Grouping and Aggregating DataFrames by Multiple Columns Using Pandas

Authored by

Mohammed looti

November 7, 2025

In modern data analysis and complex manipulation tasks using the Python ecosystem, it is an extremely common requirement to summarize and segment large datasets. Data analysts frequently encounter scenarios where they must perform sophisticated data aggregation based not just on one, but on the intersecting values of two or more distinct columns. This requirement moves beyond simple filtering and demands a powerful toolset for calculating summary statistics across specific categories.

The highly versatile Pandas library is specifically designed to handle these challenges with elegance and efficiency. Pandas leverages the combined power of its core methods, notably .groupby() and .agg(), to manage multi-column grouping. This methodology is central to effective data processing, enabling users to implement the fundamental "split-apply-combine" strategy necessary for generating granular business intelligence and actionable insights from raw data.

This comprehensive tutorial serves as an expert guide, walking you through practical, step-by-step examples. We will demonstrate exactly how to leverage these functions to group and aggregate data using multiple columns simultaneously, ensuring your analytical outputs are both precise and easy to interpret. Mastering this technique is essential for anyone working with real-world, complex datasets in Pandas.

Core Concepts: The Split-Apply-Combine Strategy

To effectively utilize Pandas for complex data manipulation, it is crucial to first internalize the underlying mechanism that orchestrates grouping and aggregation. This mechanism is formally known as the **split-apply-combine** strategy, a paradigm introduced by Hadley Wickham. This three-stage process ensures that complex operations are performed efficiently and logically across defined subsets of your data, preventing errors and optimizing computational performance when dealing with large volumes of information.

The process initiates with the **splitting** phase, which is handled primarily by the .groupby() method. When multiple column names are passed to this method (e.g., a list of strings), Pandas meticulously partitions the original DataFrame into distinct groups. Each resulting group corresponds to a unique combination of values found across the specified grouping keys. For instance, if you group by 'Country' and 'Year', every unique Country-Year pair forms its own isolated data segment ready for the next phase.

Next is the **apply** phase, where the actual calculation takes place. The .agg() function executes various statistical functions--such as calculating the mean, sum, count, or variance--on the selected data columns within each isolated group. Finally, the **combine** phase brings everything together. The results from all these group-wise operations are meticulously structured and merged back into a single, comprehensive output, which is typically a new Pandas DataFrame. When grouping by multiple columns, this output often features a MultiIndex structure, utilizing hierarchical

indexing to clearly represent the grouped keys, though this can be flattened later.

This inherent structure is paramount for generating granular summary reports. When defining multiple grouping columns, the resulting aggregated output inherently captures all unique intersections. For a sales dataset, grouping by 'Store Location' and 'Marketing Channel' ensures you receive separate aggregated metrics (e.g., total revenue) for every unique combination, such as 'Downtown Store' via 'Social Media' versus 'Suburban Store' via 'Email Campaign'.

Example 1: Grouping by Two Keys and Calculating the Average

To illustrate this functionality, let us construct a practical scenario involving player statistics. We will create a sample DataFrame that tracks several metrics, including the player's team, their position, and their recorded assists and rebounds. Our primary objective in this first example is to calculate the average number of assists, specifically broken down and grouped by the combination of both the player's **'team'** and their specific **'position'**.

We start by initializing the required data structure within a Python script. This step involves importing the necessary Pandas library and defining the raw data, which we then encapsulate into a DataFrame object. The code below sets up our initial dataset, allowing us to visualize the raw input before any aggregation takes place.

import pandas as pd

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

#view DataFrame
print(df)

team position assists rebounds
0 A G 5 11
1 B G 7 8
2 B F 7 10
3 B G 8 6
4 B F 5 6
5 M F 7 9
6 M C 6 6
7 M C 9 10

To execute the required calculation--the mean assists based on the intersection of team and position--we pass the list of grouping columns, , directly into the .groupby() method. Following the split, we apply the aggregation using .agg(), specifying that we are interested only in the 'assists' column aggregated using the `'mean'` function. Finally, the crucial `.reset_index()` method is chained to the operation, which takes the resulting hierarchical MultiIndex (created by the grouping) and converts the grouping keys back into standard columns, significantly improving the output's visual clarity and usability.

df.groupby().agg({'assists': }).reset_index()

team position assists
mean
0 A G 5.0
1 B F 6.0
2 B G 7.5
3 M C 7.5
4 M F 7.0

The resulting aggregated table provides precise statistical insights segmented by group. Analyzing the output allows for immediate, granular deductions about performance across different organizational categories. For example, we observe that the mean assists for players in the 'F' position on **Team B** is exactly **6.0** (derived from the raw values 7 and 5), while the mean assists for 'C' position players on **Team M** is **7.5** (derived from 6 and 9). This level of detail is invaluable for performance comparison and resource allocation.

Refining the Output: Renaming Columns for Clarity

While the previous output is mathematically and functionally correct, a common challenge arises from the hierarchical column indexing structure inherent when using the .agg() function with multiple statistics or specific column selections. The resulting column header for the calculated metric often presents itself as a multi-level index, which can complicate subsequent programming operations, data visualization, and reporting integration.

To ensure maximum readability and streamline any downstream processing--such as merging this summary data with other sources or exporting it to a database--it is considered best practice to rename and flatten these columns immediately after the aggregation step. This transformation simplifies the DataFrame structure, replacing the confusing MultiIndex with simple, descriptive column headers.

The following code snippet illustrates the process of performing the grouping, storing the resultant

aggregated data in a new variable (`new`), and then assigning a clean, flat list of new column names. This list must correspond exactly to the order of the columns in the aggregated DataFrame: the grouping keys followed by the calculated metric(s). This approach ensures the output is production-ready and easily accessible via standard column names like `mean_assists`.

#group by team and position and find mean assists
new = df.groupby().agg({'assists': }).reset_index()

#rename columns
new.columns =

#view DataFrame
print(new)

team pos mean_assists
0 A G 5.0
1 B F 6.0
2 B G 7.5
3 M C 7.5
4 M F 7.0

By successfully renaming the columns, we convert the initially complex structure into a clean, flat table where the calculated metric is explicitly labeled as `mean_assists`. The grouping columns, `team` and `position` (renamed `pos`), now function as clear primary keys for this summary table, dramatically simplifying readability. This technique is vital for anyone preparing data for subsequent advanced statistical modeling or integration into automated reporting pipelines.

Example 2: Grouping by Two Keys and Finding Multiple Statistics

One of the most powerful features of the .agg() function is its inherent capability to compute not just a single summary statistic, but an array of statistics simultaneously for a specific column within each defined group. This functionality allows analysts to gain a holistic view of the data distribution, measuring both central tendency and spread in a single, efficient operation.

Continuing our analysis of the player dataset, let us assume we require a more detailed perspective on the players' rebounding performance. To achieve this, we need to calculate two distinct metrics: a measure of central tendency, such as the median, and a measure of peak performance, represented by the maximum value (max). We will reuse the exact same initial Pandas DataFrame structure established in Example 1.

import pandas as pd

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

To calculate both the median and the maximum number of rebounds, we must modify the input structure provided to the .agg() dictionary. Instead of passing a single statistic string, we pass a list of strings: . This simple change instructs Pandas to iterate through the 'rebounds' column and calculate both statistics for every subgroup defined by the unique combinations of 'team' and 'position'. The use of `.reset_index()` once again ensures the output is presented clearly.

df.groupby().agg({'rebounds': }).reset_index()

team position rebounds
median max
0 A G 11 11
1 B F 8 10
2 B G 7 8
3 M C 8 10
4 M F 9 9

The final resulting table now neatly incorporates two calculated metrics for the 'rebounds' column for each unique team-position pairing. This dual aggregation offers a far more robust summary than a single statistic alone. For instance, we can observe that for **Team B, Position F**, the typical (median) rebound count is **8**, but their peak (max) performance reached **10**. Similarly, **Team M, Position C** also shows a median of **8** and a maximum of **10**. This comprehensive approach provides analysts with valuable insight into both the typical distribution and the performance ceiling for each distinct subgroup.

Additional Resources

To further advance your expertise and mastery of Pandas DataFrame manipulation and analysis techniques, we recommend exploring the following related tutorials and documentation:

How to Filter a Pandas DataFrame on Multiple Conditions
How to Count Missing Values in a Pandas DataFrame
How to Stack Multiple Pandas DataFrames