How to Easily Merge Multiple CSV Files into One with Pandas

Authored by

stats writer

November 21, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Merge Multiple CSV Files into One with Pandas*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=98911

ARABPSYCHOLOGY.COM

Introduction to Data Aggregation in Pandas

In modern data analysis workflows, it is exceedingly common to encounter scenarios where data is segmented across numerous individual source files. Specifically, working with multiple CSV files--a ubiquitous format for storing tabular data--often necessitates a reliable method for combining these segments into a unified structure for subsequent processing. The pandas library, the cornerstone of data manipulation in Python, provides exceptionally robust and intuitive tools to achieve this aggregation efficiently. The primary function employed for this purpose is `pandas.concat()`, which is specifically designed to stack or join DataFrame objects along a specific axis.

This process of data merging is fundamental, particularly when dealing with time-series data segmented by day or month, or when consolidating output logs from distributed systems. Unlike database joins (which often require matching keys), concatenation typically involves appending data rows one after the other, provided the structures are compatible. Understanding the nuances of `pandas.concat()` is critical, as it allows analysts to specify parameters such as the direction of concatenation (row-wise or column-wise), how to handle indices, and whether to force a specific type of join (e.g., outer or inner join) if the columns across the input DataFrame structures are not perfectly aligned. Once the collection of CSV files has been read into memory as a list of DataFrame objects, this powerful function performs the crucial step of knitting them together into a single, cohesive unit, ready for analysis or storage as a new, combined CSV files.

Prerequisites for Merging CSV Files

Before initiating the merging process, several fundamental requirements must be met to ensure the smooth execution of the script and the integrity of the resulting data structure. The primary prerequisite is the installation and successful import of the necessary Python libraries. Specifically, the pandas library is indispensable for handling the data structures and performing the concatenation, while the `glob` module and the `os` module are essential for navigating the file system and identifying the target source files efficiently. These modules are standard components of the Python distribution, simplifying the setup process considerably, but they must be correctly imported at the beginning of the script using standard Python syntax.

Furthermore, careful consideration must be given to the organization and structure of the source CSV files themselves. For a simple row-wise concatenation (stacking files vertically), it is highly recommended that all input files possess an identical column structure, including the column names and their sequential order. While pandas can handle mismatched columns using an outer join (filling missing values with NaN), maintaining consistency ensures a cleaner and more predictable output DataFrame. All the target files must also be accessible via a defined directory path, which needs to be correctly specified within the script, often using the `os.path` module to ensure system-agnostic path handling.

ARABPSYCHOLOGY.COM

Setting Up the Environment and Defining the File Path

The initial steps in the scripting process involve establishing the necessary imported dependencies and precisely locating the directory containing the source data. We rely on three key libraries: `pandas` for data handling, `glob` for pattern-based file searching, and `os` for interacting with the operating system, particularly for defining and manipulating file paths. Defining the path accurately is paramount; an absolute path, as shown in the subsequent code block, guarantees that the script can locate the data regardless of the current working directory. We use the raw string prefix `r` (e.g., `r'C:Users...'`) in Windows environments to prevent backslashes from being interpreted as escape sequences, ensuring the path string is handled literally.

The fundamental setup and path definition syntax required to initiate the merging process is outlined below. This snippet forms the foundation for accessing the files within the specified directory:

```
import pandas as pd
import glob
import os

#define path to CSV files
path = r'C:UsersbobDocumentsmy_data_files'

#identify all CSV files
all_files = glob.glob(os.path.join(path, "*.csv"))

#merge all CSV files into one DataFrame
df = pd.concat((pd.read_csv(f) for f in all_files), ignore_index=True)
```

In this initial block, the path variable is crucial, directing the script to the exact location of the data. For robustness, the corrected path definition within the code block above now explicitly includes the `path` variable inside `os.path.join()` to ensure the wildcard search is correctly performed within that directory, a crucial detail often overlooked in simpler implementations. This approach guarantees that only the files within the designated `my_data_files` folder are targeted for processing, preventing accidental inclusion of irrelevant CSV files elsewhere on the system.

Implementing File Discovery Using the `glob` Module

Once the file path is defined, the next logical step is to systematically identify every target CSV files within that directory. This task is perfectly suited for Python's built-in `glob` module, which is designed to find pathnames matching specified patterns, often using standard Unix shell style wildcards. By utilizing `*.csv` as the pattern, we instruct the module to return a list of every file in

the specified directory that ends with the `.csv` extension, effectively filtering out any non-CSV data or unrelated files that might reside in the same folder.

The combination of **`glob.glob()`** and **`os.path.join()`** ensures that the file discovery process is both platform-independent and precise. The `os.path.join()` function intelligently constructs the full path string using the appropriate separator for the operating system (e.g., `/` for Unix/Linux or for Windows), enhancing the portability of the code. The result of this operation is the variable **`all_files`**, which contains a list of strings, where each string represents the full, valid path to an individual CSV files intended for concatenation. This list serves as the critical input for the subsequent data loading and merging phase.

ARABPSYCHOLOGY.COM

Executing the Merge Operation with `pandas.concat()`

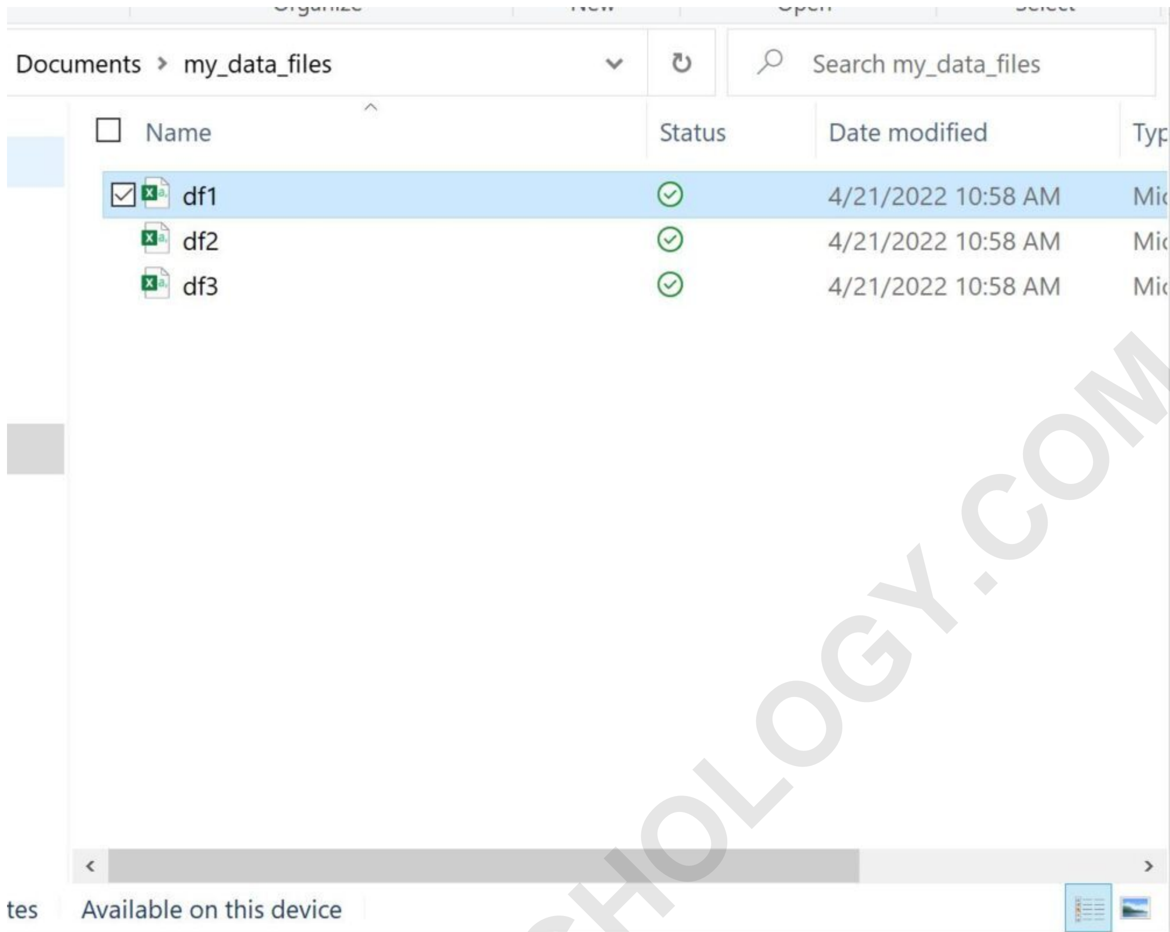
The culmination of the preparation steps is the execution of the merge operation itself, which is handled masterfully by the **`pandas.concat()`** function. This function accepts an iterable (such as the list generated by the `glob` module) containing the individual DataFrame objects that are to be combined. Since the `glob` module provides file paths, we must first iterate through that list and convert each file path into a DataFrame using **`pandas.read_csv(f)`**. This is efficiently achieved using a generator expression within the `concat` function call.

In the common scenario of vertically stacking data, the default behavior of **`pandas.concat()`** (where `axis=0`) is utilized, appending rows from one DataFrame underneath the next. A key parameter often employed in this vertical aggregation is **`ignore_index=True`**. When this parameter is set to `True`, pandas disregards the original index labels from the individual source files and assigns a completely new, sequential index (0, 1, 2, ...) to the final merged DataFrame. This prevents duplicate index labels, which can lead to ambiguity and errors during later data querying and manipulation stages, ensuring the resultant data structure is clean and ready for immediate use.

Practical Example: Combining Sample Data

To illustrate this robust process, consider a scenario where a data analyst needs to combine performance metrics for basketball players that have been logged daily into separate CSV files. Suppose we have a folder named **`my_data_files`** located on the local machine. This folder contains several CSV files, such as `df1.csv`, `df2.csv`, and `df3.csv`, each representing a distinct dataset collected over a short period.

Visually, the structure of the data repository is defined by this single folder housing all the required inputs:



Each of these files maintains a consistent schema, containing exactly two columns: **`points`** and **`assists`**. This consistency is essential for seamless vertical concatenation. For instance, if we examine the content of the first file, `df1.csv`, we can observe the simple, tabular structure that will be stacked upon the data from the other files:

```
1 "points","assists"
2 4,3
3 5,2
4 5,4
5 6,4
6 8,6
7 9,3
8
```

ARABPSYCHOLOGY.COM

By applying the previously developed syntax using the `glob` and pandas libraries, we can efficiently load and merge all data contained within the **`my_data_files`** directory into a single, comprehensive DataFrame.

The complete implementation, including the viewing of the final result, is shown below:

```
import pandas as pd
import glob
import os

#define path to CSV files
path = r'C:UsersbobDocumentsmy_data_files'

#identify all CSV files using the path variable
all_files = glob.glob(os.path.join(path, "*.csv"))

#merge all CSV files into one DataFrame using a generator expression
df = pd.concat((pd.read_csv(f) for f in all_files), ignore_index=True)
```

#view resulting DataFrame
print(df)

points assists
0 4 3
1 5 2
2 5 4
3 6 4
4 8 6
5 9 3
6 2 3
7 10 2
8 14 9
9 15 3
10 6 10
11 8 6
12 9 4

ARABPSYCHOLOGY.COM

Analyzing the Merged DataFrame Output

Upon execution of the script, the resulting DataFrame, named `df`, confirms that the aggregation was successful. The individual row data from all three source CSV files (`df1`, `df2`, and `df3`) have been sequentially appended, forming a unified table. The output clearly shows that the combined data structure retains the two original columns, **`points`** and **`assists`**, but now encompasses a total of 13 rows of data. This total number of rows is the sum of the rows from all input files, demonstrating the row-wise concatenation intended by the default `axis=0` setting of **`pandas.concat()`**.

Crucially, observe the index column on the left side, ranging from 0 to 12. Because we set **`ignore_index=True`** during the concatenation, the original, potentially overlapping or non-sequential indices from the source files were discarded. This result is a clean, contiguous index for the combined data, which is essential for streamlined analysis. If `ignore_index` had been set to `False` (the default), the output would have shown repeated index values (e.g., three rows labeled '0', three rows labeled '1', etc.), making row identification problematic and potentially requiring a separate index reset operation later.

Advanced Considerations and Alternative Approaches

While the demonstrated method is highly effective for files with uniform structures, real-world data often presents complexities that require more sophisticated handling. One common challenge

involves identifying the origin of each row after merging. This can be resolved by using the **`keys`** argument within `pandas.concat()`, which allows you to create a hierarchical index that labels which source file each block of data originated from. Alternatively, one could pre-process the dataframes by adding a new column (e.g., `source_file`) derived from the filename before concatenation, providing a permanent and explicit record of data lineage.

Another important consideration is handling files with slightly different schemas. If the column sets vary across the input files, `pandas.concat()` defaults to an **outer join**, preserving all columns from all input DataFrames and filling in the gaps with `NaN` where data is missing for a specific column in a specific source file. If, however, only the intersection of columns is desired (meaning only columns common to all files are kept), the **`join='inner'`** argument must be explicitly specified within the **`pandas.concat()`** call. These advanced controls ensure flexibility, allowing the analyst to tailor the merging strategy precisely to the requirements of the heterogeneous data sources.

Conclusion and Further Resources

The aggregation of multiple CSV files into a single, manageable DataFrame is a foundational skill in data science, efficiently streamlined by the powerful capabilities of the pandas library. By leveraging the file system navigation tools provided by the `glob` and `os` modules in combination with the high-performance data combination of `pandas.concat()`, analysts can quickly automate tedious manual file merging tasks, freeing up time for deeper analytical work. This demonstrated approach is both scalable and highly readable, forming a reliable template for handling similar data consolidation challenges across various projects.

For those interested in exploring the depth of data loading and manipulation within the pandas ecosystem, the documentation for the primary input function is highly recommended. You can find comprehensive details regarding the various parameters and advanced functionalities of the **`pandas.read_csv()`** function, which governs how each individual file is interpreted and loaded into memory before the final concatenation step. Understanding parameters related to encoding, data types, header handling, and null value interpretation is essential for robust data pipelines.

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

ARABPSYCHOLOGY.COM