How to Create a Nested DataFrame in Pandas: A Step-by-Step Guide

Authored by

stats writer

November 21, 2025

The concept of a nested DataFrame in Pandas often causes confusion because standard Pandas operations, like `concat()` or `merge()`, typically result in a single, flattened DataFrame structure. However, true nesting involves storing entire DataFrame objects as elements within the cells of a larger, outer DataFrame. This powerful technique allows for the organization of complex, hierarchical data structures where each row or group of the main dataset is associated with its own detailed, internal dataset.

Unlike simple concatenation, which requires compatible column names and data types to stitch rows together, creating a truly nested DataFrame allows us to manage distinct, unrelated datasets under a single umbrella index. This is particularly useful in scenarios involving time series analysis, processing grouped experimental results, or managing metadata linked to complex objects. By treating the individual DataFrames as scalar values within a parent container, we gain flexibility in structuring disparate information.

Understanding Nested Data Structures in Pandas

When working with large or structurally segmented datasets, data scientists often encounter situations where standard tabular representation is insufficient. A nested structure addresses this by creating a column whose elements are not simple values (like integers or strings) but rather complex objects themselves--in this case, other Pandas DataFrame objects. This approach fundamentally shifts the way we organize and interact with our data, providing a framework for multi-level analysis.

It is important to differentiate this manual nesting technique from the methods provided by the `concat()` or `merge()` functions. While those functions are designed for combining data based on shared indexes or columns, they always produce a flat result. The nested DataFrame, conversely, leverages the flexibility of the standard Python object structure within a Pandas Series. Since a Pandas Series can hold any Python object type, including list, dictionaries, or even other DataFrames, we can construct this nested hierarchy easily.

The primary benefit of adopting this architecture is the ability to keep logically segmented data together while maintaining unique indices for each sub-dataset. For instance, if you are monitoring three different product lines (A, B, and C), each with its own sales history and metrics, storing these three separate DataFrames inside a master DataFrame allows you to index the product line name and immediately access the full, complex history associated with it. This structure simplifies iteration and management when dealing with objects that maintain their own internal complexity.

The Core Technique: Creating a DataFrame of DataFrames

The most straightforward and common method for creating a nested Pandas DataFrame involves initializing a new parent Pandas object, using the `pd.DataFrame()` constructor, and passing a

dictionary where one of the columns contains a list of the DataFrames we wish to nest.

This method requires defining two essential components for the parent DataFrame: an index column (which provides context or identification for the nested data) and the data column itself (which holds the actual nested DataFrame objects). The example below demonstrates the fundamental syntax used to achieve this objective:

You can use the following syntax to nest multiple Pandas DataFrames within another DataFrame:

df_all = pd.DataFrame({'idx':, 'dfs':})

This particular example nests three DataFrames (**df1**, **df2**, **df3**) inside a larger DataFrame called **df_all**. Here, `'idx'` serves as a simple index or label for the nested object, while `'dfs'` is the column holding the actual DataFrame objects (df1, df2, df3).

The resulting structure is a two-column DataFrame where the data stored in the `'dfs'` column is complex and non-scalar. This non-standard structure is what defines it as a nested DataFrame. Once nested, accessing and manipulating these sub-DataFrames requires indexing into the parent DataFrame's Series, followed by using positional or label-based accessors.

Accessing Nested DataFrames and Data

To interact with the data contained within the nested DataFrames, we must first select the appropriate cell in the parent DataFrame. Since the column `'dfs'` is a Pandas Series containing Python objects, we can leverage standard Pandas indexing methods. The most reliable method for accessing these objects based on their position is the `iloc` accessor.

The `iloc` property is used for integer-location based indexing. When applied to the column containing the nested DataFrames (e.g., `df_all`), it returns the specific DataFrame object stored at that numerical row index. This is crucial because it retrieves the entire DataFrame object, allowing us to subsequently perform operations directly on that sub-DataFrame.

For example, to retrieve and display the first nested DataFrame in our structure, we select the `'dfs'` column and apply `iloc`, which targets the item at position zero:

#display first nested DataFrame
print(df_all.iloc)

This method ensures that the retrieved object is indeed a Pandas DataFrame, ready for any further analysis, filtering, or manipulation needed. The ability to isolate and operate on sub-components without destroying the main structural integrity of the nested data set is one of the key architectural

advantages of this approach.

Example: Detailed Implementation of Nested DataFrames

To illustrate the process clearly, we will walk through a comprehensive example. We will first define three distinct DataFrames, each representing monthly sales data for a specific group of items. These DataFrames are intentionally separate and do not require joining or merging based on common keys.

Setting Up the Base DataFrames

We begin by importing the Pandas library and creating three individual DataFrames (`df1`, `df2`, and `df3`). Each DataFrame has two columns: `'item'` (categorizing the product) and `'sales'` (the corresponding sales volume). Notice that the items across the three DataFrames are unique, emphasizing that these are distinct datasets we wish to group, not merge.

import pandas as pd

#create first DataFrame
df1 = pd.DataFrame({'item': ,
'sales': })

print(df1)

item sales
0 A 18
1 B 22
2 C 19
3 D 14
4 E 30

#create second DataFrame
df2 = pd.DataFrame({'item': ,
'sales': })

print(df2)

item sales
0 F 10
1 G 12
2 H 13
3 I 13

4 J 19

#create third DataFrame
df3 = pd.DataFrame({'item': ,
'sales': })

print(df3)

item sales
0 K 41
1 L 22
2 M 28
3 N 25
4 O 18

Each of these three DataFrames is now stored in memory as an independent object. The goal is to create one overarching structure, `df_all`, that maintains these three objects organizationally linked by a simple index.

Implementing the Nesting Operation

Now suppose that we would like to create one big DataFrame to hold all three of these DataFrames. We achieve the nesting by passing a list of our DataFrames () as the value for the designated column in the parent Pandas object constructor. We use `'idx'` to provide a positional label for clarity, though any meaningful identifier could be used.

We can use the following syntax to create the nested structure:

df_all = pd.DataFrame({'idx':, 'dfs':})

The resulting `df_all` DataFrame now looks structurally simple, having only two columns and three rows. However, the `'dfs'` column contains deep, complex data objects. When you print `df_all` directly, Pandas will typically show the object type in the cell (e.g., `<DataFrame object at 0x...>`) rather than the contents of the nested table, confirming the successful object storage.

We can then use the Pandas `iloc` function to access specific nested DataFrames. This step is crucial for working with the individual datasets contained within the primary structure.

Retrieving Specific Nested DataFrames

The true utility of the nested structure comes from the ease of retrieval. Since the DataFrames are

stored as elements in a series, retrieving them is as simple as indexing the parent series. For example, we can use the following syntax to access the first nested DataFrame (which corresponds to `df1`, due to its position in the list when `df_all` was created):

```
#display first nested DataFrame
print(df_all.iloc)
```

item sales
0 A 18
1 B 22
2 C 19
3 D 14
4 E 30

The output confirms that the operation successfully extracted the original `df1` DataFrame object, displaying its content, index, and columns exactly as defined initially. This demonstrates the integrity of the nesting process.

Similarly, accessing the second nested DataFrame (corresponding to `df2`) requires only a change in the positional index used with `iloc`:

```
#display second nested DataFrame
print(df_all.iloc)
```

item sales
0 F 10
1 G 12
2 H 13
3 I 13
4 J 19

This organized methodology allows users to manage hundreds or thousands of related, yet structurally independent, data segments efficiently within a single Pandas object, simplifying data storage and workflow orchestration.

Use Cases and Advantages of True Dataframe Nesting

While often overlooked in favor of simpler operations like joining or concatenating, the ability to nest DataFrames provides significant organizational and analytical benefits in specific contexts. Understanding when to use this approach is key to optimizing data handling in Pandas.

One major use case involves handling data from iterative processes or simulations. Imagine running a scientific experiment 50 times, where each run generates a complex results table (a DataFrame). Instead of saving 50 separate files or concatenating them into a massive table that requires complex multi-indexing to retrieve the original structure, nesting allows you to store all 50 results DataFrames in a single parent DataFrame, indexed by the run number or specific experimental parameters. This simplifies querying and aggregation across runs.

Furthermore, nested DataFrames are extremely valuable for managing metadata alongside complex objects. The parent DataFrame can hold simple descriptive statistics (e.g., mean, variance, sample size) in its scalar columns, while the nested column holds the raw, high-resolution data DataFrame used to generate those statistics. This keeps summary and detail data intrinsically linked.

Finally, this structure can facilitate cleaner data processing pipelines. If a specific function needs to be mapped across every sub-dataset--such as applying a standardization transformation or generating a sub-report--it can be applied iteratively to the `'dfs'` column using methods like `.apply()`, where the input to the function is the DataFrame object itself, simplifying the looping logic required.

Considerations for Performance and Serialization

While powerful, nesting DataFrames introduces important considerations regarding performance and data persistence. Since the nested column contains Python objects, it is considered an `object` data type by Pandas. Operations on `object` types are generally slower than optimized, vectorized operations on numerical or categorical data types, as they rely on Python loops rather than highly optimized C routines.

Another crucial factor is serialization (saving and loading the data). Standard serialization formats for Pandas, such as CSV or Excel, cannot handle storing complex Python objects like DataFrames directly. If you attempt to save a nested DataFrame to CSV, the contents of the nested column will likely be represented as unreadable string representations of the object addresses. To persist nested DataFrames, specialized formats must be used:

Pickle: Using `df_all.to_pickle('data.pkl')` is the most reliable way to save and load nested DataFrames, as Pickle is designed to serialize arbitrary Python objects.

HDF5 (PyTables): While more complex, HDF5 can manage complex hierarchical data structures, making it suitable for very large-scale nested data, provided the data types are compatible.

JSON/Parquet (Advanced): These formats might be usable if the nested DataFrames are first converted into dictionary or specialized structure formats, though this often defeats the purpose of

maintaining the DataFrame object structure.

Therefore, users must be mindful that while the nesting technique provides excellent organizational benefits, it slightly deviates from the highly optimized, columnar paradigm that defines the core strengths of the Pandas library for numerical processing. It is a tool best reserved for organizational complexity rather than high-speed calculation.

The following tutorials explain how to perform other common functions in Pandas: