Learning Pandas: Adding Rows to an Empty DataFrame

Authored by

Mohammed looti

October 27, 2025

In modern **data analysis**, the ability to dynamically manage and modify **data structures** is paramount. Within the powerful ecosystem of the **Pandas** library for **Python**, a common requirement is populating a **DataFrame** that starts empty. While older methods existed, the preferred, robust, and highly efficient mechanism for adding rows--whether a single record or a large batch--is the **`pd.concat()`** function. This approach ensures adherence to current library best practices and promotes high performance.

This comprehensive guide delves into the specifics of utilizing **`pd.concat()`** to seamlessly inject data into an initialized, empty **DataFrame**. We will meticulously review the underlying principles, examine practical, step-by-step code examples, and discuss crucial performance considerations necessary for effective data manipulation. Understanding this fundamental operation is key to mastering dynamic data handling in **Pandas**.

Before diving into runnable code, it is essential to grasp the fundamental concept: **`concat()`** works by taking a sequence (like a list) of **DataFrame** or Series objects and joining them along a specified axis. When adding rows, we are concatenating along `axis=0` (the default). Therefore, the data we wish to add must first be structured as its own miniature **DataFrame**, which is then combined with the empty target object.

Understanding the Core Syntax of `pd.concat()`

The operation of appending data to an empty target object requires two main steps: defining the new data as a separate, correctly formatted DataFrame, and then passing both the empty container and the new data into the **`pd.concat()`** function as a list. This approach treats the empty DataFrame merely as the starting point for the collection.

The general structure below illustrates how we define the incoming row data using a dictionary embedded within a list, which **Pandas** interprets as a new row or set of rows. This new object is then formally combined with the target variable, `df`, resulting in a new, populated DataFrame.

 # Define the row(s) to add as a new DataFrame
some_row = pd.DataFrame()

# Add the row(s) to the empty DataFrame using concat()
df = pd.concat()

It is important to note that concatenation, by its nature, returns a completely new object rather than modifying the original DataFrame in place. Therefore, the resultant object must be explicitly reassigned back to the variable `df` (i.e., `df = pd.concat(...)`) to reflect the changes in the target variable. This pattern is fundamental to immutable data handling in **Pandas**.

Example 1: Appending a Single Row to an Empty DataFrame

Our initial example demonstrates the simplest use case: injecting a single record into a newly initialized, empty **DataFrame**. We begin by importing the library, creating the empty container, and then defining the structure of the data we intend to introduce. This process clearly illustrates the preparatory steps required before the actual concatenation occurs.

The data for the new row is encapsulated within a list containing a single dictionary, where the dictionary keys automatically map to the column names of the resultant DataFrame. When the empty DataFrame is created without explicit column definitions, **`pd.concat()`** intelligently derives the schema (column names and implied data types) from the first non-empty DataFrame it encounters--in this case, `row_to_append`. This automatic schema inference is highly convenient when starting from scratch.

import pandas as pd

# Create an empty DataFrame named 'df'
df = pd.DataFrame()

# Define the single row to be added, structured as a DataFrame
row_to_append = pd.DataFrame()

# Use pd.concat() to add the new row to the empty DataFrame
df = pd.concat()

# Display the updated DataFrame to verify the addition
print(df)

team points
0 Mavericks 31

Upon execution, the output confirms that the single row has been successfully added. Notice that the index (`0`) is inherited from the `row_to_append` object, which is the standard behavior of `concat()`. This behavior is crucial for index management, a topic we will address in detail later, particularly when dealing with large-scale data additions.

Example 2: Appending Multiple Rows Efficiently

The true power and efficiency of **`pd.concat()`** become evident when dealing with bulk data insertion. Instead of iteratively adding single rows, which is computationally expensive due to repeated memory reallocation, it is far more efficient to define all the desired rows within a single

list of dictionaries and convert that list into a temporary **DataFrame** object.

This batch processing approach minimizes overhead and is the recommended practice for populating a **DataFrame** from an external source or generated data. By structuring the list of dictionaries beforehand, we ensure that the memory allocation for the new data happens once, leading to significant performance gains, especially when processing hundreds or thousands of records.

import pandas as pd

# Initialize an empty DataFrame
df = pd.DataFrame()

# Define multiple rows to be added, again as a DataFrame
rows_to_append = pd.DataFrame()

# Concatenate the empty DataFrame with the DataFrame containing new rows
df = pd.concat()

# Print the resulting DataFrame
print(df)

team points
0 Mavericks 31
1 Hawks 20
2 Hornets 25
3 Jazz 43

The resulting **DataFrame** `df` now contains all four records, indexed sequentially from 0 to 3. Crucially, the process remains identical to the single-row example; the only difference is the complexity of the data object passed into the list provided to the **`concat()`** function. This consistency simplifies coding patterns when dealing with varying volumes of incoming data.

Best Practices and Key Considerations for Data Integrity

While `pd.concat()` is highly versatile, using it effectively requires attention to several details concerning performance and the resulting data structure. Ignoring these considerations can lead to fragmented indices, incorrect data types, or suboptimal runtimes.

One of the most frequent issues encountered during concatenation involves **Data Type Consistency**. When combining two or more DataFrames, Pandas attempts to align the data types. If the incoming data does not match the expected type (e.g., adding string data to an integer

column), Pandas may coerce the column to a broader, less precise type (like converting integers to objects), which can negatively impact subsequent analytical operations. Always ensure that the data types of the columns in the rows you are appending match the intended schema of the target DataFrame.

Another critical factor is **Index Management**. As seen in the examples, `concat()` preserves the indices of the source DataFrames. When concatenating an empty DataFrame with a new one, the new DataFrame's indices are retained (e.g., 0, 1, 2, 3). If you are appending multiple batches of data, you will end up with duplicate index labels (multiple rows labeled '0'). To generate a clean, continuous, and unique index for the resulting DataFrame, the parameter `ignore_index=True` must be utilized. This instructs Pandas to disregard the original indices and assign a new, zero-based, monotonic index to the entire combined structure: `pd.concat(, ignore_index=True)`.

Finally, regarding **Performance for Large Data**, avoid repeatedly calling **`pd.concat()`** inside a loop if you are adding data row-by-row. Each call to `concat()` necessitates creating a brand new DataFrame object and copying all existing data, leading to quadratic time complexity ($O(N^2)$). The best practice is to collect all the incoming rows, usually in a standard **Python** list of dictionaries, and perform a single, final concatenation operation. This list-building approach ensures linear time complexity ($O(N)$), which is vastly superior for production environments dealing with massive datasets.

Alternative and Deprecated Methods

While `pd.concat()` is the modern standard, it is beneficial to understand the historical context and reasons why other methods are now discouraged or deprecated in the **Pandas** ecosystem.

The primary deprecated method is the instance method `DataFrame.append()`. This method, which performed a similar function to concatenation, was officially removed in Pandas version 2.0. The removal was driven by the desire to streamline the API and encourage the use of **`pd.concat()`**, which is fundamentally a more explicit and flexible function for combining data across both rows (axis=0) and columns (axis=1). Developers should migrate any legacy code utilizing `.append()` to the `pd.concat()` paradigm to maintain compatibility and benefit from future performance enhancements.

Another powerful alternative involves creating a list of data objects (dictionaries or lists) and initializing the DataFrame directly from that list, bypassing the need for an empty DataFrame entirely. This is generally the fastest method if all data is available upfront. If you know the column schema in advance, you can initialize the empty **DataFrame** with those columns to enforce data types immediately:

Direct Initialization: Build a complete list of dictionaries and pass it directly: `df =`

`pd.DataFrame(list_of_dictionaries)`.

Schema Enforcement: Initialize the empty DataFrame with predefined columns and data types: `df = pd.DataFrame(columns=, dtype='int32')`.

However, when the data arrives sequentially or dynamically (e.g., in a stream or from iterative database queries), using the empty DataFrame combined with a final **`pd.concat()`** step remains the most reliable and efficient way to assemble the final structure without incurring repeated copying costs.

Conclusion and Further Reading

Populating an empty **DataFrame** using **`pd.concat()`** is a foundational skill in **Pandas**, offering superior performance and API longevity compared to deprecated methods. By ensuring the input rows are correctly structured as a temporary DataFrame and managing the index explicitly, developers can handle dynamic data ingestion tasks with high efficiency and confidence.

For more detailed information, including advanced options for joining and merging, readers are encouraged to consult the official documentation:

Pandas Documentation for `pd.concat()`: A comprehensive resource covering all parameters and edge cases.

Understanding the differences between merging and concatenating DataFrames

How to effectively drop rows from a DataFrame

Renaming columns in **Pandas** for better readability

Converting a dictionary or list of dictionaries into a DataFrame