How to Combine DataFrames in Python Like R’s rbind()

Authored by

stats writer

December 3, 2025

Bridging R and Python: The Equivalent of R's rbind

For data scientists transitioning from R to Python, combining datasets vertically is a fundamental task. In R, this operation is elegantly handled by the built-in function **rbind**, which stands for "row-bind." This function allows users to stack two or more DataFrames (or similar objects) on top of each other, provided they have compatible column structures.

In the Python ecosystem, particularly when dealing with structured data, the library of choice is pandas. pandas provides the powerful and highly flexible function, pandas.concat(), which serves as the direct and enhanced equivalent of R's **rbind**. Understanding how to correctly implement this function is crucial for efficient data manipulation in Python.

The core purpose of concat() is to combine multiple Series or DataFrames along a specified axis. When replicating the behavior of **rbind**, we must ensure the concatenation occurs along the row axis, which is designated by setting the parameter `axis=0`. This operation effectively stacks the data vertically, resulting in a single, consolidated DataFrame containing all the rows from the input objects.

The **rbind** function in R, short for *row-bind*, is used to combine data frames together by their rows.

We use the concat() function from pandas to perform the equivalent function in Python:

df3 = pd.concat()

The following examples detail how to utilize this function in practical data aggregation tasks, addressing common scenarios such as identical structures and unequal column sets.

Understanding Row Binding: The Core Mechanism

Row binding involves appending data objects sequentially, meaning the rows of the second dataset are placed directly underneath the rows of the first dataset, and so on. This process is distinct from merging or joining, which typically align data horizontally based on common keys. When using concat() for row binding, we pass a list of the DataFrames we wish to combine.

The default behavior of pandas.concat() is to perform concatenation along `axis=0` (the index, or row axis), making the syntax surprisingly concise when only two data structures need to be combined. Although the parameter `axis=0` is the default, it is often good practice for clarity in production code to explicitly specify the axis, reinforcing that a vertical stack is intended, mirroring the function of **rbind**.

Here is the simplest implementation demonstrating the concept of row-binding two existing

DataFrames, `df1` and `df2`, into a new object, `df3`:

Example 1: Concatenating DataFrames with Identical Structures

To illustrate the most common use case--combining datasets that share the exact same column names and data types--let us define two sample pandas DataFrames. Both `df1` and `df2` represent team statistics, each containing columns named 'team' and 'points'. This equality in structure ensures a seamless vertical combination.

In this scenario, all data points align perfectly, and the concat() function simply stacks the rows from `df2` directly below `df1`. We first need to import the pandas library and define our initial data structures:

import pandas as pd

#define DataFrames
df1 = pd.DataFrame({'team': ,
'points': })

print(df1)

team points
0 A 99
1 B 91
2 C 104
3 D 88
4 E 108

df2 = pd.DataFrame({'team': ,
'points': })

print(df2)

team points
0 F 91
1 G 88
2 H 85
3 I 87
4 J 95

We can use the concat() function to quickly bind these two DataFrames together by their rows. Notice how the index values are preserved from the original DataFrames, leading to duplicated

index labels (0 through 4 appearing twice).

Executing the Row-Bind Operation

By simply calling concat() and passing our list of data structures, we achieve the desired vertical combination. This results in a new DataFrame, `df3`, which contains all ten rows--five from `df1` and five from `df2`--while maintaining the original two columns.

It is important to observe the resulting index values after this operation. Since pandas preserves the original index from each source DataFrame by default, we will see duplicate index labels (e.g., index `0` appears twice, `1` appears twice, and so on). While this does not harm the data itself, it can cause issues if the index is relied upon for unique identification or slicing later in the analysis pipeline.

Executing the concatenation command yields the following output, clearly showing the duplicated index labels:

#row-bind two DataFrames
df3 = pd.concat()

#view resulting DataFrame
df3

team points
0 A 99
1 B 91
2 C 104
3 D 88
4 E 108
0 F 91
1 G 88
2 H 85
3 I 87
4 J 95

Managing Index Issues: The Role of reset_index()

As observed above, the resulting DataFrame from a standard concatenation retains the original index labels. To create a continuous, unique index spanning the entire newly combined dataset--a best practice in most data cleaning workflows--we must chain the reset_index() method immediately after the concat() operation.

The reset_index() function reassigns a new default integer index starting from 0 up to N-1. By passing the argument `drop=True`, we instruct pandas to discard the old, redundant index values entirely. If `drop` were set to `False` (the default), the old index would be converted into a new column in the DataFrame, which is usually undesirable for simple row-binding.

Using reset_index() in conjunction with concat() yields a much cleaner, sequentially indexed result, which is generally what analysts expect when performing an **rbind** equivalent.

#row-bind two DataFrames and reset index values
df3 = pd.concat().reset_index(drop=True)

#view resulting DataFrame
df3

team points
0 A 99
1 B 91
2 C 104
3 D 88
4 E 108
5 F 91
6 G 88
7 H 85
8 I 87
9 J 95

Example 2: Row-Binding DataFrames with Unequal Columns

A significant advantage of pandas.concat() is its ability to handle structural differences gracefully. When the list of input DataFrames contains columns unique to individual structures, concat() performs an outer join operation by default. This means the resulting DataFrame will include all unique column names present across all input datasets. For cells where a row from one source DataFrame lacks data for a column present in another source DataFrame, the corresponding value in the new combined structure will be filled with NaN (Not a Number).

Consider the following example where `df2` contains an additional column, 'rebounds', which is absent in `df1`. This test case demonstrates how pandas ensures no data loss while managing the resulting heterogeneity:

import pandas as pd

```
#define DataFrames
df1 = pd.DataFrame({'team': ,
'points': })

df2 = pd.DataFrame({'team': ,
'points': ,
'rebounds': })

#row-bind two DataFrames
df3 = pd.concat().reset_index(drop=True)

#view resulting DataFrame
df3

team points rebounds
0 A 99 NaN
1 B 91 NaN
2 C 104 NaN
3 D 88 NaN
4 E 108 NaN
5 F 91 24.0
6 G 88 27.0
7 H 85 27.0
8 I 87 30.0
9 J 95 35.0
```

Handling Missing Values (NaN) and Type Coercion

When the resulting DataFrame contains NaN values, it is a direct consequence of the automatic column alignment during concatenation. The rows originating from `df1` are missing the 'rebounds' metric, so pandas inserts NaN placeholders in those corresponding cells. NaN stands for "Not a Number" and is the standard way Python handles missing or undefined numerical data.

This introduction of NaN often leads to automatic type coercion. If a column originally contained integers (e.g., 'points'), the presence of floating-point NaN values in the combined result will usually force the entire column's data type to shift to floating-point numbers (e.g., 99 becomes 99.0). Analysts must be aware of this coercion, as it may affect subsequent calculations.

If the requirement is to only concatenate columns that are common to all input DataFrames (an inner join behavior), the optional `join` parameter within concat() can be set to `'inner'`. This will discard any column not found in every source DataFrame, avoiding the creation of NaN values due

to unequal structure.

Advanced Parameters: The ignore_index Argument

While chaining reset_index() after concat() is highly effective for generating a clean, sequential index, pandas offers a more direct way to achieve this outcome. The `ignore_index` parameter, when set to `True` within the concat() function itself, forces the resulting DataFrame to generate a new, non-overlapping index automatically, eliminating the need for the subsequent reset_index() call.

For standard vertical stacking tasks, setting `ignore_index=True` simplifies the code and improves readability, effectively condensing two method calls into one. This is often the most idiomatic Python/Pandas way to replicate the behavior of **rbind** where a new sequential index is desired.

 # Achieving row-bind and index reset in a single step
df3 = pd.concat(, ignore_index=True)

#view resulting DataFrame (output is identical to the chained reset_index example)
df3
team points
0 A 99
1 B 91
2 C 104
3 D 88
4 E 108
5 F 91
6 G 88
7 H 85
8 I 87
9 J 95

Summary of Best Practices for Vertical Concatenation

To effectively transition from using R's **rbind** to Python's pandas library, keep the following key practices in mind:

Use pandas.concat(): This is the functional equivalent for vertical stacking of data objects.
Specify Axis (Optional but recommended): While `axis=0` is the default, explicitly setting it ensures clarity that row binding is being performed.
Index Management: Always address the index. Use `ignore_index=True` within `concat()` for a

quick, clean sequential index, or use the `.reset_index(drop=True)` method chain.
Handling Heterogeneity: Recognize that pandas defaults to an outer join, resulting in NaN values for missing entries when columns are unequal. Use `join='inner'` only if you specifically want to discard non-common columns.

Mastering the concat() function is essential for data preparation in Python, providing a flexible and robust tool for combining data sources, whether they are perfectly structured or contain disparate columns.

The following tutorials explain how to perform other common functions in Python: