How to Easily Keep Specific Columns and Drop All Others in Pandas

Authored by

stats writer

November 25, 2025

Column selection and subsetting are fundamental operations when working with data analysis in Python, particularly within the Pandas library. When preparing data for analysis or modeling, you often need to isolate a few key variables from a much larger dataset. While the standard `drop()` method is excellent for removing a few specified columns, efficiency dictates using selection techniques when the goal is to **keep** only a small subset of columns. This comprehensive guide explores the most effective and idiomatic ways to retain specific columns while discarding all others within a DataFrame.

We will focus on two primary methods that utilize indexing capabilities within Pandas: the straightforward approach using double brackets for list-based selection, and the powerful, label-based method using .loc. Both techniques achieve the desired result--a new DataFrame containing only the specified columns--but offer different advantages in terms of readability and flexibility.

Understanding Column Subsetting in Pandas

The core challenge of dropping all columns except a specific selection is best addressed not by explicitly listing all columns to be removed. Instead, the most efficient approach in Pandas is known as column subsetting or selection. This involves telling the DataFrame exactly which columns you wish to retain, and implicitly, everything else is discarded in the resulting object. This process is significantly faster and less prone to error than calculating the complement list of columns to drop, especially when dealing with hundreds of variables.

When we talk about column subsetting, we are utilizing the concept of indexing, which allows us to access a subset of data by labels (column names) or by positional integers. Since Pandas DataFrames are built on the foundations of Python list structures, accessing columns often involves passing a list of the desired column names. This list of names acts as a mask, filtering the original DataFrame to produce a view or a copy containing only the selected data attributes.

While the standard Pandas `drop()` method is widely used for explicit column removal, using it to keep specific columns requires passing the inverse list of columns (i.e., `df.drop(columns=)`). This inversion step is cumbersome and unnecessary. Therefore, the methods detailed below--double brackets and .loc--represent the standard best practice for column retention.

Method 1: Direct Column Selection Using Double Brackets

The simplest and most commonly used method for retaining specific columns in a DataFrame is through direct selection using double square brackets (`[]`). In Pandas, a single set of brackets is used to select a single column, which returns a Pandas Series. However, when we pass a list of column labels (names) within a second set of brackets, the result is a new DataFrame containing only those specified columns. This double-bracket notation is essential because it maintains the two-dimensional structure of the resulting object.

This technique is highly intuitive for Python developers, as it mirrors list-based selection patterns common across the language. The syntax requires defining a Python list containing the exact names of the columns you wish to retain. When this list is passed to the Pandas DataFrame, all columns not present in that list are effectively excluded from the resulting object. This is often the preferred method for its brevity and clarity when the row indexing (or row subsetting) is not a concern.

The general structure of this operation is straightforward: you reassign the existing DataFrame variable (`df`) to the result of the selection, effectively overwriting the original object with the desired subset. If you need to keep the original DataFrame intact, ensure you assign the result to a new variable.

Below illustrates the concise syntax for retaining only 'col2' and 'col6':

df = df]

Method 2: Advanced Selection Using .loc Indexer

For more sophisticated indexing needs, Pandas provides the attribute-based indexers, most notably .loc (label-based indexing) and `.iloc` (integer position-based indexing). The .loc indexer allows for selection based on labels (names) for both rows and columns, making it extremely powerful and explicit regarding data access. When using .loc for column retention, we specify that we want all rows, and only a list of specific columns.

The syntax for .loc always follows the pattern `df.loc`. To select all rows, we use the Python slice operator, the colon (`:`), in the row selector position. In the column selector position, we provide the list of column names we wish to keep. This explicit structure enhances code readability, especially when combining column retention with complex row filtering (e.g., keeping only certain columns for rows where a condition is met).

While using double brackets is slightly shorter, many experienced Pandas users prefer .loc because it clearly distinguishes between row and column selection, reducing ambiguity. It ensures that the operation is label-based, which is critical if the DataFrame indices are non-standard integer ranges. Furthermore, it is the safest method when chaining operations or dealing with potential SettingWithCopyWarning issues, as it is designed for both viewing and assignment.

Here is the implementation of the column retention using the .loc indexer:

df = df.loc]

Both Method 1 and Method 2 successfully subset the DataFrame, dropping all columns except

those explicitly named, such as **col2** and **col6** in these introductory examples.

Setting Up the Sample DataFrame for Demonstration

To effectively demonstrate these column subsetting techniques, we must first establish a working DataFrame. This example uses data representing statistics for various teams, providing multiple columns for clear manipulation. We use the Pandas library, commonly imported as `pd`, to construct the DataFrame from a dictionary of lists in Python.

This sample DataFrame contains eight rows (teams) and six columns (`team`, `points`, `assists`, `rebounds`, `steals`, and `blocks`). This structure allows us to simulate a typical scenario where a data professional might only be interested in performance metrics like scores and defensive stats, requiring the exclusion of unnecessary identifier columns or intermediate data.

The following code snippet demonstrates the creation and immediate visualization of our sample dataset, which we will subsequently use for both detailed examples.

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

Example 1: Retaining Columns Using Double Brackets

In this first practical example, we apply the double-bracket method (Method 1) to our sample DataFrame. Our objective is to keep only two specific performance metrics: the **points** scored by the team and the **blocks** recorded. All other columns, including `team`, `assists`, `rebounds`, and `steals`, will be discarded. This is a common requirement when feeding input features into machine learning models or focusing analysis solely on primary outcomes.

We achieve this by simply passing a Python list containing the desired column names (`'points'` and `'blocks'`) directly to the DataFrame using the double-bracket notation. The resulting object is a new DataFrame containing the same rows but only the columns specified in the list. By reassigning this result back to `df`, we update the variable to reflect the filtered dataset.

Reviewing the output confirms that the operation was successful. The resulting DataFrame, `df`, now consists exclusively of the **points** and **blocks** columns, demonstrating the efficiency and simplicity of double-bracket indexing for targeted column retention.

#drop all columns except points and blocks
df = df]

#view updated DataFrame
print(df)

points blocks
0 18 1
1 22 0
2 19 0
3 14 3
4 14 2
5 11 2
6 20 1
7 28 5

As observed in the output, only the **points** and **blocks** columns remain. All other columns, which were not included in the selection list, have been successfully dropped from the DataFrame structure.

Example 2: Retaining Columns Using the .loc Indexer

Our second example replicates the exact outcome of Example 1 but utilizes the explicit label-based indexer, .loc. This method is structurally more verbose but offers greater control, particularly when

simultaneously performing row filtering. Here, we aim to retain the same columns: **points** and **blocks**.

The key to using .loc for column retention is the inclusion of the full slice (`:`) in the row selector position. This signifies that we are selecting all rows from the original DataFrame. The column selector position then takes the list of desired column labels, similar to the method used in the double-bracket approach.

Using .loc provides clear semantic meaning: "Select all rows, but restrict the columns to this specific list." This highly explicit form is valuable in production environments where clarity regarding indexing boundaries is paramount. The resulting DataFrame is identical to the one obtained via the double-bracket method, highlighting that both approaches are valid and deliver the same outcome.

#drop all columns except points and blocks
df = df.loc]

#view updated DataFrame
print(df)

points blocks
0 18 1
1 22 0
2 19 0
3 14 3
4 14 2
5 11 2
6 20 1
7 28 5

Just as with the previous method, the output confirms that only the **points** and **blocks** columns have been retained, while all others were successfully removed.

Comparative Analysis: Double Brackets vs. .loc

Choosing between the double-bracket method and the .loc indexer often comes down to context, personal preference, and the complexity of the operation. Both methods are foundational for column subsetting in Pandas, but they cater to slightly different needs and coding styles. Understanding these differences helps in writing robust and maintainable code.

The double-bracket method is generally favored for its **simplicity and conciseness**. If the task is

strictly limited to selecting columns and requires no row filtering or specific index manipulation, `df]` is the quickest and most readable option. It is the idiomatic standard for basic column selection in the Python data science ecosystem. However, this method can sometimes be less explicit about whether a copy or a view of the original data is being returned, which can lead to the infamous SettingWithCopyWarning if subsequent modifications are attempted.

Conversely, the .loc method, requiring the syntax `df.loc]`, is superior in terms of **explicitness and capability**. By using .loc, we explicitly state that we are using label-based indexing across both axes. This is crucial for operations involving assignment or when the row index is customized. When subsetting data based on both row criteria (e.g., `where team == 'A'`) AND column criteria, .loc becomes the only viable and recommended approach.

Double Brackets: Ideal for fast, simple column selection when rows are fully retained. Highly concise.

.loc: Preferred for complex subsetting, explicit index handling, simultaneous row and column selection, and avoiding potential assignment warnings. More robust for production code.

Alternative Approaches and Best Practices

While the two primary methods discussed are the most recommended, it is worth briefly mentioning an alternative that leverages the `difference` between the full set of columns and the set of columns to keep. Although generally less efficient for simple column retention, this approach is useful when the list of columns to **drop** is computationally derived.

If you must use the standard `drop()` method, you would first calculate the list of columns to be discarded. This can be done by taking the difference between the full set of column names (`df.columns`) and the set of names you wish to retain. Once this list is calculated, you pass it to `df.drop()`, ensuring the `axis=1` parameter is set to indicate column operations. This maintains data manipulation efficiency when working with Pandas Series and DataFrames.

 # Example of using drop() by finding the complement
columns_to_keep =
columns_to_drop = df.columns.difference(columns_to_keep)
df_new = df.drop(columns=columns_to_drop, axis=1)

In summary, unless your workflow specifically requires complex mathematical set operations on the column index, stick to the direct selection methods. For basic retention, use **double brackets**.

For explicit, robust indexing that integrates row conditions, use the **.loc indexer**. These are the most idiomatic and clear ways to manage column subsets within a DataFrame in Python.