Drop Columns by Index in Pandas

Authored by

Mohammed looti

November 4, 2025

Understanding Column Indexing in Pandas

Data cleaning and preprocessing frequently require the removal of irrelevant or redundant features from a DataFrame. While most operations focus on dropping columns using their explicit names (labels), scenarios often arise where only the column's positional index number is available or practical. This technique becomes essential when dealing with datasets that have automatically generated, complex, or entirely unknown column names prior to analysis.

In the Pandas library, columns are organized using a zero-based indexing system; thus, the first column resides at position 0, the second at 1, and so forth. To successfully execute a column drop using its index, we must first interact with the DataFrame's metadata structure. Specifically, we utilize the **df.columns** attribute, which returns an Index object containing all column labels.

The central function responsible for this removal process is the versatile drop() method. When performing any column-wise operation, it is absolutely crucial to specify the **axis=1** parameter. This explicitly tells Pandas that the operation targets columns, as opposed to the default behavior of targeting rows (which corresponds to **axis=0**). Furthermore, developers commonly use the **inplace=True** argument to modify the DataFrame directly, eliminating the need for explicit variable reassignment.

Syntax for Dropping a Single Column by Index

Removing a single column based purely on its positional index requires a two-step process: translation and execution. Since the standard drop() method is designed to accept column labels (names), we must first retrieve the label associated with the specified numerical position using the **df.columns** attribute.

This translation step allows us to seamlessly integrate positional information into the label-based dropping mechanism. For instance, to remove the very first column, located at index number 0, the syntax is clear and concise, as demonstrated below:

```
# Drop the first column (index 0) from the DataFrame
df.drop(df.columns, axis=1, inplace=True)
```

This approach effectively selects the column label at the desired position and passes it as the argument to the drop() method. While alternative methods exist, such as advanced positional indexing using **iloc** (which we cover later), this label-translation technique is the standard way to interact with the **drop()** function when indices are known.

Syntax for Dropping Multiple Columns by Index

When streamlining a DataFrame, it is highly efficient to remove several features simultaneously. Instead of performing sequential, individual drop operations, Pandas allows us to pass a list of positional indices to the **df.columns** attribute. This action returns a comprehensive list of corresponding column names, which the drop() method can process in a single, atomic step.

The following demonstration outlines how to define a list targeting multiple columns--specifically, the columns located at indices 0, 1, and 3--and efficiently remove them from the dataset:

```
# Define a list containing the positional indices to drop
cols =
# Use df.columns to select the corresponding column labels
df.drop(df.columns, axis=1, inplace=True)
```

It is important to remember that the list **cols** must contain the zero-based indices of the columns intended for removal. Utilizing this list-based approach ensures optimal performance and clarity, which is especially beneficial when managing large-scale datasets where efficiency is paramount. This method provides a clean, readable way to perform aggressive feature engineering.

Handling DataFrames with Duplicate Column Names

A significant challenge arises when a DataFrame structure includes duplicate column names. If you attempt to use the standard label-translation technique (**df.drop(df.columns, ...)**), Pandas will, by default, target and remove **all** columns that share that resulting label, not just the single instance at the specified position. To ensure precise positional deletion in such non-unique structures, we must bypass the column labeling system entirely.

The most reliable solution involves leveraging positional indexing based on integer location, accessible via the powerful iloc accessor. Instead of instructing Pandas to drop a column, we redefine the DataFrame by explicitly selecting only the columns we intend to keep. This is achieved by generating a list of all current column indices and then surgically removing the specific index corresponding to the column we wish to exclude.

The following code snippet demonstrates this technique. We first generate the complete list of indices using a standard Python list comprehension based on the DataFrame's shape (number of columns). We then employ the list's built-in **remove()** method to exclude the target index. Finally, the iloc accessor is used to subset the DataFrame using the curated list of remaining column indices:

```
# Define a list of all column indices (0, 1, 2, ...)
```

cols =)]

# Drop the second column (index 1) by removing its index from the list
cols.remove(1)

Use iloc to select all rows (:) and only the remaining columns (cols)
df.iloc

This methodology guarantees that only the column located at the specific index number is excluded. This preservation of structural integrity makes the **iloc** method the superior choice for handling datasets containing non-unique or repetitive column labels.

Example 1: Dropping a Single Column by Index Position

This practical example illustrates the fundamental application of dropping a single column using its numerical position. This is a common requirement during the initial stages of data cleaning when boilerplate or automatically generated identifier columns need immediate removal.

We begin by initializing a sample DataFrame containing fictional sports team data. Our objective is straightforward: eliminate the very first column, corresponding to index number 0, which is labeled 'team'. We will use the standard label-translation method via **df.columns**.

import pandas as pd

# Create the initial DataFrame
df = pd.DataFrame({'team': ,
'first': ,
'last': ,
'points': })

# Drop the first column (index 0) from the DataFrame using df.columns
df.drop(df.columns, axis=1, inplace=True)

View the resulting DataFrame
df

first last points
0 Dirk Nowitzki 26
1 Kobe Bryant 31
2 Tim Duncan 22
3 Lebron James 29

The resulting output confirms that the 'team' column has been successfully removed. This clearly illustrates how the positional index (0) was correctly converted into the corresponding column name ('team') before the drop() method executed the removal operation on the column label.

Example 2: Dropping Multiple Columns Using a List of Indices

When conducting significant feature reduction, the ability to drop multiple columns simultaneously is a key efficiency gain. By using a list of indices, we minimize the computational overhead, allowing Pandas to perform the necessary index lookup and removal in one streamlined process.

Using the established DataFrame structure from the previous example, we now aim for a more aggressive removal. Our goal is to eliminate columns located at indices 0 ('team'), 1 ('first'), and 3 ('points'). This will leave only the column at index 2 ('last') remaining in the resulting DataFrame.

import pandas as pd

# Create the initial DataFrame
df = pd.DataFrame({'team': ,
'first': ,
'last': ,
'points': })

Define the list of indices to drop: 0, 1, and 3
cols =
# Execute the drop operation using the list of indices
df.drop(df.columns, axis=1, inplace=True)

View the resulting DataFrame
df

last
0 Nowitzki
1 Bryant
2 Duncan
3 James

As verified by the output, only the 'last' column remains. This demonstrates the superior effectiveness and elegance of passing a list of positional indices to the **df.columns** attribute, enabling batch processing of column removals.

Example 3: Dropping a Column with Duplicate Names Using iloc

This critical example showcases the necessity of using the iloc accessor when structural integrity must be maintained in the presence of duplicate column names. We construct a DataFrame where the label 'last' is intentionally duplicated. Our specific task is to remove only the first occurrence of 'last', which is situated at index 1.

If we attempted the standard **drop()** method targeting the label 'last', the operation would inadvertently remove both instances of that column label. By carefully managing the index list and using iloc for selection, we achieve precise positional deletion without disturbing other columns that share the same descriptive label.

import pandas as pd

# Create DataFrame with duplicate column name 'last'
df = pd.DataFrame({'team': ,
'last': ,
'last': ,
'points': },
columns=)

# Define list of all current column indices (0, 1, 2, 3)
cols =)]

# Remove the index corresponding to the column we want to drop (index 1)
cols.remove(1)

Use iloc to select only the remaining columns in the list
df.iloc

team last points
0 Mavs Nowitzki 26
1 Lakers Bryant 31
2 Spurs Duncan 22
3 Cavs James 29

The final DataFrame correctly retains 'team' (index 0), the second instance of 'last' (originally index 2), and 'points' (originally index 3). The column at index 1 was successfully excluded, validating the use of iloc for targeted, positional column subsetting.

Conclusion and Best Practices

The ability to drop columns using their positional index is an indispensable skill for efficient data preprocessing in Pandas. Whether your dataset is clean and straightforward or presents complex challenges like duplicate column names, a clear understanding of the difference between label-based dropping (using **df.columns**) and positional selection (via **iloc**) is crucial for robust data manipulation workflows.

As a best practice, always confirm the intended column index before executing a destructive operation. Misindexing can lead to the accidental loss of vital data features. A quick verification step, such as displaying **df.columns**, should always precede the drop function when working with indices.

For developers seeking to deepen their expertise in data manipulation within the Python ecosystem, we strongly recommend reviewing the official documentation for the methods and accessors discussed here. This foundational knowledge ensures you can handle any structural anomaly encountered during data preparation.