Learning to Select Multiple Columns in Pandas DataFrames: A Comprehensive Guide

Authored by

Mohammed looti

November 2, 2025

The Pandas library is the cornerstone of data analysis and manipulation in Python. A fundamental task when working with tabular data is selecting specific subsets of columns from a larger DataFrame. Whether you are performing preliminary data cleaning or preparing a dataset for advanced statistical modeling, mastering various column selection techniques is crucial for efficiency.

This guide explores the three primary and most efficient methods available in Pandas for selecting multiple columns simultaneously. These techniques leverage different forms of indexing, allowing users to choose columns based on their positional index or their assigned label (name).

We will demonstrate each approach using a clear, consistent example. The methods covered include:

Method 1: Selecting non-contiguous columns using positional indexing (`.iloc`).
Method 2: Selecting a continuous range of columns using positional slicing (`.iloc`).
Method 3: Selecting columns explicitly by name (List Notation).

Below is a quick reference for the three selection mechanisms:

Method 1: Select Columns by Positional Index (Non-Contiguous)

df_new = df.iloc]

Method 2: Select Columns in Index Range (Positional Slicing)

df_new = df.iloc

Method 3: Select Columns by Explicit Name

df_new = df]

Setting Up the Example DataFrame

To illustrate these concepts clearly, we will first create a sample Pandas DataFrame containing fictional basketball statistics. This DataFrame, named `df`, includes four columns representing player metrics: `points`, `assists`, `rebounds`, and `blocks`.

It is important to note the zero-based positional indexing of the columns, which is critical when using the `.iloc` accessor for positional selection:

`points`: Index 0

`assists`: Index 1
`rebounds`: Index 2
`blocks`: Index 3

The following Python code initializes and displays our working dataset:

import pandas as pd

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

#view DataFrame
df

points assists rebounds blocks
0 25 5 11 4
1 12 7 8 7
2 15 7 10 7
3 14 9 6 6
4 19 12 6 5
5 23 9 5 8
6 25 9 9 9
7 29 4 12 10

With our sample data prepared, we can now proceed to explore how each column selection method operates on this structure.

Method 1: Select Columns by Positional Index (`.iloc`)

The `.iloc` accessor in Pandas stands for **integer-location based indexing**. This means you select data based purely on the numerical position (zero-based index) of the rows and columns. This method is exceptionally useful when the column names are unknown, cumbersome, or when you need to select columns that are not adjacent to each other.

To select columns using `.iloc`, the syntax requires two components separated by a comma: `df.iloc`. To select all rows, we use the slicing operator `:` for the row component. For the columns, we pass a Python **list** of the specific zero-based indices we wish to retrieve.

For instance, if we want to retrieve the `points` (Index 0), `assists` (Index 1), and `blocks` (Index 3) columns, we specify the list for the column index parameter. Notice how `rebounds` (Index 2) is intentionally skipped, demonstrating the non-contiguous selection capability of this method.

The following code shows how to select columns in index positions 0, 1, and 3:

#select columns in index positions 0, 1, and 3
df_new = df.iloc]

#view new DataFrame
df_new

points assists blocks
0 25 5 4
1 12 7 7
2 15 7 7
3 14 9 6
4 19 12 5
5 23 9 8
6 25 9 9
7 29 4 10

The resulting DataFrame, `df_new`, successfully contains only the columns corresponding to the specified non-contiguous indices. This demonstrates the power of list-based positional indexing.

Method 2: Select Columns in Index Range (Positional Slicing)

When you need to select a sequence of contiguous columns, using Python's standard slicing notation within the `.iloc` accessor is the most concise and efficient method. This approach leverages the `start:stop` syntax, which is fundamental to sequence manipulation in Python.

A crucial rule of Python slicing applies here: the start index is **inclusive**, but the stop index is **exclusive**. If you specify a range of `0:3`, Pandas will select indices 0, 1, and 2, but will stop just before reaching index 3.

If our goal is to select the first three columns (`points`, `assists`, and `rebounds`), which correspond to indices 0, 1, and 2, we must define the range as `0:3`. If we wanted to include the `blocks` column (Index 3), the range would be `0:4`.

The following code demonstrates how to select columns starting from index 0 up to (but not including) index 3:

#select columns in index range 0 to 3 (0, 1, 2)
df_new = df.iloc

#view new DataFrame
df_new

points assists rebounds
0 25 5 11
1 12 7 8
2 15 7 10
3 14 9 6
4 19 12 6
5 23 9 5
6 25 9 9
7 29 4 12

This method is significantly cleaner than writing out a long list of individual indices, provided the columns you need are sequentially ordered within the DataFrame. It is particularly valuable when dealing with datasets that have dozens or hundreds of columns.

Method 3: Select Columns by Name (List Notation)

The most intuitive and readable way to select columns is by using their explicit names (labels). This approach is label-based, making it fundamentally different from the positional indexing used by `.iloc`. It relies entirely on the column labels defined by the user.

To select a single column in Pandas, you use single bracket notation: `df`. However, to select **multiple columns**, you must pass a Python list of the required column names within the selection brackets. This requirement results in the common **double bracket notation**: `df]`.

This approach offers superior code readability, as the script clearly states which specific data fields are being retrieved, regardless of their position within the dataset. It is the preferred method for scripts that need to be maintained or shared with others.

The following code shows how to select the `points` and `blocks` columns by their explicit names:

#select columns called 'points' and 'blocks'
df_new = df]

#view new DataFrame
df_new

points blocks
0 25 4
1 12 7
2 15 7
3 14 6
4 19 5
5 23 8
6 25 9
7 29 10

This method is generally preferred for production data analysis scripts because it makes the code resistant to changes in column order. If the underlying data source reorders the columns, selecting by name ensures the correct data is always extracted, whereas positional indexing might inadvertently select the wrong data.

Summary of Selection Techniques

Choosing the correct method for column selection depends entirely on the context of your data manipulation task. We have demonstrated that the Pandas library provides flexible tools for nearly every scenario:

Use **Method 3 (By Name)** when readability and stability against column reordering are paramount. This is the safest choice for ETL processes.
Use **Method 1 or 2 (`.iloc`)** when working with datasets where column names may not be consistent, or when you need to quickly access data based on its physical order, such as selecting the first `N` columns.
Remember that `.iloc` is purely positional and relies on zero-based indexing, while bracket notation (Method 3) is label-based.

Mastering these fundamental selection techniques is essential for performing efficient and robust data cleaning and preparation workflows in Pandas.

Additional Resources

To further enhance your skills in data manipulation, explore these related tutorials that explain how to perform other common operations in Pandas: