How to select rows by index in a Pandas DataFrame?

Authored by

stats writer

December 16, 2025

When conducting data analysis using Pandas DataFrames, one of the most frequent operations is the precise selection of rows. Unlike traditional numerical arrays, Pandas introduces two distinct indexing mechanisms: selection based on implicit positional integers (similar to array indexing) and selection based on explicit index labels (which can be numbers, strings, or dates). Mastering these selection methods is fundamental to efficient data manipulation.

To accurately select rows within a DataFrame, developers must choose between two powerful indexers: the `.iloc` function, designed for positional selection using integer indexing, and the `.loc` function, utilized for selection based on explicit label indexing. While both achieve row selection, their underlying logic and appropriate use cases differ significantly, especially when the DataFrame index has been explicitly defined or customized.

This comprehensive guide will demonstrate the practical application of both `.iloc` and `.loc`, detailing how they handle single row retrieval, multiple row selection via lists, and range-based slicing. Understanding the nuances of these functions is crucial for any data scientist or analyst working with Python and Pandas.

Example 1: Selecting Rows Using Positional Integer Indexing (`.iloc`)

The `.iloc` indexer stands for "integer location" and is strictly based on the zero-based position of the rows, irrespective of the actual index labels assigned to the DataFrame. This behavior mirrors standard array indexing in Python. If you wish to retrieve the fifth row of a DataFrame, regardless of whether its index label is 4, 100, or 'A', you must use the integer position `4` (since indexing starts at 0).

This function is indispensable when the data analyst needs to access rows based purely on their order within the dataset, such as retrieving the first 10 observations or the last 5 observations. It provides a reliable mechanism for accessing data based on implicit row number, which remains constant even if the explicit index labels are non-sequential or non-numeric.

The following code snippet demonstrates how to first construct a sample DataFrame using the NumPy library and then use `.iloc` to precisely select the row corresponding to the fifth position (index 4) within the underlying data structure:

import pandas as pd
import numpy as np

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

#create DataFrame with non-standard index (0, 3, 6, 9, 12, 15)
df = pd.DataFrame(np.random.rand(6,2), index=range(0,18,3), columns=)

#view DataFrame structure and labels
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 row (position index 4) of the DataFrame
df.iloc]

A B
12 0.963663 0.383442

In the output above, notice that although the selected row has an index label of `12`, it was retrieved by calling `.iloc]` because `12` is the label corresponding to the fifth position (index 4) in the sequence of rows. This distinction between positional index and index label is the core concept of using `.iloc` effectively.

Using `.iloc` for Selecting Multiple Rows

The true power of `.iloc` often comes into play when selecting multiple, non-contiguous rows or applying slicing across a range of positions. To select multiple rows that are not sequential, we pass a list of integer indices to the `.iloc` accessor. This list explicitly defines the positional order of the rows we wish to retrieve.

For instance, if we needed to inspect the third, fourth, and fifth rows of our DataFrame, we would provide the corresponding positional indices: `2`, `3`, and `4`. This method ensures that the selection is purely position-based, guaranteeing that we retrieve the rows in those specific sequential positions within the DataFrame.

#select the 3rd, 4th, and 5th rows of the DataFrame (positional indices 2, 3, 4)
df.iloc]

A B
6 0.423655 0.645894
9 0.437587 0.891773
12 0.963663 0.383442

Implementing Slicing Techniques with `.iloc`

When dealing with a contiguous block of rows, standard Python slicing notation significantly simplifies the selection process. The `.iloc` indexer supports range-based slicing using the syntax , where `start` is the beginning positional index (inclusive) and `stop` is the ending positional index (exclusive). This behavior aligns perfectly with Python's conventional list and array slicing.

If the requirement is to select the third, fourth, and fifth rows, we specify the starting position (2) and the position immediately following the desired range (5). The resulting slice will effectively capture indices 2, 3, and 4, providing a concise way to select a block of data based on position.

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

A B
6 0.423655 0.645894
9 0.437587 0.891773
12 0.963663 0.383442

It is essential to remember that when using `.iloc` slicing, the stop boundary is always exclusive. If a user intends to include the row at positional index 5, they must specify . This exclusivity is a fundamental rule of integer indexing in Python and Pandas.

Example 2: Selecting Rows Using Explicit Label Indexing (`.loc`)

The `.loc` indexer, which stands for "label location," is dedicated to retrieving rows based on their explicit index labels. Unlike `.iloc`, the position of the row within the DataFrame is irrelevant; only the actual values in the index column matter. This makes `.loc` the primary tool for time-series data, category lookups, or any scenario where the index itself carries semantic meaning.

When querying a DataFrame using `.loc`, the user must provide the exact label or set of labels corresponding to the desired rows. If the DataFrame's index is composed of strings (e.g., product IDs or dates), `.loc` expects those strings. If the index consists of integers (as in our example, where the index is 0, 3, 6, etc.), `.loc` treats those numbers as labels, not positions.

Using the same DataFrame created previously, which has index labels 0, 3, 6, 9, 12, and 15, we demonstrate how to use `.loc` to retrieve the row explicitly labeled `3`:

import pandas as pd
import numpy as np

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

#create DataFrame with index labels 0, 3, 6, 9, 12, 15
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

If we had attempted to use `df.iloc]`, we would have retrieved the fourth row (label 9). However, because we used `df.loc]`, we correctly retrieved the row whose index label value is exactly 3. This highlights the crucial difference: `.loc` requires the index value itself, not its position.

Selecting Multiple, Non-Contiguous Rows with `.loc`

Similar to `.iloc`, the `.loc` accessor accepts a list of desired index values to retrieve multiple rows simultaneously. This is exceptionally useful when performing lookups of specific identifiers within a large dataset, where those identifiers are stored in the index.

To select the rows corresponding to index labels 3, 6, and 9, we provide `.loc` with a list containing these specific labels. This operation guarantees the retrieval of only those rows whose indices exactly match the provided labels, regardless of where they fall in the positional order.

#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

An important consideration when using `.loc` is that if a requested label does not exist in the DataFrame index, Pandas will raise a `KeyError`, ensuring that data integrity based on explicit labels is maintained.

Slicing by Label Range Using `.loc`

Slicing with `.loc` offers a significant departure from `.iloc` and Python's standard slicing conventions. When slicing using index labels--for example, `df.loc`--the stop label is **inclusive**. This feature is highly intuitive when working with categorical or chronological data, as the user typically intends to include data up to and including the specified endpoint.

For our DataFrame, if we wish to slice from label 3 up to and including label 9, the syntax `df.loc` will include the rows labeled 3, 6, and 9. This behavior contrasts sharply with the exclusive stop boundary used by `.iloc`.

#select the rows from index label '3' up to and including '9'
df.loc

A B
3 0.602763 0.544883
6 0.423655 0.645894
9 0.437587 0.891773

This inclusive nature of label slicing is a powerful feature of Pandas, especially when dealing with data indexed by dates or identifiers where defining an exact range is critical for filtering subsets of the data.

The Fundamental Difference Between `.iloc` and `.loc`

Understanding when to use `.iloc` versus `.loc` is perhaps the most critical skill when indexing Pandas DataFrames. Although they appear similar, they serve entirely different purposes based on the indexing methodology being employed.

`.iloc` (Integer Location): This indexer is strictly positional. It relies on the inherent, zero-based order of rows within the physical structure of the DataFrame. When selecting the 5th row, you must use the integer index `4` (since 0-indexing applies). This method is immune to changes in the explicit index labels and only changes if the DataFrame is physically reordered.

`.loc` (Label Location): This indexer is strictly label-based. It requires the actual value defined in the index column for selection. If you want to select the row explicitly labeled `5`, you must use `df.loc]`, irrespective of whether that row is the 1st or the 500th row of the dataset. This method is crucial when the index carries critical descriptive information.

The distinction is most apparent when working with DataFrames whose indices are non-standard, non-sequential, or non-numeric (e.g., strings or timestamps). In such cases, attempting to use `.iloc` when aiming for a specific label, or vice versa, will lead to unexpected results or errors. Developers should always clarify whether they are interested in the position of the data (use `.iloc`) or the identity of the data (use `.loc`).

Considerations for Mixed Indexing Schemes

A common pitfall occurs when the DataFrame's explicit index labels happen to be simple integers (0, 1, 2, 3, etc.), mimicking the default positional indices. In this scenario, both `df.iloc` and `df.loc` would return the same row, as the positional index 2 corresponds exactly to the label index 2.

However, relying on this coincidence is poor practice. If the DataFrame is later subsetted or sorted, the positional index might shift relative to the label index. By explicitly choosing `.iloc` or `.loc`, the code clearly communicates whether the selection logic is based on position or label, making the code significantly more robust and readable for future maintenance.

Furthermore, these indexers are not limited to row selection; they also handle column selection. When combined, `df.loc` and `df.iloc` allow for powerful two-dimensional subsetting, enabling the retrieval of specific data points identified by both row and column indices (label or position).

Conclusion: Choosing the Right Indexer for Data Integrity

Effective data retrieval in Pandas hinges on correctly applying `.iloc` for positional access and `.loc` for label-based access. By adhering to these conventions, data professionals ensure their operations are logically sound and maintain data integrity, regardless of the complexity or custom nature of the DataFrame's index. Always default to `.loc` when working with meaningful labels (such as unique identifiers or dates) and reserve `.iloc` for situations where the row's physical sequence is the sole criterion for selection.

The following resources provide further exploration into advanced indexing techniques in Pandas:

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