How to Easily Add Rows to a Pandas DataFrame

Authored by

stats writer

December 4, 2025

Managing and manipulating data structures is fundamental in modern data analysis, and the Pandas library in Python provides robust tools for this purpose. One common operation is the necessity of adding new rows of data to an existing Pandas DataFrame, whether it is a single observation or an entire batch of records. While this task might seem straightforward, understanding the nuances of how Pandas handles data concatenation and indexing is crucial for writing efficient and maintainable code. Pandas offers several methods to achieve row addition, each suited for different scenarios, ranging from direct assignment using indexers to specialized functions designed for combining large datasets.

Historically, the primary technique involved using the dedicated append function, which efficiently merged the rows of one DataFrame onto the end of another. However, due to performance concerns related to the underlying immutability of DataFrames, this method has been deprecated in favor of more flexible and optimized alternatives. The modern approach heavily relies on the versatile concat function, which handles joining data structures along any axis. Furthermore, for highly specific insertions or single row additions, direct assignment using the powerful `.loc` indexer remains the preferred technique, offering precise control over the placement and content of the new row.

The choice of method--whether leveraging `loc`, `append`, or `concat`--depends heavily on the context: are you adding a single row iteratively, or are you batch-processing thousands of new records? Are you aiming to insert the data at the very end, or do you need to place it at a specific, designated position within the existing structure? Analyzing these requirements allows developers to select the most idiomatic and performant solution provided by the Pandas ecosystem, ensuring data integrity and efficient resource utilization throughout the data cleaning and modeling pipeline.

The Primary Method: Using `.loc` for Single Row Insertion

For adding a single row of values directly to the end of a DataFrame, the most recommended and readable approach involves using the index-based selector, `.loc`. This accessor allows assignment to new labels, and by calculating the length of the existing DataFrame's index, we can determine the exact position for the next available row. Since DataFrames are zero-indexed, the length of the index naturally corresponds to the first position immediately after the last existing row, making it perfect for appending data. This method is particularly useful when processing data iteratively or when appending small amounts of known data, as it is simple to implement and read.

The mechanics of this operation require passing a list or sequence of values that match the existing columns of the DataFrame. When the index is determined using `len(df.index)`, Pandas recognizes this as an assignment operation to a non-existent row and extends the DataFrame structure to accommodate the new data. It is essential that the list of values provided aligns precisely with the data types and order of the columns defined in the original DataFrame, thereby

preventing type coercion errors or misalignment of values.

The following snippet illustrates the powerful simplicity of using the **df.loc()** indexer to seamlessly integrate a new record into the dataset, effectively adding a row to the end of a pandas DataFrame structure. This technique is often favored in scenarios where data is generated or received one record at a time, providing a clear and explicit way to manage growing datasets.

#add row to end of DataFrame
df.loc =

Appending Multiple Rows Using the Legacy `df.append()` Method

Before the shift towards general-purpose combining functions, the **df.append()** function was the conventional way to merge the rows of one DataFrame onto the end of another. This method takes another DataFrame object (or a list of dictionaries or Series objects) as an argument and returns a completely new DataFrame containing the combined data. It is crucial to understand that, like most Data Manipulation Language (DML) operations in Pandas, this is not an in-place modification; rather, it generates a new object, which is why it often requires reassigning the result back to the original variable (e.g., `df = df.append(...)`).

A key requirement for successful appending is ensuring that the columns of the source DataFrame (the one being appended, `df2`) match the column names of the target DataFrame (`df`). If column names do not align, Pandas will attempt to match them based on names and fill any missing values with `NaN` in the resulting merged DataFrame. When combining datasets, developers frequently encounter challenges related to the index. If the original indices are sequential integers, they will likely duplicate after the append operation. To prevent duplicate indices and ensure a continuous, clean index structure in the combined data, the parameter `ignore_index = True` must be set.

Although this function is now officially deprecated in favor of `pd.concat()`, its syntax is concise and it is still encountered frequently in legacy codebases. The following code demonstrates its use for efficiently merging several rows contained within a secondary DataFrame (`df2`) into the primary DataFrame (`df`), making it suitable for batch additions where the new data is already structured as a DataFrame.

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

Modern Approach: Combining DataFrames with `pd.concat()`

The function **pd.concat()** is the definitive and recommended method for combining Pandas objects (DataFrames or Series) along a particular axis. Unlike the older `append` method, which was specialized for row-wise joining, `concat` offers superior flexibility, allowing for horizontal (axis=1) or vertical (axis=0, for rows) merging of data structures. It accepts a list of DataFrame objects to be combined, making it ideal for consolidating numerous data sources simultaneously, which is a common requirement in data aggregation tasks.

When using `pd.concat()` to add rows, setting the `axis=0` parameter explicitly instructs Pandas to perform the concatenation vertically, stacking the DataFrames one atop the other. This function automatically handles alignment based on column names, ensuring that only columns present in both or all input DataFrames are matched up, while non-matching columns are handled based on the specified join type (default is an outer join, preserving all columns). The flexibility of this function extends to index handling; similar to `append`, using `ignore_index=True` is crucial if you require a freshly generated, sequential integer index across the entirety of the resulting combined DataFrame.

While the examples shown below utilize the `.loc` and `.append` methods for clarity based on the original structure, developers writing new Python code should transition to using `pd.concat(, axis=0, ignore_index=True)`. This technique not only future-proofs the code against deprecation warnings but also provides a unified framework for all merging operations, whether adding rows, adding columns, or merging complex hierarchical datasets.

Inserting Rows at Specific Positions

Pandas DataFrames are highly optimized for sequential access, and inserting a row into the middle of a large DataFrame is an inherently costly operation, as it requires rebuilding the entire internal data structure. Therefore, Pandas does not provide a simple, dedicated `.insert()` method for rows (like the one available for columns). Instead, inserting a row at an arbitrary position (e.g., index position 3) is achieved through a combination of slicing and the concatenation function, `pd.concat()`. This approach manually segments the DataFrame, isolates the insertion point, and then reassembles the parts with the new row inserted.

The typical workflow involves three steps: first, slice the original DataFrame into two parts (the segment before the insertion point and the segment after); second, convert the new row data (which might start as a dictionary or list) into a temporary single-row DataFrame; and finally, concatenate the three resulting components in the desired order: . This explicit process, while slightly verbose, clearly communicates the intent and manages the complexity of index alignment that results from the insertion.

This method is generally reserved for situations where the order of rows is semantically critical and cannot be achieved simply by sorting the data after a standard append operation. Due to the performance impact associated with slicing and copying data, this technique should be used judiciously, especially with very large DataFrames. If many insertions are required, it is often far more efficient to collect all the data changes into a staging list or temporary DataFrame and perform a single, large concatenation operation at the end.

Example 1: Add One Row to Pandas DataFrame Using `.loc`

This practical example demonstrates the preferred technique for appending a single new row of data to the existing Pandas DataFrame using the `.loc` indexer. We begin by initializing a DataFrame containing sample basketball statistics--points, rebounds, and assists--for several players. After defining the initial structure, we utilize `len(df.index)` to identify the next sequential index position, ensuring the new row is placed directly after the last record.

The values assigned to this new index position are enclosed in a list, corresponding directly to the column order (points, rebounds, assists). This explicit assignment triggers Pandas to expand the DataFrame, creating a seventh row (index 6) that contains the new player statistics. Observing the updated DataFrame confirms that the new data has been successfully integrated into the structure, maintaining data type consistency across all columns.

This approach is highly recommended for iterative data ingestion because of its clarity and direct manipulation of the data structure. It avoids the overhead associated with creating temporary DataFrames required by concatenation methods when dealing with only one row of data.

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

Example 2: Add Several Rows to Pandas DataFrame Using `df.append()`

This example illustrates how to efficiently add multiple rows of data to a DataFrame when the new data already exists in a structured format, namely another DataFrame. We start with the original DataFrame (`df`) and define a second DataFrame (`df2`) containing three new records, ensuring that the column names in `df2` exactly match those in `df` for successful alignment.

We then invoke the **df.append()** function, passing `df2` as the data to be added. Crucially, we include the parameter `ignore_index = True`. If omitted, the resulting DataFrame would retain the original indices of both `df` (0-5) and `df2` (0-2), resulting in duplicate index labels (0, 1, 2 appearing twice), which is usually undesirable and confusing for subsequent operations. By ignoring the index, Pandas generates a clean, continuous numerical index (0 through 8) for the combined dataset.

The resultant DataFrame successfully integrates the three rows from `df2` (indices 6, 7, and 8) into the original structure. While this function is older, its use case--combining two pre-existing, aligned data structures--is clear, though modern practice dictates substituting this line with `pd.concat(, ignore_index=True)` for better compatibility and performance moving forward.

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

Performance Considerations and Best Practices

A fundamental concept in understanding DataFrames is their general immutability of size. When you add a row using methods like `.loc`, concat function, or the legacy append function, Pandas typically creates an entirely new underlying data array and copies the old data along with the new data into this fresh structure. While this process is fast for single additions or small DataFrames,

repeatedly performing this operation in a loop (e.g., adding 10,000 rows one by one) leads to significant performance degradation because of continuous memory reallocation and data copying.

For optimal performance when handling large volumes of new data, the best practice is to avoid iterative row addition entirely. Instead, data analysts should collect all the new records--perhaps as a list of Python dictionaries or tuples--and then convert this entire collection into a single, new DataFrame. This temporary DataFrame can then be combined with the existing main DataFrame in a single, highly efficient operation using `pd.concat()`. This batch operation minimizes the overhead associated with the constant rebuilding of the DataFrame structure, providing substantial speed improvements.

Another crucial performance tip involves index management. While using `ignore_index=True` is often the most straightforward way to manage indexing when appending, developers must decide whether preserving the existing index structure (which might be based on unique keys or timestamps) is more important than achieving a clean, sequential integer index. Careful planning regarding index strategy, column alignment, and utilizing vectorized operations like `pd.concat()` are key pillars for high-performance data manipulation in the Pandas framework.

Finally, a note on consistency: the two DataFrames used in combination operations (e.g., `df.append(df2)`) should always have the same column names, ideally in the same order, to successfully append the rows of one DataFrame to the end of another without introducing unexpected `NaN` values or requiring complicated data cleaning steps afterward. Consistency ensures predictable and robust data merging.

Summary of Row Addition Techniques

To summarize the methods available for expanding DataFrames in Python using the Pandas library, we can categorize the techniques based on their suitability and modern relevance:

Single Row Addition: Use the `df.loc =` method. This is the cleanest and most direct way to append a single record to the end of a DataFrame, utilizing the powerful loc indexer.

Multiple Row Addition (Recommended): Use `pd.concat(, axis=0, ignore_index=True)`. This method is versatile, highly performant for batch operations, and is the current standard for combining multiple DataFrames or Series objects vertically (by rows).

Multiple Row Addition (Legacy): The `df.append(df2, ignore_index=True)` method, while still functional in many versions, is deprecated and should be replaced by `pd.concat()` in new projects.

Insertion at Specific Positions: Utilize slicing and `pd.concat()` to split the DataFrame at the

desired point, insert the new row (created as a temporary DataFrame), and then rejoin the segments. This is generally the least performant method and should be reserved for cases where row order is critical.

Understanding these distinctions allows developers to write efficient and future-proof code, ensuring that data manipulation tasks scale effectively, whether dealing with tens of rows or millions of records.