Learn How to Select Columns by Name in Pandas DataFrames: A Comprehensive Guide with Examples

Authored by

Mohammed looti

October 28, 2025

Introduction to Column Selection in Pandas

The ability to efficiently select and manipulate specific subsets of data is fundamental to modern data analysis. When working with Python, the **Pandas library** serves as the industry standard for handling structured data, primarily through the use of the **DataFrame** object. A key task for any data scientist is isolating relevant variables or features, which necessitates accurate column selection.

While Pandas provides several methods for retrieving columns (such as simple bracket notation or attribute access), using the `.loc` accessor for selection by name offers unparalleled clarity and robustness. This method ensures that your code remains functional even if the underlying positional index of the columns changes, as it relies purely on the defined column labels.

This guide focuses specifically on utilizing the `.loc` indexer to target columns by their explicit names. We will explore three essential techniques that cover most common selection needs: selecting a single column, selecting multiple discontiguous columns, and selecting a contiguous range of columns. Mastering these methods will significantly enhance your data preparation workflow when using a Pandas DataFrame.

Understanding the Pandas `.loc` Accessor

The `.loc accessor` is Pandas' primary label-based indexing method. It is designed to allow selection based on the row and column labels (names), rather than their integer positions. This distinction is crucial for writing stable, readable code. The fundamental syntax for `.loc` is `df.loc`.

In the context of column selection, we are typically interested in retaining **all rows** while specifying which columns to keep. To instruct the `.loc accessor` to select every single row, we utilize the standard Python slicing symbol, the colon (`:`), in the row indexer position. Therefore, all column selections by name begin with `df.loc`.

The power of the `.loc accessor` lies in its flexibility in handling the column indexer argument. Depending on whether you pass a single string, a list of strings, or a slice using strings, the behavior changes to accommodate the three distinct selection requirements demonstrated below. By passing the column names as labels, we ensure that the selection is explicit and independent of the physical arrangement of the data.

The following three methods demonstrate how to select columns using the label-based approach:

Method 1: Select One Column by Name

df.loc

Method 2: Select Multiple Columns by Name

df.loc]

Method 3: Select Columns in Range by Name

df.loc

We will now illustrate these methods using a concrete Pandas DataFrame created below. This sample data represents fictional scores for five different basketball teams over seven time periods (rows).

import pandas as pd

#create DataFrame
df = pd.DataFrame({'mavs': ,
'cavs': ,
'hornets': ,
'spurs': ,
'nets': })

#view DataFrame
print(df)

mavs cavs hornets spurs nets
0 10 18 5 10 10
1 12 22 7 12 14
2 14 19 7 14 25
3 15 14 9 13 22
4 19 14 12 13 25
5 22 11 9 19 17
6 27 20 14 22 12

Example 1: Selecting a Single Column by Name

The most straightforward column selection involves isolating a single variable for analysis. While standard bracket notation (e.g., `df`) is common, using `.loc` ensures consistency across all your label-based selection operations, making your code highly predictable. This method is essential when you need to extract a specific feature to perform univariate statistical analysis or visualization.

To select just the 'spurs' column, we use the following syntax: `df.loc`. The first argument, `:`, tells Pandas to include every row index from the original DataFrame. The second argument, `'spurs'`, explicitly names the column label we wish to retrieve. Note that since we are only requesting one label, we pass it as a simple string, not enclosed in a list.

When a single column is selected using this method, Pandas returns a **Series object**, which is essentially a one-dimensional labeled array. This Series retains the original index of the DataFrame, allowing it to be easily re-merged or used for calculations relative to the full dataset.

The following code shows how to select the 'spurs' column in the DataFrame:

#select column with name 'spurs'
df.loc

0 10
1 12
2 14
3 13
4 13
5 19
6 22
Name: spurs, dtype: int64

As shown in the output, only the values from the 'spurs' column are returned, along with the column name (Name: spurs) and its data type (dtype: int64).

Example 2: Selecting Multiple Discontiguous Columns

Often, data tasks require working with several, but not all, columns simultaneously--for instance, selecting predictor variables for a model while excluding the target variable or ID fields. To select multiple columns that are not necessarily adjacent to each other, you must pass a Python list of column labels to the column indexer position of `.loc`.

This approach is highly flexible as the order of the columns in the resulting subset is determined by the order defined in the list you pass. For example, if you list , the resulting DataFrame will show the 'nets' column first, followed by 'cavs', regardless of their original arrangement in the source Pandas DataFrame.

Crucially, when selecting multiple columns, even if only two, the result is always a new **DataFrame object**, not a Series. This maintains the two-dimensional structure necessary for multi-column operations. Remember the syntax requirement: the list of column names must be enclosed within

the outer square brackets of the `.loc` method, resulting in double brackets (e.g., `df.loc]`).

The following code shows how to select the 'cavs', 'spurs', and 'nets' columns in the DataFrame:

#select columns with names cavs, spurs, and nets
df.loc]

cavs spurs nets
0 18 10 10
1 22 12 14
2 19 14 25
3 14 13 22
4 14 13 25
5 11 19 17
6 20 22 12

Only the values from the 'cavs', 'spurs', and 'nets' columns are returned, forming a new DataFrame subset.

Example 3: Selecting a Range of Columns by Name

When the desired columns are contiguous (next to each other) in the DataFrame, the most concise and efficient method is using label-based slicing with `.loc`. This technique is particularly useful in datasets where related features are grouped together, such as time series measurements or sequential categorical variables.

To perform range selection by name, you use the slice notation `'start_column':'end_column'` within the column indexer. A critical difference between label-based slicing in Pandas (used by `.loc`) and standard Python integer slicing is inclusivity. When using column names with `.loc`, **both the start and end column labels are included** in the resulting selection.

For instance, if you define the slice as `'hornets':'nets'`, Pandas will return the 'hornets' column, the 'nets' column, and all columns physically located between them in the underlying Pandas DataFrame structure. This method eliminates the need to manually list every column name, streamlining code for large datasets.

The following code shows how to select all columns between the names 'hornets' and 'nets' in the DataFrame:

#select all columns between hornets and nets
df.loc

hornets spurs nets
0 5 10 10
1 7 12 14
2 7 14 25
3 9 13 22
4 12 13 25
5 9 19 17
6 14 22 12

As expected, all of the columns between the names 'hornets' and 'nets'--including both endpoints--are returned successfully.

Best Practices and Conclusion

Selecting columns by name using the `.loc` accessor is highly recommended for professional data scripting due to its explicit, label-based nature. While other methods like direct bracket notation or `.iloc` exist, `.loc` ensures that your code operates strictly based on the labels you define, making it robust against common data manipulation issues like index rearrangement.

To summarize the best use cases for each method:

Use `df.loc` when you need to extract a single variable, resulting in a Pandas Series.
Use `df.loc]` when you need a specific, potentially non-adjacent subset of columns, resulting in a new DataFrame.
Use `df.loc` when the columns are contiguous and you wish to leverage the label-inclusive slicing feature.

By implementing these techniques, you gain precise control over your data subsets, a prerequisite for cleaning, transforming, and modeling complex data structures efficiently.

Additional Resources

The following tutorials explain how to perform other common tasks in pandas: