How to Sort a Pandas DataFrame by Multiple Columns

Authored by

stats writer

December 2, 2025

Introduction to Multi-Column Sorting in Pandas

Sorting data is a foundational requirement in data analysis, allowing analysts to organize and interpret datasets efficiently. When working with the powerful Pandas library in Python, achieving simple sorting is straightforward. However, complex datasets frequently necessitate sorting based on criteria spanning several columns--a process known as multi-column sorting. This technique is critical for establishing a definitive hierarchy within the data, especially when primary sort keys contain duplicate values. Understanding how to execute this operation correctly is essential for any data professional utilizing the Pandas framework.

The primary mechanism for sorting data within a DataFrame is the DataFrame.sort_values() method. This versatile function is designed not only for single-column sorting but also for handling complex, multi-tiered sorting logic. When sorting by more than one column, the order in which the columns are provided to the method dictates the sorting priority. The first column listed acts as the primary sort key, followed by the second as the secondary, and so on. This hierarchical sorting ensures that when rows share the same value in the primary column, the secondary column is used as a tie-breaker, providing a precise and deterministic organization of the data.

Furthermore, the sort_values() method provides exceptional control over the direction of the sort. For each column specified in the list of keys, the user can explicitly define whether the sort should be **ascending** (smallest to largest) or **descending** (largest to smallest). This granularity is crucial because often, the optimal organization requires sorting different columns in different directions--for instance, sorting records first by date descending, and then by sales ascending. Mastering the configuration of these parameters is key to leveraging the full power of DataFrames.

The Anatomy of DataFrame.sort_values()

The sort_values() function is central to organizing data in Pandas. When applying it to a DataFrame, several key parameters govern its behavior, particularly when dealing with multiple columns. The most important parameter is `by`, which accepts a single column name (a string) or, more relevantly for this topic, a **list of column names** (a list of strings). This list defines the exact sequence of sorting operations that Pandas will execute.

The second essential parameter is `ascending`. If the `by` parameter contains a list of columns, the `ascending` parameter must also accept a list of corresponding boolean values (`True` for ascending, `False` for descending). The length of the `ascending` list must strictly match the length of the `by` list. If `ascending` is omitted entirely, Pandas defaults to `True` for all specified columns, meaning every column will be sorted from smallest to largest. This explicit control over sort direction is what enables complex, multi-criteria sorting.

Beyond the core parameters, it is important to understand `inplace` and `na_position`. By default,

Pandas operations return a new DataFrame, leaving the original unchanged. Setting `inplace=True` modifies the existing DataFrame directly, which can be useful for memory management in very large datasets. The `na_position` parameter dictates where missing values (NaNs) are placed during the sort--either `'first'` or `'last'`. By default, NaNs are positioned at the end of an ascending sort and at the beginning of a descending sort, but this parameter allows precise control over their placement regardless of the sort direction.

Basic Syntax for Multi-Column Sorting

The core syntax for executing a multi-column sort is highly compact and powerful. The primary requirement is wrapping the desired column names within a Python list structure, which is then passed to the `by` argument of the method. If we want to sort a DataFrame named `df` first by `column1` and then by `column2`, the structure below provides the cleanest implementation, explicitly demonstrating how to set descending order for the first column and ascending order for the second.

It is crucial to note the positional mapping: the first boolean in the `ascending` tuple corresponds directly to the first column name in the `by` list, and the second boolean corresponds to the second column name, maintaining a one-to-one relationship across all columns involved in the sorting process.

The following example outlines the standard syntax used to sort a Pandas DataFrame by multiple columns:

df = df.sort_values(, ascending=(False, True))

In this specific syntax, `column1` will be sorted in **descending** order (`False`), serving as the primary sorting criterion. Any rows that share the same value in `column1` will then be sorted by `column2` in **ascending** order (`True`). This two-stage process guarantees a predictable and precise ordering of the entire dataset based on complex criteria.

Detailed Example: Defining the Dataset

To illustrate the mechanics of multi-column sorting, consider a statistical dataset representing player performance metrics. This DataFrame, which we will name `df`, contains columns for **points** scored, **assists** given, and **rebounds** grabbed. Our objective will be to reorganize this data based on specific criteria related to these performance indicators.

The creation of this sample data utilizes standard Pandas syntax, employing a dictionary structure where keys are the column names and values are lists representing the data points. Observing the

initial, unsorted state of the `df` is important as it establishes the baseline from which all sorting operations will proceed.

import pandas as pd

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

#view DataFrame
df

points assists rebounds
0 14 5 11
1 20 7 8
2 9 7 10
3 20 9 6
4 25 12 6
5 29 9 5
6 20 9 9
7 25 4 12

This initial DataFrame, containing eight rows, serves as our foundation. Notice that the index (0 through 7) is currently sequential, reflecting the order of creation, and there are several rows where the **points** values are duplicated (e.g., three rows have 20 points, two rows have 25 points). These duplicate values are precisely where the need for a secondary sort key, such as **assists**, becomes apparent to break the ties.

Executing a Default Multi-Column Sort

In the first scenario, we will implement a standard multi-column sort using the sort_values() method, but we will intentionally omit the `ascending` parameter. As previously established, omitting this parameter defaults the sort direction to `True` (ascending) for all specified columns. Our goal is to sort the data primarily by **points**, and secondarily by **assists**, both in ascending order.

By executing the code below, we instruct Pandas to evaluate the **points** column first, arranging all rows from the minimum point total to the maximum. Then, within groups of equal point totals, the algorithm proceeds to the secondary key, **assists**, sorting those rows from the minimum number of assists to the maximum. This demonstrates the fundamental hierarchical structure of the multi-

column sort.

The following syntax sorts the rows of the DataFrame primarily by **points** ascending, and then by **assists** ascending:

#sort by points ascending, then assists ascending
df = df.sort_values()

#view updated DataFrame
df

points assists rebounds
2 9 7 10
0 14 5 11
1 20 7 8
3 20 9 6
6 20 9 9
7 25 4 12
4 25 12 6
5 29 9 5

Upon reviewing the resulting DataFrame, we can confirm the successful application of the sorting logic. The **points** column is sorted numerically from 9 up to 29. Furthermore, observe the handling of ties: for rows where **points** equals 20 (indices 1, 3, 6), the secondary sort key, **assists**, successfully orders these rows (7, 9, 9). Where **assists** is also tied (indices 3 and 6, both having 9 assists), the original row position or the tertiary column (rebounds, which we did not specify) would maintain the order, though in this case, the rows 3 and 6 are adjacent.

Controlling Sort Direction with the `ascending` Parameter

While default ascending sorting is useful, practical data analysis often requires a mix of ascending and descending orders. For instance, we might want to rank players by the highest number of **points** first (descending), but if points are equal, we might want to prioritize players who showed more discipline, perhaps by ordering them by **assists** ascending. To achieve this mixed order, we must explicitly pass a tuple or list of boolean values to the `ascending` parameter in the sort_values() method.

The key requirement here is maintaining the alignment between the column list and the boolean list. If we specify , and we want points to be descending and assists to be ascending, the corresponding boolean list must be `(False, True)`. The `False` value applies to `'points'`, and the `True` value applies to `'assists'`. Misalignment in the length or order of these two lists will

result in a `ValueError` from Pandas, emphasizing the need for precision.

The following syntax demonstrates how to sort the primary key, **points**, in descending order, and the secondary key, **assists**, in ascending order:

#sort by points descending, then assists ascending
df = df.sort_values(, ascending = (False, True)))

#view updated DataFrame
df

points assists rebounds
5 29 9 5
7 25 4 12
4 25 12 6
1 20 7 8
3 20 9 6
6 20 9 9
0 14 5 11
2 9 7 10

Analyzing the output confirms the effectiveness of the mixed-sort strategy. The **points** column is now clearly sorted in descending order, beginning with 29. Crucially, when **points** are tied at 25 (indices 7 and 4), the secondary sort key, **assists**, is applied in ascending order (4 then 12). Similarly, for the rows tied at 20 points (indices 1, 3, 6), assists are ordered ascendingly (7, 9, 9). This demonstrates the precise hierarchical tie-breaking capability inherent in the multi-column sort_values() function.

Advanced Sorting Scenarios and Tie-Breaking Logic

The power of multi-column sorting scales seamlessly beyond just two columns. The exact syntax demonstrated for two columns--providing parallel lists to the `by` and `ascending` parameters--can be extended to sort by three, four, or any arbitrary number of columns within the DataFrame. This ability is vital when dealing with complex datasets where ties in the primary and secondary keys are common, requiring several levels of distinction to establish a unique and logical order for every row.

When sorting by multiple columns, the process relies on sophisticated tie-breaking logic. Pandas performs the sort iteratively: it sorts based on the first column. If any groups of rows share the same value in that first column, the algorithm then restricts its focus to those tied rows and applies the sort based on the second column. If ties still persist, the third column is used, and so on, until

either a unique order is established for all rows, or all specified columns have been exhausted. If rows remain tied after all specified columns are evaluated, their final relative order will depend on their original order within the DataFrame (stable sort).

To implement sorting by three columns, say , with the goal of prioritizing overall offensive output (points descending), rewarding teamwork (assists descending), and minimizing defensive reliance (rebounds ascending), the syntax would involve three booleans: `ascending=(False, False, True)`. This layered approach ensures that the most impactful sorting criterion is always addressed first, followed by criteria of decreasing importance, resulting in a highly customized and meaningful organization of the data structure.

Handling Missing Values and Performance Considerations

An often-critical aspect of sorting real-world data involves managing missing values, represented as `NaN` (Not a Number) in DataFrames. By default, the sort_values() method handles NaNs by placing them at the end of the sorted output if the column is being sorted ascendingly, and at the beginning if descendingly. However, analysts sometimes require missing data to be consistently grouped at the top or bottom, irrespective of the column's sort direction.

This control is achieved using the `na_position` parameter. Setting `na_position='first'` ensures that all rows containing a NaN in the relevant sorting column are moved to the top of the dataset. Conversely, setting `na_position='last'` places these rows at the bottom. This parameter accepts a single string value and applies consistently across all columns being sorted. For example, if sorting by , and `date` has NaNs, setting `na_position='first'` would group all rows with missing dates at the beginning, before sorting the remaining, non-missing data by both columns.

Regarding performance, the underlying sorting algorithm used by Pandas is generally optimized, utilizing fast C implementations. However, sorting is inherently a resource-intensive operation (typically O(N log N) complexity). When sorting extremely large DataFrames by many columns, performance optimization becomes important. If memory consumption is a concern, setting `inplace=True` can prevent the creation of a temporary copy of the DataFrame, although this means the original data is permanently modified. For maximum speed, ensure that the columns being sorted are of appropriate, non-object dtypes (e.g., numeric types are faster to compare than strings).

Summary of Multi-Column Sorting Best Practices

Multi-column sorting using the sort_values() method is a fundamental skill for effective data manipulation in Pandas. Achieving successful and logical sorting hinges on adherence to a few best practices. Firstly, always ensure that the list of columns passed to the `by` parameter

accurately reflects the desired priority (Primary key first, secondary key second, and so on). The order of the columns is the order of evaluation.

Secondly, when specifying custom sort directions, absolute alignment between the `by` list and the `ascending` list (tuple of booleans) is non-negotiable. A mismatch in length will cause an immediate error, while a mismatch in order will produce incorrect results without warning. It is highly recommended to explicitly define the sort direction for all columns, rather than relying on the default `ascending=True`, especially in collaborative projects, to enhance code readability and predictability.

Finally, always consider the stability of the sort and the handling of edge cases, such as missing data. Pandas uses a stable sort algorithm, meaning that if two rows remain tied even after applying all specified sort keys, their relative order from the original DataFrame is preserved. Using the `na_position` parameter appropriately ensures that data analysts can control where missing values are grouped, preventing them from skewing visual interpretations or further processing steps.

Further Documentation and Resources

For comprehensive details regarding all optional parameters, exceptions, and advanced usage scenarios of this critical function, readers are strongly encouraged to consult the official documentation. The complete documentation for the Pandas **sort_values()** function is the authoritative source for optimizing sorting logic and understanding internal behavior.