Split a Pandas DataFrame into Multiple DataFrames

Authored by

Mohammed looti

November 3, 2025

In data analysis, particularly when working with large datasets, it is frequently necessary to divide the data into smaller, manageable subsets. This segmentation technique is fundamental for crucial tasks such as creating **training and testing datasets** for machine learning models, isolating data segments for specialized **visualization**, or enabling efficient **batch processing**. The most straightforward and robust technique for splitting a Pandas DataFrame based purely on numerical row positions relies on the powerful iloc indexer.

This method leverages standard Python slicing notation to define the exact boundaries of the split. The following concise syntax demonstrates how to divide a DataFrame into two distinct parts based on a specified row index, effectively creating two new DataFrames:

#split DataFrame into two DataFrames at row 6 (the 7th row)
df1 = df.iloc
df2 = df.iloc

We will now dive into practical, detailed examples demonstrating this fundamental slicing methodology, first by splitting a DataFrame into two primary halves, and subsequently by segmenting it into multiple non-overlapping parts.

Introduction to Positional Splitting with Pandas

The capacity to accurately segment data is indispensable within the modern **data science workflow**. Regardless of whether one is handling sequential **time-series data** that requires ordered analysis or preparing data for rigorous **cross-validation** procedures, precise row-based separation is paramount. Pandas provides specialized tools tailored for this purpose, with the iloc attribute serving as the principal mechanism for positional indexing.

The term iloc is shorthand for "integer-location." This indexer facilitates selection based purely on the absolute numerical position of the rows and columns, beginning the count at zero. Critically, the iloc method ignores any custom index labels (such as unique IDs, strings, or dates) that the DataFrame might possess, focusing solely on the structural order. When performing a split, we define the exact range of indices to extract, which results in the creation of new, entirely independent DataFrames.

A crucial principle to recall when utilizing Python slicing conventions is the concept of the **semi-open interval**: the starting index is always inclusive, while the stopping index is always exclusive. Consequently, if we establish a split point at index 6, the initial resulting DataFrame (`df1`) will encompass rows indexed 0 through 5. The subsequent DataFrame (`df2`) will then commence precisely at index 6, ensuring no row is duplicated or omitted during the segmentation.

The Core Mechanism: Understanding iloc Indexing

The iloc property functions exactly like slicing a standard Python list or array. The general syntax for slicing a DataFrame is structured as `df.iloc`. Since our objective here is row-based division while maintaining all original columns, we only need to specify the row range, allowing the column specification to default to the entire column set.

By defining the strategic split point, we generate non-overlapping slices that collectively cover the entire range of the original data. For instance, to partition a DataFrame, `df`, of size N at index K, we execute two distinct slicing assignments:

The first DataFrame captures the initial segment: `df.iloc`. If K is defined as 6, this slice captures indices 0, 1, 2, 3, 4, and 5.

The second DataFrame captures the remainder: `df.iloc`. If K is 6, this slice begins precisely at index 6 and continues through N-1.

This robust dual assignment procedure is guaranteed to ensure that every single row from the initial Pandas DataFrame is allocated to exactly one of the resulting subsets, thereby preserving complete data integrity throughout the splitting process.

Example 1: Splitting a Pandas DataFrame into Two Segments

This first practical illustration demonstrates the most common scenario: dividing a single Pandas DataFrame into two separate subsets, which may be equal or unequal in size, based on a predefined integer index position. For clear illustration, we will initialize a sample DataFrame consisting of 12 rows.

We begin by importing the requisite Pandas library and applying the slicing logic to achieve the desired two-way split at index position 6:

import pandas as pd

#create DataFrame
df = pd.DataFrame({'x': ,
'y': })

#view DataFrame
df

x y
0 1 5

1 1 7
2 1 7
3 3 9
4 3 12
5 4 9
6 5 9
7 5 4
8 5 3
9 6 3
10 7 1
11 9 10

#split original DataFrame into two DataFrames at index 6
df1 = df.iloc
df2 = df.iloc

#view resulting DataFrames
print(df1)

x y
0 1 5
1 1 7
2 1 7
3 3 9
4 3 12
5 4 9

print(df2)
x y
6 5 9
7 5 4
8 5 3
9 6 3
10 7 1
11 9 10

Careful observation of the output confirms the effectiveness of the split. **df1** successfully isolates the first six rows of the original data (indices 0 through 5, due to the exclusive stop index). Correspondingly, **df2** captures the remaining six rows, commencing precisely at index 6 and extending to the end. Note that the original index labels are maintained in both new DataFrames.

This preservation is often valuable for tracing data lineage back to the source, although these indices can be explicitly reset if required for the subsequent analysis.

Example 2: Segmenting a DataFrame into Multiple Parts

Although splitting data into two segments (e.g., training/testing) is the most frequent use case, the inherent capability of iloc slicing permits segmenting the data into any arbitrary number of subsets. To achieve this, we must define a series of sequential, non-overlapping slices that, when combined, span the entirety of the original DataFrame's indices.

In this specific demonstration, we will partition our 12-row dataset into three distinct DataFrames. This requires establishing two intermediate slicing points. Utilizing our 12-row structure, we choose to split the data first at index 3, and then again at index 6. This careful selection results in the following segmentation logic:

The **First Segment** captures the beginning, rows 0, 1, and 2, defined by the slice `:3`.

The **Second Segment** captures the middle, rows 3, 4, and 5, defined by the slice `3:6`.

The **Third Segment** captures the remainder, rows 6 through the end, defined by the slice `6:`.

The implementation below illustrates how these sequential slices are applied to generate the three subsets:

import pandas as pd

#create DataFrame (same as Example 1)
df = pd.DataFrame({'x': ,
'y': })

#split into three DataFrames
df1 = df.iloc
df2 = df.iloc
df3 = df.iloc

#view resulting DataFrames
print(df1)

x y
0 1 5
1 1 7
2 1 7

print(df2)

x y
3 3 9
4 3 12
5 4 9

print(df3)

x y
6 5 9
7 5 4
8 5 3
9 6 3
10 7 1
11 9 10

This technique underscores the flexibility of sequential array slicing. By precisely defining the start and stop indices for each intermediate segment, data professionals gain granular control over the size, composition, and boundaries of every resulting DataFrame, making this technique highly scalable for N-segment partitioning projects.

Advanced Considerations: Ensuring Data Independence with .copy()

While using `iloc` for splitting is highly efficient and reliable, developers must remain cognizant of certain underlying behaviors within Pandas, particularly regarding the distinction between **"copies"** and **"views."**

By default, when a segment of a DataFrame is sliced, Pandas often attempts to return a memory-efficient "view" of the original data whenever feasible. However, performing subsequent modifications on a view can sometimes lead to the notorious SettingWithCopyWarning. This warning arises because changes made to the new slice might not propagate correctly back to the original source, or, conversely, may unintentionally alter the source data, leading to unpredictable results and data integrity issues.

To guarantee true **independence** between the newly created segments and the source DataFrame, the industry best practice dictates explicitly forcing a deep copy of the slice. This is achieved by appending the `.copy()` method directly after the slicing operation:

 # Ensure df1 is a deep copy and fully independent
df1 = df.iloc.copy()

Utilizing `.copy()` ensures that any subsequent data manipulation, transformation, or modification applied to the new DataFrame will exclusively affect that subset, providing stability and predictability in complex data pipelines and ensuring that your original dataset remains immutable.

Summary of Best Practices for DataFrame Splitting

Splitting a Pandas DataFrame based on precise row position is an essential and foundational skill for effective data manipulation and preparation. The primary tool for this task is the `iloc` indexer, which ensures consistent, position-based slicing irrespective of the DataFrame's custom index labels.

Key principles for executing a successful and robust DataFrame split include:

Always use the `iloc` indexer for position-based, integer row selection.

Adhere strictly to standard Python slicing syntax: the start index is inclusive, but the stop index is exclusive.

Verify that all created slices collectively cover the full extent of the original DataFrame without leaving any gaps or causing overlaps.

Explicitly employ the `.copy()` method when necessary to ensure the resulting DataFrames are fully independent copies of the original data source, mitigating potential warnings and side effects.

Expanding Your Pandas Proficiency

To further refine your skills in data wrangling and manipulation using the powerful Pandas library, we recommend exploring tutorials and documentation covering related data partitioning techniques and functions:

Understanding **Boolean indexing** for filtering rows based on complex conditional logic.

Mastering techniques for merging and joining multiple DataFrames after individual processing is complete.

An in-depth study detailing the functional differences and appropriate use cases for the `iloc` (integer location) and `loc` (label location) indexers.

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