**How To Use .groupby() Effectively As A Data Scientist**

Asa data scientist, it is important to use the right tools and techniques to get the most out of the data. The Pandas library is a great tool for data manipulation, analysis, and visualization, and it is an essential part of any data scientist’s toolkit. However, it can be challenging to use Pandas efficiently, and this can lead to wasted time and effort. Fortunately, there are a few best practices that can help data scientists get the most out of their Pandas experience. From using vectorized operations to taking advantage of built-in functions, these best practices will help data scientists quickly and accurately analyze and visualize data using Pandas. Understanding and applying these best practices will help data scientists increase their productivity and accuracy, allowing them to make better decisions faster.



Photo by [Martin Sanchez](https://unsplash.com/@martinsanchez?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

In this article, we will use how to use the **.groupby()** function effectively to group the entries of a DataFrame according to the values of a specific feature. The**.groupby()** method is applied to a DataFrame and groups it according to a feature. Then, we can apply some simple or more complicated functions on that grouped object. This is a very important tool for every **data scientist** working on tabular or structured data as it will help you to manipulate data easily and in a more effective way.

**Table of Contents:**

1. **Why do We need Efficient Coding?**
2. **Common functions used with .groupby()**
3. **Missing value imputation using .groupby() & .transform()**
4. **Data filtration using the .groupby() & .filter()**

**You can find the codes and datasets used in this article in this GitHub repository:**

**[GitHub - youssefHosni/Efficient-Python-for-Data-Scientists](https://github.com/youssefHosni/Efficient-Python-for-Data-Scientists" \t "_blank)**

[Learn how to write efficient python code as a data scientist How To Write Python Clean Code [Article] Write Efficient…](https://github.com/youssefHosni/Efficient-Python-for-Data-Scientists" \t "_blank)

[github.com](https://github.com/youssefHosni/Efficient-Python-for-Data-Scientists" \t "_blank)

***Join the***[***Medium membership***](https://youssefraafat57.medium.com/membership)***program to continue learning without limits. I’ll receive a small portion of your membership fee if you use the following link, at no extra cost to you.***

**[Join Medium with my referral link - Youssef Hosni](https://youssefraafat57.medium.com/membership" \t "_blank)**

[Read every story from Youssef Hosni (and thousands of other writers on Medium). Your membership fee directly supports…](https://youssefraafat57.medium.com/membership" \t "_blank)

[youssefraafat57.medium.com](https://youssefraafat57.medium.com/membership" \t "_blank)

**1. Why do We need Efficient Coding?**

Efficient code is code that executes faster and with lower computational meomery. In this article, we will use the **time()**function to measure the computational time. This function measures the current time so we will assign it to a variable before the code execution and after and then calculate the difference to know the computational time of the code. A simple example is shown in the code below:

import time  
# record time before execution  
start\_time = time.time()  
# execute operation  
result = 5 + 2  
# record time after execution  
end\_time = time.time()  
print("Result calculated in {} sec".format(end\_time - start\_time))

Let’s see some examples of how applying efficient code methods will improve the code runtime and decrease the computational time complexity: We will calculate the square of each number from zero, up to a million. At first, we will use a list comprehension to execute this operation, and then repeat the same procedure using a for-loop.

First using list comprehension:

#using List comprehension   
  
list\_comp\_start\_time = time.time()  
result = [i\*i for i in range(0,1000000)]  
list\_comp\_end\_time = time.time()  
print("Time using the list\_comprehension: {} sec".format(list\_comp\_end\_time -  
list\_comp\_start\_time))

https://miro.medium.com/max/492/0*GPg-J9H2qz_TZbXP.png

Now we will use for loop to execute the same operation:

# Using For loop  
  
for\_loop\_start\_time= time.time()  
result=[]  
for i in range(0,1000000):  
 result.append(i\*i)  
for\_loop\_end\_time= time.time()  
print("Time using the for loop: {} sec".format(for\_loop\_end\_time - for\_loop\_start\_time))

https://miro.medium.com/max/476/0*vnv9AQ4zNfE4nnnP.png

We can see that there is a big difference between them, we can calculate the difference between them in percentage:

list\_comp\_time = list\_comp\_end\_time - list\_comp\_start\_time  
for\_loop\_time = for\_loop\_end\_time - for\_loop\_start\_time  
print("Difference in time: {} %".format((for\_loop\_time - list\_comp\_time)/  
list\_comp\_time\*100))

https://miro.medium.com/max/365/0*JksMrLUJc_ybNYJZ.png

Here is another example to show the effect of writing efficient code. We would like to calculate the sum of all consecutive numbers from 1 to 1 million. There are two ways the first is to use brute force in which we will add one by one to a million.

def sum\_brute\_force(N):  
 res = 0  
 for i in range(1,N+1):  
 res+=i  
 return res  
  
# Using brute force  
bf\_start\_time = time.time()  
bf\_result = sum\_brute\_force(1000000)  
bf\_end\_time = time.time()  
  
print("Time using brute force: {} sec".format(bf\_end\_time - bf\_start\_time))

Another more efficient method is to use a formula to calculate it. When we want to calculate the sum of all the integer numbers from 1 up to a number, let’s say N, we can multiply N by N+1, and then divide by 2, and this will give us the result we want. This problem was actually given to some students back in Germany in the 19th century, and a bright student called Carl-Friedrich Gauss devised this formula to solve the problem in seconds.

def sum\_formula(N):  
 return N\*(N+1)/2  
   
# Using the formula  
formula\_start\_time = time.time()  
formula\_result = sum\_formula(1000000)  
formula\_end\_time = time.time()  
  
print("Time using the formula: {} sec".format(formula\_end\_time - formula\_start\_time))

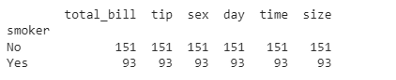
https://miro.medium.com/max/420/0*C_gpP0KZX8djtL8G.png

After running both methods, we achieve a massive improvement with a magnitude of over 160,000%, which clearly demonstrates why we need efficient and optimized code, even for simple tasks.

**2. Common functions used with .groupby()**

One of the simplest methods to apply to an aggregated group is the **.count().**In the example below we will apply this to the restaurant dataset. At first, we group the restaurant data according to whether the customer was a smoker or not. Then, we apply the **.count()** method. We obtain the count of smokers and non-smokers.

restaurant = pd.read\_csv('restaurant\_data.csv')  
  
restaurant\_grouped = restaurant.groupby('smoker')  
print(restaurant\_grouped.count())

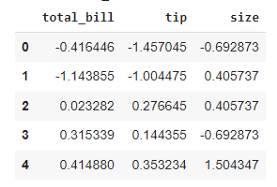


It is no surprise that we get the same results for all the features, as the **.count()** method counts the number of occurrences of each group in each feature. As there are no missing values in our data, the results should be the same in all columns.

After grouping the entries of the DataFrame according to the values of a specific feature, we can apply any kind of **transformation** we are interested in. Here, we are going to apply the z score, a normalization transformation, which is the distance between each value and the mean, divided by the standard deviation. This is a very useful transformation in statistics, often used with the z-test in standardized testing. To apply this transformation to the grouped object, we just need to call the .transform() method containing the lambda transformation we defined.

This time, we will group according to the type of **meal**: was it a dinner or a lunch? As the z-score transformation is a group-related transformation, the resulting table is just the original table. For each element, we subtract the mean and divide it by the standard deviation of the group it belongs to. We can also see that numerical transformation are applied only to numerical features of the DataFrame.

zscore = lambda x: (x - x.mean() ) / x.std()  
  
restaurant\_grouped = restaurant.groupby('time')  
  
restaurant\_transformed = restaurant\_grouped.transform(zscore)  
restaurant\_transformed.head()



While the **transform()** method simplifies things a lot, is it actually more efficient than using native Python code? As we did before, we first group our data, this time according to **sex**. Then we apply the z-score transformation we applied before, measuring its efficiency. We omit the code for measuring the time of each operation here, as you are already familiar with this. We can see that with the use of the transform() function, we achieve a massive speed improvement. On top of that, we’re only using one line to perform the operation of interest.

restaurant.groupby('sex').transform(zscore)  
  
mean\_female = restaurant.groupby('sex').mean()['total\_bill']['Female']  
mean\_male = restaurant.groupby('sex').mean()['total\_bill']['Male']  
std\_female = restaurant.groupby('sex').std()['total\_bill']['Female']  
std\_male = restaurant.groupby('sex').std()['total\_bill']['Male']  
  
for i in range(len(restaurant)):  
 if restaurant.iloc[i][2] == 'Female':  
 restaurant.iloc[i][0] = (restaurant.iloc[i][0] - mean\_female)/std\_female  
 else:  
 restaurant.iloc[i][0] = (restaurant.iloc[i][0] - mean\_male)/std\_male

**3. Missing value imputation using .groupby() & .transform()**

Now that we have seen why and how to use the **transform()** function on a grouped pandas object, we will address a very specific task that is imputing missing value.

Before we actually see how we can use the **transform()**function for missing value imputation, we will see how many missing values there are in our variable of interest in each of the groups. We can see below the number of data points in each of the “**time”**features and they are 176+68 = 244.

prior\_counts = restaurant.groupby('time')  
prior\_counts['total\_bill'].count()

https://miro.medium.com/max/656/0*jan48gduyxbkhF1i.png

Next, we will create a **restaurant\_nan** dataset, in which the total bill of 10% random observations was set to **NaN** using the code below:

import pandas as pd  
import numpy as np  
  
p = 0.1 #percentage missing data required  
  
mask = np.random.choice([np.nan,1], size=len(restaurant), p=[p,1-p])  
restaurant\_nan = restaurant.copy()  
restaurant\_nan['total\_bill'] = restaurant\_nan['total\_bill'] \* mask

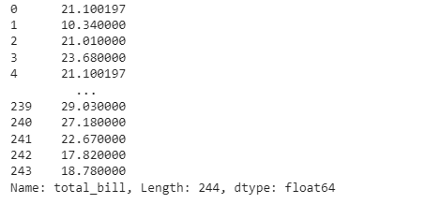
Now, let’s print the number of data points in each of the “**time”**feature and we can see that they are now 155+62 = 217. Since the total data points we have are 244 then the missing data points are 24 which is equal to 10%.

prior\_counts = restaurant.groupby('time')  
prior\_counts['total\_bill'].count()

https://miro.medium.com/max/503/0*8k0X7MDKPenFzDVh.png

After counting the number of missing values in our data, we will show how to fill these missing values with a group-specific function. The most common choices are the mean and the median, and the selection has to do with the skewness of the data. As we did before, we define a **lambda**transformation using the **fillna**() function to replace every missing value with its group average. As before, we group our data according to the time of the meal and then replace the missing values by applying the pre-defined transformation.

# Missing value imputation  
  
missing\_trans = lambda x: x.fillna(x.mean())  
restaurant\_nan\_grouped = restaurant\_nan.groupby('time')['total\_bill']  
restaurant\_nan\_grouped.transform(missing\_trans)



As we can see, the observations at index 0 and index 4 are exactly the same, which means that their missing value has been replaced by their group’s mean.

Also, we can see the computation time using this method is 0.007 seconds.

https://miro.medium.com/max/481/0*ibkz5UIlPDV1x0aF.png

Let’s compare this with the conventional method:

start\_time = time.time()  
mean\_din = restaurant\_nan.loc[restaurant\_nan.time =='Dinner']['total\_bill'].mean()  
mean\_lun = restaurant\_nan.loc[restaurant\_nan.time == 'Lunch']['total\_bill'].mean()  
  
for row in range(len(restaurant\_nan)):  
 if restaurant\_nan.iloc[row]['time'] == 'Dinner':  
 restaurant\_nan.loc[row, 'total\_time'] = mean\_din  
 else:  
 restaurant\_nan.loc[row, 'total\_time'] = mean\_lun  
print("Results from the above operation calculated in %s seconds" % (time.time() - start\_time))

https://miro.medium.com/max/656/0*oiIT5Arlfq4Ok8Np.png

We can see that using the **.transform()** function applied on a grouped object performs faster than the native Python code for this task.

**4. Data filtration using the .groupby() & .filter()**

Now we will discuss how we can use the .**filter()** function on a grouped pandas object. This allows us to include only a subset of those groups, based on some specific conditions.

Often, after grouping the entries of a DataFrame according to a specific feature, we are interested in including only a subset of those groups, based on some conditions. Some examples of filtration conditions are the number of missing values, the mean of a specific feature, or the number of occurrences of the group in the dataset.

We are interested in finding the mean amount of tips given, on the days when the mean amount paid to the waiter is more than 20 USD. The **.filter()** function accepts a **lambda** function that operates on a DataFrame of each of the groups. In this example, the lambda function selects “total\_bill” and checks that the **mean()** is greater than 20. If that lambda function returns True, then the **mean()** of the tip is calculated. If we compare the total mean of the tips, we can see that there is a difference between the two values, meaning that the filtering was performed correctly.

restaurant\_grouped = restaurant.groupby('day')  
filter\_trans = lambda x : x['total\_bill'].mean() > 20  
restaurant\_filtered = restaurant\_grouped.filter(filter\_trans)

print(restaurant\_filtered['tip'].mean())

https://miro.medium.com/max/433/0*VLAxtEELcSdvwxrv.png

print(restaurant['tip'].mean())

https://miro.medium.com/max/508/0*o9mkY2tVKKlNZE5T.png

If we attempt to perform this operation without using **groupby()**, we end up with this inefficient code. At first, we use a list comprehension to extract the entries of the DataFrame that refer to days that have a mean meal greater than $20 and then use a for loop to append them into a list and calculate the mean. It might seem very intuitive, but as we see, it’s also very inefficient.

t=[restaurant.loc[restaurant['day'] == i]['tip'] for i in restaurant['day'].unique()  
 if restaurant.loc[restaurant['day'] == i]['total\_bill'].mean()>20]  
restaurant\_filtered = t[0]  
  
for j in t[1:]:  
 restaurant\_filtered=restaurant\_filtered.append(j,ignore\_index=True)