Learning Pandas: Mastering Grouping and Aggregation by Multiple Columns

Authored by

Mohammed looti

November 15, 2025

Introduction to Advanced Grouping and Aggregation in Pandas

In the thriving domain of data analysis and manipulation, the pandas library stands out as the indispensable toolkit for handling structured data within the Python ecosystem. While fundamental data operations are straightforward, unlocking truly valuable insights often demands sophisticated techniques, particularly when navigating complex datasets characterized by multiple categorical variables. The cornerstone of these advanced techniques in pandas is the powerful `groupby()` function, which enables complex aggregation and transformation operations. This functionality is absolutely essential for generating summary statistics, revealing latent trends across distinct subsets of data, and preparing raw information for subsequent stages in the data science pipeline, such as machine learning or detailed reporting.

Modern real-world datasets are inherently multi-dimensional; data points are rarely defined by a single factor. Instead, they are commonly categorized by two, three, or even more related attributes. Consider scenarios such as a global retailer tracking sales volume segmented by both country and product category, or a healthcare institution analyzing patient outcomes based on both treatment protocol and age demographic. To properly interpret the complex interaction effects of these variables, it becomes mandatory to group the data based on the unique combinations formed by these categorical columns. Utilizing multi-column grouping allows analysts to transcend simple grand totals and monolithic averages, enabling them to acquire deep, granular, and ultimately actionable insights into their data structure.

This comprehensive guide is dedicated to achieving mastery over multi-column grouping using the `groupby()` method in pandas. We will specifically demonstrate the practical steps required to segment a DataFrame using two distinct categorical columns simultaneously, and then apply diverse aggregation functions to a designated numerical target column. Through practical, hands-on examples--including calculating the conditional mean, identifying the maximum value, and counting group occurrences--we will illustrate how to effectively transform raw, messy data into structured, actionable intelligence, all built upon a clear and representative sample DataFrame designed for clarity.

Understanding the Split-Apply-Combine Strategy

The conceptual underpinning of the **`groupby()`** method is the highly effective "split-apply-combine" paradigm, now a cornerstone of efficient data processing in Python. This methodological strategy ensures that complex computations are executed systematically across relevant data subsets before the final results are intelligently merged back into a cohesive structure. When a data scientist invokes the **`groupby()`** method in pandas, the system executes the following three critical operations in sequence:

Split: The initial DataFrame is logically divided into multiple independent groups. This partitioning is governed by the unique combinations of values present in the specified grouping columns (in our advanced scenario, two columns). Each unique combination forms its own subset.
Apply: A chosen aggregation function--such as **`mean()`**, **`max()`**, or **`sum()`**--is then independently applied to the relevant data within each of the previously formed groups. This step calculates the required summary statistic tailored specifically for every unique combination.
Combine: Finally, the distinct results generated during the "Apply" phase from all individual groups are efficiently assembled and merged back together. The final output structure is typically a Series or a DataFrame, usually organized using a multi-index that clearly labels and identifies the source groups.

When dealing with grouping by multiple factors, the syntax remains remarkably intuitive and easy to read. We define our grouping criteria by passing an explicit Python list of column names directly to the **`groupby()`** method. After defining the groups, we use bracket notation to select the specific numerical column designated for aggregation, and then chain the appropriate function call (e.g., `.mean()`). This streamlined approach ensures that analysts can concentrate on the core analytical question rather than grappling with complex programming implementation details.

The standardized structure for grouping by two variables and subsequently applying an aggregation function to a third numerical variable is clearly demonstrated in the schematic below. Here, `var1` and `var2` represent the categorical grouping keys, while `var3` is the target variable for the mathematical calculation:

df.groupby().mean()

In this specific example, the DataFrame referenced as `df` is first partitioned based on every unique combination of values found in `var1` and `var2` columns. Once these distinct groups have been successfully established, the **`mean()`** aggregation function is applied exclusively to the values contained within the `var3` column for each group. The final result is presented as a Series where the index structure mirrors the unique combinations of `var1` and `var2`, and the corresponding data points represent the computed means of `var3` for those specific sub-populations.

Preparing the Data: Creating the Sample DataFrame

To effectively demonstrate the robust capability and inherent simplicity of grouping data by two distinct columns, we begin by constructing a specialized sample DataFrame. This synthetic dataset is deliberately structured to mimic a common analytical scenario often encountered in sports performance tracking, where we monitor player statistics based on their team affiliation and their specific playing position. This design is optimal for our purposes because it clearly delineates the data into two primary categorical dimensions (`team` and `position`) alongside a single numerical

metric (`points`) that is ideal for subsequent aggregation.

The foundational step for any pandas operation is importing the necessary library, typically aliased as `pd`. Following the import, we define our dataset using a standard Python dictionary format, which pandas then seamlessly converts into our working DataFrame. The two categorical columns, `team` (values 'A' or 'B') and `position` (values 'G' for Guard or 'F' for Forward), will serve as the composite keys for our grouping operations, while the `points` column will be the central numerical variable we seek to analyze and summarize.

The code snippet below initializes our sample data structure and immediately outputs a view of the resulting DataFrame. This allows us to inspect the raw data distribution and confirm the structure before we proceed with applying any sophisticated analytical transformations or the **`groupby()`** method:

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

Our resulting DataFrame `df` comprises ten distinct player records. This structure is perfectly configured to demonstrate how simultaneous grouping by `team` and `position` can precisely isolate and analyze performance metrics. We are now ready to analyze how the combined effect of team membership and a specific playing role impacts the scores recorded in the 'points' column.

Example 1: Calculating the Conditional Mean Score

Calculating the mean, or average, is typically the initial and most fundamental step in statistical analysis, serving as a critical measure of central tendency within any defined population. When we implement grouping across two distinct categories--in this case, `team` and `position`--we elevate the analysis beyond simple overall averages to calculate granular, conditional means. This powerful technique allows us to answer highly specific business or analytical questions, such as, "What is the average scoring performance achieved by Forwards on Team A?" This conditional level of detail is absolutely vital for accurate performance assessment and precise, localized benchmarking.

To execute this operation, we invoke the `groupby()` method, supplying the list of column names to establish our composite groups. Next, we select the `'points'` column using bracket notation as the designated target for our calculation. Finally, we chain the `mean()` function to trigger the aggregation process. This clean, fluid sequence ensures that the calculation is meticulously isolated to the desired subsets of the data.

The following syntax efficiently calculates the average points for every unique pairing of team and position:

#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, elegantly indexed by a hierarchical multi-index where the unique combinations of 'team' and 'position' define the index levels. Analyzing these results yields immediate and actionable performance data: We can clearly observe that players on **Team B** occupying position **G** (Guard) demonstrate the highest average score at **25.5** points, signifying a strong core competency in that specific role. Conversely, **Team B**'s players in position **F** (Forward) record the lowest average at **12.0** points. This direct comparison effectively illuminates performance disparities across different roles and teams, offering valuable analytical depth that would be completely masked by a simple overall average calculation.

Example 2: Identifying Peak Performance with the Maximum Value

While measures of central tendency, such as the mean, provide valuable context, the identification of extreme values is equally critical for fully understanding potential and peak performance capabilities within a dataset. Calculating the maximum recorded value within each group is essential for key analytical tasks, including pinpointing potential outliers, establishing best-case scenarios, or recognizing high-achieving individuals. In the specific context of our athlete dataset, determining the maximum points scored, meticulously grouped by both `team` and `position`, allows us to immediately identify the single highest individual performance achieved within every unique team-role combination.

To execute this specialized analysis, the procedural steps remain fully consistent with the previous example, requiring only a change in the final aggregation function. We initiate the process by grouping the data using via the `groupby()` method, then isolate the `'points'` column, and finally apply the `max()` function. This chaining efficiently returns the absolute highest score recorded for the numerical variable within each distinct group defined by the two categorical columns.

The following code snippet executes the calculation to determine the maximum recorded points:

#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

The resulting output is once again a clearly structured, hierarchically indexed Series. Interpretation of these maximum values immediately reveals the locations of peak performance: The highest score across the entire compiled dataset, a remarkable **35** points, was achieved by a player belonging to **Team B** in position **G**. In contrast, the top score recorded for **Team A** in position **F** was **25** points. This specific type of aggregation proves invaluable for the rapid identification of top performers and for setting realistic, category-specific performance benchmarks across various operational segments.

Example 3: Counting Occurrences and Analyzing Group Volume

Moving beyond numerical summary statistics like the mean and maximum, it is critically important

to understand the fundamental structural composition of the data itself. Counting the number of entries (occurrences or volume) within each defined group is essential for assessing the distribution, balance, and volume of data dedicated to each category combination. In the context of our player performance example, using counting allows us to precisely determine the headcount for every unique team and position pairing, which is crucial for subsequent analysis relating to resource allocation, sample size validation, or ensuring the statistical robustness of comparative metrics.

To effectively count the occurrences of each unique combination of `team` and `position`, we again leverage the highly versatile `groupby()` method, specifying our two grouping columns. However, rather than selecting a specific numerical column to aggregate, we directly apply the `size()` function immediately after the grouping operation. The `size()` method is the preferred tool in this scenario because it counts all rows belonging to the group, including those with potential missing or null values, thereby providing the true and complete size of the group partition.

The following syntax calculates the total number of players within each defined team-position group:

#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 resultant Series offers a transparent overview of the player distribution. For instance, we instantly observe a significant structural imbalance: **Team B** heavily concentrates its resources in the **G** position with **4** players, while only **1** player is currently assigned to the **F** position on the same team. This structural insight is exceptionally vital for interpreting numerical results--if we were comparing the average points, we must acknowledge that the average computed for Team B, position F, is derived from a sample size of only one, making its average score highly susceptible to individual variance and less statistically reliable than the other larger groups.

Conclusion and Expanding Your Pandas Aggregation Toolkit

The capability to execute sophisticated aggregation operations by grouping data across multiple categorical columns stands as a foundational and indispensable skill for anyone engaged in

effective data analysis using Python. As we have conclusively demonstrated throughout these examples, the `groupby()` function within pandas provides a highly robust, efficient, and syntactically readable mechanism necessary for summarizing complex datasets and uncovering previously obscured patterns. Whether the analytical objective involves calculating nuanced conditional averages, identifying points of peak performance, or simply mapping out the compositional structure of the underlying data using precise counts, the "split-apply-combine" strategy is instrumental in transforming raw data streams into detailed, actionable business intelligence.

A critical technical detail to master when advancing beyond basic single-column grouping is handling the resulting hierarchical index, or multi-index. The output of multi-column grouping is often a Series or a DataFrame with this layered index structure. Understanding how to seamlessly interact with, manipulate, and potentially flatten this index (most commonly achieved using the `.reset_index()` method) is paramount for performing smooth subsequent data manipulations, merging operations with other datasets, or preparing the data for visualization tools.

We strongly encourage all data practitioners to extend their analytical exploration far beyond the basic core functions of `mean()`, `max()`, and `size()`. The powerful `groupby()` object natively supports a vast array of other highly valuable aggregation methods, including `sum()`, `min()`, `median()`, `std()` (standard deviation), and variance (`var()`). Moreover, for scenarios requiring multiple aggregations simultaneously, the flexible `.agg()` method provides a single, concise syntax to compute diverse statistical profiles in one streamlined operation, thereby offering a complete statistical overview of every defined group and drastically enhancing the efficiency of your data analysis workflows.