How to Get Last Row in Pandas DataFrame (With Example)

Authored by

stats writer

November 22, 2025

Accessing specific data points within large datasets is a fundamental requirement in data analysis. When working with the Pandas library in Python, the primary data structure utilized is the DataFrame. Often, analysts need to retrieve the most recent or final entry, which corresponds to the last row of the structure. Efficiently extracting this final record is crucial for tasks ranging from real-time monitoring to summarizing aggregated results.

Fortunately, Pandas provides highly optimized methods for positional selection, bypassing the need for manual iteration or complex conditional logic. The most direct and robust technique involves utilizing the .iloc accessor, which stands for integer location indexing. This method allows users to pinpoint data based purely on its numerical position within the rows and columns, regardless of the explicit labels assigned to those indices.

This guide delves into the specifics of using .iloc combined with negative indexing to reliably select the final row of any DataFrame. Furthermore, we will differentiate between the two primary ways this selection can be executed: retrieving the result as a Pandas Series or maintaining it as a single-row DataFrame. Understanding these nuances ensures that subsequent data manipulations are performed using the correct data structure type, thereby preventing common processing errors.

Understanding Positional Indexing in Pandas

Positional indexing is a cornerstone of efficient data manipulation within the Pandas ecosystem. Unlike label-based selection, which relies on explicit row names (like dates or IDs), positional indexing uses zero-based integers to identify elements. This is particularly useful when the row labels are non-unique, default integer ranges, or irrelevant to the immediate task of sequential access. Understanding this concept is critical because the method used to locate the last row inherently relies on the physical order of the data within the DataFrame structure.

The use of negative integers in positional indexing follows standard Python list behavior. When an index of is supplied, it instructs the accessor to start counting from the end of the sequence rather than the beginning. Therefore, always refers to the last element, to the second-to-last, and so forth. This powerful convention allows developers to write concise and predictable code that reliably targets the boundary conditions of the dataset, regardless of its overall size.

It is essential to distinguish between positional indexing methods like .iloc and label-based indexing methods like .loc. While .loc would require knowing the explicit label of the last row (which can be variable), .iloc only requires the relative position. Since we are always interested in the final position, .iloc provides the most efficient and robust solution for this specific task.

The Crucial Role of the .iloc Accessor

The .iloc accessor is the designated tool within Pandas for accessing data based on integer

position. It is designed specifically for this purpose and handles both single element selection and complex slicing operations. When applied to a DataFrame, the first argument refers to the row index, and the optional second argument refers to the column index. In scenarios where only the row index is provided, as is the case when retrieving an entire row, all columns are implicitly selected.

The efficiency of .iloc stems from its direct interaction with the underlying NumPy array representation of the data. By relying on simple integer calculations rather than hash map lookups associated with label indexing, .iloc executes quickly, even when dealing with extremely large datasets. This high performance makes it the preferred method when sequential or positional access is necessary, such as grabbing the first or last record.

When selecting a single row, the output format is determined by whether we use a scalar index (e.g.,) or a slice (e.g.,). This distinction is fundamental because it governs the resulting data structure--either a Pandas Series or a single-row DataFrame, respectively. Choosing the right method depends entirely on what subsequent operations the user intends to perform on the extracted data.

Identifying the Last Row: The Two Key Methods

There are two principal ways to use .iloc to retrieve the last row, dictated by the desired output format. Both methods utilize the negative index , but the syntax differs slightly to signal whether a single element extraction (Series) or a slicing operation (DataFrame) is intended.

The first approach treats the last row as a singular entity, accessing it directly via its position. The output is a Pandas Series, where the column names of the original DataFrame become the indices of the resulting series, and the values are the data points from that specific row. This format is ideal for fast inspection, calculation across that single row's attributes, or direct conversion to a dictionary.

The second, and equally important, approach involves slicing the DataFrame to include only the last row. By defining a slice that starts at the position and extends to the end of the DataFrame, the structure is maintained. The output is a single-row DataFrame, which is essential if the extracted data needs to preserve the original structure, retain column data types, or be immediately concatenated back into another DataFrame or used in functions expecting DataFrame input.

Method 1: Retrieving the Last Row as a Pandas Series

To extract the last row as a Pandas Series, the syntax is exceptionally concise. We pass the integer directly to the .iloc accessor without using slicing notation (the colon symbol). This operation tells Pandas to select the element at that precise index. Since DataFrames are two-

dimensional, selecting a row reduces the dimensionality, resulting in a one-dimensional Series object.

The primary benefit of using the Series output is its simplicity for quick data access and attribute extraction. If, for instance, an analyst only needs to retrieve the value of a single column for the last entry, doing so from a Series is straightforward: **last_row**. This avoids the need for double-bracket indexing or redundant steps required when working with a single-row DataFrame.

Here is the precise implementation for generating a Pandas Series from the final row of a DataFrame named **df**:

last_row = df.iloc

This approach is highly memory efficient and computationally light, making it suitable for iterative processing where the overhead of creating numerous temporary DataFrame objects would be detrimental to performance. It is the go-to solution when the structural context of the original data is no longer necessary for the operation at hand.

Method 2: Retrieving the Last Row as a Single-Row DataFrame

In contrast, if maintaining the two-dimensional structure and column headers is necessary, we must use slicing notation within .iloc. By specifying a slice that begins at the negative index , we instruct Pandas to return a subset of the original DataFrame. This guarantees that the result retains the essential attributes of a DataFrame, including index metadata and column data types, which are often lost or converted when extracting a Series.

The critical syntactical difference lies in the inclusion of the colon: **df.iloc**. This notation implies a range selection, starting at the last row and proceeding to the implicit end of the DataFrame. Since slicing operations in Pandas always return a structure of the same dimensionality as the source, the output is guaranteed to be a DataFrame, even if it contains only one row.

This method is invaluable when preparing data for input into functions or libraries that strictly require a DataFrame object, or when the extracted row must retain its original index label for tracing or joining purposes. For example, if the original index represented timestamps, preserving the index label in the resulting single-row DataFrame is essential for maintaining data integrity.

last_row = df.iloc

Practical Demonstration: Setting Up the Sample DataFrame

To illustrate these concepts practically, we will establish a sample DataFrame representing player

statistics. This dataset contains ten records (rows 0 through 9) and three columns: assists, rebounds, and points. The final row, index 9, contains the record we aim to extract using both methods previously discussed.

We begin by importing the necessary Pandas library and then constructing the DataFrame explicitly from a dictionary of lists. Defining the data structure clearly ensures that the resulting row selections are easily verifiable against the source data.

The following code snippet sets up the environment and displays the complete dataset. Observe the last row (index 9) carefully, as its values (11 assists, 14 rebounds, 12 points) will be the target output for all subsequent extraction examples.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'assists': ,
'rebounds': ,
'points': })

#view DataFrame
print(df)

assists rebounds points
0 3 1 20
1 4 3 22
2 4 3 24
3 5 5 25
4 6 2 20
5 7 2 28
6 8 1 15
7 12 1 29
8 15 0 11
9 11 14 12

Detailed Analysis of Example 1: Series Output

Using the **df.iloc** command extracts the last row (index 9) as a Pandas Series. This transformation fundamentally changes the way the data is stored and accessed. The original column labels ('assists', 'rebounds', 'points') are converted into the Series index, while the original index label (9) is preserved as the **Name** attribute of the resulting Series object.

The resulting Series provides a vertical representation of the row data. The data type for all elements in the Series is determined by the combination of data types present in the original row, defaulting typically to **int64** or **object** if mixed types are present. This format simplifies vectorized operations performed solely on the attributes of this single entry.

Below is the code execution and the subsequent output, confirming that the last row is extracted successfully and presented as a one-dimensional object:

```
#get last row in Data Frame as Series
last_row = df.iloc

#view last row
print(last_row)
```

```
assists 11
rebounds 14
points 12
Name: 9, dtype: int64
```

To formally verify the data structure type, we can use the **type()** function in Python. This confirmation is crucial in scripting environments where the expected input type for subsequent functions must be guaranteed.

```
#view type
type(last_row)
```

```
pandas.core.series.Series
```

Detailed Analysis of Example 2: DataFrame Output

When executing **df.iloc**, the result is maintained as a standard DataFrame. The key difference from the Series output is the retention of the two-dimensional matrix structure. The columns remain column headings, and the row maintains its original index label (9). This preservation of structure ensures compatibility with standard DataFrame operations, such as merging, joining, or immediate saving to a file format like CSV or SQL.

Although this method requires slightly more overhead than creating a Series, the benefits often outweigh the minor performance difference, particularly when the analyst intends to perform column-wise filtering or comparisons that rely on the DataFrame syntax. For instance, accessing a specific cell requires using **last_row.iloc** or **last_row.values**, reflecting the two-dimensional nature.

The following demonstration confirms that the last row is returned with its structure intact, including the column labels positioned horizontally above the data:

#get last row in Data Frame as DataFrame
last_row = df.iloc

#view last row
print(last_row)

assists rebounds points
9 11 14 12

Similar to Example 1, we use the **type()** function to confirm the structural integrity of the output. The expected result confirms that Pandas successfully returned a **DataFrame** object, ready for further tabular processing.

#view type
type(last_row)

pandas.core.frame.DataFrame

Choosing Between Series and DataFrame Output

The decision between extracting the last row as a Pandas Series (**df.iloc**) or a DataFrame (**df.iloc**) is a common point of confusion for new Pandas users. The choice is not arbitrary; it must be driven by the requirements of the next step in the data pipeline. A general rule of thumb is to use the Series output for introspection and mathematical computation on the row's values, and the DataFrame output when structural consistency is paramount.

When working with multiple rows or needing to maintain column metadata (such as during joins or schema validation), the DataFrame output is mandatory. For instance, if you were appending this last row to a log of "final entries," the append function would typically expect a DataFrame input. Conversely, if you simply need to calculate the sum of points and rebounds for that final entry, the Series output allows for direct attribute access and simpler arithmetic.

A key difference often overlooked is how missing values are handled. While both structures manage missing data, working with a Series simplifies operations when only value extraction is required, as it reduces the syntactic overhead associated with the two-dimensional container. Analysts should consciously decide whether they need the row's DataFrame context or simply its contents before executing the extraction command.

Alternative Approaches to Accessing Final Data

While .iloc is the preferred and most versatile method for positional indexing, Pandas offers alternatives, particularly the **.tail()** function. The **.tail(n)** method is specifically designed to retrieve the last **n** rows of a DataFrame, and it always returns a DataFrame, making it syntactically cleaner for the DataFrame output requirement.

Using **df.tail(1)** is functionally equivalent to **df.iloc**. This method is often more readable for non-technical users, as the intent to retrieve the final record is explicitly clear. However, **.tail()** does not offer the flexibility to easily return a Series, nor can it be used for complex two-dimensional slicing that .iloc permits.

Another, less common method involves retrieving the index of the last element and then using .loc for retrieval: **last_index = df.index; df.loc**. This approach guarantees that even if the DataFrame index is not a simple integer range (e.g., if it uses dates or strings), the correct final record is accessed by label. While powerful, this two-step process is generally slower and less direct than the dedicated .iloc solution for simple positional access.