Understanding Data Selection with Pandas: A Guide to loc and iloc

Authored by

Mohammed looti

November 1, 2025

When conducting data analysis in Python, efficiently and accurately selecting subsets of data is perhaps the most fundamental skill. The Pandas library provides two extraordinarily powerful, yet frequently confused, accessors for this task: **loc** and **iloc**. While both functions allow users to extract rows and columns from a DataFrame, they employ fundamentally different mechanisms rooted in how they interpret the index.

A solid understanding of these two methods--label-based selection versus positional selection--is essential for writing robust, scalable, and predictable data code. Misusing **loc** where **iloc** is intended, or vice versa, is a common source of bugs, especially when working with datasets that utilize non-default, complex, or non-sequential row indices. We will delve into the precise operation of each accessor and provide clear examples to ensure you select your data with confidence.

Understanding the Core Distinction: Label vs. Position

The central difference between **loc** and **iloc** lies in the type of argument they accept and how they map that argument to the underlying data structure. This distinction is known as explicit versus implicit indexing. The **loc** accessor operates using explicit indexing, meaning it relies entirely on the assigned names, known as **index labels**, of the rows and columns. Conversely, the **iloc** accessor uses implicit indexing, which relies on the zero-based **integer positions** of the rows and columns, mirroring how standard Python sequences (like lists) are indexed.

This divergence is critically important because a DataFrame's visual row order (its integer position) might not align with its explicit index labels. For example, if a dataset is indexed by dates, or if the user performs an operation like resetting the index, the physical order (position 0, 1, 2...) remains constant, but the label (Jan 1, Jan 2, Jan 3...) dictates how **loc** accesses the data. Therefore, the choice between **loc** and **iloc** must be determined by whether you are interested in the semantic meaning of the row (the label) or its physical location (the position).

To summarize the functionality and usage criteria:

loc: Used for selection based on specific, meaningful **labels** (names) of rows and columns. This is the preferred method for working with indices that have semantic value (e.g., timestamps, unique IDs).

iloc: Used for selection based on specific, zero-based **integer positions** (sequential location) of rows and columns. This is the preferred method for selecting data based purely on physical order (e.g., the first 5 rows or the last column).

Deep Dive into the loc Accessor (Label-Based Indexing)

The **loc** method is the preferred and most readable tool when your data selection criteria depend

on the actual names assigned to the rows and columns. Utilizing **loc** enhances code clarity, as the selection criteria directly reference the data's metadata. This is particularly valuable in production environments where indices represent real-world identifiers, such as stock tickers, geographical identifiers, or dates.

When utilizing **loc**, selection arguments are passed in the format `df.loc`. Both selectors must provide the exact label, a list of exact labels, or a boolean series. One crucial aspect of **loc** is its behavior when slicing ranges: the end label is always **inclusive**. If you specify a range like `df.loc`, the resulting subset will include the row labeled 'Start', all rows between, and the row labeled 'End'.

Beyond simple label selection, **loc** is also the primary mechanism in Pandas for powerful Boolean Indexing. This allows data analysts to filter rows based on complex conditional logic applied to column values (e.g., selecting all customers where 'spending' is greater than $500). This flexibility makes **loc** indispensable for complex data filtering and manipulation.

Practical Application of loc for Row and Column Selection

To clearly demonstrate the mechanics of **loc**, we will establish a sample DataFrame that utilizes distinct, named row indices. This structure ensures that the integer position and the label are separate, highlighting the power of label-based indexing.

We start with the following setup:

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

Using **loc**, we can target specific rows based solely on their index labels. By passing a list of labels, we can select non-contiguous rows in a single operation:

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

team points assists
E B 12 6
F B 9 5

Furthermore, **loc** facilitates precise subsetting by allowing simultaneous selection of specific rows and columns using their respective labels. Here, we target rows 'E' and 'F' and limit 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, observe the inclusive nature of slicing with **loc**. The statement below selects rows starting at label 'E' through the very last label, 'H'. It also selects columns up to and including the label 'assists':

#select rows starting from 'E' through the end, and columns up to and including 'assists'
df.loc

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

Mastering the iloc Accessor (Integer-Based Indexing)

The **iloc** accessor is designed for accessing data based purely on its sequential, physical location within the DataFrame, completely disregarding any named indices. This method is highly

consistent with standard Python list and array indexing, making it intuitive for selecting data based on position rather than semantic meaning.

When using **iloc**, all arguments provided must be integers or sequences of integers representing the zero-based positions of the rows and columns. For instance, if you want the fifth row, you request position 4; if you want the third column, you request position 2. This strict reliance on integer position ensures that **iloc** selections are always predictable based on the physical order of the data.

The most critical difference when slicing with **iloc** is that the stop position is always **exclusive**. This behavior mirrors Python's standard slicing convention, where the element at the index specified as the end point is excluded from the result. For example, selecting yields elements at positions 0, 1, 2, 3, and 4, excluding the element at position 5. This consistency with native Python sequencing is a significant benefit when performing iterative or algorithmic selections where precise positional control is necessary.

Practical Application of iloc for Positional Slicing

We will reuse the same DataFrame to illustrate how **iloc** ignores the explicit labels ('A' through 'H') and instead relies on the physical positions (0 through 7).

Referencing our sample 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

To select rows 'E' and 'F', we must recognize that these correspond to the 4th and 5th physical rows (positions 4 and 5, since indexing starts at zero). To include position 4 and 5, we must specify the slice , where 6 is the exclusive stop point:

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

team points assists
E B 12 6
F B 9 5

We can simultaneously select rows and columns using their integer positions. For example, to select rows at positions 4 and 5 (`4:6`) and the first two columns, 'team' (0) and 'points' (1), we specify the column range as `0:2` (excluding position 2, which is 'assists'):

#select rows in range 4 through 6 and columns in range 0 through 2
df.iloc

team points
E B 12
F B 9

The colon operator (**:**) maintains its standard Python slicing function in **iloc**, allowing us to select ranges from a specific point to the end of the data structure. The following example selects rows from position 4 to the end of the DataFrame, and columns from the start up to (but not including) position 3:

#select rows from position 4 through end of rows and columns up to (but not including) position 3
df.iloc

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

Crucial Differences in Slicing Behavior

The most common point of confusion and error when transitioning between **loc** and **iloc** is the handling of slice boundaries. Analysts must clearly distinguish between label inclusivity and positional exclusivity when using the colon (**:**) operator:

loc (Label-based): The stop label is **inclusive**. When slicing, the element identified by the label used as the stopping point will always be included in the resulting subset.

iloc (Position-based): The stop position is **exclusive**. When slicing, the element at the integer position used as the stopping point will *not* be included, maintaining consistency with native Python slicing conventions.

This difference in inclusivity demands careful attention. If your DataFrame uses the default sequential integer index (0, 1, 2, 3...), using **loc** with these integers will treat them as *labels*, resulting in inclusive slicing, which is often not the intended behavior when the user is thinking positionally.

As a best practice, always choose the accessor that best reflects the intent of the selection. Use **loc** when the index has semantic meaning (e.g., filtering based on a date range or specific customer IDs). Use **iloc** exclusively when the selection criterion is the physical order of the data (e.g., retrieving the first or last N elements). Sticking to this guideline dramatically improves code readability, maintainability, and reduces the likelihood of subtle indexing bugs.

Additional Resources

The following tutorials explain how to perform other common operations in Pandas:

How to Select Rows Based on Column Values in Pandas