How to Easily Group by Two Columns and Aggregate Data in Pandas

Authored by

stats writer

November 20, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Group by Two Columns and Aggregate Data in Pandas*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=98386

The ability to perform complex aggregation across multiple categories is a cornerstone of effective data analysis. In the realm of Python data science, the Pandas library offers robust tools for this purpose, specifically leveraging the power of its `groupby()` method. When working with tabular data, analysts frequently need to segment their information based on two or more criteria simultaneously before calculating summary statistics. This technique is essential for deriving meaningful insights, such as calculating the average performance across different teams and positions in a sports dataset, or determining maximum sales volume based on region and product type. Understanding how to group a DataFrame by two columns and efficiently apply aggregation functions is a core skill for any user working with Pandas.

This comprehensive guide will detail the methodology for achieving multi-column grouping and aggregation using standard Pandas syntax. We will explore the primary functions required, demonstrate practical application through concrete examples, and discuss best practices for structuring the resulting data. Mastering this technique allows for powerful data transformation, leading to clearer reporting and more precise analytical conclusions. We will focus on combining the `groupby()` method with various statistical functions to produce insightful summaries from raw datasets.

Understanding the Core Methods: `groupby()` and `agg()`

The process of grouping and aggregating data in Pandas fundamentally relies on a split-apply-combine strategy. The `groupby()` method initiates the "split" phase, dividing the rows of the DataFrame into groups based on the unique combinations found in the specified key columns. When two columns are passed to this method, Pandas creates distinct groups for every unique pairing of values present across those two dimensions. This segmentation is critical because subsequent operations are applied independently to each generated group.

Following the grouping, the "apply" phase is executed, typically utilizing the `agg()` method or a direct statistical function call (like `.mean()` or `.max()`). The `agg()` method is particularly versatile, allowing users to apply one or more aggregation functions to one or more columns within the resulting groups. To handle aggregation across multiple columns or apply different functions to different columns, a dictionary structure is passed to the `agg()` method, mapping column names to desired functions (e.g., {'points': 'mean', 'assists': 'sum'}).

Although it is often sufficient to use a simplified syntax like `.mean()` immediately after `groupby()` when calculating a single aggregate for a single target column, understanding the explicit role of `.agg()` is crucial for more complex or multi-output aggregation tasks. The final step, "combine," merges the results of the applied functions into a new series or DataFrame, often resulting in a hierarchical (MultiIndex) row structure that summarizes the source data according to the group keys.

Essential Syntax for Double Grouping

To perform grouping based on two distinct features, you must supply a list containing both column names to the `groupby()` method. This tells Pandas exactly which dimensions should define the boundaries of the groups. Following the grouping definition, you select the specific column you wish to aggregate and apply the desired aggregation function. The standard structure remains elegant and highly readable, making complex data summaries straightforward to implement.

You can use the following basic syntax with the **groupby()** function in Pandas to group by two columns and aggregate another column:

df.groupby().mean()

This specific structure is highly efficient. It instructs Pandas to first segment the DataFrame `df` based on the unique combination of values found in `var1` and `var2`. Once the groups are established, the operation targets the `var3` column exclusively, calculating the arithmetic **mean** of all `var3` values within each defined group. The result is a concise summary statistic detailing the central tendency of `var3` relative to the groupings of `var1` and `var2`.

It is important to remember that the aggregate function chosen (in this case, `.mean()`) determines the type of summary produced. Depending on the analytical goal, one might choose `.sum()`, `.max()`, `.count()`, or the highly flexible `agg()` method to calculate multiple metrics simultaneously. The next sections will provide practical, concrete demonstrations of this powerful syntax using a real-world example dataset.

Setting up the Example DataFrame

To illustrate the practical application of double-column grouping and aggregation, we will construct a simple but representative Pandas DataFrame. This dataset simulates performance scores in a sports context, allowing us to group players by both their assigned team and their specific playing position. Analyzing the scores across these two dimensions provides far richer insight than analyzing them based on team or position alone.

We begin by importing the necessary Pandas library and then defining the data structure using lists, which are subsequently converted into a DataFrame. The columns defined are `team` (categorical), `position` (categorical), and `points` (numerical), which will serve as our aggregation target.

The following examples show how to group by two columns and aggregate using the following Pandas DataFrame:

import pandas as pd

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

As displayed by the output of the `print()` function, our DataFrame contains ten records. Note the variability in the number of records associated with certain combinations. For example, Team A has three players listed as position 'F', while Team B has four players listed as position 'G'. This inherent imbalance in group sizes highlights the necessity of using aggregation functions to standardize comparisons across these combined categories, ensuring that conclusions are drawn from summary statistics rather than raw observation counts.

This dataset will serve as the foundation for our subsequent examples, demonstrating how minor adjustments to the aggregation function can yield fundamentally different, yet equally valuable, analytical insights when paired with the same dual-column grouping strategy.

Example 1: Calculating Grouped Means

One of the most common applications of grouping is calculating the arithmetic mean, or average, of a numerical column. When grouping by two columns--`team` and `position` in our example--we generate the mean points scored for every unique combination of team and position. This allows us to compare the average performance of a specific position across different teams, or the average performance of different positions within the same team.

To achieve this, we chain the `.mean()` method directly after defining the grouping criteria and selecting the target column, `points`. This streamlined syntax efficiently executes the split-apply-combine process inherent to Pandas aggregation.

We can use the following syntax to calculate the mean value of the **points** column, grouped by the **team** and **position** columns:

#calculate mean of points grouped by team and position columns
df.groupby().mean()

team position
A F 23.0
G 18.5
B F 12.0
G 25.5
Name: points, dtype: float64

The resulting output is a Pandas Series indexed by a MultiIndex (Team and Position), showing the calculated mean for each group. Analyzing this output reveals several key performance insights:

The mean points value for players on team A in position F is **23.0**.
The mean points value for players on team A in position G is **18.5**.
The mean points value for players on team B in position F is significantly lower at **12.0**.
Team B's players in position G exhibit the highest average performance, achieving **25.5** mean points.

This simple aggregation step provides a normalized perspective on player performance, immediately highlighting high-performing and underperforming group segments based on the combination of team and position. The use of floating-point numbers in the result, denoted by `dtype: float64`, is standard for mean calculations in Pandas, reflecting the potential for fractional averages.

Example 2: Determining Maximum Values per Group

While the mean provides insight into typical performance, finding the maximum value within each group is crucial for identifying outliers or record-setting performances. In the context of our sports data, calculating the maximum points scored grouped by `team` and `position` reveals the highest single score achieved by any player within that specific team-position segment.

We substitute the `.mean()` method from the previous example with the `.max()` method. This

change in the aggregation function is the only modification required to shift the analytical focus from central tendency to extremity. The grouping mechanism remains identical, ensuring the maximum is calculated correctly for each unique combination of the two specified columns.

We can use the following syntax to calculate the max value of the **points** column, grouped by the **team** and **position** columns:

#calculate max of points grouped by team and position columns
df.groupby().max()

team position
A F 25
G 22
B F 12
G 35
Name: points, dtype: int64

Examining the results provides a clear view of the top individual performances within each segment:

The max points value for players on team A in position F is **25**.
The max points value for players on team A in position G is **22**.
The highest individual score overall belongs to Team B, position G, reaching **35** points.

This analysis, unlike the mean calculation, is resistant to the sample size of the group; it focuses purely on the single greatest observation. This is invaluable when screening data for top performers or identifying maximum capacities. Notice that the resulting data type is `int64`, as the aggregation function (max) returns a value identical in type to the input column (points), which contained only integers.

Example 3: Counting Occurrences and Group Sizes

Beyond summarizing numerical data (like mean or max), aggregation is frequently used to determine the size or frequency of the groups themselves. Counting the occurrences provides a powerful measure of the dataset's distribution, indicating how many records belong to each unique combination defined by the grouping columns. This is essential for understanding representation and potential biases in the data structure.

To count the number of rows associated with each combination of `team` and `position`, we utilize the `.size()` method immediately following the `groupby()` call. Unlike `.mean()` or `.max()`, the

`.size()` method does not require selecting a specific target column (e.g.,) because it operates directly on the size of the generated groups.

We can use the following syntax to count the occurrences of each combination of the **team** and **position** columns:

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

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

The resulting Series shows the count of observations for each segment:

There are **3** players on team A in position F.
There are **2** players on team A in position G.
Team B in position F has the lowest count, with only **1** player represented.
Team B in position G has the highest representation, with **4** players.

These counts are fundamental for contextualizing the numerical aggregations performed in the previous examples. For instance, the high mean score for Team B, position F (12.0) is based on only a single observation, making it highly susceptible to change if more data were introduced. Conversely, the mean for Team B, position G (25.5), is derived from four observations, lending it greater statistical stability. Always pair numerical summaries with group sizes for robust data interpretation.

Advanced Aggregation with Multiple Functions

While chaining methods like `.mean()` is convenient for single aggregate calculations, real-world data analysis often requires calculating several statistics simultaneously (e.g., mean, max, and count) for the same group keys. The `agg()` method provides the flexibility needed to perform these multi-functional aggregation tasks efficiently within a single operation.

To use `agg()` effectively, we pass a list of desired function names (as strings) to the method immediately after defining the grouping and selecting the target column. This results in a DataFrame output where the columns are the names of the aggregate functions applied, and the index remains the MultiIndex defined by the grouping columns (`team` and `position`).

Consider the requirement to find the mean, minimum, and maximum points for every team-position combination. The syntax would look like this:

#Calculate mean, min, and max simultaneously
df.groupby().agg()

mean min max
team position
A F 23.0 20 25
G 18.5 15 22
B F 12.0 12 12
G 25.5 14 35

This approach transforms the summary output from a simple Series (as seen in Examples 1 and 2) into a structured DataFrame, simplifying comparative analysis. The ability to calculate numerous metrics in one call significantly cleans up analytical code and improves execution speed for complex reporting tasks. Furthermore, the `agg()` method can be extended to handle custom user-defined functions if standard Pandas operations are insufficient for the task.

Flattening the Output using `reset_index()`

A typical consequence of using the `groupby()` method is the creation of a MultiIndex (hierarchical index) in the resulting Series or DataFrame. While a MultiIndex is computationally powerful, it can often be inconvenient for subsequent operations, such as merging the results back into another DataFrame, saving the data to a CSV file, or generating final reports where the grouping variables must appear as regular columns.

To convert the index levels back into standard columns, the `reset_index()` method is applied as the final step in the aggregation chain. This method effectively "flattens" the output, turning the previously used grouping keys (`team` and `position`) from index levels into traditional columns alongside the aggregated results.

Let's take the multi-functional aggregation from the previous section and apply `reset_index()`:

#Flattening the output DataFrame
agg_df = df.groupby().agg().reset_index()

print(agg_df)

team position mean min max
0 A F 23.0 20 25

1 A G 18.5 15 22
2 B F 12.0 12 12
3 B G 25.5 14 35

The resulting DataFrame, `agg_df`, now has a default integer index (0, 1, 2, 3), and the columns `team`, `position`, `mean`, `min`, and `max` are all readily accessible for further manipulation or visualization. Using `reset_index()` is generally considered best practice when the final output is intended for consumption by non-Pandas tools or when the user prefers a flat, SQL-like structure over a hierarchical index.

Conclusion and Best Practices

Grouping and aggregation across two columns using the `groupby()` method is a fundamental and powerful operation in Pandas. By supplying a list of two column names--such as --you instruct the library to perform detailed, segmented analysis, yielding metrics that are far more insightful than single-column aggregations.

To ensure optimal performance and clarity when performing these multi-dimensional summaries, consider the following best practices:

Verify Data Types: Ensure that the grouping columns (`team`, `position`) are appropriate categorical or object types, and the aggregated column (`points`) is numeric (integer or float).
Use `.agg()` for Multi-Metric Reporting: Whenever two or more statistics (e.g., mean and standard deviation) are required, utilize the `agg()` method for cleaner, more efficient code rather than performing multiple separate aggregations.
Flatten the Output: Unless specifically required for advanced hierarchical indexing techniques, always conclude the aggregation chain with `reset_index()` to generate a standard, easy-to-read DataFrame ready for presentation or export.

Mastering this combination of grouping and summarizing techniques enables data analysts to quickly move from raw data observation to sophisticated, evidence-based conclusions, making the Pandas library an indispensable tool for data manipulation and statistical reporting.