Pandas: Use loc to Select Multiple Columns questions

Authored by

stats writer

November 22, 2025

RECOMMENDED CITATION

stats writer (2025). *Pandas: Use loc to Select Multiple Columns questions*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=99335

Mastering Column Selection with Pandas loc

The Pandas library stands as an indispensable tool for data manipulation and analysis in Python. A central element of this library is the ability to efficiently subset data, allowing users to isolate specific rows and columns for focused processing. Among the most potent indexing mechanisms provided by Pandas is the loc accessor. Unlike integer-based indexing, the loc method operates purely on labels, providing a highly readable and stable way to interact with data. This label-based approach is particularly advantageous when selecting multiple columns, especially when those columns are non-contiguous or when clarity in code is paramount.

When working with large, complex datasets, the need to select several columns simultaneously is routine. While simple bracket notation can achieve this, the loc method offers a standardized syntax that applies equally well to row selection, ensuring consistency across all indexing operations. By leveraging loc for column selection, developers gain precise control over which attributes of the data are retained, enhancing both the precision of data extraction and the ease of subsequent operations like updates or aggregations.

This detailed guide explores the fundamental techniques for using the loc indexer to retrieve multiple columns from a DataFrame. We will examine two primary methodologies: selecting columns by passing an explicit list of column labels, and selecting contiguous columns using standard Python slicing notation applied to the column labels. Understanding these methods is crucial for anyone aiming to write robust, efficient, and maintainable data analysis scripts using Pandas.

Understanding the Syntax of the loc Accessor

The loc accessor follows a mandatory structure: it requires two arguments separated by a comma, mirroring the traditional structure of matrix indexing: `df.loc`. The power of loc lies in its versatility in accepting various inputs for these indexers, including single labels, lists of labels, boolean arrays, or slice objects.

When the objective is strictly to select columns, without imposing any filtering criteria on the rows, the row indexer is typically set to the colon symbol (`:`). This colon acts as the standard Python slice operator, signifying that we wish to select all rows present in the DataFrame. The column indexer then becomes the focus, defining precisely which columns are returned in the resulting subset.

Consequently, when using `df.loc` to select multiple columns by label, the core challenge simplifies to formulating the correct argument for the second position (the column indexer). We will now delve into the two most common and effective ways to structure this column indexer, allowing for flexible selection of both non-contiguous and contiguous column sets based on their respective string labels.

You can use the **loc** function in Pandas to select multiple columns in a DataFrame by label.

Here are the two most common ways to achieve multi-column selection:

Method 1: Explicit Selection Using a List of Column Labels

The first and most flexible method involves passing a standard Python list of column names to the column indexer position of the `.loc` function. This approach is ideal for selecting columns that are scattered throughout the DataFrame and do not appear next to each other in the original structure. Because the list explicitly defines the columns sought, the order in which they appear in the resulting subset DataFrame will exactly match the order specified within the input list.

This method requires the column labels to be provided as strings enclosed within the square brackets of the list. It offers unparalleled precision, ensuring that only the columns explicitly named are returned, regardless of the relative positions of other columns. The general syntax for this approach is concise and highly readable:

df.loc]

In the example syntax shown above, `df.loc` is called, the row indexer `:` ensures all rows are included, and the column indexer specifically requests only those two columns. This technique is highly recommended whenever the desired subset of columns is small, non-sequential, or requires a specific output order different from the source DataFrame.

Method 2: Selecting a Range of Columns Using Slicing

The second powerful technique utilizes Python's slicing notation directly on the column labels. This method is exceptionally efficient when the required columns are adjacent or sequential within the original DataFrame. Unlike standard Python slicing, which typically excludes the end index, the loc accessor slicing is **inclusive** of both the start and end column labels specified.

To use this method, you pass the starting column label, followed by the colon (`:`) slice operator, and then the ending column label, all enclosed as a single string argument (not a list of strings) to the column indexer. The loc method then automatically selects the starting column, the ending column, and every column positioned between them in the DataFrame structure.

The syntax for selecting a contiguous range of columns is straightforward:

df.loc

This slicing technique provides a much cleaner and less verbose approach than manually listing

every column label when dealing with large sequential subsets. It is important to remember the inclusive nature of loc slicing, as this behavior differs fundamentally from standard Python slicing conventions used in libraries like NumPy or built-in Python sequences.

Setting Up the Example DataFrame

To demonstrate these two powerful techniques in a practical context, we will first create a sample DataFrame representing statistical data for different sports teams. This example dataset contains both categorical data ('team') and numerical performance metrics ('points', 'assists', 'rebounds'). The structure of this DataFrame is essential for illustrating how column selection operates based on the defined labels.

We import the Pandas library and construct the dataset using a dictionary of lists, ensuring that each column is clearly labeled. Observing the output structure allows us to confirm the order of the columns, which is critical for correctly implementing Method 2 (range slicing).

import pandas as pd

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

#view DataFrame
print(df)

team points assists rebounds
0 A 5 11 6
1 A 7 8 7
2 A 7 10 7
3 A 9 6 6
4 B 12 6 10
5 B 9 5 12
6 B 9 9 10
7 B 4 12 9

Example 1: Selecting Non-Contiguous Columns by Label List

We now implement Method 1 to select specific performance metrics from our sample DataFrame. Suppose we are interested only in the offensive metrics of 'points' and the defensive metrics of

'rebounds', deliberately excluding 'assists' and the grouping column 'team'. Since 'points' and 'rebounds' are not adjacent in the source DataFrame (they are separated by 'assists'), using a range slice would be inappropriate. The most suitable method is to pass a list containing the exact labels we wish to retrieve.

The following code demonstrates how to use the **loc** function with a list to select only the 'points' and 'rebounds' columns. Notice that the row indexer is `:`, indicating selection of all rows, and the column indexer is the list .

#select points and rebounds columns
df.loc]

points rebounds
0 5 6
1 7 7
2 7 7
3 9 6
4 12 10
5 9 12
6 9 10
7 4 9

Crucially, observe that every row from the original DataFrame is returned, but only the two specified columns, 'points' and 'rebounds', are retained in the resulting subset. The use of a list of labels for column selection provides complete autonomy over the structure of the output.

Furthermore, a key feature of using a list for selection is that the order in which you specify the column labels dictates the order in which they appear in the resultant DataFrame. This capability is extremely useful for reporting or visualization purposes where a specific presentation sequence is required, even if it differs from the original DataFrame structure.

For example, we can easily reverse the output order to place the 'rebounds' column before the 'points' column by simply swapping their positions within the input list passed to the loc function:

#select rebounds and points columns
df.loc]

rebounds points
0 6 5
1 7 7
2 7 7

3 6 9
4 10 12
5 12 9
6 10 9
7 9 4

Example 2: Selecting Contiguous Columns via Label Slicing

Now, let's turn our attention to Method 2: using slicing with column labels to select a contiguous block of data. Based on our sample DataFrame, the columns 'points', 'assists', and 'rebounds' are sequentially positioned. If the goal is to extract all three of these columns, employing label slicing is the most elegant solution.

We initiate the slice with the starting label, 'points', and conclude it with the ending label, 'rebounds'. Because the loc accessor performs inclusive slicing, the result will contain all columns, including both the specified start and end points. The row indexer remains `:` to capture all observations.

The following code snippet executes this range selection:

#select all columns between points and rebounds columns
df.loc

points assists rebounds
0 5 11 6
1 7 8 7
2 7 10 7
3 9 6 6
4 12 6 10
5 9 5 12
6 9 9 10
7 4 12 9

The output clearly confirms that all columns residing between 'points' and 'rebounds' (inclusive of both) are returned. This includes 'points', 'assists', and 'rebounds'. This method is highly dependent on the current ordering of columns in the DataFrame; if the columns were to be rearranged, the slice would select whichever columns happen to fall within that range of labels. Therefore, while convenient, it demands attention to the internal structure of the data.

Advanced Considerations: loc vs. iloc for Column Selection

While this article focuses on the label-based indexing provided by the loc accessor, it is vital to distinguish it from its counterpart, the iloc accessor. Both are used for data subsetting, but they operate on fundamentally different principles, leading to different best-use scenarios, particularly concerning selecting multiple columns.

The iloc function handles selection strictly by integer position, starting from 0. If you wanted to select the second, third, and fourth columns (indexed 1, 2, and 3, respectively) using iloc, you would use `df.iloc]` or `df.iloc`. Notice that iloc slicing for ranges, unlike loc, follows standard Python conventions and is exclusive of the stop index.

The choice between loc and iloc generally boils down to stability and readability. loc provides greater stability: if the internal column order changes, but the column names ('labels') remain the same, your code using loc will continue to retrieve the correct data. Conversely, code relying on iloc will break or, worse, retrieve incorrect data if the column order is modified. Therefore, for robust scripts that must withstand changes in underlying data structure, **loc** is overwhelmingly preferred for column selection.

loc's dependence on string labels also significantly improves code readability, making it immediately clear to any reader exactly which data attributes are being accessed. In contrast, integer positions require cross-referencing with the DataFrame structure to determine which columns correspond to indices 1, 2, or 3.

Note: To select columns by integer index position, use the iloc function instead.

Summary and Best Practices for Multi-Column Selection

The loc accessor is the authoritative method in Pandas for selection based on column labels, offering a powerful combination of precision, stability, and clarity. Whether you need to extract disparate columns using an explicit list of labels or require an entire sequential block using label slicing, loc provides the canonical syntax.

For selecting non-contiguous columns or controlling the resulting column order, always pass a Python list of column labels: `df.loc]`. If the columns are adjacent in the DataFrame and you need all columns between a start and end point, use label slicing, remembering that the stop label is inclusive: `df.loc`.

Adopting `.loc` as the standard tool for label-based subsetting ensures that your data processing pipelines are resilient to changes in column order and maximizes the readability of your code. By internalizing these techniques, you move closer to becoming a proficient user of the Pandas

ecosystem.

How to Select Rows Based on Column Values in Pandas