Learning to Import Excel Files with Merged Cells into Pandas

Authored by

Mohammed looti

January 31, 2026

Introduction: Navigating Merged Cells When Importing Excel to Pandas

In the realm of data science and processing, it is exceptionally common to encounter data sourced from external formats, particularly legacy spreadsheets like those created in Excel (E: 1). While Excel offers powerful visual tools for organizing and presenting information, certain formatting choices--most notably **merged cells**--can introduce significant complexity when translating that data into a programming environment. The library of choice for data manipulation in Python, pandas (P: 1), relies fundamentally on a clean, rectangular structure, which often conflicts directly with the visual grouping implied by merged cells.

The core challenge lies in the difference between visual presentation and underlying data structure. Excel uses merged cells for aesthetic purposes, allowing a single value (like a group name) to span multiple rows or columns. However, the inherent tabular nature of a pandas DataFrame (DF: 1) demands that every cell contain a distinct, non-ambiguous value. When pandas attempts to import a spreadsheet with vertical merged cells, it correctly identifies that only the top-most cell holds the actual data, treating all subsequent cells in the merged region as effectively empty. This interpretation results in the pervasive appearance of **missing values**, which are standardly represented as NaN (Not a Number) (N: 1).

Although NaN values serve as necessary placeholders for absent data, they are not the desired outcome when the merged cells were intended to carry forward an explicit label or categorical identifier. If left unaddressed, these missing values can severely hinder subsequent data analysis (DA: 1) and aggregation tasks, compromising the overall data integrity (DI: 1). Fortunately, pandas is equipped with robust functionality to mitigate this issue. The most efficient and standard technique involves employing the `pandas.DataFrame.fillna()` (F: 1) function, specifically utilizing the 'forward fill' method. This article will thoroughly explore the mechanics of this conflict and provide a practical, detailed solution to seamlessly integrate Excel data containing merged cells into a usable DataFrame.

Understanding Merged Cells and Their Structural Conflict with DataFrames

Within Excel (E: 2), **merged cells** are fundamentally a formatting attribute. They allow a user to visually combine two or more adjacent cells, often vertically, to create a single, larger display area. This practice is extremely useful for generating clear visual groupings, such as centering a primary header across several columns or listing a category name once at the beginning of a group of related entries. For example, if a spreadsheet tracks sales data, merging the 'Region' cell vertically across all sales transactions for that region prevents redundant repetition and improves human readability.

However, this visual convenience creates a critical ambiguity when the data is programmatically ingested by a library like pandas (P: 2). A pandas DataFrame (DF: 2) is built upon the strict

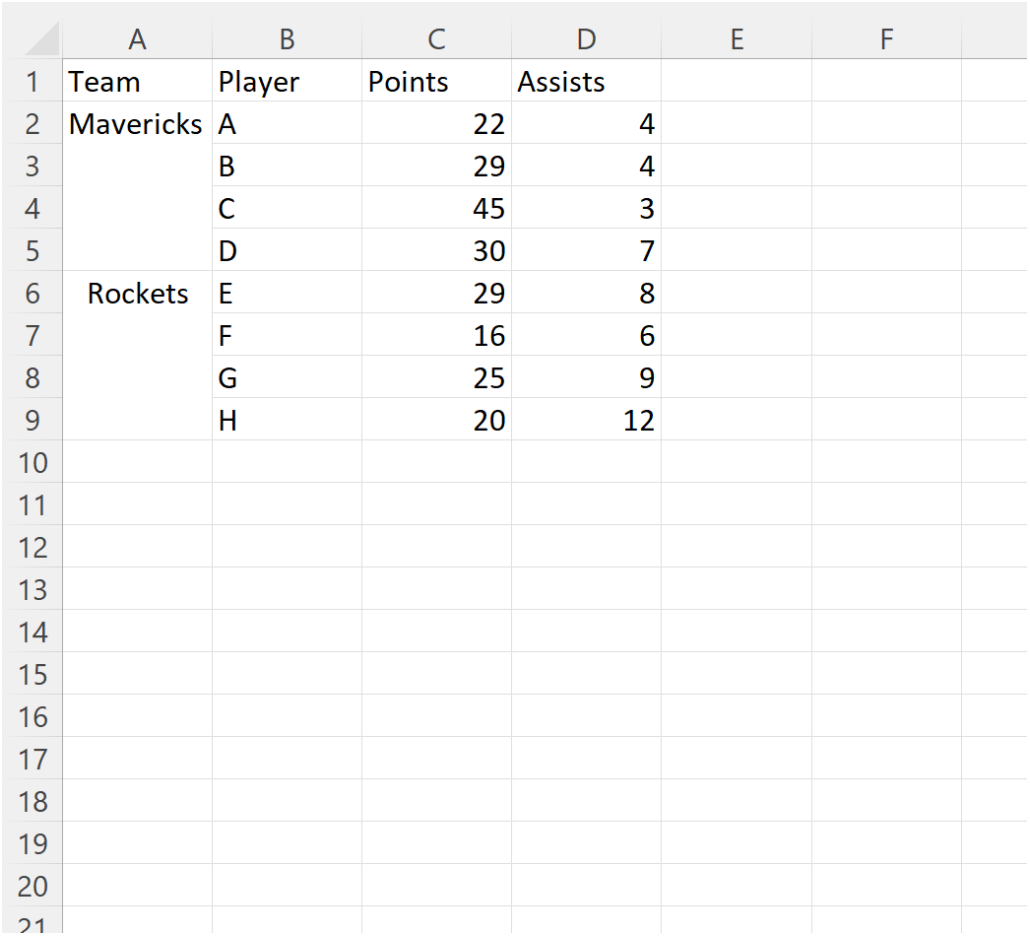
mathematical concept of a matrix, where every row and column intersection must contain an independent, defined value. When pandas processes an Excel file, it reads the underlying XML structure rather than interpreting the visual formatting. In Excel's internal data model, only the top-most or left-most cell of the merged block actually holds the categorical value; the remaining cells in that block are considered empty or null from a data perspective.

As a direct consequence of this structural interpretation, pandas inserts NaN (N: 2) values into the DataFrame wherever an empty cell existed within the merged region. This behavior is crucial for maintaining the rectangular integrity of the DataFrame, but it simultaneously corrupts the intended logical grouping of the dataset. For data scientists, relying on a column containing these unexpected NaN values for tasks like grouping, filtering, or calculating descriptive statistics will inevitably lead to inaccurate results or operational errors. Therefore, the step of 'unmerging' the data--by propagating the initial valid observation downwards--becomes a mandatory precursor to any meaningful data analysis (DA: 2).

Initial Import: Using `pandas.read_excel()` and Observing the Results

The initial and most fundamental step in this process is utilizing pandas' dedicated function for spreadsheet ingestion: `read_excel()` (RE: 1). This highly robust function is designed to handle various Excel formats (such as .xlsx, .xls) and offers comprehensive control over the import process, allowing specifications for sheet names, header rows, and index columns. Despite its versatility, it adheres strictly to the interpretation of the underlying data model, which means it cannot preemptively resolve the issue of merged cells.

To illustrate this challenge, we consider a hypothetical Excel file named **`merged_data.xlsx`**. This file tracks professional basketball statistics, including columns for Team, Player ID, Points, and Assists. Crucially, the **Team** column uses merged cells to visually associate multiple players with a single team name. As shown in the visual representation below, players A through D are grouped under 'Mavericks', and players E through H under 'Rockets'.



	A	B	C	D	E	F
1	Team	Player	Points	Assists		
2	Mavericks	A	22	4		
3		B	29	4		
4		C	45	3		
5		D	30	7		
6	Rockets	E	29	8		
7		F	16	6		
8		G	25	9		
9		H	20	12		
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						

Upon executing the import command using `read_excel()` (RE: 2), the resulting DataFrame confirms the expected structural interpretation. The following code snippet demonstrates the import and the subsequent display of the DataFrame's initial state, clearly revealing the impact of the merged cells on the data representation:

import pandas as pd

#import Excel fie
df = pd.read_excel('merged_data.xlsx'**)**

#view DataFrame
print(df)

Team Player Points Assists
0 Mavericks A 22 4
1 NaN B 29 4
2 NaN C 45 3
3 NaN D 30 7

4 Rockets E 29 8
5 NaN F 16 6
6 NaN G 25 9
7 NaN H 20 12

This output verifies that while the team name appears correctly for the first player in each group (row 0 and row 4), the subsequent rows (1, 2, 3, 5, 6, and 7) within the 'Team' column are populated with `NaN`. This necessitates an immediate data cleaning step. Without resolving these missing values, any operation requiring complete categorical assignments--such as determining the average points scored per team or counting the number of players on each roster--would yield incomplete or erroneous results, underscoring the necessity of value propagation.

The Solution: Propagating Values using `df.fillna(method='ffill')`

The effective solution to rectify the NaN (N: 3) values introduced by Excel (E: 3) **merged cells** lies in the targeted application of the `DataFrame.fillna()` (F: 2) method provided by pandas (P: 3). This method is explicitly designed for missing data imputation and offers several strategies for value replacement. For vertically merged cells, the ideal strategy is **forward filling**, which propagates the last known valid observation downward along the index.

The forward filling mechanism is invoked by setting the `method` parameter to `'ffill'` (forward fill). This instructs pandas to scan the specified axis and, upon encountering an NaN value, replace it with the most recent non-NaN value found immediately prior to it. When dealing with vertically grouped data originating from merged cells, this behavior perfectly replicates the logical intent of the original spreadsheet grouping: the first team name encountered is carried forward to all subsequent rows until a new team name is found.

Equally critical is the selection of the `axis` parameter. Since the merged cells span rows (i.e., they are vertical groupings within a single column), we must ensure that the filling operation proceeds vertically, following the index. We achieve this by setting `axis=0` (or `axis='index'`). This tells pandas to look up and down the DataFrame rows, rather than left and right across the columns, ensuring that the team name from row 0 propagates down to rows 1, 2, and 3, and the team name from row 4 propagates down to rows 5, 6, and 7. The combined effect of `method='ffill'` and `axis=0` restores the categorical integrity of the data structure.

Applying this solution transforms our dataset into a clean, analytical format:

#fill in NaN values with team names
df = df.fillna(method='ffill', axis=0)

#view updated DataFrame

```
print(df)
```

```
Team Player Points Assists
0 Mavericks A 22 4
1 Mavericks B 29 4
2 Mavericks C 45 3
3 Mavericks D 30 7
4 Rockets E 29 8
5 Rockets F 16 6
6 Rockets G 25 9
7 Rockets H 20 12
```

The resulting pandas DataFrame (DF: 3) is now properly denormalized, with every player row correctly assigned to a team. This transformation is vital for subsequent processing, enabling accurate aggregation, filtering, and preparation for advanced statistical modeling without risking errors due to missing categorical values.

Deeper Dive: The Critical Role of the `axis` Parameter

While `method='ffill'` defines the type of value replacement (forward propagation), the `axis` parameter in the `DataFrame.fillna()` (F: 3) method determines the direction of that propagation, making it crucial for correctly interpreting data structures like merged cells. Understanding the difference between `axis=0` and `axis=1` is fundamental for mastering data imputation in pandas (P: 4).

Specifying **`axis=0`**, which refers to the index, dictates a vertical operation. When combined with forward filling, pandas processes the data column by column, moving downwards through the rows. If a cell is NaN (N: 4), the function searches the preceding rows within that same column for the last valid value to use for imputation. This is the required behavior for handling the vast majority of vertically **merged cells** commonly found in Excel (E: 4) spreadsheets, where the grouping label is at the top of a block and needs to be extended down the column.

In contrast, setting **`axis=1`**, which refers to the columns, initiates a horizontal operation. Here, pandas processes the data row by row, moving across the columns. If a cell contains NaN, forward fill looks to the preceding columns within that same row to find the last valid observation. This directional choice would be necessary if the data structure involved horizontal grouping--for instance, if a general category was listed in the first column, and its attributes were spread across subsequent columns where the category name was merged horizontally. However, for our standard merged cell scenario, using `axis=1` would incorrectly propagate values across different attributes (e.g., trying to fill a missing 'Points' value with the 'Player ID' value), resulting in

meaningless data. Therefore, careful consideration of the data orientation and the intended direction of value flow is paramount for successful data integrity (DI: 2).

Conclusion and Best Practices for Data Integrity

Dealing with Excel (E: 5) files containing **merged cells** represents a fundamental step in data preprocessing, but it is a challenge that pandas (P: 5) handles systematically. The import function, `read_excel()` (RE: 3), reliably identifies the empty space resulting from merging and correctly replaces it with NaN (N: 5) values, laying the groundwork for precise imputation.

The cornerstone of the solution is the strategic application of the `DataFrame.fillna()` (F: 4) method, specifically using the combination of `method='ffill'` and `axis=0`. This procedure executes a meticulous **forward fill**, ensuring that categorical values from the top of the merged blocks are accurately propagated downwards. This technique successfully transforms ambiguous or incomplete raw data into a clean, rectangular pandas DataFrame (DF: 4) structure, which is mandatory for reliable statistical analysis, aggregation, and machine learning pipeline readiness.

It is important to remember that while forward filling is the best practice for resolving vertical merged cells, `fillna()` offers flexibility for other scenarios. Analysts can choose `'bfill'` (backward fill) if the valid observation is at the bottom of the block, or use static values (e.g., `df.fillna('N/A')`) or calculated statistics (e.g., mean or median) for numerical imputation. The selection of the imputation method must always be guided by a clear understanding of the data source and the intended meaning of the missing values. By adhering to these best practices, practitioners can confidently process complex spreadsheet data, guaranteeing the highest possible degree of data integrity (DI: 3) throughout their analytical workflow.

Additional Resources for Advanced Pandas Techniques

To further refine your skills in using pandas DataFrame (DF: 5) objects and tackling diverse data preparation challenges, we recommend exploring tutorials and documentation covering the following related topics. These resources will enhance your ability to perform complex data analysis (DA: 3) efficiently:

In-Depth Guide to Data Imputation Techniques and Missing Value Handling in Pandas

Exploring Advanced Parameters of the `DataFrame.fillna()` (F: 5) Method

Mastering Data Reshaping and Pivoting for Hierarchical Datasets