue  120000            70  type A
10   Blue  160000           130  type B
14   Blue  142000           150  type B
16   Blue  400000           230  type B
-------------------------------------------
GroupName:  Red
   colour   sales  transactions product
3     Red   90000           920  type A
7     Red  300000           250  type A
11    Red   90000           860  type B
15    Red  130000           170  type B
17    Red  350000           280  type B
-------------------------------------------
GroupName:  Yellow
    colour   sales  transactions product
0   Yellow  100000           100  type A
4   Yellow  200000           230  type A
8   Yellow  250000           250  type A
12  Yellow   90100           980  type B
-------------------------------------------
```

With the above example, I hope we have developed some clarity on the GroupBy object along with some of its attributes and methods. With this, now lets move forward to the next stage, which is **APPLY**.

APPLY: Apply some function on the Object.

Apply step can performed in three ways: **Aggregation, Transformation, & Filtering.** We all have good amount of experience in using Aggregation with GroupBy objects, but most of us might not have the same experience with the Transformation and Filtering. Here, we will discuss all the three with special focus on Transformation.

AGGREGATION:

I am assuming that we are already comfortable with applying the aggregation functions with GroupBy object, therefore I will start of with some interesting features of this function.

Aggregating in the Groups created by multiple columns:

By choosing multiple columns to create the group, we increase the granularity of the aggregation. For instance, while splitting we created 4 groups based on the column 'colours', which has 4 categories of colours, so we had 4 groups. Now, if include 'product' column, having 2 categories ('type A' and 'type B'), along with the 'colour' column, then we will be having total 8 categories (ex. 'type A-Blue', 'type A-Black' ..) in total (4 x 2). This would be more clear from the below mentioned code.

Groupby two columns and aggregation

```
data_prod_colour_index=data_sales.groupby(['product','colour'], as_index=True).sum()          # Note: as_index=True
```

```
data_prod_colour_index
```

product	colour	sales	transactions
type A	Black	295000	270
	Blue	200000	890
	Red	390000	1170
	Yellow	550000	580
type B	Black	350000	410
	Blue	702000	510
	Red	570000	1310
	Yellow	90100	980

The above code used the aggregation function as sum(), thus we get the sum of sales and the transactions to the level of granularity defined by the combination of the 'product' and 'colour' columns.

It is to be noted that we have used the parameter 'as_index=True', therefore we can see the 'product' and the 'colour' column as the index. On the contrary, if we take the same parameter as False then in our output we will not get the 'product' and 'colour' columns as the index but as the columns.

```
Groupby without index as grouped column
```

```
data_prod_colour_Noindex = data_sales.groupby(['product','colour'],as_index=False).sum()

data_prod_colour_Noindex
```

	product	colour	sales	transactions
0	type A	Black	295000	270
1	type A	Blue	200000	890
2	type A	Red	390000	1170
3	type A	Yellow	550000	580
4	type B	Black	350000	410
5	type B	Blue	702000	510
6	type B	Red	570000	1310
7	type B	Yellow	90100	980

Custom Aggregation grouped by Multiple Columns

In previous example we used only single type of aggregation function for all the columns; however, if we want to aggregate different columns with different aggregation functions then we can use the custom aggregation functionality of the aggregation function. For doing this we can pass on the dictionary to the aggregation function stating the column name as 'key' and function name as 'value'. Interestingly, we can also pass the multiple aggregation functions to a column. Let us see an example code below for more clarity.

```
## Custom Aggregation with GroupBy using Dictionary as a parameter inside aggregation function 'agg()'
```

```
data_sales.groupby(['product','colour'], as_index=True).agg({'sales': np.sum, 'transactions':[np.median,'count']})
```

		sales	transactions	
		sum	median	count
product	colour			
type A	Black	295000	135	2
	Blue	200000	445	2
	Red	390000	585	2
	Yellow	550000	230	3
type B	Black	350000	205	2
	Blue	702000	150	3
	Red	570000	280	3
	Yellow	90100	980	1

Real World Application of Aggregation function with the GroupBy Object:

Example 1:

Few days back one of my friends asked me to calculate volatility of different groups of the data. I suggested him to use ‘coefficient of variation’ as a measure of volatility. *Why only ‘coefficient of variation’ but why not ‘variance’ , we will discuss this in next article*. While writing this article I realized that I can use similar kind of example to show the application of the aggregation function in business. Just for the sake of example I have used the aforementioned sales data to illustrate its application.

In this example we are trying to find ‘which product and its colour combination have lowest variation. We have done this by , grouping the dataframe based on product and colour columns and then by calculating the coefficient of variation for each group. The code below will make it more clearer.

```python
## Define the function
def coeff_of_Variation(x):
    co_vn = x.std()/x.mean()
    return(co_vn)
```

```python
## use the above defined function in the aggregation.
cvn= data_sales.groupby(['product','colour'])['sales'].agg([coeff_of_Variation,np.mean,np.std])
```

```python
print(cvn)   ## The group 'type A- Black' has the lowest volatility
```

```
                coeff_of_Variation           mean            std
product colour                                                  
type A  Black             0.023970  147500.000000    3535.533906
        Blue              0.282843  100000.000000   28284.271247
        Red               0.761500  195000.000000  148492.424049
        Yellow            0.416598  183333.333333   76376.261583
type B  Black             0.202031  175000.000000   35355.339059
        Blue              0.615563  234000.000000  144041.660640
        Red               0.736842  190000.000000  140000.000000
        Yellow                 NaN   90100.000000            NaN
```

Example 2:

Till now we have seen how to group by multiple categorical columns. In our examples we have only seen the application of Groupby aggregation function while grouping on the basis of existing categorical columns; however, sometimes we may need to group by the numeric columns. How do we do that??

The answer is simple and it is just a two step process. Firstly, convert the numeric column to a categorical column. This can be done by binning/ bucketing, and mapping. For binning read this article, further in this example we have used mapping to convert the numeric column to categorical column. Secondly, group the dataframe based on new category and use the aggregate function to get aggregation based on the new categorical column. The code below is self explanatory and would help you to go deeper.

```
## Lets create a series for so that Number of transactions is greater than 250

greater_than_250 = data_sales['transactions']>250            ## A pandas series will be created.
```

```
greater_than_250.head()                    ## have a look at the series. the series has a binary outcome for each row
```

```
0    False
1    False
2     True
3     True
4    False
Name: transactions, dtype: bool
```

```
## In this step we convert the binary seies into the Categories. lets map the False as 'under250' and True as 'over250'.

categ_col= greater_than_250.map({True:'over250', False:'under250'})          ## Pass the mapping through the dictionary
```

```
categ_col.head()   ## Data After Mapping of sales column
```

```
0    under250
1    under250
2     over250
3     over250
4    under250
Name: transactions, dtype: object
```

```
## Calculate the mean of sales for these two categories (under250, and over250 ). It is to be noted that that the 'category_col'
## was not the part of the dataframe data_sales, still the group was created based on them. It is because the
## indexing of the 'category_col' is same as the dataframe data_sales.

data_sales.groupby(categ_col)['sales'].mean()
```

```
transactions
over250     141683.333333
under250    191416.666667
Name: sales, dtype: float64
```

TRANSFORMATION:

Transform function has high potential utility in Feature Engineering. It is a function/method used in conjunction with the GroupBy object. It is a bit counter-intuitive, therefore a bit difficult to understand. If somebody have used Tableau LOD (Fixed) function, then it will be easier for them to understand the transform function. The figure below illustrate the Split-Apply-Combine using transform function.

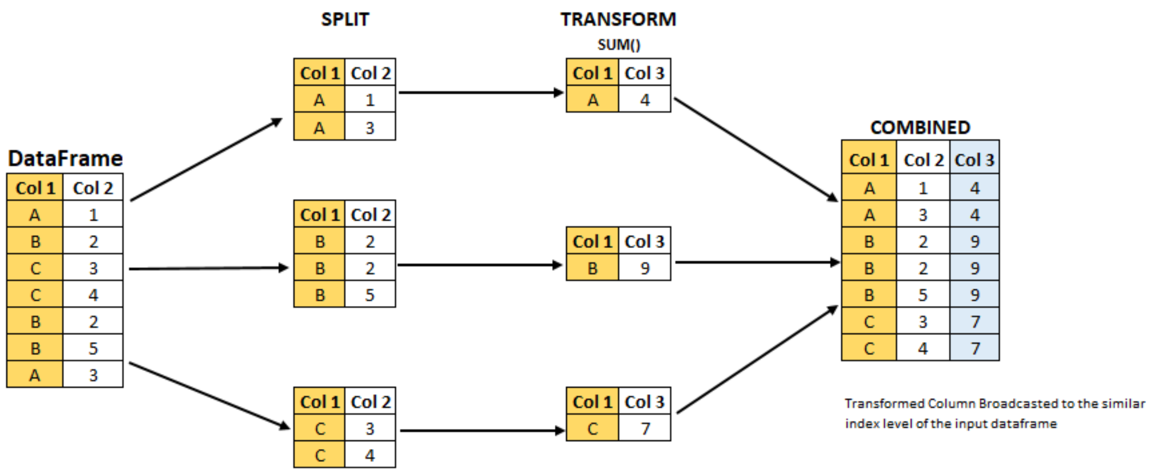


Figure B: Split-Transform-Combine

During aggregation we saw that shape of the input dataframe got reduced (Number of rows got reduced); however, it is to be noted that on using transform method the resulting output dataframe have same number of rows in output as in input. The output retained the length of the dataframe and it happened in two stages. Firstly, in the apply stage, the transform function (sum(); as shown in the Figure B) was applied , and at this stage the number of rows got reduced. Secondly, in the combine stage, the result of Apply stage was broadcasted to the original level of granularity, thereby producing the dataframe having the same length as the length of the dataframe in the input stage.

To make it more clear, we can use an example code using the same fictitious data set, which we have been using in this article.

```
### Apply the transform function (standard deviation:std()) on the sales column grouped by product column

### It is to be noted that the output has the same number of rows as in input
```

```
data_sales['sales_product_std'] = data_sales.groupby('product',as_index=True)['sales'].transform('std')
```

```
data_sales.loc[:,['colour','product','sales','transactions', 'sales_product_std']]
```

	colour	product	sales	transactions	sales_product_std
0	Yellow	type A	100000	100	75929.426297
1	Black	type A	150000	150	75929.426297
2	Blue	type A	80000	820	75929.426297
3	Red	type A	90000	920	75929.426297
4	Yellow	type A	200000	230	75929.426297
5	Black	type A	145000	120	75929.426297
6	Blue	type A	120000	70	75929.426297
7	Red	type A	300000	250	75929.426297
8	Yellow	type A	250000	250	75929.426297
9	Black	type B	200000	110	110783.301991
10	Blue	type B	160000	130	110783.301991
11	Red	type B	90000	860	110783.301991
12	Yellow	type B	90100	980	110783.301991
13	Black	type B	150000	300	110783.301991
14	Blue	type B	142000	150	110783.301991
15	Red	type B	130000	170	110783.301991
16	Blue	type B	400000	230	110783.301991
17	Red	type B	350000	280	110783.301991

Since now we have some understanding of the transform function, so now lets talk about its utility in data cleaning and Feature Engineering.

We can define a custom function and use it to transform the column. For instance, we can use a common example of standardization of a column based on group category through transform function. *It is to be noted that this standardization is not applied directly on entire column, but it is applied on the column based on the group(mean is the group mean and the std dev is the std dev of the group)*. The example below will throw more light on this concept.

```python
## Using custom function to transform. Standardization of column 'sales' grouped by column 'colour'

data_sales['sales_stdzed_colour'] = data_sales.groupby('colour')['sales'].transform(lambda x: (x-x.mean())/x.std())

print(data_sales.loc[:,['colour', 'sales', 'sales_stdzed_colour']])
```

```
    colour   sales  sales_stdzed_colour
0   Yellow  100000            -0.770946
1    Black  150000            -0.433682
2     Blue   80000            -0.794706
3      Red   90000            -0.824083
4   Yellow  200000             0.513429
5    Black  145000            -0.626430
6     Blue  120000            -0.478090
7      Red  300000             0.872558
8   Yellow  250000             1.155617
9    Black  200000             1.493795
10    Blue  160000            -0.161474
11     Red   90000            -0.824083
12  Yellow   90100            -0.898099
13   Black  150000            -0.433682
14    Blue  142000            -0.303951
15     Red  130000            -0.500913
16    Blue  400000             1.738221
17     Red  350000             1.276520
```

In the above code, we have used lambda function and within the lambda function we have used the two methods mean and standard deviation(std) on each row of the dataframe. *A confusion may arise from here that 'how a transform function is doing the row wise computation, when it is meant to do a group wise computation?'*. The whole story behind the scene is explained in the code below.

```python
## Calculation of Group Level mean and its broadcasting on to the original data frame.

data_sales['X.mean'] = data_sales.groupby('colour')['sales'].transform('mean')
```

```python
## Calculation of Group Level std dev and its broadcasting on to the original data frame.

data_sales['X.std'] = data_sales.groupby('colour')['sales'].transform('std')
```

```python
## Simple row wise standardization based on X, X.mean , and X.std columns

data_sales['simple_stdzed'] = (data_sales['sales']-data_sales['X.mean'])/data_sales['X.std']
```

```python
## Print both the transformed column and the column with simple calculation and see the difference between the two.
## Both the columns 'simple_stdzed', and 'sales_stdzed_colour' are exactly similar.
## Mean and Std Dev has been calculated based on groups thus we can see the values(X.mean,X.std) repeating for same colours.

print(data_sales.loc[:,['colour', 'sales','X.mean','X.std','simple_stdzed','sales_stdzed_colour']])
```

```
    colour   sales  X.mean          X.std  simple_stdzed  sales_stdzed_colour
0   Yellow  100000  160025   77858.862694      -0.770946            -0.770946
1    Black  150000  161250   25940.637360      -0.433682            -0.433682
2     Blue   80000  180400  126336.059777      -0.794706            -0.794706
3      Red   90000  192000  123773.987574      -0.824083            -0.824083
4   Yellow  200000  160025   77858.862694       0.513429             0.513429
5    Black  145000  161250   25940.637360      -0.626430            -0.626430
6     Blue  120000  180400  126336.059777      -0.478090            -0.478090
7      Red  300000  192000  123773.987574       0.872558             0.872558
8   Yellow  250000  160025   77858.862694       1.155617             1.155617
9    Black  200000  161250   25940.637360       1.493795             1.493795
10    Blue  160000  180400  126336.059777      -0.161474            -0.161474
11     Red   90000  192000  123773.987574      -0.824083            -0.824083
12  Yellow   90100  160025   77858.862694      -0.898099            -0.898099
13   Black  150000  161250   25940.637360      -0.433682            -0.433682
14    Blue  142000  180400  126336.059777      -0.303951            -0.303951
15     Red  130000  192000  123773.987574      -0.500913            -0.500913
16    Blue  400000  180400  126336.059777       1.738221             1.738221
17     Red  350000  192000  123773.987574       1.276520             1.276520
```

I hope from the above code the transformation operation is clear. We should now move to **Filter** operation of the Apply Section**.**

FILTERING:

As it is evident from the the name itself, it is used to filter the groups from the dataframes. The below mentioned code illustrates the operation.

```
If we want to filter the dataframe such that it contains only those colours that has average number of transaction greater than the average of some other colour.
```

```python
grouped = data_sales.groupby('colour')

Blue_avg_transaction= grouped['transactions','sales'].mean().loc['Blue','transactions']
Black_avg_transaction= grouped['transactions','sales'].mean().loc['Black','transactions']
Yellow_avg_transaction= grouped['transactions','sales'].mean().loc['Yellow','transactions']
Red_avg_transaction= grouped['transactions','sales'].mean().loc['Red','transactions']

print('Blue_avg_transaction: ', Blue_avg_transaction)
print('Black_avg_transaction:', Black_avg_transaction)
print('Yellow_avg_transaction: ', Yellow_avg_transaction)
print('Red_avg_transaction:', Red_avg_transaction)
```

```
Blue_avg_transaction:  280
Black_avg_transaction: 170
Yellow_avg_transaction:  390
Red_avg_transaction: 496
```

```python
## The output shows that

filt_df = grouped.filter(lambda x: x['transactions'].mean() > Black_avg_transaction)

print(filt_df.iloc[:,[0,2]])
```

```
    colour  transactions
0   Yellow           100
2     Blue           820
3      Red           920
4   Yellow           230
6     Blue            70
7      Red           250
8   Yellow           250
10    Blue           130
11     Red           860
12  Yellow           980
14    Blue           150
15     Red           170
16    Blue           230
17     Red           280
```

Filtering is easy to understand operation and with this APPLY section of *SPLIT-APPLY-COMBINE* comes to an end. Now, we will move to the last part, which is **COMBINE**.

COMBINE: Combine the result to get a new Object.

During the above discussions, the combine section has already been covered; however, there is one important point regarding this which I would like to share.

```python
## Aggregation function mean() applied on only one column 'sales'. Also note as_index=True.
d1 = data_sales.groupby('colour',as_index=True)['sales'].mean()
```

```python
type(d1) ## The output is Pandas Series
```

```
pandas.core.series.Series
```

```python
print(d1)
```

```
colour
Black     161250
Blue      180400
Red       192000
Yellow    160025
Name: sales, dtype: int64
```

```python
## Aggregation function mean() applied on only two columns
d2 = data_sales.groupby('colour',as_index=True).agg({'sales':np.mean, 'transactions': np.sum})
```

```python
type(d2)     ## The Output is Pandas DataFrame
```

```
pandas.core.frame.DataFrame
```

```python
print(d2)
```

```
         sales  transactions
colour
Black   161250           680
Blue    180400          1400
Red     192000          2480
Yellow  160025          1560
```

```
## Aggregation function mean() applied on only one column 'sales'.But now the result is dataframe bec parameter as_index=True
d3 = data_sales.groupby('colour',as_index=False)['sales'].mean()
```

```
type(d3)      ## The Output is Pandas DataFrame
```

```
pandas.core.frame.DataFrame
```

```
print(d3)
```

```
   colour   sales
0   Black  161250
1    Blue  180400
2     Red  192000
3  Yellow  160025
```

Aggregation doesn’t always lead to the creation of the dataframe. It depends primarily upon the parameter ‘as_index’, if the value of this parameter is ‘True’ then it depends upon the number of columns on which we are applying the aggregation function.

End Note

With this we come to the end of this article. I hope the codes and the related discussion would help the readers not only in developing the better intuitive understanding of the Split-Apply-Combine strategy, but also in application of this technique in data mining.

Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore

Share your thinking.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Write on Medium

About

Help

Legal

Medium