Learning Pandas: Selecting Multiple Columns with loc

Authored by

Mohammed looti

March 5, 2026

Data manipulation is central to effective data analysis, and the pandas library in Python provides robust tools for this purpose. Among its most essential features is the **`loc`** indexer, which allows users to select data based on labels--a fundamentally powerful capability when working with structured data. This article focuses specifically on leveraging **`loc`** to select multiple columns within a DataFrame efficiently and clearly. Using **`loc`** ensures that your code remains readable and resistant to changes in the underlying data structure, unlike methods relying purely on integer positions.

There are two primary and highly versatile methods for using **`loc`** to extract subsets of columns. Both methods rely on the structure `df.loc`, where the colon (`:`) in the row selection position signifies that all rows should be included. The key difference lies in how the column selection is specified, offering flexibility depending on whether the columns are adjacent or scattered throughout the **DataFrame**.

Understanding the Structure of Pandas `loc` Indexer

The **`loc`** property is meticulously designed for strictly label-based indexing, meaning that both the row and column selectors must refer to the actual index names or column headings, respectively. When selecting columns, we utilize the second argument of the **`loc`** indexer. To select all rows while focusing on specific columns, we universally employ the slice object `:` for the row indexer. This approach is highly efficient as it instructs pandas to include every record in the **DataFrame** for the resulting column subset.

We will examine the two standard syntaxes for powerful column selection. The first method uses a Python list of column names, which is the most flexible approach, ideal for selecting non-contiguous columns or for reordering existing columns. The second method leverages standard Python slicing syntax to select a range of adjacent columns based on their starting and ending labels, providing a concise method for extracting contiguous subsets.

These two techniques cover nearly every scenario required for column selection using labels:

Method 1: Select Multiple Columns by Name

df.loc]

Method 2: Select All Columns in Range

df.loc

Prerequisite: Setting Up the Sample DataFrame

To demonstrate these methods practically, we will utilize a sample **DataFrame** representing fictional sports statistics. This dataset includes various key metrics such as team identification, points scored, assists recorded, and rebounds obtained. Understanding the structure and column names of this initial **DataFrame** is the fundamental first step before performing any selection operations, as `loc` relies entirely on these specific string labels.

The following code block imports the necessary pandas library and constructs the sample data. We are creating a **DataFrame** with eight rows of data, distributed across four distinct columns. Note the column labels: `team`, `points`, `assists`, and `rebounds`. These exact strings will serve as the label inputs for all subsequent `loc` calls.

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

The resulting **DataFrame**, `df`, is now ready for analysis. We can clearly see the structure: the `team` column is first, followed by the numerical metrics. We will now proceed to use the `loc` indexer to efficiently subset these columns based on explicit label referencing, starting with the non-contiguous selection method.

Method 1: Selecting Multiple Columns Using a List of Labels

When you need to select specific columns that are scattered across the **DataFrame** or require them in a particular order, passing a standard Python list of column names to **`loc`** is the ideal approach. This technique is highly flexible because the columns do not need to be adjacent in the source data. The syntax uses the format `df.loc]`.

For instance, if our analysis requires us to focus only on the `points` scored and the `rebounds` obtained, excluding the `team` and `assists` data, we simply supply those two column names in a list. The code below demonstrates this operation, resulting in a new **DataFrame** that preserves all original rows but includes only the requested columns.

#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

A significant advantage of this list-based method is the precise control it offers over column sequencing. The order in which you define the column names within the Python list dictates the final arrangement of columns in the returned **DataFrame**. This feature is particularly useful when preparing data for visualization tools or machine learning models that expect inputs in a specific, required sequence.

To illustrate this reordering capability, we can simply reverse the list of column names. Even though `points` precedes `rebounds` in the original **DataFrame**, we can force `rebounds` to appear first in the output by adjusting the list provided to **`loc`**. This capability ensures that data presentation is always tailored exactly to the analytical need.

#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

Method 2: Selecting a Contiguous Range of Columns

When the desired columns are adjacent within the **DataFrame** structure, using Python's slicing notation combined with label referencing provides the most concise and efficient method. This technique avoids the manual listing of numerous column names, making the code cleaner and easier to maintain when dealing with large blocks of sequential variables.

The syntax for range selection is `'start_label':'end_label'`. A critical feature of **`loc`** label-based slicing, which is a key distinction from standard Python integer slicing, is that the ending label is **inclusive**. This means that both the starting column and the ending column, along with all columns positioned strictly between them, will be included in the resultant subset **DataFrame**. This design choice simplifies range selection when working with column names.

In our sample **DataFrame**, the columns `points`, `assists`, and `rebounds` are all adjacent. If we wish to select this entire contiguous block of statistical metrics, we can specify the starting column (`points`) and the ending column (`rebounds`) using the slicing operator. The operation ensures that `assists` is automatically included in the resulting output.

#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

As confirmed by the output, all columns between `points` and `rebounds` in the original **DataFrame**,

including the specified endpoints, are successfully returned. This method proves invaluable for extracting logical groupings of data, such as a time series of metrics or a set of calculated features, in a single line of code.

Key Considerations: `loc` vs. `iloc` for Column Selection

While **`loc`** is the definitive method for selecting columns by name (label), it is often contrasted with its counterpart indexer, **`iloc`**. Understanding the fundamental distinction between these two indexers is paramount for writing robust and stable pandas code. Specifically, **`loc`** operates exclusively on labels (string names or index objects), whereas **`iloc`** operates exclusively on zero-based integer positions (indices).

If your requirement involves selecting columns based strictly on their numerical position--for example, selecting the first column (index 0) and the third column (index 2), regardless of their current names--you must use **`iloc`**. The usage pattern for **`iloc`** mirrors that of **`loc`** in structure, but it accepts integers or lists of integers instead of string labels for the column selector. For instance, if we wanted the `points` (position 1) and `rebounds` (position 3) using indices, the command would be `df.iloc]`.

The choice between **`loc`** and **`iloc`** should always prioritize code stability and clarity. If the column names are descriptive and fixed (e.g., `'points'`), using **`loc`** is highly recommended, as it prevents code failure if a column is inserted earlier in the **DataFrame**. Conversely, if you are reading a source file where only the position is guaranteed (e.g., the third column is always the target variable), then **`iloc`** is the safer choice. For most analytical tasks, relying on **`loc`** and explicit labels leads to more maintainable code.

Summary and Next Steps

Leveraging the **`loc`** indexer in pandas provides powerful, label-based mechanisms for selecting columns, whether you require a custom-ordered list of non-contiguous columns or a contiguous range of variables. By using the appropriate syntax--either a list of strings or the inclusive label-based slice--data professionals can efficiently subset large **DataFrames**, making data preparation tasks faster and more reliable.

For those looking to expand their skills beyond column selection, the next logical step is to master row selection and conditional filtering, which often works in tandem with column selection using **`loc`**.

How to Select Rows Based on Column Values in Pandas

Note: To select columns based on their numerical index position rather than their label, utilize the

`iloc` function instead.