Pandas: Merge Columns Sharing Same Name

Authored by

Mohammed looti

April 11, 2026

Introduction to Column Merging in Pandas

In the realm of data manipulation and data cleaning, encountering datasets with duplicate column names is a common challenge. This often arises from integrating data from various sources, erroneous data entry, or specific data collection methodologies. When such situations occur, consolidating these identically named columns into a single, cohesive representation becomes essential for effective analysis and model building. The Pandas library in Python provides robust tools to address this, enabling users to merge columns that share the same name while preserving all relevant information.

This guide will walk you through a practical method for merging columns with identical names within a Pandas DataFrame. We will explore a powerful combination of custom functions and built-in Pandas operations to achieve this consolidation efficiently. The core idea involves grouping columns by their names and then applying a custom aggregation logic to combine the values from each group.

The fundamental syntax for accomplishing this merge involves defining a specific function to handle the concatenation of values and then applying this function across the grouped columns of your DataFrame. This approach ensures flexibility, allowing you to control how the values are combined, such as using a comma or semicolon as a separator.

 # Define function to merge columns with same names together
def same_merge(x): return ','.join(x.astype(str))

# Define new DataFrame that merges columns with same names together
df_new = df.groupby(level=0, axis=1).apply(lambda x: x.apply(same_merge, axis=1))

The following sections will delve into this syntax, explaining each component and demonstrating its application with a practical example.

Understanding the Core Merging Logic

The solution presented above leverages several key Pandas operations to effectively merge columns with identical names. At its heart is the custom function, `same_merge`, which is designed to concatenate non-null values from a given series.

The `same_merge` function: This function takes a Pandas Series (`x`) as input. Inside the function, `x.notnull()` filters the Series to include only non-null values. These non-null values are then converted to string type using `.astype(str)` to ensure they can be joined. Finally, `','.join()` concatenates these string representations, separated by a comma.

The `groupby()` method: This is a powerful feature in Pandas used for splitting data into groups based on some criteria. In our case, `df.groupby(level=0, axis=1)` is crucial.

`level=0` specifies that grouping should occur based on the outermost level of the column index. Since our columns are simple strings (e.g., 'A', 'B'), this effectively groups columns by their names.

`axis=1` indicates that the grouping operation should be performed along the columns (horizontally), rather than rows (vertically).

The `apply()` method with a lambda expression: After grouping, the `apply()` method is used to execute a function on each group. Here, a lambda expression is employed: `lambda x: x.apply(same_merge, axis=1)`.

The outer `apply()` operates on the grouped DataFrame sections (each representing a set of columns with the same name).

The inner `x.apply(same_merge, axis=1)` applies our `same_merge` function row-wise (`axis=1`) to each of these grouped sections. This means for each row, it takes the values from the columns within that group (e.g., all 'A' columns), passes them to `same_merge`, and receives a single concatenated string.

This intricate yet elegant combination allows Pandas to iterate through the DataFrame, identify columns that share names, and then aggregate their row-wise values according to the logic defined in `same_merge`.

Practical Example: Consolidating Duplicate Columns

To illustrate the merging process, let's consider a scenario where you have a Pandas DataFrame containing several columns, some of which possess identical names. For instance, imagine a dataset where 'A' and 'B' represent different categories of measurements, but due to data collection variations, multiple columns exist for each category (e.g., 'A' and 'A1', 'B' and 'B1').

First, we will set up our example DataFrame using Pandas and NumPy to include some missing values (`np.nan`), which are common in real-world data and need proper handling during merging.

import pandas as pd
import numpy as np

# Create DataFrame with initial distinct columns
df = pd.DataFrame({'A': ,
'A1': ,
'B': ,

'B1': })

Rename columns to create duplicate names for demonstration
df.columns =

# View the DataFrame with duplicate column names
print(df)

A A B B
0 5.0 NaN 2.0 5.0
1 6.0 12.0 7.0 NaN
2 8.0 NaN NaN 6.0
3 NaN 10.0 NaN 15.0
4 4.0 NaN 2.0 1.0
5 NaN 6.0 4.0 NaN
6 NaN 4.0 NaN 4.0

As you can observe from the output above, our DataFrame `df` now explicitly features two columns named 'A' and two columns named 'B'. This setup perfectly mimics the kind of data structure where column merging becomes necessary. Our goal is to consolidate these pairs into single 'A' and 'B' columns, with their respective row values combined.

Step-by-Step Implementation and Output Analysis

Now, let's apply the merging logic we discussed earlier to our example DataFrame. We will use the `same_merge` function and the `groupby().apply()` combination to achieve the desired consolidation. The process will merge columns with identical names and concatenate their non-null values, using a comma as the default separator.

 # Define function to merge columns with same names together (re-iterated for clarity)
def same_merge(x): return ','.join(x.astype(str))

# Define new DataFrame that merges columns with same names together
df_new = df.groupby(level=0, axis=1).apply(lambda x: x.apply(same_merge, axis=1))

# View the newly created DataFrame
print(df_new)

A B
0 5.0 2.0,5.0
1 6.0,12.0 7.0

2 8.0 6.0
3 10.0 15.0
4 4.0 2.0,1.0
5 6.0 4.0
6 4.0 4.0

Upon executing the code, we obtain `df_new`, a transformed DataFrame. The output clearly demonstrates that columns originally named 'A' have been successfully merged into a single 'A' column, and similarly for 'B'. For rows where both original columns had values (e.g., row 1 for 'A' with values 6.0 and 12.0), these values are now concatenated as "6.0,12.0". Where only one value existed, it remains as is. This process effectively consolidates the information, providing a cleaner and more manageable data structure.

It's important to note how `notnull()` plays a crucial role here. By filtering out `np.nan` values before concatenation, we prevent the merged string from containing "nan" entries, which would otherwise clutter the data and potentially complicate downstream analysis. This ensures that only meaningful data points contribute to the final merged string.

Customizing the Separator for Merged Values

The flexibility of the `same_merge` function allows for easy customization of the separator used to join the values. While a comma (`,`) is a common choice, you might need a different separator depending on your specific data requirements or subsequent data processing steps. For example, a semicolon (`;`) might be preferred if the data itself could contain commas, preventing ambiguity.

To change the separator, you simply modify the character within the `join()` method inside the `same_merge` function definition. This small change provides significant control over the output format of your merged columns.

Let's demonstrate this by changing the separator from a comma to a semicolon.

 # Define function to merge columns with same names together, using a semicolon as separator
def same_merge(x): return ';'.join(x.astype(str))

# Define new DataFrame that merges columns with same names together
df_new = df.groupby(level=0, axis=1).apply(lambda x: x.apply(same_merge, axis=1))

# View the new DataFrame with semicolon-separated values
print(df_new)

A B
0 5.0 2.0;5.0
1 6.0;12.0 7.0
2 8.0 6.0
3 10.0 15.0
4 4.0 2.0;1.0
5 6.0 4.0
6 4.0 4.0

As expected, the resulting DataFrame now displays the merged values separated by a semicolon, demonstrating the ease with which you can adapt this solution to various output formats. This adaptability is particularly beneficial when preparing data for export to different systems or for specific analytical tools that require distinct delimiters.

Potential Use Cases and Best Practices

Merging columns with duplicate names, as demonstrated, is more than just a technical exercise; it addresses practical challenges in data management. Here are some scenarios where this technique proves invaluable:

Survey Data Consolidation: Imagine a survey where respondents could select multiple options for a question, and each selection was recorded in a separate column (e.g., 'Interest_1', 'Interest_2', which you might rename to 'Interest', 'Interest'). Merging them consolidates all interests into a single field per respondent.

Integrating Data from Disparate Sources: When combining datasets from different systems, column names might accidentally overlap or be intentionally duplicated to represent similar but distinct attributes. This method helps in unifying these attributes.

Handling Inconsistent Data Entry: In cases of manual data entry, the same piece of information might be recorded in slightly different column names over time, or multiple entries for the same attribute might inadvertently create duplicate columns.

Preprocessing for Machine Learning: For many machine learning models, having a clean, consolidated feature set is crucial. Merging redundant columns can simplify feature engineering and improve model performance.

While powerful, it's important to consider some best practices when applying this merging technique:

Data Type Awareness: Ensure that the data types of the columns you are merging are compatible

or can be meaningfully converted to strings. Numeric data will be converted to strings for concatenation, which might impact subsequent numeric operations.

Separator Choice: Select a separator that is unlikely to appear within your actual data values to avoid parsing issues later.

Handling All Nulls: If a row has only null values across all duplicate columns, the `same_merge` function will return an empty string. You might want to post-process these to `None` or `np.nan` if an empty string is not suitable for your downstream analysis.

Column Order: The order of concatenated values might depend on the internal order of columns after grouping. If order is critical, ensure your data or column naming scheme accounts for this.

Conclusion and Further Learning

Effectively managing and cleaning data is a cornerstone of any robust data analysis or data science workflow. The ability to merge columns sharing identical names in Pandas, as detailed in this guide, provides a flexible and powerful solution for consolidating disparate information within your DataFrame. By understanding the interplay of the `groupby()`, `apply()`, and a custom aggregation function, you can transform messy data into a clean, structured format ready for further processing and insights.

This technique is a valuable addition to your data manipulation toolkit, allowing you to handle common data inconsistencies with elegance and precision. The adaptability of the separator also ensures that the output can be tailored to various downstream applications, making this method highly versatile.

For those looking to deepen their expertise in Pandas and data handling, exploring additional resources is always recommended. The official Pandas documentation is an excellent starting point for understanding other advanced functionalities.

Additional Resources

The following tutorials explain how to perform other common operations in Pandas, further enhancing your data processing capabilities:

Link 1: How to Reshape Pandas DataFrame
Link 2: How to Merge Multiple DataFrames in Pandas
Link 3: How to Convert Multiple Columns to One Column in Pandas