How to Easily Append Two Pandas DataFrames

Authored by

stats writer

December 4, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Append Two Pandas DataFrames*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=104867

ARABPSYCHOLOGY.COM

Combining or concatenating datasets is a fundamental operation in data science and Python programming, especially when working with the powerful Pandas library. Appending two Pandas DataFrames--a process often referred to as vertical concatenation--involves stacking rows from one DataFrame onto another to form a single, unified dataset. This is essential for merging data collected at different times or from distinct sources that share an identical columnar structure. While older versions of Pandas supported the dedicated `.append()` method, modern best practices strongly advocate for the use of the robust `pd.concat()` function. Understanding the nuances of `pd.concat()`, including how it handles indices and alignment, is critical for efficient data manipulation.

To successfully append two DataFrames, the process generally requires two prerequisites: the creation of the individual DataFrames themselves, and then the application of the chosen concatenation function. Both the legacy `.append()` method and the current standard, `pd.concat()`, accept a list or dictionary of DataFrames as their primary input argument. They then return a new, consolidated DataFrame containing the combined data. Crucially, `pd.concat()` offers far more flexibility than `.append()`, allowing for not only vertical (row-wise) concatenation but also horizontal (column-wise) merging, depending on the specified `axis` parameter. For standard appending, where we stack rows, we will focus on the default behavior where **axis=0**.

When performing vertical concatenation (appending rows), the primary function utilized is `pd.concat()`. This function is designed to handle sequences of DataFrames, combining them along a specified axis. For simple appending, the syntax is straightforward, requiring only the list of DataFrames to be combined. Below is the basic structure used to combine `df1` and `df2` into a new DataFrame called `big_df`:

big_df = pd.concat(, ignore_index=True)

This example highlights the use of the `ignore_index` argument, which is frequently necessary when appending DataFrames to ensure clean, continuous indexing in the resulting output. The following sections will delve into practical applications and the significance of various arguments within the `pd.concat()` function.

Understanding the Shift from `.append()` to `pd.concat()`

Historically, the Pandas library provided the `.append()` instance method directly on a DataFrame object. While intuitive, this method was fundamentally limited and inefficient, especially when dealing with a large number of DataFrames or complex merging scenarios. It was designed primarily for basic row stacking. Due to performance concerns and to streamline the API, `.append()` was officially deprecated in newer versions of Pandas (starting around version 1.4) and

ARABPSYCHOLOGY.COM

is slated for removal in future releases. Therefore, it is strongly recommended that all new code development utilizes the more powerful and flexible `pd.concat()` function.

The `pd.concat()` function is designed to perform concatenation along an axis. By default, it operates on `axis=0`, which corresponds to the row index, effectively stacking DataFrames vertically (appending). If `axis=1` is specified, it performs column-wise concatenation (joining side-by-side). This versatility makes `pd.concat()` the superior choice for all types of consolidation tasks. Furthermore, `pd.concat()` is optimized to handle a list of objects simultaneously, making it far more performant when combining many DataFrames compared to repeatedly calling `.append()` in a loop, which would result in numerous costly data copies.

When transitioning from using `.append()`, developers should note that the core functionality--stacking rows--is perfectly replicated by `pd.concat()`. The key difference lies in the syntax: `.append()` was called on an existing DataFrame (e.g., `df1.append(df2)`), whereas `pd.concat()` takes a list of DataFrames as its first argument (e.g., `pd.concat()`). Adopting this modern approach ensures code compatibility and leverages the performance improvements built into the latest versions of the Pandas library.

Example 1: Appending Two Pandas DataFrames Vertically

This first example demonstrates the most common use case: combining two DataFrames that possess identical column names and data types. We create two sample DataFrames, `df1` and `df2`, which track three variables (`x`, `y`, and `z`) across different observations. We then use `pd.concat()` to join them row by row.

Notice the inclusion of `ignore_index=True`. As detailed later, this parameter is crucial for generating a new, clean integer index (starting from 0) for the resulting combined DataFrame, preventing duplicate indices that would arise if the original indices were maintained.

import pandas as pd

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

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

#append two DataFrames together

ARABPSYCHOLOGY.COM

```
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
```

The resulting `combined` DataFrame seamlessly integrates the 9 rows from `df1` and the 3 rows from `df2`, resulting in a total of 12 rows. Because we used `ignore_index=True`, the index runs continuously from 0 to 11. This outcome perfectly illustrates the definition of appending: adding new observations (rows) to the end of an existing dataset while maintaining the original set of variables (columns).

Handling Appending with Mismatched Columns and Data Alignment

A common scenario in real-world data handling is attempting to append DataFrames that do not share an identical set of columns. For instance, one DataFrame might contain an extra variable that the others lack. The `pd.concat()` function intelligently manages these discrepancies using its `join` parameter, which defaults to `'outer'`.

When `join='outer'` is used (the default behavior), `pd.concat()` performs a union of all columns present across all input DataFrames. If a row originates from a DataFrame that lacked a particular column, the resulting cell for that observation and column will be filled with a Not a Number (NaN) value. This ensures that no data is lost and that the resulting combined DataFrame contains all variables seen across the inputs. For example, if `df1` has columns A, B, C, and `df2` has columns B, C, D, the combined DataFrame will have columns A, B, C, D.

Conversely, if a strict column match is required, the user can specify `join='inner'`. Using

ARABPSYCHOLOGY.COM

`join='inner'` tells `pd.concat()` to perform an intersection of the column names. The resulting DataFrame will only contain the columns that are common to all input DataFrames. Any column unique to only one or a subset of the DataFrames will be discarded in the output. Choosing between `'outer'` and `'inner'` depends entirely on whether the analysis requires the full set of variables (including NaNs) or only the consistently available variables.

Example 2: Appending More Than Two Pandas DataFrames

A key advantage of using the `pd.concat()` function is its scalability. Unlike the iterative nature often required by the deprecated `.append()` method, `pd.concat()` is explicitly designed to accept an arbitrarily long sequence (list or tuple) of DataFrames. This makes combining three, ten, or even hundreds of DataFrames simultaneously a trivial task.

The following code illustrates how to concatenate three distinct DataFrames (`df1`, `df2`, and `df3`), each containing the same column structure (`x` and `y`), into a single output DataFrame.

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
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

ARABPSYCHOLOGY.COM

7 61 12
8 77 19

The resulting DataFrame contains 9 rows in total, demonstrating the efficient stacking of all three inputs. Crucially, the order of the DataFrames in the input list () strictly dictates the order of the resulting rows in the combined output. This predictability is vital for maintaining data integrity and chronological sequence if the inputs are time-series based.

The Critical Role of Index Management: Using `ignore_index`

One of the most frequent challenges encountered when appending DataFrames is managing the resulting index. If the DataFrames being combined originated from separate files or were created independently, they often share identical index labels (e.g., indices 0, 1, 2, etc.). When these are stacked using `pd.concat()` without modification, the resulting DataFrame will contain redundant, non-unique index values.

While duplicate index labels do not strictly violate the rules of a DataFrame, they can lead to unexpected behavior and errors when attempting to slice, locate, or merge data using functions like `.loc`. To prevent this, the `ignore_index=True` argument is used, which discards the original indices and assigns a fresh, sequential integer index starting from 0 to the newly created, combined DataFrame.

To illustrate the effect of omitting `ignore_index=True`, consider the previous example where we appended `df1`, `df2`, and `df3`. When the argument is excluded, the indices are preserved, resulting in repeated index labels, as shown in the output below:

#append all three DataFrames together
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

ARABPSYCHOLOGY.COM

2 77 19

Note how the index repeatedly cycles through 0, 1, and 2. While this preservation might be desirable if the index represents a unique identifier (like a date or ID) that needs to be maintained, it is generally detrimental for standard integer-based indices when simply stacking data. For most basic appending tasks, setting `ignore_index=True` is the recommended practice for generating a clean, usable index in the output.

Advanced Parameters for Data Concatenation

While the primary use of `pd.concat()` for appending relies on the default `axis=0` and management of `ignore_index`, understanding additional parameters allows for sophisticated data integration techniques. Two particularly important parameters are `keys` and `verify_integrity`.

The `keys` parameter allows you to create a hierarchical index (a MultiIndex) on the resulting combined DataFrame, clearly indicating which input DataFrame each row originated from. If you pass a list of strings to `keys` (one for each input DataFrame), these keys are used as the highest level of the new index. For instance, concatenating `df1` and `df2` with `keys=` would result in an index where rows from `df1` are prefixed by 'SourceA' and rows from `df2` by 'SourceB'. This feature is invaluable for auditing the source of data after consolidation.

The `verify_integrity` parameter is a crucial safety mechanism. If set to `True`, `pd.concat()` will check if the resulting index contains any duplicates. If the original DataFrames had indices that overlap, and `ignore_index=False`, setting `verify_integrity=True` will cause the function to raise a ValueError instead of producing a DataFrame with non-unique indices. This prevents silent errors caused by index duplication and forces the developer to explicitly handle the index structure, usually by setting `ignore_index=True` or using the `keys` argument for MultiIndexing.

Performance and Efficiency in Large-Scale Appending

When working with extremely large datasets--a common scenario in high-volume data processing using Python--performance becomes a significant factor. The architectural design of `pd.concat()` inherently makes it highly efficient compared to iterative appending. When Pandas concatenates objects, it attempts to minimize the number of underlying memory allocations and copies required, especially when the data types and block layouts of the input DataFrames are homogenous.

A key performance tip, especially when combining many DataFrames that reside in a list, is to ensure that the DataFrames are placed in the correct order before the concatenation call. While `pd.concat()` handles ordering naturally based on the input sequence, performance benefits can sometimes be realized if DataFrames with identical data types or structures are grouped together.

ARABPSYCHOLOGY.COM

Furthermore, avoiding repeated concatenation calls (e.g., concatenating `df1` and `df2`, then concatenating the result with `df3`, and so on) is vital. Always combine all DataFrames in a single call: `pd.concat()`.

If the data volume is so massive that loading all DataFrames into memory simultaneously becomes restrictive, alternative solutions such as using Dask or other distributed computing frameworks that support out-of-core processing become necessary. However, for most common use cases involving datasets that fit within available RAM, `pd.concat()` remains the fastest and most idiomatic way to append DataFrames in the Pandas library.

Summary of Best Practices for Pandas Concatenation

Appending DataFrames is a core data preparation step, and mastering the `pd.concat()` function is essential for modern Python data analysis. Key takeaways include prioritizing `pd.concat()` over the deprecated `.append()` method, ensuring that the DataFrames are supplied as a list argument, and diligently managing the index structure.

Always use `pd.concat()` for combining multiple DataFrames efficiently.
Use `ignore_index=True` to create a fresh, sequential index unless the original index labels must be preserved for specific analytical reasons.
Understand the difference between `join='outer'` (union of columns, introduces NaNs) and `join='inner'` (intersection of columns, drops unique columns).
For complex merging scenarios, consider using the `keys` parameter to track the origin of each row.

For a comprehensive understanding of all available parameters and functionalities, you can refer to the official documentation for the `pandas.concat()` function.

You can find the complete online documentation for the **pandas.concat()** function .

Further Exploration in Pandas

The techniques discussed here form the foundation of data consolidation. The following resources explain how to perform other common functions in pandas, moving beyond simple row stacking to more complex merging and joining operations:

ARABPSYCHOLOGY.COM