How to Extract the Top N Rows by Group Using Pandas

Authored by

stats writer

November 28, 2025

The ability to efficiently analyze subset data is fundamental in modern data science. When working with large datasets, it is often necessary to segment the data based on categorical variables and then extract the most relevant observations from each segment. This highly specific operation--finding the top N rows within distinct groups--is a common challenge that the **Pandas** library in **Python** addresses elegantly.

This comprehensive guide explores the primary and most robust technique for accomplishing this task using built-in **Pandas** functionalities. We will focus specifically on how to combine the powerful **groupby()** function with the slicing mechanism provided by the **head()** method. Mastering this combination allows analysts to quickly identify high-performing records, outliers, or simply the first few observations relevant to any specified grouping structure within a **DataFrame**.

Understanding this methodology is crucial for tasks ranging from identifying the top sales performers in each region to finding the N most recent logs per server. By leveraging these optimized functions, data professionals can perform complex group-wise selection operations without resorting to slow, iterative loops, ensuring efficient and scalable data processing.

The fundamental **Pandas** syntax for extracting the top N rows, segmented by a specific column, is remarkably concise. You can use the following basic syntax to get the top N rows by group in a **Pandas DataFrame**:

df.groupby('group_column').head(2).reset_index(drop=True)

This particular syntax will return the top **2** rows for every unique category defined within the group_column. This is a crucial distinction: the result is not just the top two rows overall, but the top two rows after the internal ordering applied by the **groupby()** operation, which typically preserves the initial order of the rows within each group.

To adjust the selection size, simply change the value inside the **head()** function to return a different number of top rows. This straightforward adaptability is why this method remains a cornerstone for grouped data manipulation in **Pandas**. The following examples show how to use this syntax with a specific sample **DataFrame**:

import pandas as pd

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

#view DataFrame

print(df)

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

Understanding the Core Concept: Grouping in Data Analysis

The core mechanism behind extracting the top N records per group relies entirely on the powerful split-apply-combine strategy implemented efficiently within **Pandas**. When we invoke the **groupby()** method, the **DataFrame** is logically split into multiple sub-frames, with each sub-frame containing rows that share the same unique value(s) in the specified grouping column(s).

Crucially, the **groupby()** operation itself does not immediately alter the data structure; rather, it returns a GroupBy object. This object holds all the necessary metadata to understand which rows belong to which group, enabling efficient subsequent operations. The power of this object lies in its ability to apply functions to each individual group independently. When we chain the **head()** method immediately after the GroupBy object, **Pandas** iterates through these internal groups and extracts the specified number of rows from the beginning of each group. This process ensures that the selection is performed group-wise, not globally, thereby fulfilling the requirement of finding N rows *per* group.

It is paramount to understand the implication of using **head()** without prior sorting. Since **Pandas** typically preserves the row order within each group as it existed in the original **DataFrame**, **head(N)** will return the first N rows for that group based on the initial index sequence. If the goal is to find the top N records based on a specific metric (e.g., the highest 5 scores), an explicit sorting step using sort_values() must precede the grouping, or an alternative function like nlargest() should be utilized, which we will discuss later.

Example 1: Selecting Top N Rows Based on a Single Column Group

Our first demonstration utilizes only the **team** column for grouping. This scenario is common when

analyzing data segmented by primary identifiers, such as departments, countries, or, in our case, distinct teams. By setting N=2, we instruct **Pandas** to extract the first two records found sequentially for Team A and the first two records for Team B, based on the original data order. The following code shows how to return the top 2 rows, grouped exclusively by the **team** variable:

```
#get top 2 rows grouped by team
df.groupby('team').head(2).reset_index(drop=True)
```

team position points
0 A G 5
1 A G 7
2 B G 9
3 B G 9

The resulting **DataFrame**, which maintains a clean, zero-based index after the reset_index call, clearly illustrates the successful application of the group-wise selection. For Team A, we retrieved rows with 5 and 7 points (original indices 0 and 1). For Team B, we retrieved rows with 9 and 9 points (original indices 5 and 6). This confirms that the operation successfully isolated the top 2 records for each group based on their position in the input data.

The output displays the top 2 rows, grouped by the **team** variable. The critical takeaway here is the automatic application of the **head()** operation internally to every unique subset generated by the **groupby()** object, streamlining what would otherwise require complex loops.

Example 2: Advanced Grouping by Multiple Variables

Grouping often requires the consideration of multiple categorical dimensions simultaneously. For instance, we might not just want the top records per team, but the top records per team *and* per position. This multi-level grouping provides a much finer level of granularity in the analysis, essential for complex hierarchical data structures. The **groupby()** function easily accommodates this requirement by accepting a list of column names rather than a single string.

When multiple columns are specified, **Pandas** creates unique subgroups based on the combination of values across those columns. In our example, the groups will be defined by the tuple (Team, Position): (A, G), (A, F), (B, G), and (B, F). The **head(2)** method is then applied individually to these four distinct groups, ensuring we select the top two entries for each unique combination. The following code shows how to return the top 2 rows, grouped by the **team** and **position** variables:

```
#get top 2 rows grouped by team and position
```

df.groupby().head(2).reset_index(drop=True)

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

The output displays the top 2 rows, grouped by the **team** and **position** variables. This detailed selection confirms eight total rows: two for each of the four unique groups. For example, Group (A, G) returns 5 and 7 points, and Group (A, F) returns 9 and 12 points. This demonstrates the seamless scalability of the **groupby()** and **head()** combination when dealing with increasing complexity in grouping variables.

The Importance of Order: Sorting Before Grouping

While `groupby().head(N)` extracts the first N rows based on the current sequence, in most real-world applications, "Top N" implies selection based on the maximum values of a metric column, such as sales or scores. To achieve this desired ranked selection, an explicit sorting step must be introduced prior to the grouping and selection phases. The standard way to enforce a ranked selection is by using the sort_values() method.

When applying sort_values(), it is crucial to sort by the metric column (e.g., points) in descending order to ensure the highest values are positioned at the top of each potential group. This modification transforms the operation into finding the top performers, not just the first entries. For example, to find the top 2 players per team based on points, the command chain becomes: `df.sort_values(by='points', ascending=False).groupby('team').head(2).reset_index(drop=True)`. This sequence ensures that within Team A, the players with the highest points are selected first, and the same principle is applied independently to Team B, thereby transforming the positional selection into a meaningful, ranked selection.

Alternative Method: Utilizing nlargest() for Efficiency

While sorting followed by groupby().head() is effective, **Pandas** offers a highly optimized alternative specifically designed for ranking-based selections: the nlargest() method. This method

is often preferred for performance reasons, especially on very large **DataFrames**, as it avoids a full sort of the entire dataset, which can be computationally expensive (O(N log N)).

When applied directly to a GroupBy object, nlargest() automatically finds the N rows with the highest values in the designated metric column within each subgroup. The syntax is concise and immediately conveys the intent: `df.groupby('group_column').nlargest(N)`. Using nlargest() is the most idiomatic and generally recommended approach for metric-based Top N queries in **Pandas**.

It is important to note a structural difference in its output: when applied to a GroupBy object, nlargest() returns a Series with a MultiIndex, where the first level is the grouping key and the second level is the original index of the selected rows. If a standard **DataFrame** output is required, further processing, such as converting the Series back to a **DataFrame** and using reset_index(), is necessary to achieve a clean structure.

Deep Dive into the reset_index() Function

The reset_index(drop=True) command is essential in finalizing the output **DataFrame** after group-wise selection. When the **groupby()** operation is performed and rows are filtered out (as is the case with **head(N)**), the resulting **DataFrame** retains the original index labels of the retained rows. This results in a discontinuous index sequence.

The reset_index() method resolves this by discarding the existing index structure and replacing it with a new, sequential, zero-based integer index. The argument drop=True ensures that the old, discontinuous index values are completely discarded and not added back into the **DataFrame** as a new column. This guarantees the final **DataFrame** is clean, structurally sound, and ready for immediate use, making it an essential final step in achieving readable, standardized output.

Practical Use Cases and Performance Considerations

The technique of finding the top N rows per group is fundamental to numerous analytical tasks. Common applications include financial analysis (tracking the top five stock trades per sector), web analytics (identifying the three most frequent user actions per country), or database management (locating the ten most recent error logs generated by each server instance). The versatility of the groupby().head(N) structure, particularly when combined with efficient sorting or nlargest(), makes it indispensable across various analytical domains.

When dealing with massive datasets, performance choice is key. The chained method using sort_values() followed by groupby().head(N) forces **Pandas** to perform a full sort of the entire **DataFrame**, which has an algorithmic complexity of O(N log N). This overhead can be substantial.

For performance-critical ranked selection, the nlargest() approach is highly recommended. It uses specialized quickselect algorithms to find the N largest elements without fully sorting the remainder of the data within each group. This approach is significantly faster (closer to O(N * k * log k) complexity, where k is N) for small selection sizes (k), providing a performance advantage essential for large-scale data manipulation in **Python**.

Conclusion: Mastering Grouped Data Selection

Selecting the top N rows by group is a core skill for any **Python** data analyst utilizing **Pandas**. We have explored the primary method, which couples the foundational power of **groupby()** with the simplicity of **head()**, demonstrating its flexibility in handling both single and multiple grouping columns. We also emphasized the necessity of using reset_index(drop=True) to ensure clean, structured output.

By internalizing these techniques, and knowing when to use the optimized nlargest() alternative for ranked selections, data professionals can perform intricate data segmentations with confidence, speed, and precision, ensuring that the most relevant information is always extracted cleanly and efficiently for deeper analysis.