Learning How to Add Rows to a Pandas DataFrame in Python

Authored by

Mohammed looti

November 4, 2025

Working with the Pandas library in Python is an essential skill for modern data manipulation and analytical tasks. A frequent requirement during data cleaning, preparation, or aggregation is the need to incorporate new observations, which typically involves inserting one or more rows into an existing DataFrame. While the concept of adding a row seems straightforward, selecting the most effective method depends significantly on the scale of the operation--whether you are adding a single, isolated record or combining multiple large datasets--and whether your priority is code simplicity or execution speed.

This comprehensive guide details the most effective and idiomatic techniques for row insertion in Pandas. We will begin by exploring the methods suitable for single-row additions, transition into legacy techniques for bulk merging, and conclude with the modern, high-performance best practices, focusing heavily on the recommended utility, pd.concat().

Technique 1: Adding a Single Record with the .loc indexer

When the task involves adding a singular, isolated row of data rather than stacking an entire block of rows from another source, the preferred and most established methodology is utilizing the .loc indexer. This method leverages Pandas' powerful label-based indexing capabilities, allowing for direct assignment of values to a specific index position.

To successfully append a single row to the very end of a DataFrame, we must calculate the exact index label for the new row. Since DataFrames typically use zero-based indexing, the index position immediately following the last existing row is equivalent to the current length of the index. Therefore, we use df.loc to target the next available slot for insertion.

This generalized syntax provides a highly effective and readable approach for non-iterative insertions, ensuring the new record is placed correctly without disrupting the existing data structure. It is paramount that the list of values being assigned to the new row precisely matches the number and order of columns in the target DataFrame; any structural mismatch will trigger a ValueError.

The concise code pattern for this operation is as follows:

#add row to end of DataFrame
df.loc =

Detailed Example: Implementing Single-Row Addition

Using the df.loc() accessor to modify a DataFrame in place is the recommended technique for adding a singular record. We specifically use len(df.index) as the index label because, in a zero-indexed sequence, the length of the list or index always points to the position immediately

succeeding the last element, thus ensuring the new row is correctly positioned at the bottom of the table.

The example below illustrates this principle by first constructing a sample DataFrame intended to hold player statistics. Following the initial creation, we demonstrate the insertion of a new record for a player with 20 points, 7 rebounds, and 5 assists, ensuring the assigned values correspond accurately to the established column structure (points, rebounds, assists).

import pandas as pd

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

#view DataFrame
df

points rebounds assists
0 10 7 11
1 12 7 8
2 12 8 10
3 14 13 6
4 13 7 6
5 18 4 5

#add new row to end of DataFrame
df.loc =

#view updated DataFrame
df

points rebounds assists
0 10 7 11
1 12 7 8
2 12 8 10
3 14 13 6
4 13 7 6
5 18 4 5
6 20 7 5

As clearly demonstrated by the output, the new row is successfully integrated into the DataFrame at index position 6. This method ensures data integrity and preserves the original structure while efficiently appending the new observation.

Technique 2 (Legacy): Combining Multiple Rows using `.append()`

In scenarios requiring the combination of two complete datasets vertically--stacking one set of rows on top of another--the traditional method used by data scientists was the df.append() function. This function was designed specifically for merging two DataFrames where the goal is to expand the row count while maintaining the existing column structure.

A non-negotiable prerequisite for using .append() is that both DataFrames must possess the exact same column names and corresponding data types. If the column names differ between the source and target DataFrames, the resulting combined DataFrame will introduce missing values (NaN) wherever columns did not align, potentially compromising data quality. Furthermore, because Pandas DataFrames are fundamentally designed to be immutable, the .append() method does not modify the original DataFrame in place; instead, it returns an entirely new DataFrame containing the combined data. Consequently, the result must always be explicitly reassigned back to the original variable.

The key parameter in this vertical merging process is `ignore_index = True`. When appending two DataFrames that might share common index labels (e.g., if both start counting their rows from 0), setting this parameter to `True` instructs Pandas to discard the original, potentially conflicting index values and generate a fresh, sequential index (0, 1, 2, ...) for the final combined dataset. Omitting this parameter would preserve the original indices, leading to non-unique index values in the final structure, which is generally undesirable for clean data management.

#append rows of *df2* to end of existing DataFrame
df = df.append(df2, ignore_index = True)

Detailed Example: Legacy Row Appending

To demonstrate the combination of multiple rows, we first define two separate DataFrames, df and df2, ensuring that their column names and order match precisely. We then invoke the .append() function, utilizing the critical ignore_index = True setting, to stack the rows of df2 onto the end of df, resulting in a single, cohesive, and updated dataset.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'points': ,

```
'rebounds': ,
'assists': })

#view DataFrame
df

points rebounds assists
0 10 7 11
1 12 7 8
2 12 8 10
3 14 13 6
4 13 7 6
5 18 4 5

#define second DataFrame
df2 = pd.DataFrame({'points': ,
'rebounds': ,
'assists': })

#add new row to end of DataFrame
df = df.append(df2, ignore_index = True)

#view updated DataFrame
df

points rebounds assists
0 10 7 11
1 12 7 8
2 12 8 10
3 14 13 6
4 13 7 6
5 18 4 5
6 21 7 11
7 25 7 3
8 26 13 3
```

The resulting DataFrame, df, now contains all rows from the original df followed immediately by the three rows from df2, complete with a newly generated, sequential index spanning from 0 to 8.

Modern Best Practice: Bulk Concatenation with pd.concat()

Although df.append() is currently functional, it has been officially deprecated by the Pandas development team and may be removed in future versions. For all new projects and performance-critical applications, the comprehensive pd.concat() function is the highly recommended replacement for combining multiple DataFrames or Series.

pd.concat() is preferred because it offers greater speed, increased flexibility, and a more unified approach to joining data structures. To replicate the row-stacking behavior of the deprecated .append() method, DataFrames intended for combination must be passed to pd.concat() inside a standard Python list, and the concatenation axis must be explicitly defined.

For vertical concatenation (stacking rows), we set the critical parameter to **`axis=0`**. This signals to Pandas that the operation should proceed along the index (row-wise). As with the legacy method, the inclusion of `ignore_index = True` remains the standard practice to ensure the resulting DataFrame receives a clean, new, and sequential index, preventing potential index conflicts.

 # Using pd.concat() to append df2 to df
df = pd.concat(, ignore_index = True, axis = 0)

By using pd.concat(), developers not only adhere to current best practices but also utilize a function that unifies data joining: whether stacking rows (`axis=0`) or joining columns side-by-side (`axis=1`).

Critical Performance Considerations: Mutability and Efficiency

Understanding the underlying architecture of a DataFrame is vital for achieving efficient data processing. DataFrames are constructed around NumPy arrays, which are highly optimized for fast reading and vectorized calculations but are fundamentally immutable in their structure. This immutability implies a critical performance caveat: whenever you "add" a row using methods like df.loc or the deprecated df.append(), Pandas is forced to allocate memory for and create a completely new copy of the entire DataFrame, moving the old data and then appending the new row. This process is computationally expensive.

For this structural reason, repeatedly adding single rows within a Python loop (e.g., inside a data processing or scraping routine) is extremely inefficient. Each iteration forces a full copy of the dataset, leading to severe performance degradation as the dataset size increases.

To ensure maximum performance when collecting data iteratively, the most performant strategy is to delay all row insertion until the processing loop is fully complete. The recommended workflow for collecting thousands of new records involves three straightforward steps:

Collect all the new data rows (often as dictionaries or lists) into a standard Python list.
After the loop finishes, convert the entire list of new data into a single, cohesive, new DataFrame.
Use pd.concat() just once to combine the original DataFrame with the newly created DataFrame.

This list-and-concat approach drastically minimizes the number of memory allocations and full DataFrame copies required, resulting in a significantly faster operation, particularly when dealing with large volumes of new data.

Summary of Row Insertion Methods

Selecting the appropriate row insertion method is key to writing clean, maintainable, and efficient Pandas code. The following summary outlines the best use cases for each technique:

Single Row Insertion (Recommended): Use the df.loc accessor by assigning a new row of values to the position indicated by len(df.index). This is the ideal method for quick, infrequent, or manual additions of a single record.
Multiple Rows/Bulk Insertion (Modern Standard): Utilize the pd.concat() function. It requires passing a list of DataFrames to be joined and setting `axis=0` for vertical stacking. This is the high-performance choice for combining large datasets.
Legacy Multiple Rows (Deprecated): The df.append() method is still functional but should be systematically replaced by pd.concat() in all current and future development projects.

Regardless of the chosen method--be it single row insertion or bulk concatenation--the consistency of column names and data types between the source and target data structures is absolutely paramount for a successful and error-free append operation.

Additional Resources

To further deepen your expertise in data manipulation within Pandas, we highly recommend exploring the official documentation specifically covering joining and merging data structures. This documentation provides extensive details on advanced concatenation, merging, and complex joining operations.

We particularly encourage consulting the comprehensive documentation for pd.concat(), which offers insights into additional parameters and sophisticated techniques for handling complex data integration scenarios beyond simple row stacking.