Learning Pandas: How to Add a Column from One DataFrame to Another

Authored by

Mohammed looti

October 29, 2025

Introduction: Essential Data Integration with Pandas

In the fast-paced realm of **data analysis** and transformation, the **Pandas** library within **Python** stands out as an indispensable tool. Its core structure, the **DataFrame**, provides a flexible, two-dimensional, tabular format that simplifies complex data operations immensely. A frequent and critical requirement for data professionals is the integration of disparate information sources, which often necessitates the addition of a **column** from one DataFrame into another. This seemingly simple operation is foundational for enriching datasets, combining related metrics, and ensuring data readiness for advanced analytics or sophisticated **machine learning** pipelines.

The ability to efficiently consolidate data elements is paramount across numerous applications. Whether you are merging customer demographics with transactional histories, consolidating results from multiple experiments, or augmenting sensor readings with contextual metadata, successful data integration relies on robust tools. Pandas provides powerful and versatile methods specifically designed to handle these integration challenges effectively, ensuring data manipulation remains both flexible and reliable.

This article serves as a comprehensive guide to mastering two primary techniques for moving a column between existing Pandas DataFrames. We will provide clear, practical explanations and walk through real-world examples to demonstrate how these methods function. Proficiency in these techniques is crucial for anyone engaging in tabular data wrangling in Python, as they form the fundamental building blocks for all more intricate data restructuring tasks. We will specifically cover how to append a column to the end of a DataFrame and how to insert a column at a precise, custom position, granting you optimal control over your resulting data structure.

Understanding the Two Primary Column Addition Strategies

When the need arises to transfer a column (represented internally as a **Series**) from a source DataFrame to a target DataFrame, developers typically employ one of two core strategies. These methods offer flexibility depending on the exact requirements of the project, particularly whether the precise placement of the new column is mandatory or if simply extending the existing structure suffices. Gaining a clear understanding of the mechanical differences between these approaches enables more efficient and intentional data manipulation workflows.

The first and most commonly used method involves straightforward direct assignment, which naturally results in the new column being appended to the rightmost position of the target DataFrame. This method is often the quickest and most intuitive way to expand a dataset. The second, more controlled strategy utilizes the dedicated `.insert()` function. This function grants granular control by allowing the user to specify the exact zero-based integer index where the column should reside.

Both direct assignment and the `.insert()` method rely critically on the alignment of the **index** between the source Series and the target DataFrame. Pandas automatically handles index alignment during these operations. If the indices match, the data is mapped perfectly row-by-row. However, if the indices do not align, Pandas will intelligently align them, potentially introducing `NaN` (Not a Number) values for rows in the target DataFrame that do not have a corresponding index entry in the source Series. Careful index management is therefore key to successful data integration.

Method 1: Appending a Column via Direct Assignment

One of the most accessible ways to incorporate a new data Series into an existing DataFrame is by appending it as a new **column** to the far-right side. This technique is highly recommended when the sequential order of columns is not a strict requirement, or when the objective is simply to expand the dataset with supplementary features that naturally extend its current structure.

This approach capitalizes on the direct assignment capabilities inherent to Pandas DataFrames, treating the structure much like an advanced **Python dictionary** where column names function as unique keys. When you assign a Series object to a new column name using standard bracket notation (e.g., `df = series_data`), Pandas automatically recognizes this as an instruction to create and append the new column to the end of the DataFrame.

The syntax for this operation is concise, intuitive, and highly efficient, making it the preferred choice for rapid data enrichment tasks. It requires minimal code and is easy to read, mirroring the standard mechanism used to add a new key-value pair to any Python dictionary structure.

 # This code snippet demonstrates how to add 'some_col' from df2 to df1, placing it at the last column position.
df1= df2

Method 2: Inserting a Column at a Precise Position using .insert()

While direct assignment offers speed and simplicity, there are numerous analytical and reporting scenarios where the precise placement of a new **column** is absolutely critical. This may be necessary for logical grouping of related variables, adhering to specific output formats, or complying with pre-defined data schemas. For these scenarios, **Pandas** provides the sophisticated **`.insert()`** method.

The `.insert()` method empowers developers to specify the exact zero-based integer position where the new column should be placed within the target **DataFrame**. It accepts three mandatory arguments: the insertion location index (`loc`), the name of the new column (`column`), and the data

itself (`value`). Crucially, the `loc` parameter dictates the index of the column *before which* the new column will be inserted. For example, setting `loc` to `2` ensures the new column occupies the third overall position (since Python utilizes 0-based indexing).

Using `.insert()` guarantees that your DataFrame maintains the exact column sequence required, which is often vital for downstream tasks, regulatory compliance, or seamless integration with other software tools that rely on a fixed schema. An important distinction is that `.insert()` modifies the target DataFrame in place, meaning it alters the existing object directly rather than returning a new DataFrame copy. As with direct assignment, ensuring the index of the source data aligns correctly with the DataFrame's index is paramount for accurate data mapping and successful integration.

 # This code demonstrates how to insert 'some_col' from df2 into the third column position (index 2) of df1.
df1.insert(2, 'some_col', df2)

Data Preparation: Setting the Stage for Practical Examples

To effectively illustrate the operational differences between these two methods, we must first initialize two representative sample **Pandas DataFrames**. This setup simulates a common real-world scenario where associated data is stored across separate tables that need to be harmonized. Our first DataFrame, named `df1`, will contain core information about sports teams, including their designated position and accumulated points. Our second DataFrame, `df2`, will hold supplementary statistics, specifically rebound data, corresponding to the same teams.

These simple but representative datasets provide a clear and verifiable context for understanding how columns can be either seamlessly appended or strategically inserted. This hands-on approach directly mirrors the challenges of integrating related data attributes in real-world data science projects. We must ensure that the index alignment is preserved, which is achieved here by using default sequential indices for both DataFrames.

The following **Python** code snippet initializes both DataFrames. Note that the essential first step is importing the **Pandas** library, typically aliased as `pd`, which grants access to the DataFrame structure and its manipulative methods. The output of the print commands clearly displays the initial structure of both datasets before any column modification is performed.

import pandas as pd

# Create the first DataFrame, 'df1', containing team, position, and points data.
df1 = pd.DataFrame({'team': ,
'position': ,

'points': })

# Display the initial structure of df1 to the console.
print(df1)

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

# Create the second DataFrame, 'df2', containing team and rebounds data.
df2 = pd.DataFrame({'team': ,
'rebounds': })

# Display the initial structure of df2 to the console.
print(df2)

team rebounds
0 A 12
1 A 7
2 A 8
3 A 8
4 B 5
5 B 11

Practical Demonstration: Executing Column Transfers

With both our sample DataFrames, `df1` and `df2`, successfully prepared, we can now move on to the practical demonstration of both column addition techniques. These examples are designed to concretely illustrate how each approach alters the structure of the target DataFrame, clearly showing the precise differences between utilizing direct assignment and the controlled placement afforded by the `.insert()` method. Analyzing the resulting DataFrame structure in each case is essential for fully grasping the advantages and implications of each technique.

Example 1: Appending 'rebounds' Using Direct Assignment

In this first practical application, we employ Method 1 (Direct Assignment) to transfer the **'rebounds'** **column** from the source `df2` DataFrame to `df1`. The specific objective here is to

append this new column to the absolute end of `df1`, effectively expanding its informational breadth without disrupting the existing order of 'team', 'position', and 'points'. This procedure is highly typical when simply adding new attributes or metrics to an established primary dataset.

The elegance of this method lies in its simplicity and reliance on Pandas' automatic index handling. By executing the direct assignment `df1 = df2`, **Pandas** intelligently aligns the rebound values based on the DataFrame **index**. Because both DataFrames were initialized with identical default integer indices, the alignment is seamless, ensuring that every rebound count correctly corresponds to its respective row in `df1`. This demonstrates the ease with which DataFrames can be augmented when indices are compatible.

Inspect the resulting DataFrame carefully after this operation. The **'rebounds'** column will appear as the fourth and rightmost column, having been smoothly integrated alongside the existing data. This perfectly illustrates the straightforward efficiency of using direct assignment for non-position-critical column additions.

 # Add the 'rebounds' column from df2 to df1, placing it at the last column position.
df1= df2

# Display the updated DataFrame to see the newly added column.
print(df1)

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

As confirmed by the output, the **'rebounds'** column has been successfully transferred from `df2` and appended to the last position of `df1`, concluding the data integration task using the direct assignment technique.

Example 2: Inserting 'rebounds' at Index 2 using .insert()

For our second demonstration, we utilize Method 2, leveraging the **`.insert()`** method to place the **'rebounds'** **column** from `df2` into a highly specific location within `df1`. Our aim is to insert it at the third column position, which corresponds to the index `2` within the standard zero-based indexing system of **Python**. This ability to meticulously control column order is indispensable for maintaining data integrity and clarity.

The command `df1.insert(2, 'rebounds', df2)` precisely instructs **Pandas** on the placement. The `2` signifies that the new column, named **'rebounds'**, must be inserted *before* the column currently occupying index 2. Since our original `df1` columns were 'team' (index 0), 'position' (index 1), and 'points' (index 2), the **'rebounds'** column is consequently positioned between 'position' and 'points'. This capability for meticulous, index-based placement is what fundamentally differentiates `.insert()` from simple direct assignment.

The resulting DataFrame clearly verifies that the **'rebounds'** column has been seamlessly integrated into `df1` at the designated index 2. This demonstration highlights the power of `.insert()` to enforce a precise column sequence, ensuring that your DataFrames are structured exactly as required for logical coherence, specific reporting standards, or compatibility with external systems where column sequence is a critical schema element.

 # Insert the 'rebounds' column from df2 into df1 at the third column position (index 2).
df1.insert(2, 'rebounds', df2)

# Display the updated DataFrame to verify the new column's position.
print(df1)

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

The output confirms that the **'rebounds'** column has been accurately added to the third column position (index 2) of `df1`, showcasing the precision and control inherent in the `.insert()` method.

Conclusion: Strategic Data Integration and Next Steps

We have thoroughly examined two highly effective and distinct methods for transferring a **column** from a source DataFrame to a target DataFrame in Pandas: using direct assignment for simple appending and employing the **`.insert()`** method for precise, index-based placement. Both techniques are foundational elements of efficient **data manipulation** when working with tabular data in **Python**, offering vital flexibility based on specific organizational or analytical needs.

The choice between these two methods should be strategic, driven by your data structure requirements and overarching analytical goals. Direct assignment is the ideal, concise method for

rapid additions where the final column order is secondary. Conversely, `.insert()` is indispensable when maintaining a fixed, predefined column schema is necessary--whether for improving human readability, ensuring compatibility with downstream systems, or meeting strict reporting requirements.

It is important to recognize that while these methods handle column addition, Pandas offers even more powerful relational tools for combining DataFrames based on common keys, such as **`.merge()`** and **`.join()`**. These functions operate like SQL joins and are essential for complex data consolidation where row-level alignment is based on shared identifier columns rather than just the sequential index. Mastering the fundamentals of column addition, however, remains a crucial prerequisite for advancing to these more sophisticated data wrangling operations.

Additional Resources for Pandas Proficiency

To further refine your expertise in Pandas and explore the more advanced techniques available for robust **data manipulation**, we recommend engaging with the following high-quality documentation and tutorials:

Pandas User Guide: Merge, join, and concatenate
Pandas Cheat Sheet for Data Science
Real Python: Merging, Joining, and Concatenating Data with Pandas