# Learning How to Sort Pandas DataFrames by Multiple Columns

Authored by

**Mohammed looti**

November 1, 2025

## Introduction to Sorting DataFrames

Sorting data is a fundamental requirement in nearly all data analysis tasks. When working with the powerful Pandas library in Python, data is typically stored within a two-dimensional labeled structure known as a DataFrame. While sorting by a single column is straightforward, real-world datasets often necessitate a more nuanced approach, requiring organization based on the hierarchical order of multiple columns simultaneously.

This advanced technique, commonly referred to as multi-column sorting, becomes indispensable when the primary sorting criterion (e.g., 'Category') results in ties. These ties must then be systematically resolved using secondary, tertiary, or even quaternary criteria (e.g., 'Value' or 'Date'). Mastering this multi-level organization ensures that your data presentation is not only logical and structured but also provides immediate, actionable insights, greatly enhancing comparative analysis.

The Pandas library provides the highly flexible and essential `sort_values()` method to handle these complex sorting requirements. This method is the definitive tool for rigorously organizing the rows of a DataFrame according to the values found in one or more specified columns, allowing data scientists to define precise sorting hierarchies.

## Understanding the Core Syntax of sort_values()

To effectively sort a DataFrame using multiple columns, the `sort_values()` method relies on two primary arguments: a list of columns defining the sorting keys, and an optional, corresponding list or tuple of boolean values that specify the desired sorting direction for each key.

The initial argument is a list of column names, which is crucial as it establishes the strict hierarchy of the sort. The column listed first dictates the primary sort order. If any rows share identical values in this primary column (a tie condition), the second column in the list is automatically invoked to break that tie. This process continues sequentially down the list of specified columns until all rows are uniquely ordered or the list is exhausted.

The basic syntax structure for executing a multi-column sort, where `column1` serves as the primary key and `column2` acts as the secondary key, is demonstrated below. It is vital to note how the `ascending` parameter receives a tuple whose length and order correspond exactly to the columns specified in the sort list:

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

In this specific, powerful example, the DataFrame `df` is first sorted by `column1` in **descending** order (indicated by the boolean value `False`). Subsequently, for all groups of rows that share the

exact same `column1` value, they are then sorted by `column2` in **ascending** order (indicated by `True`). Grasping this precise, paired relationship between the list of columns and the `ascending` tuple is absolutely crucial for efficient and effective data manipulation using Pandas.

## Prerequisites: Establishing the Example Dataset

Before proceeding with complex sorting demonstrations, it is essential to establish a robust and representative sample DataFrame. This dataset is designed to represent performance statistics--including columns for points, assists, and rebounds--and will serve as the foundation for illustrating precisely how hierarchical sorting affects the final row order. This setup utilizes the Pandas library initialized under the standard alias `pd`, a widely accepted convention in modern Python development environments.

We begin by defining a simple dictionary containing our structured data, which is then swiftly converted into a Pandas DataFrame object. A key feature of this sample data is the deliberate inclusion of several duplicated values in the `points` column (specifically, the values 20 and 25). These repetitions are intentional, as they create the necessary "tie conditions" that will force the secondary sorting criteria (the `assists` and `rebounds` columns) to be fully utilized.

The following comprehensive code block generates and displays the initial, unsorted state of our structured data:

**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 original dataset is currently indexed sequentially, reflecting the order in which the data was first entered (from index 0 through 7). We are now prepared to systematically sort this data using different combinations of the `points` and `assists` columns, thereby demonstrating the full operational power and flexibility of the `sort_values()` method within Pandas.

## Practical Example 1: Implementing a Default Ascending Hierarchy

For our initial practical scenario, the objective is to sort the rows primarily based on the number of **points**, and then apply a secondary sort based on the number of **assists**. A critical feature of the `sort_values()` function is its default behavior: if the `ascending` parameter is entirely omitted, the method automatically assumes an ascending order (sorting from the smallest value to the largest) for all columns specified in the list.

To execute this default sort, we apply the method by providing only the list of columns . Functionally, this call is identical to explicitly setting `ascending=(True, True)`. Carefully observe the output below, which organizes the data meticulously: first by the points scored, and then immediately resolves any ties encountered in the points column by referencing the assists column.

**#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

By reviewing the resulting sorted DataFrame, the clear hierarchical sort structure is evident. The `points` column is organized perfectly from 9 up to 29. More importantly, when the `points` value is tied at 20 (originally indices 1, 3, and 6), the `assists` column successfully dictates the final sub-order (7, followed by two instances of 9). It is worth noting that where both `points` and `assists` are tied (indices 3 and 6), their relative order remains stable based on their original input position, unless a third, tie-breaking sorting column is introduced.

## Practical Example 2: Fine-Grained Control with Mixed Sort Directions

In advanced data analysis scenarios, it is frequently necessary to mix sort directions. For instance, we might want to rank the primary column in **descending** order (to find the top performers) while simultaneously applying an **ascending** order to the secondary column (perhaps to prioritize those top scorers who achieved their scores with the fewest assists). Achieving this nuanced organization requires explicitly defining the sort direction for every column using the flexible `ascending` parameter.

The `ascending` parameter must be supplied as a tuple or list of boolean values. Crucially, the sequence and length of this boolean list must correspond exactly to the sequence and number of columns provided in the primary sort list. The boolean value `False` instructs Pandas to sort in descending order (largest to smallest), while `True` maintains the ascending order (smallest to largest).

We will now execute a sort where the data is first organized by **points descending**, followed by **assists ascending**. This precise behavior is achieved by setting the `ascending` tuple to the sequence `(False, True)`:

**#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

A thorough examination of this output confirms that the specified mixed-direction behavior has been perfectly executed:

The `points` column dominates the initial sort, starting with the largest value (29) and proceeding downward.
Where `points` are tied at 25 (indices 7 and 4), the `assists` column resolves the tie by sorting in

ascending fashion (4 comes before 12).
Similarly, where `points` are tied at 20 (indices 1, 3, and 6), the `assists` column correctly sorts them ascendingly (7, then the two instances of 9).

This powerful demonstration highlights the versatility and precision offered by the `ascending` parameter, providing developers with complete, fine-grained control over the presentation and organization of complex datasets using the `sort_values()` method.

## Advanced Usage: Scaling and Persistence

While our focus has primarily been on sorting using two columns, the underlying mechanism of the `sort_values()` method is intrinsically scalable. This robust design allows analysts to effortlessly extend the syntax to sort by any practical number of columns required for deep analysis. Scaling involves a simple process: increasing the length of the column list and ensuring the corresponding `ascending` tuple matches this new length precisely.

For a complex scenario where you might need to sort by `points` descending, followed by `assists` ascending, and finally break any remaining ties using `rebounds` descending, the function call remains clean and readable:

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

It is critically important to recall a core principle of Python object methods: the `sort_values()` method, by default, operates immutably--meaning it returns a new, sorted DataFrame object but leaves the original object unmodified. If the goal is to permanently apply the sorting changes to the existing DataFrame, you have two primary options. The first, and generally preferred method in modern Python coding practices, is to explicitly assign the result back to the original variable (as demonstrated throughout all preceding examples: `df = df.sort_values(...)`). Alternatively, you can utilize the less-favored `inplace=True` argument within the function call.

## Summary and Recommendations for Data Mastery

The proficiency to execute multi-column sorting is recognized as a fundamental and indispensable skill for anyone leveraging the Pandas library for rigorous data analysis. By carefully specifying an ordered list of columns and exercising precise control over the corresponding `ascending` parameter, users gain absolute authority over how their data is hierarchically arranged, which is essential for facilitating unambiguous comparisons and generating highly insightful reports.

We have successfully covered two critical paradigms: the rapid default ascending sort across all columns, and the nuanced, mixed-direction sort exemplified by the powerful `ascending=(False,`

`True)` tuple. These examples vividly illustrated how secondary and tertiary sort criteria are efficiently utilized to systematically break ties established by the preceding primary criteria.

For exploring detailed edge cases, such as managing missing values using the `na_position` argument or controlling sorting stability via the `kind` argument, users should consult the official Python and Pandas documentation. The complete and authoritative documentation for the Pandas `sort_values()` function remains the definitive resource.

## Additional Resources for Pandas Mastery

To significantly enhance your ability to manipulate complex DataFrame structures, the following specialized tutorials provide guidance on other common and crucial operations within Python and the Pandas ecosystem:

How to filter rows based on multiple complex conditions.
Advanced techniques for grouping and aggregating data effectively using `groupby()`.
Robust methods for merging and joining multiple DataFrames efficiently.

By integrating these fundamental data transformation and organization techniques, you will be exceptionally well-equipped to tackle the most demanding data manipulation challenges encountered in modern data analysis workflows.