How to Count Group Sizes with Pandas groupby() and size()

Authored by

stats writer

November 20, 2025

ARABPSYCHOLOGY.COM

RECOMMENDED CITATION

stats writer (2025). *How to Count Group Sizes with Pandas groupby() and size()*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=98381

One of the most fundamental tasks in data analysis involves summarizing data based on categorical criteria. Specifically, data scientists often need to determine the exact number of records, or rows, that belong to specific predefined groups within a dataset. In the context of the Pandas library, this common requirement is expertly handled by combining the powerful **groupby()** function with the concise **size()** method. Mastering this combination is essential for efficiently calculating the frequency distribution across one or multiple columns of your DataFrame.

The methodology is straightforward yet immensely powerful: we first instruct Pandas to partition the dataset--the groupby() operation--based on the unique values found in the column(s) of interest. This action creates a specialized GroupBy object, which internally manages these partitioned subsets. Subsequently, applying the size() method to this object immediately calculates the total count of rows within each resulting group. This technique reliably provides the count regardless of null values, making it the preferred function for determining group membership size.

Understanding the Pandas GroupBy Operation

The **groupby()** function in Pandas implements the crucial "split-apply-combine" paradigm, a concept adopted from R and fundamental to relational data manipulation. The **split** phase involves breaking the primary DataFrame into smaller segments based on the values in the specified key column(s). For instance, if you group by a 'City' column, Pandas internally creates temporary groups for London, Paris, Tokyo, and so forth.

The **apply** phase is where aggregation or transformation occurs. When calculating group size, this phase is executed by the **size()** method. Unlike many other aggregation methods (like **sum()** or **mean()**), which calculate a statistical measure on a specific data column, **size()** simply counts the total number of rows present within that group segment. The subsequent **combine** phase then merges these individual group results into a single output structure, typically a Pandas Series, where the index represents the unique grouping keys and the values represent the counts.

This process ensures that the resulting structure is clean, indexed by the grouping variable(s), and highly optimized for performance. It is important to note that when using groupby(), the choice of the aggregation function determines the output format. While **count()** counts non-null values in columns, size() counts every row, making it the unambiguous choice for determining group cardinality, or the total number of items in the group.

The Core Methods for Counting Group Occurrences

The core mechanism for counting rows within distinct groups relies on chaining the **groupby()** function with the **size()** method. Below are the primary syntax patterns used, ranging from simple single-variable grouping to complex, sorted multi-variable analysis. Understanding these variations

ARABPSYCHOLOGY.COM

allows for precise control over the output and enhances the utility of grouped counting in advanced data analysis workflows.

Method 1: Count Occurrences Grouped by One Variable

This method is the simplest implementation, ideal for understanding the distribution of a single categorical feature within the dataset.

df.groupby('var1').size()

Method 2: Count Occurrences Grouped by Multiple Variables

When data requires multi-dimensional categorization, this method groups the data based on the unique combinations of two or more variables, returning a MultiIndex Series.

df.groupby().size()

Method 3: Count Occurrences Grouped by Multiple Variables and Sort by Count

For prioritizing the most frequent or infrequent combinations, the output of the **size()** method is chained with the **sort_values()** function, typically used with `ascending=False` to show the largest groups first.

df.groupby().size().sort_values(ascending=False)

Setting up the Sample DataFrame

To illustrate these methods practically, we will utilize a sample DataFrame named `df`. This dataset simulates athlete performance data, featuring categorical columns like `team` and `position`, alongside a numerical column `points`. This structure provides ideal data for demonstrating how grouping and counting operations reveal underlying distributions within structured data.

The DataFrame is initialized using the standard Pandas constructor, defining ten rows of synthetic data that represent common scenarios encountered in real-world data science tasks. Pay close attention to the repetition of values in the `team` and `position` columns, as these repetitions are what the groupby() function will count.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,

ARABPSYCHOLOGY.COM

'position': ,
'points': })

#view DataFrame
print(df)

team position points
0 A G 15
1 A G 22
2 A F 24
3 A F 25
4 A F 20
5 B G 35
6 B G 34
7 B G 19
8 B G 14
9 B F 12

Example 1: Counting Occurrences Grouped by One Variable

In this initial example, we focus on determining the distribution of observations across a single categorical column: `team`. This operation reveals whether the data is balanced between Team A and Team B, or if one team is disproportionately represented in our dataset. This is a common first step in exploratory data analysis to check for potential data biases or structure.

We use the standard **groupby()** syntax, specifying only the 'team' column, and immediately follow it with the size() method. The result is a Pandas Series where the unique team names ('A' and 'B') become the index, and the corresponding values are the total counts of rows associated with those teams.

#count occurrences of each value in team column
df.groupby('team').size()

team
A 5
B 5
dtype: int64

From the output, we immediately ascertain that the dataset is perfectly balanced in terms of team membership. Both Team A and Team B occur precisely **5** times in the `team` column. This simple

ARABPSYCHOLOGY.COM

calculation provides crucial contextual information necessary before moving onto more complex aggregations or statistical modeling.

Example 2: Counting Occurrences Grouped by Multiple Variables

Often, simple single-variable counts are insufficient; we require counts based on the intersection of multiple features. For instance, we may need to know how many forwards (F) are on Team A versus how many guards (G) are on Team B. This requires grouping by both the `team` and `position` columns simultaneously.

When grouping by multiple columns, we pass a list of column names () to the groupby() function. The resulting output is a Pandas Series with a MultiIndex, where the unique combinations of 'team' and 'position' form the hierarchical index, and the corresponding values reflect the count of rows matching that specific combination.

#count occurrences of values for each combination of team and position
df.groupby().size()

team position
A F 3
G 2
B F 1
G 4
dtype: int64

Interpreting this MultiIndex Series is crucial for understanding the data distribution. The output clearly shows the specific counts for each unique combination:

Team A has **3** players designated as 'F' (Forward).
Team A has **2** players designated as 'G' (Guard).
Team B has only **1** player designated as 'F'.
Team B has **4** players designated as 'G', making it the most frequent combination in the dataset.

This result provides a much richer understanding of the data structure than the single-variable count alone, highlighting the compositional difference between the two teams.

Example 3: Sorting Group Counts for Enhanced Analysis

While the previous example provided the counts, the results were ordered alphabetically by the grouping variables (Team A then B, then Position F then G). In many analytical scenarios, the primary goal is to quickly identify the most or least frequent group combinations. This is achieved

ARABPSYCHOLOGY.COM

by chaining the **size()** output with the **sort_values()** method.

By default, **sort_values()** sorts in ascending order. To achieve the common goal of finding the largest groups first (i.e., the highest frequency combinations), we explicitly pass the argument `ascending=False`. This immediately reorders the output series based on the calculated counts, regardless of the alphabetical order of the grouping keys.

#count occurrences for each combination of team and position and sort
df.groupby().size().sort_values(ascending=False)

team position
B G 4
A F 3
G 2
B F 1
dtype: int64

The output shows the count of each combination of **team** and **position** values, now sorted by count in descending order. This visualization makes it instantly clear that 'Team B Guards' (count of 4) is the largest subgroup, followed by 'Team A Forwards' (count of 3). Conversely, 'Team B Forwards' (count of 1) is the least represented subgroup.

Note: To sort by count in ascending order, simply omit the `ascending=False` parameter within the **sort_values()** function, or set it explicitly to `True`. This technique is invaluable when seeking outliers or sparsely populated groups within your dataset.

Why Use size() Instead of count()?

A frequent point of confusion for new Pandas users is the distinction between the **size()** and **count()** methods when applied to a GroupBy object. While both return counts, they serve fundamentally different purposes, and only **size()** is reliably used to determine the total number of rows in each group.

The **count()** method is an aggregation function that counts the number of **non-null** values for **each column** within the group. If your DataFrame has five columns, applying **count()** after a groupby() operation will return a DataFrame with five columns of counts. If a column contains missing values (NaNs), that column's count for that group will be less than the actual number of rows.

In contrast, the size() method is not an aggregation function tied to specific columns; it is a property of the GroupBy object itself. It calculates the total number of rows (including those with NaNs in any column) for each group and returns a single Series. Therefore, **size()** is the mathematically

ARABPSYCHOLOGY.COM

correct and robust method for calculating group cardinality, regardless of data completeness in other columns. For determining "how many items are in this group," **size()** is the definitive solution.

ARABPSYCHOLOGY.COM