

How to Easily Drop Multiple Columns in Pandas

Authored by
stats writer

November 21, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Drop Multiple Columns in Pandas*. PSYCHOLOGICAL SCALES. Retrieved from <https://scales.arabpsychology.com/?p=98812>

In the expansive and crucial domain of data manipulation using the [Pandas](#) library in Python, effectively managing the structure of your data is paramount. One of the most common tasks data scientists face is the systematic removal of unnecessary or redundant data fields, typically represented as columns within a [DataFrame](#). Pandas offers a robust and flexible tool for this purpose: the `drop()` function. This function allows users to remove rows or columns based on specified labels or indices, making data cleaning an efficient process.

When the objective is to eliminate multiple columns simultaneously, the core mechanism involves passing a collection--specifically a list--of the column names or index positions you wish to discard. Crucially, successful column deletion requires careful specification of the [axis argument](#). In Pandas, data operations occur along two primary axes: axis 0 represents the rows (index), and axis 1 represents the columns. To ensure that the function targets columns, you must explicitly set `axis=1`. Conversely, setting `axis=0` (which is the default behavior) instructs the function to operate on the rows.

Understanding and correctly utilizing the `axis` parameter is the foundation for efficient structural modifications within Pandas. Furthermore, mastering the various ways to specify the columns--be it by explicit naming, index range, or conditional selection--provides maximum control over the resulting [DataFrame](#). The methods outlined in this guide demonstrate the flexibility available to drop multiple columns, catering to situations where you know the names, the indices, or a range thereof.

Understanding the Pandas `drop()` Function and `axis` Parameter

The [drop\(\)](#) function is arguably the most utilized tool for removing data components within a Pandas structure. Its signature flexibility comes from its ability to handle both rows and columns based on the `axis` parameter. When performing column deletion, it is considered best practice to pass the column names via the `columns` argument, which implicitly sets `axis=1`, although explicitly setting `axis=1` alongside the `labels` argument is also valid.

When working with large, high-dimensional datasets, the efficiency gains from dropping many columns at once, rather than iteratively, are substantial. The function expects an iterable--typically a Python list--containing the identifiers of the columns targeted for removal. If this list is omitted or incorrectly formatted, the function will default to attempting row deletion, likely leading to a `KeyError` if the provided labels do not match the row index.

The four primary strategies detailed below offer different approaches depending on the availability of metadata--whether you have the column names readily available, or if you need to rely on their zero-based integer index positions. Mastering these methods is crucial for streamlining the initial data preparation phase of any statistical project.

You can use the following methods to drop multiple columns from a Pandas DataFrame:

Method 1: Drop Multiple Columns by Name (Explicit List)

Method 2: Drop Columns in a Range by Name (Using the `loc` accessor)

Method 3: Drop Multiple Columns by Index Position (Explicit List)

Method 4: Drop Columns in a Range by Index Position (Using Slicing)

Preparation: Setting Up the Example DataFrame

To provide a practical demonstration of these column-dropping techniques, we will first establish a sample DataFrame representing statistical data for various teams. This setup allows us to clearly illustrate the input and resulting output for each method, enhancing comprehension of the syntax and effect of the `drop()` function.

We will import the Pandas library and then construct a dictionary that is subsequently converted into our sample DataFrame. This initial step is standard practice in data analysis environments, ensuring a reproducible environment for testing operations.

import pandas as pd

```
#create DataFrame
```

```
df = pd.DataFrame({'team': ,  
'points': ,  
'assists': ,  
'rebounds': ,  
'steals': })
```

```
#view DataFrame
```

```
print(df)
```

```
team points assists rebounds steals
```

```
0 A 18 5 11 4
```

```
1 B 22 7 8 5
```

```
2 C 19 7 10 10
```

```
3 D 14 9 6 12
```

```
4 E 14 12 6 4
```

```
5 F 11 9 5 8
```

```
6 G 20 9 9 7
```

```
7 H 28 4 12 2
```

The resultant DataFrame contains eight rows and five columns, offering a sufficiently complex

structure to test the various column deletion methods effectively. Note that in all subsequent examples, we assume the initial DataFrame is reset to this state before applying the dropping operation, unless specified otherwise, to demonstrate the isolated effect of each method.

Method 1: Dropping Multiple Columns by Name (Explicit List)

This is arguably the most common and straightforward method when working with well-labeled datasets. If you know the exact names of the columns you intend to remove, you simply pass them as a standard Python list to the `drop()` function, utilizing the `columns` parameter.

The use of the `columns` parameter is preferred over the older method of using `labels` combined with `axis=1`, as it enhances code readability and reduces the chance of errors related to axis configuration. By specifying the column names explicitly, we maintain a high degree of control and clarity regarding which data fields are being pruned. This method is highly recommended for scripts where column names are stable and predefined.

The following code demonstrates how to drop the **points**, **rebounds**, and **steals** columns by name from our sample DataFrame:

```
#drop multiple columns by name  
df.drop(columns=, inplace=True)
```

```
#view updated Dataframe  
print(df)
```

```
team assists  
0 A 5  
1 B 7  
2 C 7  
3 D 9  
4 E 12  
5 F 9  
6 G 9  
7 H 4
```

As illustrated in the output, only the `team` and `assists` columns remain, confirming the successful removal of the specified fields. Note that the `inplace=True` argument ensures the modification is applied directly to the existing DataFrame, preventing the need for explicit variable reassignment.

Method 2: Dropping Column Ranges Using Indexing (loc)

Sometimes, data columns are organized sequentially, and it becomes more efficient to drop a contiguous block of columns rather than listing every name individually. Pandas provides the powerful `loc` accessor, which allows label-based indexing, to select a range of columns based on their starting and ending names. This technique is particularly useful when dealing with data derived from systems that output extensive feature sets in a predictable order.

The syntax `df.loc` selects all rows (indicated by `:`) and all columns ranging from `start_col` to `end_col`, inclusively. However, we cannot pass this selection directly to the `columns` parameter of the `drop()` function, as it expects column labels, not the actual data subset. Instead, we use the selection to retrieve the relevant column labels.

A common pattern for dropping a range defined by names involves using the `columns` attribute of the DataFrame indexer to extract the labels themselves. The following code demonstrates how to drop every column between **points** and **rebounds**, including both endpoints, by leveraging the underlying label indexing capabilities:

#drop columns in range by name

```
df.drop(columns=df.loc.columns, inplace=True)
```

```
#view updated Dataframe
```

```
print(df)
```

```
team steals
```

```
0 A 4
```

```
1 B 5
```

```
2 C 10
```

```
3 D 12
```

```
4 E 4
```

```
5 F 8
```

```
6 G 7
```

```
7 H 2
```

In this specific outcome, the columns `points`, `assists`, and `rebounds` were dropped, leaving only `team` and `steals`. This confirms that `loc`, when used for column range selection in conjunction with `.columns`, provides a clean list of labels for the `drop()` function.

Method 3: Removing Columns by Integer Index Position

In scenarios where datasets lack standardized or meaningful column names, or when dealing with

intermediate results where column names might be generic (e.g., '0', '1', '2'), relying on zero-based integer index positions becomes necessary. Pandas allows column deletion by providing a list of these positional indices to the `drop()` function.

To implement this, we must first access the list of column names using `df.columns` and then use standard Python [indexing](#) notation to select the names corresponding to the desired index positions. For example, if we want to drop the first, fourth, and fifth columns, we would pass the list of indices to the column indexer, and the resulting list of names is then passed to the `drop()` function.

The following code shows how to drop the columns in index positions 0, 3, and 4 from the DataFrame. Referring back to our initial setup, these indices correspond to `team` (0), `rebounds` (3), and `steals` (4):

```
#drop multiple columns by index  
df.drop(columns=df.columns, inplace=True)
```

```
#view updated Dataframe  
print(df)
```

```
points assists  
0 18 5  
1 22 7  
2 19 7  
3 14 9  
4 14 12  
5 11 9  
6 20 9  
7 28 4
```

The resulting DataFrame retains only `points` (index 1) and `assists` (index 2). While effective, relying on integer indexing should be used cautiously, especially in dynamic data pipelines, as adding or removing columns elsewhere in the pipeline could silently shift the indices and lead to unintended deletions.

Method 4: Deleting Columns Within an Index Range (Slicing)

Similar to using named ranges, we can also leverage Python's powerful slicing syntax directly on the column index to specify a contiguous range of columns for deletion based purely on their positions. This method simplifies the removal of adjacent columns without needing to manually list all index numbers.

When slicing the column index, standard Python slicing rules apply: the start index is inclusive, and the stop index is exclusive. For instance, a slice of selects elements at index 1, 2, and 3, but stops before index 4. This is a crucial distinction compared to the label-based slicing used with `loc`, which is inclusive of the stop label.

The following code demonstrates how to drop a range of columns using index slicing. We specify `df.columns`, which targets the columns at index positions 1, 2, and 3. These correspond to `points`, `assists`, and `rebounds` in our original DataFrame:

#drop columns by index range

```
df.drop(columns=df.columns, inplace=True)
```

```
#view updated Dataframe
```

```
print(df)
```

```
team steals
```

```
0 A 4
```

```
1 B 5
```

```
2 C 10
```

```
3 D 12
```

```
4 E 4
```

```
5 F 8
```

```
6 G 7
```

```
7 H 2
```

The resulting DataFrame shows that only the columns `team` (index 0) and `steals` (index 4) remain, confirming the successful removal of the middle three columns specified by the index range . It is essential to remember that the syntax `df.columns` specifies columns in index positions 1 *up to* (but not including) 4. Thus, this syntax successfully drops the columns in index positions 1, 2, and 3.

The Critical Role of the inplace Parameter

Throughout these examples, the argument `inplace=True` has been consistently applied within the `drop()` function calls. Understanding the function of this parameter is vital for managing memory and variable assignments in Pandas operations.

By default, most Pandas methods, including `drop()`, return a new DataFrame containing the results of the operation while leaving the original DataFrame unchanged. This behavior supports functional programming paradigms and helps prevent accidental data corruption. However, when working with very large datasets, creating a copy of the entire DataFrame just to drop a few

columns can consume significant memory and processing time.

The argument `inplace=True` tells Pandas to drop the columns directly within the existing DataFrame object, modifying it in place without returning a new object or requiring variable reassignment. This saves memory and often results in cleaner code, as you do not need to write `df = df.drop(...)`. It is a powerful feature but must be used judiciously, as the original data state cannot be easily recovered once the operation is executed in place.

Conclusion and Best Practices for Column Deletion

Pandas offers comprehensive methods for managing DataFrame structure, and the ability to drop multiple columns is a fundamental component of the data preparation toolkit. Whether you opt for label-based deletion using explicit names (Method 1), or utilize positional indexing for ranges (Methods 3 and 4), the key is specifying the column identifiers accurately and ensuring the `axis` parameter is implicitly or explicitly set to 1.

For most production environments, using column names (Methods 1 and 2) is highly recommended over positional indexing. Column names provide semantic meaning and are less susceptible to breaking changes if the data source schema is slightly altered, whereas positional indices are inherently unstable across data iterations. Furthermore, always consider whether `inplace=True` is appropriate for your specific memory constraints and workflow requirements.

For further reference and a deeper understanding of all available parameters and functionalities, consult the comprehensive documentation for the [Pandas drop\(\) function](#) on the official Pandas website.