Learning to Select Rows by Index in Pandas DataFrames: A Tutorial on .iloc and .loc

Authored by

Mohammed looti

November 6, 2025

In the dynamic world of Python-based data analysis, the ability to efficiently select specific subsets of data from a large dataset is not merely useful--it is fundamental. When working with the powerful pandas DataFrame structure, one of the most frequent requirements is isolating rows based on their specific position or identifying index label. Mastering this operation is key to effective data manipulation.

Pandas provides two highly specialized accessors for this task: .iloc and .loc. While both facilitate row selection, they operate on entirely different principles. Understanding this core distinction is critical for data scientists and analysts, as misusing them can lead to subtle but significant indexing errors that compromise data integrity.

If your objective is to select rows based purely on their physical sequence within the dataset--their zero-based position--you must utilize the **.iloc** function, which relies on integer indexing. Conversely, if the selection criteria rely on the explicit names or identifiers assigned to the rows, known as label indexing, then the **.loc** function is the appropriate and precise tool.

Understanding Pandas Indexing Fundamentals

A pandas DataFrame is uniquely designed to support two distinct indexing mechanisms simultaneously, providing incredible flexibility for data handling. The first is positional indexing, which is the implicit, zero-based ordering that reflects the physical arrangement of the rows, much like an element in a standard Python list or array. This inherent order is always present, regardless of what custom labels are applied.

The second mechanism is the explicit index. This consists of the custom labels assigned to each row, which serve as meaningful identifiers. While these labels often default to a sequential series of integers (0, 1, 2, ...), they can be customized to use non-sequential numbers, unique strings, descriptive categories, or timestamp values. This explicit index allows for highly intuitive data retrieval based on meaningful names rather than arbitrary positions.

The choice between the **.iloc** and **.loc** accessors hinges entirely on which of these two underlying indexing systems you intend to leverage for your data selection. Attempting to select a custom label using the positional accessor **.iloc**, or trying to use a physical position with the label accessor **.loc**, will inevitably lead to runtime errors or, more dangerously, unexpected data extraction results that introduce errors into downstream analysis.

Selecting Rows Using Positional Indexing (.iloc)

The **.iloc** accessor is derived from the term "integer location." Its operation is strictly confined to accessing data based on the integer position of the rows and columns, completely disregarding any explicit index labels that might be assigned to the DataFrame. This method adheres rigorously

to the zero-based counting convention common in Python, where the initial row is located at index 0, the second at index 1, and so on.

To illustrate its function, consider a scenario where we initialize a DataFrame with deliberately non-sequential index labels (0, 3, 6, 9, 12, 15). We then use **.iloc** to select a row based solely on its physical order. In the following example, we specifically target the 5th physical row, which corresponds to the positional index 4, demonstrating that the custom label (12) is irrelevant to the selection process:

import pandas as pd
import numpy as np

#make this example reproducible
np.random.seed(0)

#create DataFrame with custom index (0, 3, 6, ...)
df = pd.DataFrame(np.random.rand(6,2), index=range(0,18,3), columns=)

#view DataFrame
df

A B
0 0.548814 0.715189
3 0.602763 0.544883
6 0.423655 0.645894
9 0.437587 0.891773
12 0.963663 0.383442
15 0.791725 0.528895

#select the 5th physical row of the DataFrame (positional index 4)
df.iloc]

A B
12 0.963663 0.383442

Practical Application of .iloc (Lists and Slicing)

The utility of the .iloc accessor extends well beyond the selection of a single row. It offers powerful mechanisms for retrieving multiple, non-contiguous rows by passing a list of desired integer positions, or for extracting a continuous block of rows using standard Python slicing notation. This flexibility makes it indispensable for operations that require precise selection based on data order.

To select several rows that are not adjacent--for example, the 3rd, 4th, and 5th physical rows (corresponding to positional indices 2, 3, and 4)--we simply pass a list containing these specific integer locations to the **.iloc** accessor. The result is a DataFrame subset containing only those rows, ordered based on the input list:

#select the 3rd, 4th, and 5th physical rows of the DataFrame
df.iloc]

A B
6 0.423655 0.645894
9 0.437587 0.891773
12 0.963663 0.383442

Furthermore, for extracting a continuous sequence of rows, we can employ slicing. It is crucial to remember that **.iloc** strictly adheres to Python's standard slicing conventions: the starting index is always inclusive, but the stopping index is always exclusive. Therefore, to select positions 2, 3, and 4, we must specify the slice as , where 5 is the index immediately following the last desired row. This ensures accurate retrieval of the intended data block:

#select the 3rd, 4th, and 5th physical rows using slicing
df.iloc

A B
6 0.423655 0.645894
9 0.437587 0.891773
12 0.963663 0.383442

Selecting Rows Using Label Indexing (.loc)

In contrast to positional indexing, the **.loc** accessor--short for "label location"--is dedicated to selecting rows and columns using the explicit index labels defined within the pandas DataFrame. This method is the preferred approach when the index structure contains meaningful identifiers such as unique database IDs, date stamps, or descriptive string values. It allows users to query data based on business logic or semantic meaning rather than physical arrangement.

A key behavioral difference distinguishes .loc from standard Python slicing: when performing a slice operation (e.g., `df.loc`), the stop label is **inclusive**. This means that the row corresponding to the final label specified in the slice will be included in the output, providing a more intuitive experience when dealing with labeled data ranges.

Using the same DataFrame setup with custom index labels (0, 3, 6, 9, 12, and 15), we demonstrate how **.loc** precisely selects the row whose index label is exactly 3. Note that this is the second physical row, but we refer to it by its label, not its position:

import pandas as pd
import numpy as np

#make this example reproducible
np.random.seed(0)

#create DataFrame
df = pd.DataFrame(np.random.rand(6,2), index=range(0,18,3), columns=)

#view DataFrame
df

A B
0 0.548814 0.715189
3 0.602763 0.544883
6 0.423655 0.645894
9 0.437587 0.891773
12 0.963663 0.383442
15 0.791725 0.528895

#select the row with index label '3'
df.loc]

A B
3 0.602763 0.544883

Similar to **.iloc**, the **.loc** accessor permits the selection of multiple, non-contiguous rows by supplying a list of the desired index labels. This list does not require any specific ordering, provided that all the labels specified exist within the DataFrame's explicit index. This capability is essential for querying specific data points scattered throughout a large index:

#select the rows with index labels '3', '6', and '9'
df.loc]

A B
3 0.602763 0.544883
6 0.423655 0.645894
9 0.437587 0.891773

The Essential Difference Between .iloc and .loc

For individuals new to the pandas DataFrame library, the distinction between **.iloc** and **.loc** is often a primary source of confusion. However, the fundamental principle governing their use is straightforward and must be adhered to strictly: **.iloc** exclusively uses physical position, and **.loc** exclusively uses the explicit label.

A common scenario that complicates this understanding is when the explicit index labels happen to be the default sequential integers (0, 1, 2, 3, ...). In such cases, using **.iloc** with an integer (e.g., `df.iloc`) and using **.loc** with the same integer (e.g., `df.loc`) will yield the same row. Relying on this accidental overlap, however, is poor practice and can lead to fragile code that breaks if the index is ever customized or reset.

To write robust and readable code, you must consciously decide whether your intent is to retrieve the "Nth physical row" (requiring **.iloc**) or to retrieve the "row explicitly labeled N" (requiring **.loc**). This intentional choice ensures that your selection logic remains sound even as the DataFrame structure evolves.

.iloc Selection: This method selects rows based on their **integer index** (position). To select the 5th row in a DataFrame, you must use `df.iloc]`, remembering that the physical index count begins at 0.

.loc Selection: This method selects rows based on their **labeled index**. To select the row whose explicit index label is 5, regardless of its physical location within the DataFrame, you would use `df.loc]` directly.

Slicing Behavior: When slicing, **.iloc** is exclusive of the stop value (aligning with standard Python list behavior), while **.loc** is uniquely inclusive of the specified stop label.

Additional Resources for Pandas Mastery

For those committed to advancing their proficiency in data manipulation and achieving precise data selection within the Pandas ecosystem, the following resources offer valuable supplemental guidance on related indexing and data management topics:

How to Get Row Numbers in a Pandas DataFrame
How to Drop Rows with NaN Values in Pandas
How to Drop the Index Column in Pandas