# Learn How to Remove the First Column in a Pandas DataFrame Using Python

Authored by

**Mohammed looti**

November 1, 2025

When conducting thorough data analysis using the Pandas DataFrame structure in Python, practitioners frequently encounter the need to refine or restructure their datasets. A particularly common scenario involves the accidental inclusion of an extraneous index column during data import, which typically manifests as the very first column (index 0). Removing this unwanted element is a fundamental and necessary step in achieving effective data manipulation and preparing data for subsequent modeling or reporting.

The powerful **Pandas** library offers exceptional flexibility, providing not just one, but several distinct and efficient mechanisms for structural modifications like column removal. While the desired outcome--a DataFrame without the initial column--remains constant, the underlying philosophy and performance characteristics of these methods vary significantly. Understanding these differences allows developers to select the most appropriate method based on criteria such as code readability, execution speed, and the permanence of the modification.

In this comprehensive guide, we will dissect three primary techniques utilized by **Pandas** experts to drop the first column based on its positional index. We will explore the strengths and specific use cases for each approach, enabling you to master this essential cleaning operation:

**Method 1:** Leveraging the explicit and versatile `.drop()` function.
**Method 2:** Employing efficient positional slicing using the `iloc` indexer.
**Method 3:** Utilizing the native Python `del` statement for dictionary-like deletion.

## Establishing the Reproducible Sample DataFrame

To accurately demonstrate the mechanics of column removal, we first require a stable and representative dataset. The following example constructs a sample **Pandas DataFrame** focused on sports statistics. This structure will clearly illustrate how each subsequent method targets and eliminates the column located at index position 0. For this tutorial, we assume the necessary **Pandas** library has already been successfully imported into your Python working environment.

Our sample code initializes a DataFrame containing four columns: 'team', 'position', 'assists', and 'rebounds'. The 'team' column, which resides at index 0, will be the target of all removal operations. Notice that standard **Pandas** behavior automatically assigns a row index (0 through 7) separate from the columns we define.

**import pandas as pd**

#create DataFrame
df = pd.DataFrame({'team': ,
'position': ,
'assists': ,

'rebounds': })

#view DataFrame
df

team position assists rebounds
0 A G 5 11
1 A G 7 8
2 A F 7 10
3 A F 9 6
4 B G 12 6
5 B G 9 5
6 B F 9 9
7 B F 4 12

The column labeled 'team' is unambiguously located at column index 0. The objective of the forthcoming demonstrations is not merely to remove the column named 'team', but specifically to remove the column that occupies the **first positional index** (0), irrespective of its current label. This distinction is vital when dealing with dynamically generated or imported datasets where column names might be unknown or generic.

## Method 1: Utilizing the drop() Function

The .drop() function represents the standardized and most explicit mechanism within **Pandas** for removing rows or columns. While `.drop()` traditionally accepts a list of labels (names), we can dynamically target the first column by extracting its label using positional indexing on the DataFrame's column list: `df.columns`.

Executing this operation requires careful specification of the function's parameters to ensure the intended behavior. We pass the identified column label to the `columns` argument (or the older `labels` argument combined with `axis=1`). Crucially, we must set the parameter `axis=1`, which tells **Pandas** to look horizontally across the columns rather than vertically across the rows (the default behavior). The combination of these settings ensures that the function correctly interprets the input as a column label intended for deletion.

To make the change permanent on the original DataFrame object, we include the optional parameter `inplace=True`. When this parameter is set, the function modifies the DataFrame in place and returns `None`. If `inplace` were omitted (or set to `False`), the function would return a completely new DataFrame object with the specified column removed, leaving the original `df` untouched. Using `inplace=True` is convenient for data cleaning pipelines but requires caution, as

the action cannot be easily undone.

**#drop first column of DataFrame**
**df.drop(columns=df.columns, axis=1, inplace=True)**

#view updated DataFrame
df

position assists rebounds
0 G 5 11
1 G 7 8
2 F 7 10
3 F 9 6
4 G 12 6
5 G 9 5
6 F 9 9
7 F 4 12

As confirmed by the output, the 'team' column has been successfully removed, and the DataFrame now begins with the 'position' column at index 0. This method is highly favored in production environments due to its self-documenting nature; the intent to "drop" data is explicitly clear from the function name.

## Method 2: Using Positional Indexing with iloc

The second technique harnesses the power of the iloc indexer, which is designed for pure integer-location based indexing. Rather than explicitly deleting a column, this method achieves the desired outcome by performing a selection operation, effectively creating a new DataFrame that includes all existing data *except* the first column.

The concise syntax for this operation is `df.iloc`. This expression is interpreted as a standard slicing operation applied to the two dimensions of the DataFrame. The first element, `:`, indicates that we select all rows from the beginning to the end. The second element, `1:`, is the key to this method; it specifies that we select all columns starting from index position 1 and extending to the end of the column axis. Since column indices start at 0, this slicing successfully bypasses and excludes the column located at index 0.

A crucial distinction from the `.drop(inplace=True)` method is that `iloc` always returns a view or a copy of the selected data. Therefore, to make the structural change permanent, the resulting sliced DataFrame must be explicitly reassigned back to the original variable, `df = df.iloc`. This reassignment ensures the original DataFrame variable now points to the new, modified structure,

completing the removal process. This method is often the fastest of the three, as slicing operations are highly optimized internally within **Pandas** for speed.

**#drop first column of DataFrame**
**df = df.iloc**

#view updated DataFrame
df

position assists rebounds
0 G 5 11
1 G 7 8
2 F 7 10
3 F 9 6
4 G 12 6
5 G 9 5
6 F 9 9
7 F 4 12

For users already comfortable with Python slicing notation, this technique offers unparalleled conciseness and is a preferred choice when optimizing for computational performance, especially when dealing with extremely large datasets where the overhead of function calls might be a factor.

## Method 3: Employing the Native Python del Statement

The final method leverages the native Python `del` keyword. This method is possible because a Pandas DataFrame behaves structurally much like a standard **Python** dictionary, where the column names serve as the dictionary keys. The `del` statement is a powerful, low-level command used to remove bindings from a namespace, and when applied to a DataFrame column, it removes that column key and its associated data entirely.

Since `del` requires the exact column name (key), we must again retrieve the label of the first column dynamically using the expression `df.columns`. The full statement then becomes `del df]`. This syntax is remarkably succinct and executes the modification directly on the original object, meaning it is inherently an in-place operation, similar to using the `inplace=True` argument in Method 1.

While `del` offers the simplest syntax for in-place column removal, it is generally considered the least flexible of the three options. It provides no mechanism for selecting based on conditions or returning a new object; its only function is immediate, permanent deletion. Due to its abrupt nature and lack of error handling capabilities compared to a dedicated **Pandas** function, its usage is often

limited to quick, interactive data exploration sessions rather than complex, robust production code where clarity and error management are paramount.

**#drop first column of DataFrame**
**del df]**

#view updated DataFrame
df

position assists rebounds
0 G 5 11
1 G 7 8
2 F 7 10
3 F 9 6
4 G 12 6
5 G 9 5
6 F 9 9
7 F 4 12

We have successfully used three distinct techniques to execute the same fundamental data manipulation task. The ability to choose between these methods provides flexibility tailored to specific coding contexts, whether prioritizing maximum speed, maximum readability, or minimum code footprint.

## Comparative Analysis: Selecting the Optimal Method

Although all three techniques achieve the goal of dropping the first column, data professionals must weigh factors beyond mere functionality. The choice often boils down to a balance between code clarity, performance characteristics, and the need for flexibility within a larger data pipeline. Understanding these trade-offs is key to writing professional and maintainable **Pandas** code.

The .drop() function (Method 1) is generally considered the gold standard for column and row removal. Its explicit naming makes the intention of the code immediately obvious, even to those unfamiliar with **Pandas** specifics. Furthermore, it offers superior control through parameters like `axis` and `inplace`, allowing the user to decide whether to modify the object directly or return a new copy. This robustness and clarity make `.drop()` the recommended choice for collaborative projects or complex scripts where maintainability is a primary concern.

Conversely, the `.iloc` slicing method (Method 2) excels in computational efficiency. Because **Pandas** implements highly optimized routines for indexing and selecting contiguous data blocks, slicing operations often incur less overhead than generalized function calls like `.drop()`. While the

syntax `df.iloc` is extremely concise, it requires a solid understanding of iloc and array slicing conventions. This method is best employed when working with very large datasets where milliseconds in execution time matter, or when the code needs to be as succinct as possible.

Finally, the native `del` statement (Method 3), while the quickest to write, lacks the flexibility and safety features of the other two methods. Since it operates directly on the DataFrame's internal structure using the column name as a key, it is only suitable for simple, in-place deletion and offers no easy way to remove data based on positional index without first retrieving the label. Due to its potential for causing unexpected errors if column names are not guaranteed, its use should be reserved primarily for interactive exploration and immediate, non-recoverable cleanup tasks.

**The drop() function (Method 1):** This method provides the best balance of readability, robustness, and flexibility. It is the professional standard for data cleaning operations.
**`.iloc` Slicing (Method 2):** Recommended when performance and conciseness are prioritized over explicit function naming, particularly effective for trimming edges of the DataFrame.
**`del` Statement (Method 3):** Use cautiously. Best suited for quick, interactive sessions where the exact column name is known and rapid, in-place modification is required.

For standard data manipulation tasks, favoring the **Pandas**-specific `.drop()` function ensures the highest degree of code clarity and maintainability for future developers.

## Additional Resources for Pandas Mastery

Mastering column and row manipulation is foundational to effective use of the **Pandas** library. For those looking to deepen their expertise, exploring related topics will significantly enhance their Pandas DataFrame skills:

Further study into the official documentation regarding advanced indexing, multi-indexing, and conditional slicing techniques will provide a more comprehensive understanding of the structural power available within **Pandas**. Specifically, reviewing the documentation for `.loc` (label-based indexing) and the various methods for renaming and reordering columns is highly recommended.

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

For more detailed information on indexing, slicing, and column manipulation in Pandas, consult the official documentation.