Learning How to Print Specific Rows in Pandas DataFrames

Authored by

Mohammed looti

May 29, 2026

Understanding Row Selection in Pandas

The ability to precisely select and retrieve specific rows is fundamental when working with tabular data using the Pandas library in Python. A DataFrame, the primary data structure in Pandas, organizes data into rows and labeled columns, requiring specialized methods for access. Unlike simple Python lists or arrays, DataFrames have two primary indexing systems: one based on numerical position (zero-based integer indexing) and another based on explicit labels defined in the Index. To effectively extract a single row or a subset of rows, we must distinguish between these two indexing types.

Pandas provides two powerful accessor functions specifically designed for row selection: `.iloc` and `.loc`. Choosing the correct method depends entirely on how you intend to identify the desired row--whether you know its physical location within the structure or its associated label. Misunderstanding the distinction between these methods often leads to frustrating errors, especially when dealing with DataFrames where the explicit row labels (the Index) do not align sequentially with the physical row positions.

This guide will thoroughly explore both techniques, providing clarity on when and how to deploy `.iloc` for positional retrieval and `.loc` for label-based retrieval. We will demonstrate how to select a single row and multiple rows, ensuring the output is correctly formatted and easily interpreted. Mastering these indexing techniques is essential for efficient data manipulation and analysis in Python.

Method 1: Positional Indexing with `iloc`

The `.iloc` accessor is used for integer-location based indexing. The 'i' in `.iloc` stands for 'integer,' signifying that it operates purely on the physical, zero-based position of the rows and columns, similar to how one would index a standard Python list. If you need the 4th row of the DataFrame, regardless of what its explicit label might be, `.iloc` is the tool you must use. This method is incredibly reliable when data order is critical or when the index labels are complex or non-unique.

To select and print a specific row using its position, you pass the integer position within double square brackets to the `.iloc` function. The use of double brackets (`[]`) is important because it ensures that the output is returned as a DataFrame, preserving the structure, rather than a Pandas Series (which would result from single brackets). For instance, if you wish to extract the fourth row (which corresponds to index position 3, as counting starts at 0), the syntax is remarkably straightforward.

This approach is particularly useful in iterative processes or when dynamically selecting rows based on their order of appearance in the dataset. Note the required format for selecting the row at

position 3:

Selecting Row Based on Index Position (Position 3):

print(df.iloc])

Setting Up Our Example DataFrame

Before diving into the practical examples of row selection, we must first define the DataFrame we will be using throughout this tutorial. This sample dataset represents fictional player statistics, including points, assists, and rebounds, and importantly, utilizes custom alphabetical labels (A through H) for its Index. This configuration provides a clear illustration of why differentiating between positional indexing (`iloc`) and label indexing (`loc`) is crucial.

We import the Pandas library and construct the DataFrame. Notice how the explicit index labels are assigned during the creation process. This setup ensures that we can effectively demonstrate the distinct behaviors of the two primary row selection methods. When the DataFrame is printed, observe that the labels (A, B, C...) appear on the left, separate from the implicit zero-based positions (0, 1, 2...).

The following Python code initializes our sample data structure, which we will query in the subsequent examples:

import pandas as pd

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

#view DataFrame
print(df)

points assists rebounds
A 18 4 3
B 22 5 9
C 19 5 12
D 14 4 4
E 10 9 4
F 11 12 9

G 20 11 8
H 28 8 2

Applying `iloc` for Row Retrieval by Position

Using `.iloc` allows us to target data based purely on its sequential position, ignoring the custom index labels (A, B, C, etc.). In our example DataFrame, position 3 corresponds to the row labeled 'D'. This method confirms that even when explicit labels exist, `.iloc` strictly adheres to the zero-based counting system inherent in computer science.

The following code demonstrates how to retrieve and print the row situated at index position 3. The output clearly shows the data for row 'D'. This is a critical distinction: we queried position 3, but the result includes the label 'D', confirming which row was selected based on its position:

Example 1: Printing a Single Row by Positional Index (3)

#print row located at index position 3
print(df.iloc])

points assists rebounds
D 14 4 4

Furthermore, `.iloc` is highly versatile and allows for the selection of multiple non-contiguous rows by simply passing a list of integer positions within the double brackets. If we wanted to analyze the data for the rows at position 3 and position 5, we include both integers in the list. This functionality is vital for extracting specific subsets of data quickly and efficiently based on their order in the structure.

Here is the demonstration for selecting multiple rows based on their positional index:

Selecting Multiple Rows by Positional Index (3 and 5)

#print rows located at index positions 3 and 5
print(df.iloc])

points assists rebounds
D 14 4 4
F 11 12 9

This code successfully retrieves the rows corresponding to labels 'D' and 'F', proving the efficacy of positional selection using `.iloc` when the numerical position is known.

Method 2: Label-Based Indexing with `loc`

In contrast to `.iloc`, the `.loc` accessor specializes in label-based indexing. The 'loc' stands for 'location,' referring to the explicit row label defined in the Index of the DataFrame. This method is preferred when the meaningful identifier of the row (such as a date, a name, or a category label) is known, rather than its arbitrary physical position. When using `.loc`, the user must provide the exact label, ensuring strict type matching (e.g., using a string label if the index consists of strings).

Using `.loc` enhances code readability, as it ties the selection directly to the semantic meaning of the data. If you know you need the statistics for player 'C', you can directly query using 'C' without needing to recall that 'C' happens to be at position 2. Like `.iloc`, when selecting a single row that is intended to be returned as a DataFrame structure, the label should be enclosed within double square brackets `]`.

If the user were to use single brackets (e.g., `df.loc`), the result would be returned as a Pandas Series, which is a one-dimensional array. While a Series is acceptable for many operations, returning a DataFrame (using double brackets) often provides a more consistent output format, especially when chaining operations or preparing data for subsequent analysis steps that require a two-dimensional structure.

The core advantage of `.loc` becomes apparent when dealing with sorted or filtered DataFrames where the original positional order has been disrupted, but the row labels remain intact.

Practical Application of `loc` for Index Labels

We will now use our sample DataFrame to demonstrate label-based selection. Suppose we need to extract the data for the player identified by the label 'C'. We pass 'C' as a string within the double brackets to the `.loc` accessor.

The following code executes this selection, printing the row associated with the label 'C'. Notice that the output is confined strictly to the data row corresponding to this label:

Example 2: Printing a Single Row by Index Label ('C')

#print row with index label 'C'
print(df.loc])

points assists rebounds
C 19 5 12

Just like `.iloc`, the `.loc` function supports selecting multiple rows simultaneously. To retrieve the

data for players 'C' and 'F', we provide a list containing both labels. This flexibility allows analysts to quickly gather disparate data points based on known identifiers.

Here is the command for retrieving both the row labeled 'C' and the row labeled 'F':

Selecting Multiple Rows by Index Labels ('C' and 'F')

#print rows with index labels 'C' and 'F'
print(df.loc])

points assists rebounds
C 19 5 12
F 11 12 9

The resulting output confirms that only the rows matching the specified index labels were returned, maintaining the structure of the original DataFrame. Understanding the difference between these two powerful indexing methods--`.iloc` for position and `.loc` for labels--is foundational to advanced data manipulation in Pandas.

Summary of Accessor Methods and Further Reading

The selection of rows in a DataFrame is optimized by using the appropriate accessor: `.iloc` for positional lookup (using integers) and `.loc` for label lookup (using the explicit index values). Always remember to use double brackets `]` when your goal is to return the output as a DataFrame structure, especially when selecting a single row, to ensure consistency in subsequent processing steps.

When working with large, complex datasets, confusion often arises when the Index has been reset, or when dealing with multi-indexing. In such scenarios, relying on `.iloc` is safer if you strictly need the Nth row, while `.loc` is invaluable for maintaining data integrity by selecting based on unchanging, explicit identifiers. Choosing the right tool simplifies your data pipeline, making your code more resilient and easier to debug.

For those seeking to expand their knowledge of data handling within the Pandas ecosystem, the following resources provide additional depth on common data manipulation operations and advanced indexing techniques.

Additional Resources

The following tutorials explain how to perform other common operations in pandas, building upon the foundational knowledge of row selection provided here:

Tutorial on Boolean Indexing in Pandas

Understanding MultiIndex Hierarchical Indexing

How to Select Columns in a Pandas DataFrame

Advanced Slicing Techniques using `.loc` and `.iloc`