How to Easily Reorder Columns in Your Pandas DataFrame

Authored by

stats writer

December 5, 2025

The ability to manipulate the structure of a DataFrame is an essential skill for effective data analysis using the Python ecosystem. One of the most frequent requirements when preparing data for visualization or modeling is altering the sequence of columns. While the Pandas library provides several sophisticated methods for achieving this, the most common and straightforward technique involves indexing the DataFrame using a defined list of column names in the desired order. Mastering column reordering ensures that your datasets are properly formatted for downstream consumption by other analytical tools or libraries.

Several key methodologies exist for controlling column order within a DataFrame. Beyond simple list indexing, users can employ reassignment using the `df.columns` attribute, utilize the specialized function df.reindex(), or perform precise positional adjustments using `df.loc` and `df.iloc` for more complex slicing operations. Furthermore, when integrating new data, the `df.insert()` method offers granular control over where a new column should be placed relative to existing features. Understanding these varied approaches allows data scientists to select the most efficient and readable method based on the complexity of the data restructuring task.

The fundamental principle for reordering columns in Pandas involves supplying the DataFrame with a subset of its columns, listed in the exact sequence you wish them to appear. This technique is highly efficient because it leverages Pandas' built-in indexing optimization, making it the preferred method for quick rearrangement. You must ensure that the list of columns provided is complete and contains all necessary columns, otherwise, any omitted columns will be dropped from the resulting DataFrame. This method is concise and is generally the first approach recommended for restructuring column organization.

The Idiomatic Approach: Column List Reassignment

The most concise and widely adopted method for rearranging columns in a DataFrame uses double square brackets to pass a list of column names. This syntax simultaneously selects the columns and defines their output order. If you intend to reorganize columns without dropping any, simply pass a complete list of all column names in their new desired arrangement.

To execute this standard reordering, you construct a list containing the names of the existing columns and assign this list back to the DataFrame using the indexing operator. The resulting DataFrame will maintain all rows and data values but present the columns in the newly specified sequence. This approach is often encapsulated in a single, readable line of Python code, making it highly effective for scripting and interactive sessions.

For instance, if you have columns named 'column1', 'column2', and 'column3', and you wish to prioritize 'column2' and 'column3' before 'column1', the syntax is remarkably straightforward. This technique works by creating a copy of the subset defined by the list, ensuring that the original DataFrame remains unchanged unless the result is explicitly assigned back to the original variable

name.

You can use the following syntax to quickly change the order of columns in a Pandas DataFrame:

df]

Setting Up the Sample DataFrame

To thoroughly demonstrate the various methods of column reordering, we will first create a sample DataFrame representing fictitious sports statistics. This dataset includes columns for 'points', 'assists', and 'rebounds'. Establishing a consistent dataset allows us to verify the output of each restructuring technique accurately.

The initialization process involves importing the Pandas library and then defining a dictionary where keys represent column names and values are lists of data points. This dictionary is then passed to the `pd.DataFrame()` constructor, which materializes our sample data structure ready for manipulation.

The following example code defines and displays the initial structure of our sample data. Notice the initial column order: 'points', 'assists', and 'rebounds'. Our goal in the subsequent examples will be to change this inherent order using different methodologies.

import pandas as pd

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

#display DataFrame
df

points assists rebounds
0 25 5 11
1 12 7 8
2 15 7 10
3 14 9 6
4 19 12 6
5 23 9 5
6 25 9 9
7 29 4 12

Example 1: Changing Column Order by Name

This example utilizes the most frequently used method: passing an explicit list of column names to index the DataFrame. We aim to rearrange the existing columns so that 'rebounds' appears first, 'assists' second, and 'points' last. This structure is often desired when prioritizing defensive statistics over offensive ones.

By defining the list `` and applying it directly to the DataFrame indexer, we instruct Pandas to fetch and present the data according to this new sequence. Note that no data is changed; only the metadata defining the column presentation order is altered. This simplicity is why this method is favored for reordering static datasets.

The resulting DataFrame confirms that the rows and values remain intact, but the columns are now presented in the required sequence. This immediate visual confirmation is highly useful during exploratory data analysis (EDA) when trying to quickly structure data for comparative views.

#change order of columns by name
df]

rebounds assists points
0 11 5 25
1 8 7 12
2 10 7 15
3 6 9 14
4 6 12 19
5 5 9 23
6 9 9 25
7 12 4 29

Advanced Reordering with New Column Insertion

Often, reordering is required not just among existing columns, but also when introducing a completely new feature. The challenge here is not just calculating the new feature, but placing it precisely within the existing structure. While one could calculate the new column and then use the list reassignment method shown above, Pandas provides the robust `df.insert()` method specifically designed for positional column insertion.

The `df.insert()` method is a powerful in-place operation that allows users to specify exactly where a new column should reside. Unlike simple assignment (e.g., `df = values`), which always places the new column at the end, `df.insert()` requires a positional integer index, the column name, and the corresponding values.

Using `df.insert()` is crucial when the order of data presentation is paramount, such as when preparing fixed-width files or when downstream consumers expect specific columns to appear in a rigid sequence. We will now explore how to use this function to insert a new column, 'steals', at both the beginning and the end of our sample DataFrame.

Example 2: Inserting a New Column at the Beginning

To insert a new column at the very start of the DataFrame, we utilize the `df.insert()` method and specify the index position as `0`. In Python and Pandas, the index `0` refers to the absolute first position, making this the simplest way to prepend a column to an existing structure. We first define the data for the new column, 'steals', and then execute the insertion command.

The `df.insert()` function modifies the DataFrame directly (it is an in-place operation and does not return a new object). This means that after execution, the original DataFrame variable (`df`) now reflects the new column order. Since we specified `0`, 'steals' immediately precedes 'points', shifting all original columns one position to the right.

This technique is indispensable when working with identifiers or primary keys that must always occupy the leftmost column position for clarity and convention. It demonstrates the high degree of control Pandas offers over the structural layout of the dataset.

#define new column to add
steals =

#insert new column in first position
df.insert(0, 'steals', steals)

#display dataFrame
df
steals points assists rebounds
0 2 25 5 11
1 3 12 7 8
2 3 15 7 10
3 4 14 9 6
4 3 19 12 6
5 2 23 9 5
6 1 25 9 9
7 2 29 4 12

Example 3: Inserting a New Column at the End

While assigning a new column using standard indexing (`df = values`) naturally places it at the end, using `df.insert()` provides a method that is conceptually aligned with positional insertion, even for the last position. To achieve this, we must dynamically calculate the index of the position immediately following the last existing column.

This calculation is performed using `len(df.columns)`, which returns the total number of columns currently in the DataFrame. If a DataFrame has three columns (indexed 0, 1, 2), its length is 3. Passing the integer 3 to `df.insert()` will place the new column at index 3, which is precisely the last spot.

This dynamic calculation is preferable to hardcoding index values, as it ensures that the column insertion remains correct even if the DataFrame structure changes upstream. It is a robust way to guarantee the new column is appended regardless of the dataset's current width.

#define new column to add
steals =

#insert new column in last position
df.insert(len(df.columns), 'steals', steals)

#display dataFrame
df

points assists rebounds steals
0 25 5 11 2
1 12 7 8 3
2 15 7 10 3
3 14 9 6 4
4 19 12 6 3
5 23 9 5 2
6 25 9 9 1
7 29 4 12 2

Summary of Column Reordering Strategies

Effective column reordering is a foundational step in data preparation. For simple rearrangements involving only existing columns, the list indexing method (`df]`) offers the highest level of readability and efficiency. This method is concise and is suitable for most restructuring tasks where no new data is introduced.

When the task requires introducing new data elements and placing them at specific positions, the `df.insert()` method provides the necessary positional control. By leveraging integer indices, developers can precisely control whether the new column appears at the beginning, middle, or end of the existing column set. This method is particularly valuable when adherence to strict output schemas is required.

By mastering both the list indexing approach for existing columns and the positional insertion capabilities of `df.insert()`, data analysts can ensure their Pandas DataFrames are perfectly structured for any subsequent analysis, modeling, or reporting tasks.

How to Combine Two Columns in Pandas