Append Two Pandas DataFrames (With Examples)

Authored by

Mohammed looti

November 3, 2025

The task of combining data is a core necessity in nearly every data analysis project. When utilizing the powerful Pandas library within Python, the definitive method for stacking two or more datasets vertically--a process universally known as appending--is achieved through the versatile **`pd.concat()`** function. This function is engineered to combine objects along a specified axis, making it suitable for both row-wise appending (vertical) and column-wise joining (horizontal).

Crucially, appending differs fundamentally from database-style merging, which aligns data based on common key values. Appending, by contrast, simply takes all rows from one Pandas DataFrame and places them sequentially below the rows of another DataFrame. This operation presupposes that the datasets share a compatible schema, meaning they generally possess the same column names and data types. If columns do not align perfectly, the resulting DataFrame will introduce missing values, typically represented as **`NaN`**, where data is absent.

To execute a basic vertical append of two DataFrames, say `df1` and `df2`, into a single cohesive structure, these objects must be passed as a list argument to the concatenation function. A standard and highly recommended practice is using the **`ignore_index=True`** argument. This vital setting ensures that the resulting combined DataFrame receives a clean, new, sequential index starting from zero, effectively preventing the ambiguity caused by duplicate index labels inherited from the original DataFrames.

big_df = pd.concat(, ignore_index=True)

The subsequent sections provide detailed, practical examples that demonstrate how to implement this syntax effectively. We will first examine the simple combination of two DataFrames and then scale the operation to seamlessly integrate multiple DataFrames, emphasizing throughout the critical necessity of proper index management for robust data structures.

Understanding the Core Parameters of pd.concat()

The pd.concat() function is an exceptionally flexible tool within the Pandas ecosystem, designed to handle various aggregation tasks. When deployed for appending, it operates along `axis=0` (the default behavior), which dictates that the operation stacks objects row-wise, effectively combining them vertically. To master data aggregation, it is crucial to understand the principal parameters that govern how **`pd.concat()`** executes its combination logic.

`objs`: This is the mandatory first parameter, which accepts a list or dictionary containing the sequence of Pandas objects (DataFrames or Series) intended for concatenation. The explicit order

in which objects are listed in this sequence determines their final arrangement within the output DataFrame.

`axis`: This parameter specifies the dimension along which the concatenation should occur. Setting `axis=0` (the default) performs vertical stacking (appending rows), while setting `axis=1` results in horizontal stacking (joining columns side-by-side).

`join`: This controls how column alignment is managed when input DataFrames have differing sets of columns. The default setting, `join='outer'`, results in the union of all column names across all inputs, filling any resultant empty cells with the missing data indicator, **`NaN`**. Conversely, `join='inner'` computes the intersection of column names, retaining only those columns universally present in all input DataFrames.

`ignore_index`: A fundamental boolean parameter for managing the final index. Setting this to `True` explicitly discards the original, potentially overlapping index labels and assigns a brand new, clean, contiguous integer index to the combined result, starting reliably from zero.

For straightforward vertical appending tasks where DataFrames share an identical or highly similar structure, the primary focus remains on defining the `objs` list and utilizing **`ignore_index`** for robust index handling. If the source DataFrames contain minor variations in column naming, relying on the default `join='outer'` setting provides maximum data retention, although users should be prepared to handle the introduction of null values.

Selecting the appropriate axis and join method empowers the function to manage highly complex data aggregation requirements. However, for the vast majority of standard vertical appending scenarios, the combination of `axis=0` and `join='outer'` (both defaults) proves to be the most suitable and efficient choice.

Example 1: Vertical Appending of Two Identical DataFrames

This foundational example clearly illustrates the process of combining two Pandas DataFrames, `df1` and `df2`, that share an identical structural layout, containing columns 'x', 'y', and 'z'. By invoking **`pd.concat()`**, we seamlessly integrate all the rows of `df2` directly beneath those of `df1`, thereby producing a single, consolidated dataset named `combined`.

It is imperative to note the strategic inclusion of **`ignore_index=True`** here. In our setup, `df1` implicitly holds index values 0 through 8, and `df2` holds index values 0 through 2. Failure to set `ignore_index` to `True` would result in the `combined` DataFrame containing duplicate index labels

(0, 1, and 2 would appear twice). This duplication generates ambiguity and is a common source of errors during subsequent data slicing, selection, or alignment operations.

The following code block executes the creation and concatenation process, yielding a final DataFrame that successfully contains all 12 rows originating from the two input DataFrames, accompanied by a clean, sequential index ranging from 0 through 11.

import pandas as pd

#create two DataFrames
df1 = pd.DataFrame({'x': ,
'y': ,
'z': })

df2 = pd.DataFrame({'x': ,
'y': ,
'z': })

#append two DataFrames together, resetting the index
combined = pd.concat(, ignore_index=True)

#view final DataFrame
combined

x y z
0 25 5 8
1 14 7 8
2 16 7 10
3 27 5 6
4 20 7 6
5 12 6 9
6 15 9 6
7 14 9 9
8 19 5 7
9 58 14 9
10 60 22 12
11 65 23 19

Example 2: Scaling Appending to Multiple DataFrames

A significant advantage of the pd.concat() function is its inherent scalability and capability to process an arbitrary number of objects simultaneously. When dealing with numerous datasets that require vertical stacking--such as accumulating data from daily sensor readings or monthly financial snapshots--developers can avoid cumbersome sequential appending operations. Instead, all source DataFrames are simply included within the required list argument.

In this demonstration, we instantiate three distinct DataFrames: `df1`, `df2`, and `df3`, all slightly simplified here to contain only 'x' and 'y' columns. The concatenation function processes the entire list efficiently in a single, streamlined call. This design feature ensures that **Pandas** remains highly effective for large-scale data analysis workflows.

Consistent with best practices, we again employ **`ignore_index=True`** to guarantee that the resulting `combined` DataFrame possesses a clean, continuous sequence of index labels running from 0 up to 8. This method is strongly advisable whenever the original index values are non-unique or lack intrinsic meaning (e.g., they are default integers generated during DataFrame initialization).

import pandas as pd

#create three DataFrames
df1 = pd.DataFrame({'x': ,
'y': })

df2 = pd.DataFrame({'x': ,
'y': })

df3 = pd.DataFrame({'x': ,
'y': })

#append all three DataFrames together, resetting the index
combined = pd.concat(, ignore_index=True)

#view final DataFrame
combined

x y
0 25 5
1 14 7

2 16 7
3 58 14
4 60 22
5 65 23
6 58 10
7 61 12
8 77 19

The Critical Importance of Index Management During Appending

As clearly demonstrated throughout the preceding examples, index management represents arguably the single most critical consideration when performing vertical concatenation of DataFrames. If the developer chooses to omit the **`ignore_index=True`** argument, Pandas honors and preserves the original index labels of every component DataFrame. While this default behavior can be appropriate if the index serves as a unique, meaningful identifier (such as date stamps or unique primary keys), it often leads to highly confusing and error-prone results when default, non-unique integer indices are utilized.

When DataFrames are appended without index resetting, the resulting structure inevitably contains redundant index values. For example, if three DataFrames, each indexed sequentially as 0, 1, 2, are combined, the final structure will exhibit the index pattern 0, 1, 2, 0, 1, 2, 0, 1, 2. This lack of uniqueness violates the typical expectation of index labels within a single DataFrame. Consequently, attempting data selection using methods like `.loc` will ambiguously return all three rows that correspond to the label 0, potentially causing logic errors in downstream processing.

The following output illustrates the direct consequence of appending the three DataFrames from Example 2 without explicitly setting the ignore_index parameter. Observe how the index sequence restarts for each block of data originating from the input DataFrames.

#append all three DataFrames together (default index handling)
combined = pd.concat()

#view final DataFrame
combined

x y
0 25 5
1 14 7

2 16 7
0 58 14
1 60 22
2 65 23
0 58 10
1 61 12
2 77 19

If the original index values are genuinely unique identifiers that must be retained but not used as the primary access key after concatenation, an alternative approach is to convert the index into a regular column using `df.reset_index(inplace=True)` prior to combining. However, for the majority of straightforward vertical appending operations, setting **`ignore_index=True`** directly within the pd.concat() function remains the simplest and most robust solution for achieving a clean dataset.

Distinguishing Concatenation from Key-Based Merging

Data professionals must clearly understand the conceptual difference between appending (vertical concatenation along `axis=0`) and merging (joining based on shared key values, typically executed using pd.merge()). Appending is utilized when introducing new observations (rows) that conform to the existing column structure. Merging, conversely, is necessary when incorporating new variables (columns) that must be precisely aligned with existing observations using relational keys.

While the focus of this article has been vertical stacking, **`pd.concat()`** is also capable of performing horizontal joining by explicitly setting `axis=1`. When `axis=1` is applied, DataFrames are placed side-by-side, and the alignment of rows is performed exclusively based on their shared index. If the indices across the input DataFrames do not perfectly match, missing values, represented by NaN, will be introduced according to the specified `join` parameter (inner or outer).

To illustrate, if the goal was to combine `df1` and `df2` based on their index alignment (0, 1, 2) rather than stacking them vertically, the syntax would be: `pd.concat(, axis=1)`. This operation is conceptually analogous to a column join but relies purely on index correspondence rather than requiring explicit key columns, as pd.merge() does.

A firm grasp of these three fundamental data manipulation operations--vertical concatenation (appending rows), horizontal concatenation (index-based joining), and key-based merging--is

absolutely foundational for effective and efficient data wrangling using Pandas DataFrames.

Further Resources for Mastering Data Aggregation

For data scientists seeking comprehensive mastery of data aggregation techniques, particularly those involving advanced features such as concatenating heterogeneous data types or managing complex hierarchical indices, it is recommended to consult the official documentation for the **`pandas.concat()`** function.

The following curated resources provide tutorials that explain how to execute other common functions in Pandas, offering crucial context for determining when appending is the optimal choice compared to alternative methods like merging, joining, or reshaping data:

How to Merge DataFrames (Key-based Joins)
Understanding GroupBy Operations in Pandas
Handling Missing Data (NaN) in DataFrames