How to Choose Between Pandas loc vs. iloc for Data Selection

Authored by

stats writer

December 3, 2025

The distinction between Pandas **loc** and **iloc** is fundamental to effective data manipulation. At its core, **loc** is **label-based**, meaning it operates using the explicit row and column labels defined in the DataFrame index. Conversely, **iloc** is **integer index-based**, requiring users to specify rows and columns solely by their numerical position, starting from zero. Mastering these two methods of data access is crucial for anyone working extensively with the Pandas library, as incorrect selection can lead to subtle yet significant data errors.

When selecting subsets of rows and columns from a DataFrame, **loc** and **iloc** are the primary accessors provided by Pandas. While both achieve the goal of subsetting data, their methodology differs significantly based on how they interpret the selection criteria--labels versus positions. This detailed guide will explore the mechanisms of each function, provide practical examples, and highlight the critical scenarios where their behaviors diverge, ensuring robust and predictable data selection.

Understanding the subtle yet important difference between these two accessors is key to writing clean, reliable data science code:

loc (Location): Selects data using explicit row and column **labels**, or with a Boolean array.
iloc (Integer Location): Selects data using numerical **integer positions** (zero-based indexing).

The following sections will demonstrate these functions in practice, starting with the foundation of indexing in Pandas.

Understanding Pandas Indexing Fundamentals

Before diving into **loc** and **iloc**, it is essential to grasp how Pandas organizes data. Every DataFrame consists of two primary axes: the row index (Axis 0) and the column index (Axis 1). The row index often defaults to simple integers (0, 1, 2, ...), but it can be set to meaningful categorical or temporal identifiers, such as dates or unique IDs. The column index, conversely, uses labels (names) for each feature or variable.

When a DataFrame is created, it inherently has two parallel ways of addressing data points: the explicit labels assigned by the user (used by **loc**) and the implicit, zero-based integer positions (used by **iloc**). These two indexing schemes often align when the default integer index is used, but they decouple immediately when a custom index is applied. This decoupling is precisely why understanding the difference between label-based access and integer-based access is non-negotiable for intermediate and advanced Pandas users.

The distinction becomes crucial when dealing with filtered or sorted DataFrames. If you sort a DataFrame, the explicit labels (A, B, C, etc.) remain attached to their original rows, but their integer positions (0, 1, 2, etc.) change based on the new order. **loc** will find the original row by its label

regardless of its position, while **iloc** will access the row currently residing at that numerical position.

Deep Dive into loc: The Label Selector

The **loc** accessor is the preferred method when you need to select data based on the names assigned to the index and columns. Because it relies on explicit labels, the resulting code is often more readable and less prone to errors stemming from changes in row order. The general syntax for **loc** is `df.loc`, where both selectors can be a single label, a list of labels, a slice of labels, or a Boolean array.

A significant characteristic of **loc** is its handling of slices. When performing slicing, **loc** is **inclusive** of both the start and the stop label specified in the slice. For instance, slicing from 'A' to 'C' will include rows labeled 'A', 'B', and 'C'. This behavior mirrors standard Python dictionary lookup logic where keys are used, rather than sequence indexing.

Let us begin by initializing a DataFrame with custom labels to clearly illustrate the label-based nature of **loc**. Notice how the index labels are characters 'A' through 'H', which are distinct from the integer positions (0 through 7).

Example 1: Using loc in Pandas

Suppose we have the following Pandas DataFrame:

import pandas as pd

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

#view DataFrame
df

team points assists
A A 5 11
B A 7 8
C A 7 10
D A 9 6
E B 12 6
F B 9 5
G B 9 9

H B 4 12

We can use **loc** to select specific rows of the DataFrame based on their index labels. By passing a list of row labels, we retrieve only those specific records:

#select rows with index labels 'E' and 'F'
df.loc]

team points assists
E B 12 6
F B 9 5

To further refine the selection, we can specify both row and column labels. Here, we retrieve rows 'E' and 'F', but restrict the output to the 'team' and 'assists' columns:

#select 'E' and 'F' rows and 'team' and 'assists' columns
df.loc,]

team assists
E B 12
F B 9

Finally, the slicing capability of **loc** allows us to select continuous ranges based on label order. Note the use of the `:` argument, which is applied to the row axis starting at 'E', and to the column axis up to and *including* 'assists'.

#select rows from 'E' until the end, and columns up to 'assists' (inclusive)
df.loc

team points assists
E B 12 6
F B 9 5
G B 9 9
H B 4 12

Advanced loc Usage: Conditional Filtering

While simple label selection is straightforward, **loc** truly shines when used for conditional selection via Boolean arrays. A Boolean array is a series of `True` or `False` values, where the length of the array matches the number of rows in the DataFrame. When passed to **loc**, it returns only those

rows corresponding to a `True` value.

This method is the canonical way to filter data based on criteria applied to column values. For example, filtering all rows where the 'points' column is greater than 8 generates a Boolean array, which **loc** then uses to return the subsetted DataFrame. This ability to combine filtering logic with data selection is one of the most powerful features of the Pandas library, allowing for complex, readable data queries.

Furthermore, remember the behavior of label slicing with **loc**. Unlike standard Python list slicing, which is half-open (exclusive of the stop index), **loc** treats the slice labels inclusively. When dealing with hierarchical indices or time-series data, this inclusive nature often makes label-based slicing intuitive, as users typically want to include data up to the specified end point label, such as a final date.

Deep Dive into iloc: The Positional Selector

The **iloc** accessor is used exclusively for selection based on the integer position of the rows and columns, irrespective of any explicit labels they might possess. It treats the DataFrame as a grid or a matrix, where every row and every column is numbered sequentially starting from 0. This method is especially useful when the row labels are irrelevant or when you need to access data based on computed numerical positions, such as retrieving the first five rows or the last three columns.

The syntax is analogous to **loc**: `df.iloc`. However, the interpretation of slicing in **iloc** adheres strictly to Python's standard conventions. Slicing with **iloc** is **exclusive** of the stop position, meaning a slice like `4:6` will include positions 4 and 5, but not 6. This is a critical distinction from **loc** and a frequent source of indexing errors for new users.

Example 2: Using iloc in Pandas

Using the same initialized Pandas DataFrame, we now demonstrate positional access with **iloc**. Although the index labels are characters (A, B, C...), their integer positions remain 0, 1, 2, and so on.

import pandas as pd

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

#view DataFrame
df

team points assists
A A 5 11
B A 7 8
C A 7 10
D A 9 6
E B 12 6
F B 9 5
G B 9 9
H B 4 12

We use **iloc** to select rows based on their numerical position. To select the rows corresponding to index labels 'E' and 'F', we must remember that 'E' is at position 4 and 'F' is at position 5. To include both, the slice must stop at position 6 (exclusive):

#select rows in index positions 4 through 5 (not including 6)
df.iloc

team points assists
E B 12 6
F B 9 5

Similarly, when selecting columns, we use their numerical positions (0 for 'team', 1 for 'points', 2 for 'assists'). To select columns 'team' (0) and 'points' (1), the slice must stop at position 2 (exclusive):

#select rows in range 4 through 6 and columns in range 0 through 1 (not including 2)
df.iloc

team points
E B 12
F B 9

The positional slicing can also be used to select large ranges. Here, we select rows starting from position 4 until the end, and columns from the beginning up to position 3 (exclusive, meaning columns 0, 1, and 2):

#select rows from 4 through end of rows and columns up to third column (exclusive)
df.iloc

team points assists
E B 12 6
F B 9 5
G B 9 9
H B 4 12

Key Differences and Edge Cases: Slicing Behavior

The most crucial practical difference between **loc** and **iloc** lies in their behavior regarding slicing boundaries. This difference often trips up users transitioning between positional indexing and label-based indexing:

loc (Label-Based): Slicing is **inclusive** of the stop label. If you slice `df.loc`, the result will contain rows A, B, and C.
iloc (Integer-Based): Slicing is **exclusive** of the stop integer position. If you slice `df.iloc`, the result will contain rows at positions 0, 1, and 2, but not 3.

This disparity is intentional, reflecting the nature of the indices they manipulate. **loc** treats the index as a set of keys, where you explicitly name the start and end points of the key range. **iloc** treats the positions as a Python sequence, adhering to standard sequence slicing behavior where the stop index specifies the first element *not* to include.

Another important edge case involves DataFrames where the explicit index labels are integers. In such cases, if you use **loc**, it will search for the explicit integer *label*. If you use **iloc**, it will search for the implicit integer *position*. If the DataFrame is unsorted or has skipped index values, **loc** and **iloc** will return vastly different results, emphasizing why it is essential to always know whether you are addressing the label or the position when dealing with integer labels.

When to Choose loc vs. iloc

Choosing between **loc** and **iloc** depends entirely on the context and the nature of the index you are working with. A general rule of thumb dictates that code should prioritize clarity and robustness.

Use loc for Readability and Stability: When your row or column indices have meaningful names (e.g., dates, names, unique IDs), **loc** is almost always the better choice. It makes the code self-documenting (e.g., `df.loc` is clearer than using hardcoded integer positions) and is stable against sorting or filtering operations that shift the integer positions.
Use iloc for Positional Operations: iloc is ideal when you need to access data based on relative position, such as retrieving the first row (`df.iloc`), the last row (`df.iloc`), or selecting the first N

columns regardless of their names. It is also necessary when the DataFrame uses the default integer index and you prefer Python-standard exclusive slicing.

In summary, **loc** handles explicit indexing, focusing on names and labels, while **iloc** handles implicit indexing, focusing on zero-based positions. Understanding this core difference and the inclusive/exclusive slicing behavior ensures accurate and efficient data access in Pandas.

Summary of Differences

To crystallize the distinction, here is a quick reference list summarizing the key operational differences between the two methods:

Basis of Selection: loc uses labels (index names, column names); **iloc** uses integer positions (0, 1, 2...).
Slicing Behavior: loc includes the stop label (inclusive); **iloc** excludes the stop position (exclusive).
Input Types: loc accepts labels, lists of labels, Boolean arrays, or callable functions; **iloc** accepts integers, lists of integers, integer slices, or callable functions.
Use Case: loc is best for filtering by values, date ranges, or specific categorical keys; **iloc** is best for selecting arbitrary rows/columns based on order (e.g., top 5 rows).

By consistently applying the appropriate accessor based on whether you are querying by label or by position, you can leverage the full power and flexibility of the Pandas data manipulation toolkit.

How to Select Rows Based on Column Values in Pandas