How to Easily Calculate Median by Group in Pandas

Authored by

stats writer

December 3, 2025

When conducting data analysis, it is often critical to summarize data not for the entire dataset, but for specific subgroups defined by categorical variables. A common statistical measure required in this context is the median, which provides a robust measure of central tendency, less susceptible to outliers compared to the mean. For users working within the Python ecosystem, the Pandas library offers an exceptionally efficient and intuitive mechanism for performing such calculations using a combination of its core methods.

To successfully calculate the group-wise median in Pandas, we primarily rely on the powerful split-apply-combine strategy, implemented via the groupby() function and the median() function. The process begins by segmenting the DataFrame based on one or more grouping keys. Once grouped, the median calculation is applied independently to the specified numerical column within each segment. The resulting series or DataFrame provides the calculated median value corresponding to every unique group identified.

Understanding this workflow is essential for effective data manipulation. The initial output of the grouping and aggregation operation is typically a Pandas Series with a MultiIndex, especially when grouping by multiple keys. To convert this structured output back into a standard, flat DataFrame where the grouping variables appear as standard columns, the `reset_index()` method is frequently appended to the chain of operations. This method is crucial for ensuring the resulting data structure is clean and ready for further analysis or reporting.

Core Syntax for Grouped Median Calculation

The standard methodology for computing the median value when grouped by a single variable in Pandas adheres to a clean, chained syntax pattern that maximizes readability and efficiency. This pattern leverages the core functions responsible for data segregation and statistical aggregation. Mastering this fundamental syntax is the first step toward advanced data manipulation techniques within the library.

The syntax below illustrates the necessary components. We first call the groupby() function on the target DataFrame (`df`), passing the column name used for grouping (e.g., `'group_variable'`) as a list. We then select the numerical column for which the median should be calculated (e.g., `'value_variable'`). Finally, we apply the median() function. Appending `reset_index()` ensures the grouping variable moves from the index back into a regular column, making the result easier to interpret and use.

You can use the following basic syntax to calculate the median value by group in pandas:

df.groupby().median().reset_index()

For scenarios requiring more granular insights, Pandas allows grouping across multiple categorical dimensions simultaneously. This is achieved simply by passing a list containing all relevant grouping column names (e.g., `'group1'`, `'group2'`) to the groupby() function. This produces a hierarchical grouping structure where the median is calculated for every unique combination of the specified group variables. This functionality is crucial when analyzing complex datasets that require stratification based on two or more distinct attributes.

You can also use the following syntax to calculate the median value, grouped by several columns:

df.groupby().median().reset_index()

The following examples show how to use this syntax in practice.

Placeholder for Ad Block

Example 1: Finding Median Value by One Group (Team)

Our analysis begins by preparing a sample DataFrame that represents typical sports statistics, including categorical data like team and position, and numerical data like points and rebounds. This setup is necessary to illustrate the practical application of grouping techniques. We import Pandas and define the data dictionary, which serves as the foundation for our sample DataFrame object.

This dataset allows us to isolate performance metrics based on membership in two distinct teams ('A' and 'B'). By structuring the data in this manner, we create the perfect environment for demonstrating how the groupby() function segments the data and how the median() function aggregates the numerical columns within those segments.

Suppose we have the following pandas DataFrames:

import pandas as pd

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

#view DataFrame
df

team position points rebounds
0 A G 5 11
1 A G 7 8
2 A F 7 10
3 A F 9 6
4 B G 12 6
5 B G 9 5
6 B F 9 9
7 B F 4 12

Our primary objective in this example is to determine the central tendency of scoring performance based solely on the 'team' membership. This involves applying the `groupby()` function using the 'team' column, and then aggregating the 'points' column specifically using the `median()` aggregation method. This calculation provides insight into how the typical scoring output differs between Team A and Team B, filtering out potential noise from individual star performances which the mean might overemphasize.

We can use the following code to find the median value of the 'points' column, grouped by team:

#calculate median points by team
df.groupby().median().reset_index()

team points
0 A 7.0
1 B 9.0

The interpretation of these results reveals clear distributional differences. For Team A, the ordered points values are . The median, being the average of the two middle values (7 and 7), is 7.0. For Team B, the ordered points are . The median is the average of 9 and 9, which is 9.0. This indicates that Team B generally has a higher typical scoring output than Team A, providing a more robust measure than simply comparing average scores.

From the output we can see:

The median points scored by players on team A is **7**.
The median points scored by players on team B is **9**.

Data analysis often requires aggregating multiple numerical variables based on the same grouping key. Instead of running separate `groupby()` operations for 'points' and 'rebounds', Pandas allows us to pass a list of numerical columns to be aggregated simultaneously. This approach significantly

streamlines the code and ensures that all relevant metrics are calculated efficiently within a single operation. This capability is a hallmark of the flexibility provided by the Pandas aggregation framework.

When aggregating multiple columns, we pass a list of column names, such as ``, immediately after the `groupby()` call. Unlike the single-column selection using brackets ``, using double brackets `]` maintains the resulting structure as a DataFrame rather than collapsing it into a Series. Crucially, when aggregating multiple columns, the resulting structure is already columnar, and calling `reset_index()` is often unnecessary unless the grouping column is explicitly desired as a standard index column.

Note that we can also find the median value of two variables at once:

#calculate median points and median rebounds by team
df.groupby()].median()

points rebounds
team
A 7.0 9.0
B 9.0 7.5

Placeholder for Ad Block

Example 2: Find Median Value by Multiple Groups (Team and Position)

Moving beyond simple team comparisons, data analysts often need to segment data based on a hierarchy of factors. In our sports example, performance metrics like points are likely influenced by both the 'team' and the 'position' played. To calculate the median performance based on these two categorical factors combined, we extend the list passed to the groupby() function to include both dimensions: . This creates four distinct subgroups (A-F, A-G, B-F, B-G), and the median() function is applied individually to each of these unique combinations.

The resulting series from this multi-level grouping initially uses a MultiIndex, where both 'team' and 'position' define the rows. To transform this into a flat, readable table, we again utilize the `reset_index()` method. This step converts the hierarchical index levels into standard columns, making the final output immediately consumable for reporting and comparison purposes.

The following code shows how to find the median value of the 'points' column, grouped by team and position:

#calculate median points by team and position
df.groupby().median().reset_index()

team position points
0 A F 8.0
1 A G 6.0
2 B F 6.5
3 B G 10.5

The detailed output provides precise statistical metrics for each subgroup. For instance, the median points for Team B players in the 'G' position (10.5) is significantly higher than that of Team A players in the same position (6.0), a contrast not apparent when grouping only by team. Conversely, Team A's 'F' position players (8.0) perform slightly better than Team B's 'F' players (6.5) based on the median score. This granularity is essential for pinpointing performance strengths and weaknesses across stratified categories.

The median points scored by players in the 'F' position on team A is **8**.
The median points scored by players in the 'G' position on team A is **6**.
The median points scored by players in the 'F' position on team B is **6.5**.
The median points scored by players in the 'G' position on team B is **10.5**.

Conceptual Differences: Median vs. Mean in Grouped Analysis

While the mean (average) is perhaps the most common measure of central tendency, selecting the median for grouped analysis often yields more meaningful results, particularly in datasets prone to skewness or containing extreme outliers. The median represents the 50th percentile of the data; half of the observations lie above it, and half lie below. This position-based measure makes it inherently robust to unusual data points that might disproportionately pull the mean in one direction.

In contexts such as salary data, housing prices, or, as demonstrated here, individual sports statistics, a single exceptional value (e.g., a massive outlier score) can severely distort the mean, making it a poor representation of the typical value for the group. By using the median() function within the groupby() function chain, analysts ensure that the summary statistics accurately reflect the majority of the observations within each defined subgroup, leading to more reliable conclusions during the data analysis phase.

Choosing the appropriate aggregation method is a critical decision in Pandas grouping operations. If the distribution of the value variable within a group is highly symmetrical and free of outliers, the mean and median will be similar. However, if the distribution is skewed, relying on the median

provides a more conservative and often more representative measure of the typical value. The ability to switch effortlessly between aggregation functions like `median()`, `mean()`, `sum()`, or `std()` within the Pandas framework allows analysts to adapt their statistical approach based on the underlying data characteristics.

Advanced Grouping: Handling Missing Data and Data Types

When performing group-wise calculations using Pandas, it is crucial to consider the impact of missing data (NaN values) and the data types of the columns involved. By default, the `median()` aggregation function, like most statistical aggregation functions in Pandas, automatically excludes NaN values when performing the calculation. This behavior is generally desirable, ensuring that missing observations do not artificially skew the resulting group median.

However, analysts must be aware that if a specific group contains only missing values for the targeted column, the resulting median for that group will also be NaN. It is good practice during the initial stages of data analysis to utilize methods such as `df.isnull().sum()` to identify the prevalence of missing data, and potentially employ imputation techniques or filtering prior to running the groupby() function and the median() function to maintain data integrity and interpretability. Ensuring that the value column is of a numeric data type is also mandatory, as the median function cannot operate on string or object columns.

Summary of Grouped Aggregation Techniques

The `groupby().median()` chain is a cornerstone technique for effective data summarization in the Pandas library. It embodies the powerful split-apply-combine paradigm, allowing complex statistical queries to be executed with concise and readable Python syntax. Whether the goal is to group by one variable, multiple variables, or aggregate several value columns simultaneously, the core syntax remains adaptable and highly performant, even on large DataFrame objects.

By consistently applying the principles demonstrated in the examples--defining the grouping keys, selecting the target value column(s), applying the aggregation function (`median`), and utilizing `reset_index()` for clarity--analysts can quickly transform raw data into structured, actionable summary statistics. This ability to derive robust central tendency measures for distinct subgroups is indispensable for rigorous analytical work.

Further Reading

The following tutorials explain how to perform other common functions in pandas: