How to Fix: first argument must be an iterable of pandas objects, you passed an object of type “DataFrame”

Authored by

stats writer

November 25, 2025

Data manipulation is a fundamental task in modern data science, and **Pandas** is the cornerstone library for data handling in **Python**. One of the most common operations is combining multiple datasets, typically accomplished using the `pd.concat()` function. However, users frequently encounter a specific and frustrating error when performing this operation: `TypeError: first argument must be an iterable of pandas objects, you passed an object of type "DataFrame"`.

This guide serves as a detailed resource for understanding, troubleshooting, and permanently resolving this particular **TypeError**. We will delve into the technical reasons behind the error, focusing specifically on the requirements of the `pd.concat()` function concerning its primary input parameter. Understanding the concept of an **iterable** is key to resolving this issue, as the function strictly expects a collection of objects rather than a single object reference.

We will walk through practical examples, first demonstrating how the error is mistakenly generated, and subsequently illustrating the correct syntax necessary to successfully merge two or more **Pandas DataFrames**. By the end of this tutorial, you will possess a robust understanding of proper concatenation techniques, ensuring clean and efficient data workflows in your projects.

Analyzing the Core Error Message

The error message itself provides crucial diagnostic information, pinpointing exactly where the function failed. The message states: `TypeError: first argument must be an iterable of pandas objects, you passed an object of type "DataFrame"`. Let us break down the components of this notification to understand its implications for code execution.

TypeError: first argument must be an iterable of pandas objects, you passed an object of type "DataFrame"

The core problem lies in the distinction between a single object and an **iterable** object. The `pd.concat()` function is designed to take a sequence--such as a **list** or a tuple--containing all the Pandas DataFrame objects intended for combination. When a user mistakenly passes two separate, non-wrapped DataFrames as arguments, Python interprets the first argument as a single DataFrame object, which violates the function's parameter requirements.

For example, if you attempt to use `pd.concat(df1, df2)`, the function sees `df1` as the primary argument it is supposed to iterate over, expecting it to be a list or tuple of DataFrames. Since `df1` is itself a single DataFrame (a non-iterable collection of DataFrames), the function raises the **TypeError**. This strict requirement ensures that `pd.concat()` can handle an arbitrary number of inputs efficiently, not just two, which is why grouping the inputs into an iterable structure is mandatory.

Understanding the Concept of Iterables in Concatenation

In Python, an **iterable** is an object capable of returning its members one at a time. Common examples include lists, tuples, dictionaries, and strings. The `pd.concat()` function relies on this property to know which objects it needs to stack together. When combining data structures like **DataFrames**, Pandas requires that all items to be concatenated are bundled within a single iterable container.

The iterable acts as a containerized queue for the concatenation process. If you provide , the `pd.concat()` function iterates through this list, processing `df1`, then `df2`, and finally `df3`, stacking them along the specified axis (typically the row axis, or axis 0). If you fail to wrap the DataFrames in brackets (creating a list), the function attempts to treat the first DataFrame itself as the sequence to iterate over, leading to the reported error.

This design choice is critical for flexibility. Imagine needing to combine twenty DataFrames; passing them all as separate positional arguments would be cumbersome and prone to error. By requiring a single **list** or tuple, Pandas ensures that the function signature remains clean, accepting one primary argument (the iterable of objects) followed by optional parameters like `axis` and `ignore_index`. Therefore, the simple addition of square brackets transforms the separate DataFrame objects into a single, valid list object that adheres to the function's strict input requirement.

How to Reproduce the Error Step-by-Step

To fully grasp the mechanism of the error, let us first establish a scenario where this **TypeError** is intentionally triggered. We start by defining two separate, simple **Pandas DataFrames**, `df1` and `df2`, which we intend to combine vertically.

First, ensure you have imported the Pandas library, typically aliased as `pd`. The following code block initializes our sample data structures, allowing us to proceed with the flawed concatenation attempt.

import pandas as pd

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

print(df1)

x y z

0 25 5 8
1 14 7 8
2 16 7 10
3 27 5 6
4 20 7 6
5 15 6 9
6 14 9 6

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

print(df2)

x y z
0 58 14 9
1 60 22 12
2 65 23 19

Now, we proceed with the incorrect attempt to combine these two DataFrames using the `pd.concat()` function. The error occurs because we pass `df1` and `df2` as separate positional arguments instead of wrapping them in a single **list** or tuple. Observe the resulting traceback:

#attempt to append two DataFrames together
combined = pd.concat(df1, df2, ignore_index=True)

#view final DataFrame
print(combined)

TypeError: first argument must be an iterable of pandas objects, you passed an object
of type "DataFrame"

As clearly demonstrated by the traceback, the function fails immediately upon execution. Python interprets the first positional argument, `df1`, and recognizes it as a type `DataFrame`, not the expected **iterable** container. This confirms that even if we are only combining two DataFrames, the requirement for an outer container (the list or tuple) is strictly enforced by the `pd.concat()` signature.

The Solution: Wrapping DataFrames in an Iterable

Resolving this **TypeError** is remarkably straightforward once the underlying requirement is understood. The fix involves ensuring that the first argument passed to the **pd.concat()** function is a list of the DataFrames you wish to combine. The square brackets around the DataFrame names create the necessary list object, fulfilling the iterable requirement.

By changing the function call from `pd.concat(df1, df2, ...)` to `pd.concat(, ...)`, we transform two separate positional arguments into a single positional argument: a list containing two elements, both of which are Pandas DataFrame objects. This list is the iterable that the function is explicitly designed to handle, allowing it to successfully loop through and stack the components.

It is good practice to use lists for this purpose, as lists are mutable and clearly denote a sequence of elements. However, tuples `()` would also suffice, as tuples are also considered iterables in Python. Regardless of the container type chosen, the key takeaway is that all objects intended for concatenation must reside within a single, ordered container structure.

Implementing the Correct Concatenation Syntax

Let us now apply the solution to our previous example, ensuring the correct syntax is used. We simply enclose `df1` and `df2` within square brackets:

#append two DataFrames together using the correct syntax
combined = pd.concat(, ignore_index=True)

#view final DataFrame
print(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 15 6 9
6 14 9 6
7 58 14 9
8 60 22 12
9 65 23 19

As evidenced by the successful output, the function executes without raising the **TypeError**. The

resulting `combined` **DataFrame** now contains all the rows from `df1` followed immediately by all the rows from `df2`. We also successfully used the `ignore_index=True` parameter, which ensures that a continuous, clean numerical index is generated for the new, combined data structure, rather than preserving the potentially overlapping indices of the original DataFrames.

This method is highly scalable. If you had fifty DataFrames (e.g., `df_list = `), the exact same function call, `pd.concat(df_list)`, would work seamlessly, provided `df_list` is a single **iterable** containing the DataFrames. This reinforces why understanding the input requirement of the function is far more important than memorizing specific syntax patterns for combining just two DataFrames.

Alternative Methods and Best Practices

While `pd.concat()` is the most robust and flexible tool for joining Pandas objects, it is worth noting that for simple two-DataFrame vertical joining, the older `.append()` method was often used. However, the use of `.append()` has been officially discouraged by the **Pandas** development team, and it is now deprecated in favor of **pd.concat()**.

The deprecation of `.append()` emphasizes the shift towards centralized, iterable-based joining provided by `pd.concat()`. Using `pd.concat()` consistently, even for simple tasks, ensures code longevity and maintains adherence to modern library standards. Furthermore, `pd.concat()` offers superior performance, especially when dealing with many DataFrames, as it optimizes memory allocation during the joining process.

When working with concatenation, always double-check the `axis` parameter. By default, `pd.concat()` operates along `axis=0` (row-wise stacking). If you intended to perform a side-by-side join (combining columns), you must explicitly set `axis=1`. The requirement for the first argument to be an **iterable** remains consistent, regardless of the specified axis.

Summary of Key Takeaways

Successfully avoiding the `TypeError: first argument must be an iterable...` depends entirely on recognizing and satisfying the input requirements of the **pd.concat()** function. This error is not a bug in your data or the library itself, but a syntax error caused by misinterpreting the function's signature.

To summarize the critical points required for seamless DataFrame concatenation:

Always provide the DataFrames you wish to combine as elements within a single, ordered **iterable** container, such as a list or a tuple `()`.
Do not pass DataFrames as separate positional arguments (e.g., avoid `pd.concat(df1, df2)`).

Use `pd.concat(, ...)` for vertical stacking (default `axis=0`) or horizontal joining (when `axis=1`).

By adopting the habit of wrapping your objects in brackets immediately when calling `pd.concat()`, you eliminate this common point of failure and ensure robust data aggregation in your **Python** data manipulation routines. Mastering this basic concept is essential for efficient work with the Pandas library.