How to Easily Select Multiple Columns in Pandas DataFrames

Authored by

stats writer

December 3, 2025

The ability to efficiently select and manipulate data subsets is fundamental to effective data analysis using Pandas. When working with complex datasets, you often need to isolate specific features or fields--meaning you must select multiple columns from a DataFrame. Pandas offers several robust and flexible mechanisms for achieving this, primarily relying on indexers like `.loc` and `.iloc`, as well as standard Python list syntax.

Understanding these methods is crucial for writing clean, readable, and performant code. Whether you need to select columns based on their explicit names (labels) or their numeric positions (integers), Pandas provides specialized tools tailored for the task. This guide will explore the most common and effective ways to select multiple columns, detailing the underlying principles of label-based versus position-based indexing and providing practical, reproducible examples for each technique.

The core concept revolves around passing a list of desired column identifiers (either names or integer indices) directly to the DataFrame or its indexers. When a list is used, the returned result is always a new DataFrame composed exclusively of the selected columns, ready for subsequent operations such as filtering, transformation, or aggregation. Mastery of these selection techniques is the first step toward advanced data processing in Python.

Understanding Pandas Indexing: .loc vs. .iloc

Before diving into specific selection methods, it is vital to distinguish between the two primary DataFrame indexers: `.loc` and `.iloc`. These methods govern how Pandas interprets your selection criteria--whether you are referencing the column by its **explicit label** or its **implicit integer position**. Using the correct indexer ensures accurate and predictable data retrieval, particularly when column headers might be numeric or contain special characters.

The `.loc` attribute is strictly **label-based**. When selecting columns using `.loc`, you must provide the exact column names (e.g., 'points', 'assists'). This indexer is inclusive of both the start and stop labels when using Python slicing syntax, which is a key differentiator from standard Python list slicing. Conversely, the `.iloc` indexer is **integer position-based**. It uses the zero-based index of the columns, treating the columns like elements in a list. This means if you have four columns, their positions are 0, 1, 2, and 3. Importantly, `.iloc` follows standard Python slicing syntax, meaning the start index is inclusive, but the stop index is exclusive.

When selecting multiple columns, both `.loc` and `.iloc` require two primary arguments: the selection for the rows (the first position before the comma) and the selection for the columns (the second position after the comma). To select all rows while specifying column subsets, we use the colon operator (`:`) in the row position. This colon effectively tells Pandas, "include everything in this axis."

Setting Up the Example DataFrame

To illustrate the three primary methods for column selection, we will use a small sample DataFrame detailing player statistics. This example will provide a clear, visual context for how each code snippet manipulates the data structure. Remember that consistent naming and structure in your initial data setup is crucial for reliable selection operations downstream.

We begin by importing the Pandas library and defining our dataset. Our example DataFrame, named `df`, contains four columns: `points` (position 0), `assists` (position 1), `rebounds` (position 2), and `blocks` (position 3). Note the distinct labels and their corresponding zero-based index positions.

The following code block generates the data we will use throughout the remaining examples. You can run this code in your environment to follow along precisely with the outputs shown for each selection method.

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

This DataFrame `df` is the baseline for all subsequent selection operations. Pay close attention to the indices (0-7 for rows) and the column position mapping (0 for 'points', 1 for 'assists', 2 for 'rebounds', 3 for 'blocks') as we apply the different selection methods.

Method 1: Selecting Disparate Columns Using Integer Positions (.iloc)

The first powerful technique involves using the `.iloc` indexer to select columns based on their integer positions, even if those positions are non-contiguous. This method is particularly useful when you need to select the first, second, and fourth columns, regardless of their specific names. By relying on positional indexing, you gain flexibility when dealing with dynamically generated column names or when the order of columns is the critical factor.

To select multiple, non-sequential columns using `.iloc`, you must pass a Python list of integers as the column selector argument. As mentioned previously, we use `:` to select all rows. If we want to retrieve 'points' (position 0), 'assists' (position 1), and 'blocks' (position 3), we pass the list to the column dimension of `.iloc`. Notice that 'rebounds' (position 2) is intentionally skipped, demonstrating the non-contiguous nature of this selection capability.

The following code demonstrates how to execute this selection. The result, `df_new`, is a new DataFrame containing only the columns corresponding to the specified integer indices. This confirms that `.iloc` effectively isolates columns based solely on their order within the original DataFrame structure.

#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

Observe that the column 'rebounds' is absent from `df_new`. This confirms that `.iloc` successfully used the provided list of integers to construct the resulting DataFrame subset.

Method 2: Selecting Contiguous Columns Using Integer Ranges (.iloc Slicing)

When the columns you wish to select are located next to each other in a continuous block, using

Python slicing syntax with `.iloc` provides the most concise and efficient solution. Slicing eliminates the need to manually list every index number, making the code cleaner, especially when dealing with dozens of sequential columns.

As standard for Python slicing syntax, the format is , where the `start` index is inclusive and the `stop` index is exclusive (i.e., the column at the `stop` position is not included). To select the first three columns ('points', 'assists', 'rebounds'), which occupy positions 0, 1, and 2, we must define the range from 0 up to, but not including, 3. Thus, the slice selector is `0:3`.

Applying this slice to the column dimension of `.iloc` extracts the desired block of columns. This technique is computationally fast and highly readable, making it the preferred method for adjacent column selection when using positional indexing. It is a fundamental technique every Pandas user should master for quick data subsetting.

#select columns in index range 0 to 3 (exclusive of 3)
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

The resulting DataFrame `df_new` contains 'points', 'assists', and 'rebounds'. The final column, 'blocks' (position 3), is correctly excluded, adhering to the standard conventions of Python slicing.

Method 3: Selecting Columns by Explicit Name List (Standard Indexing)

The most straightforward and generally recommended method for selecting multiple columns is by using their explicit names. This approach enhances code clarity and stability because it is independent of the column order; even if the columns are rearranged in the source data, the code will still select the correct features, provided their labels remain unchanged. Since this method relies on labels rather than positions, we use standard Python indexing notation on the DataFrame itself, bypassing the need for `.loc` or `.iloc` when selecting all rows.

To select multiple columns by name, you pass a list of strings--where each string is the exact column header--directly inside the square brackets of the DataFrame. Note the use of double square brackets: the outer brackets denote the selection operation on the DataFrame, and the inner brackets define the Python list of column names being passed for selection. This syntax is concise and highly idiomatic in Pandas.

For our example, if we want to isolate 'points' and 'blocks', we construct the list and apply it to `df`. This method is preferred in production environments because it makes the code robust against structural changes like reordering columns. The explicit naming clearly documents which features are being used.

#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

The output `df_new` successfully includes only the 'points' and 'blocks' columns, demonstrating the simplicity and effectiveness of label-based list selection.

Advanced Selection: Using .loc for Label Slicing

While standard list indexing (Method 3) is great for disparate column selection by name, the `.loc` indexer offers a powerful alternative for selecting a contiguous range of columns using their labels. Unlike `.iloc`, `.loc` slicing is **inclusive** of both the starting and ending labels, making it highly intuitive for label-based operations.

If you wanted to select all columns starting from 'points' up to and including 'rebounds', you would use the slice `'points':'rebounds'` in the column selection argument of `.loc`. This selection method is particularly useful when dealing with data schemas where columns are grouped logically, such as time series data where you need all features between two specific dates.

The structure is `df.loc`. Using our example DataFrame, if we want the columns from 'points' through 'rebounds', the code looks like this. This illustrates how label-based slicing provides excellent semantic clarity in the code, linking directly to the meaning of the data:

#select columns from 'points' through 'rebounds' (inclusive)
df_new_loc = df.loc

#view new DataFrame
df_new_loc

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 successfully retrieves 'points', 'assists', and 'rebounds'. Because `.loc` includes the stop label, this achieved the same result as `df.iloc`, but using explicit names rather than abstract integer positions.

Conclusion: Choosing the Right Selection Technique

Selecting multiple columns in Pandas is a fundamental skill, and the choice of method should be guided by context: whether you rely on column names or their positions, and whether the columns are contiguous or disparate. For maximum code readability and resilience against structural changes, **Method 3 (selecting by explicit name list)** is generally the preferred approach.

Here is a quick summary of when to use each primary technique:

List of Names (`df]`): Use for non-contiguous columns when labels are known. This is the most robust and readable method.
Label Slicing (`df.loc`): Use for contiguous columns when labels are known and you prioritize clarity over positional dependence. Remember that this method is inclusive.
List of Indices (`df.iloc]`): Use when columns have been dynamically generated or when you strictly need to select the Nth column regardless of its label.
Index Slicing (`df.iloc`): Use for contiguous columns when using positional indexing is faster or when labels are unknown. Remember that this method is exclusive of the stop index.

By mastering these various column selection strategies, you ensure that your data preparation steps are efficient, accurate, and easily maintained. Always strive to use explicit labels where possible, reserving positional indexing for situations where structural order is the primary determinant of selection.