How to Count Observations by Group in Pandas

Authored by

stats writer

December 17, 2025

Analyzing and summarizing data often requires calculating the frequency of entries based on specific categories. In the Pandas library, a crucial tool for data analysis in Python, counting the number of observations within distinct groups is achieved efficiently using a combination of methods.

This process typically involves two fundamental steps: first, segmenting the data using the groupby() method to create groups based on one or more columns, and second, applying the size() method to count the items in those newly formed groups. Mastering this technique provides immediate insights and critical summary information about your dataset's distribution.

As mentioned, counting the frequency of records segregated by category is a common requirement when working with a Pandas DataFrame. This powerful combination of methods allows for rapid aggregation without requiring complex loops or conditional logic.

The basic pattern for counting group occurrences utilizes the **groupby()** function followed immediately by the **size()** function. Below is the fundamental syntax for this operation:

df.groupby('column_name').size()

To illustrate these techniques practically, we will utilize a sample dataset representing sports team statistics. This example DataFrame contains columns for 'team', 'division', and 'rebounds'.

import numpy as np
import pandas as pd

#create pandas DataFrame
df = pd.DataFrame({'team': ,
'division':,
'rebounds': })

#display DataFrame
print(df)

team division rebounds
0 A E 11
1 A W 8
2 B E 7
3 B E 6
4 B W 6
5 C W 5
6 C E 12

The code block above initializes our sample data, which we will use throughout the subsequent examples. Understanding the structure of this initial dataset is essential before proceeding with group aggregation techniques.

Example 1: Counting Frequencies Using a Single Variable

The simplest application of group counting involves aggregating records based on a single categorical column. In this example, we aim to determine the total number of entries associated with each unique 'team' present in our DataFrame.

The following Python code executes this aggregation using **groupby('team')** followed by **size()**. The result is a frequency count indexed by the grouped column.

```
#count total observations by variable 'team'
df.groupby('team').size()
```

```
team
A 2
B 3
C 2
dtype: int64
```

The output clearly shows the distribution of records across the teams. We can summarize the results as follows:

Team A appears 2 times in the dataset.
Team B appears 3 times in the dataset.
Team C appears 2 times in the dataset.

It is important to recognize that the output generated by the **size()** method is a Series object, where the grouped key (team) becomes the index. For better usability and integration into further analysis steps, transforming this output into a standard DataFrame is often necessary. We achieve this transformation using the **reset_index()** function, which converts the index back into a column and allows us to assign a descriptive name ('obs') to the count column.

```
df.groupby('team').size().reset_index(name='obs')
```

```
team obs
0 A 2
1 B 3
2 C 2
```

Example 2: Sorting Group Counts After Aggregation

Once the group counts have been calculated and converted back into a DataFrame using **reset_index()**, it is often necessary to arrange the results based on the frequency count itself. This allows for quick identification of the most or least frequent groups.

To achieve this sorting capability, we chain the **sort_values()** function onto our aggregation pipeline. This function takes arguments that specify which column to sort by (in our case, the 'obs' count column) and whether the sorting should be ascending or descending.

By specifying **ascending=True**, as demonstrated below, we arrange the groups from the smallest count to the largest count. Conversely, setting **ascending=False** would display the most frequent groups first.

df.groupby('team').size().reset_index(name='obs').sort_values(, ascending=True)

team obs
0 A 2
2 C 2
1 B 3

Example 3: Grouping Observations by Multiple Criteria

A more complex, yet common, scenario involves counting observations based on the unique combinations across two or more columns. To perform multi-level grouping, we simply pass a list of column names into the **groupby()** function.

For instance, if we want to know how many records belong to each unique pairing of 'team' and 'division', the list is provided. The **size()** method then counts the records within each resulting subgroup.

The following code groups the data first by 'team', and then further segments those teams by 'division', providing a granular count of entries for every possible combination:

#count observations grouped by team and division
df.groupby().size().reset_index(name='obs')

team division obs
0 A E 1
1 A W 1
2 B E 2

3 B W 1
4 C E 1
5 C W 1

Analyzing the resulting DataFrame allows us to draw precise conclusions about the data distribution based on the combined criteria:

There is 1 record corresponding to Team A in Division E.
There is 1 record corresponding to Team A in Division W.
There are 2 records corresponding to Team B in Division E, indicating this subgroup has the highest frequency in the dataset.
There is 1 record corresponding to Team B in Division W.
There is 1 record corresponding to Team C in Division E.
There is 1 record corresponding to Team C in Division W.

Conclusion and Further Reading

The combination of the **groupby()** and **size()** methods provides a powerful and idiomatic way within Pandas to calculate frequency counts across one or multiple columns. This is a fundamental operation in exploratory data analysis (EDA), offering immediate statistical summaries of categorical variables.

Remember that while **size()** is effective for simple counts, for calculating specific aggregate statistics (like sum or mean) on other columns within the groups, you would typically use methods like `.sum()` or `.mean()` instead of `.size()`.

Always consider using **reset_index()** after aggregation to convert the resulting Series back into a usable DataFrame format for subsequent data manipulation or visualization tasks.

How to Calculate the Sum of Columns in Pandas
How to Calculate the Mean of Columns in Pandas
How to Find the Max Value of Columns in Pandas