Learning How to Slice Columns in Pandas DataFrames: A Comprehensive Guide

Authored by

Mohammed looti

October 27, 2025

Fundamentals of Column Selection in Pandas DataFrame

Selecting, or slicing, specific columns from a large dataset is a fundamental operation in data analysis using the **Pandas library** in Python. Whether you are preparing data for machine learning, generating specific reports, or simply cleaning up a messy dataset, the ability to accurately and efficiently subset your data is critical. Pandas offers highly flexible and powerful indexing methods to achieve this, primarily categorized into two approaches: selection by **labels (names)** and selection by **integer positions**.

This guide will walk you through four distinct methods for slicing columns, providing practical examples for each. These methods rely on two core Pandas accessors: `.loc` and `.iloc`. Understanding the nuances between these two tools is essential for mastery, as they serve fundamentally different purposes regarding data access.

The four primary methods for achieving column slicing in a Pandas DataFrame are summarized below, illustrating the syntax before diving into the live examples.

Summary of Column Slicing Syntax

Here is a quick reference for the various techniques used for column selection, focusing on how the accessors handle the column argument (the second parameter after the comma in the slice notation):

Method 1: Slice by Specific Column Names (Non-Contiguous)

df_new = df.loc]

Method 2: Slice by Column Names in Range (Contiguous)

df_new = df.loc

Method 3: Slice by Specific Column Index Positions (Non-Contiguous)

df_new = df.iloc]

Method 4: Slice by Column Index Position Range (Contiguous)

df_new = df.iloc

Understanding loc and iloc for Data Selection

The core distinction in Pandas indexing rests on whether you are referencing data using explicit **labels** (like column names or index names) or implicit **integer positions** (starting from zero). The colon `:` used as the first argument in `df.loc` signifies that we are selecting **all rows**, thus focusing our selection entirely on the columns.

The `.loc` accessor is strictly **label-based**. When slicing columns using `.loc`, you must provide the actual names of the columns as strings. Crucially, when using a range slice (e.g., `'col1':'col4'`), `.loc` is **inclusive** of both the start and end labels. This is highly beneficial when you need to select a block of columns where you know the boundary names.

Conversely, the `.iloc` accessor is strictly **integer position-based**. It treats the DataFrame as a large array, relying on the default Python indexing convention where the first column is position 0, the second is position 1, and so on. When using a range slice (e.g., `0:3`), `.iloc` follows standard Python slicing rules, meaning the selection is **exclusive** of the stop position.

Preparing the Example Pandas DataFrame

To demonstrate these four methods effectively, we will utilize a sample DataFrame containing performance statistics for eight different teams. This DataFrame includes a mix of string columns (`team`) and numerical columns (`points`, `assists`, etc.).

The following code block sets up our base DataFrame, which we will slice in the subsequent examples. Note that the column names serve as the **labels**, while their position (0 through 5) serves as the **integer index**.

import pandas as pd

#create DataFrame with six columns
df = pd.DataFrame({'team': ,
'points': ,
'assists': ,
'rebounds': ,
'steals': ,
'blocks': })

#view DataFrame
print(df)

team points assists rebounds steals blocks
0 A 18 5 11 4 1

1 B 22 7 8 3 0
2 C 19 7 10 3 0
3 D 14 9 6 2 3
4 E 14 12 6 5 2
5 F 11 9 5 4 2
6 G 20 9 9 3 1
7 H 28 4 12 8 5

Slicing Columns Using loc (Label-Based Indexing)

Example 1: Slice by Specific Column Names

When you need to select a subset of columns that are not necessarily adjacent to one another, passing a list of the column **labels** to the `.loc` accessor is the most straightforward technique. This is particularly useful for feature engineering where only certain attributes are required. In this example, we isolate the identifying column (`team`) and a key performance metric (`rebounds`).

The syntax `df.loc]` specifies that we want all rows (`:`) and only the columns explicitly listed in the Python list.

#slice columns team and rebounds
df_new = df.loc]

#view new DataFrame
print(df_new)

team rebounds
0 A 11
1 B 8
2 C 10
3 D 6
4 E 6
5 F 5
6 G 9
7 H 12

Example 2: Slice by Column Names in Range

If the required columns form a continuous block within the DataFrame, using a range slice with `.loc` significantly simplifies the code compared to listing every single column name. We can select

all columns starting from `team` and ending with `rebounds` by providing the start and stop labels separated by a colon.

A key advantage of `.loc` is its **inclusivity** when slicing ranges. The resulting DataFrame will contain `team`, `points`, `assists`, and `rebounds`--the column specified as the endpoint is always included in the result. This behavior contrasts directly with standard Python list slicing.

#slice columns between team and rebounds
df_new = df.loc

#view new DataFrame
print(df_new)

team points assists rebounds
0 A 18 5 11
1 B 22 7 8
2 C 19 7 10
3 D 14 9 6
4 E 14 12 6
5 F 11 9 5
6 G 20 9 9
7 H 28 4 12

Slicing Columns Using iloc (Positional Indexing)

Example 3: Slice by Specific Column Index Positions

The `.iloc` accessor is deployed when column names might be unknown, dynamically generated, or simply when working directly with the physical arrangement of the data. Similar to `.loc`, we can select non-contiguous columns by passing a list, but this time containing the **integer positions** of the desired columns.

To select the `team` column (position 0) and the `rebounds` column (position 3), we pass the list . This technique is robust even if column names change, provided their relative positions remain constant.

#slice columns in index positions 0 and 3
df_new = df.iloc]

#view new DataFrame
print(df_new)

team rebounds
0 A 11
1 B 8
2 C 10
3 D 6
4 E 6
5 F 5
6 G 9
7 H 12

Example 4: Slice by Column Index Position Range

For selecting a contiguous block of columns using their numerical positions, we use the range slice notation with `.iloc`. This is equivalent to Example 2, but utilizes integer indices instead of labels.

When we use the range `0:3`, we are instructing Pandas to select columns starting at position 0 up to, but **not including**, position 3. This means the resulting DataFrame will contain columns at positions 0, 1, and 2 (`team`, `points`, and `assists`). The column at index position 3 (`rebounds`) is excluded, illustrating the standard Python slicing convention.

#slice columns in index position range between 0 and 3
df_new = df.iloc

#view new DataFrame
print(df_new)

team points assists
0 A 18 5
1 B 22 7
2 C 19 7
3 D 14 9
4 E 14 12
5 F 11 9
6 G 20 9
7 H 28 4

Note: When using an index position range with `.iloc`, the last index position in the range will not be included. For example, the `rebounds` column at index position 3 is not included in the new DataFrame created by `0:3`.

Key Differences and Best Practices

Choosing between `.loc` and `.iloc` often depends on the stability and nature of your data structure. If you are dealing with data where column order might shift (e.g., merging different datasets or adding new features dynamically), relying on **column names using `.loc`** is generally safer and more readable, as the code remains valid regardless of positional changes.

Conversely, `.iloc` is invaluable when processing data generated by automated pipelines where you know the exact order of columns, or when performing iterative processes that rely on stepping through indices. Remember the critical difference: `.loc` ranges are **inclusive** of the end label, while `.iloc` ranges are **exclusive** of the end position.

For most analytical tasks, especially those requiring clarity and maintainability, experts recommend prioritizing the use of `.loc` with explicit column names. This practice makes the resulting code self-documenting, as the reader immediately understands which fields are being selected without needing to cross-reference the DataFrame's positional structure.

Additional Resources for Pandas Mastery

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