Learning to Select Columns by Index in Pandas DataFrames

Authored by

Mohammed looti

November 1, 2025

When performing rigorous data analysis using the powerful Pandas library in Python, analysts frequently encounter the need to select specific columns within a DataFrame. This selection process is typically straightforward when using explicit column names (labels). However, mastering how to efficiently retrieve data based on its numerical position--its index value--is a fundamental skill for advanced data manipulation.

The ability to navigate a DataFrame using both positional and label-based indexing is crucial for building robust and flexible workflows. Positional selection relies heavily on the .iloc accessor, which is specifically designed for integer indexing. Conversely, selecting columns based on their descriptive names is managed by the .loc accessor, which specializes in label-based indexing. Understanding the distinct behaviors of these two indexers is key to unlocking Pandas' full potential.

This comprehensive guide provides expert, practical examples detailing the application of both the **.iloc** and **.loc** functions. We will ensure you can accurately and confidently select columns, regardless of whether your criteria are numerical positions or descriptive string labels, thereby enhancing the stability and efficiency of your data processing pipelines.

Understanding Positional vs. Label-Based Indexing

DataFrames are structured as two-dimensional containers, analogous to tables in SQL or traditional spreadsheets. Every axis--rows and columns--in a DataFrame is associated with two distinct types of identifiers: the explicit index (the label, typically a column name string) and the implicit index (the integer position, which always starts at 0).

The selection mechanism in Pandas is governed by indexers. The two primary indexers used to access specific subsets of data are .iloc and .loc. While both methods are incredibly versatile and handle row filtering, our focus here is strictly on their application for column retrieval, where the differences in their operational logic become critical.

It is essential to recall the standard syntax for these indexers: they accept two arguments separated by a comma--the first for row selection, and the second for column selection. When our objective is to select columns while including all rows, we pass the full range selector (the colon, **:**) in the row position, followed by the specific column index or label specification.

Leveraging .iloc for Integer Position Selection

The .iloc accessor is engineered exclusively for selecting data based on the **zero-based integer position** of the column or row. This means if you wish to access the fifth column of your DataFrame, you must request index position 4. This positional method is highly effective when column names are unknown, or when processing data programmatically based on the column

count, such as iterating over all numeric columns regardless of their labels.

To demonstrate this functionality, we will use a sample DataFrame containing fictional sports statistics. We will then use .iloc to precisely retrieve the column located at index position 3, which corresponds to 'rebounds' in our dataset.

import pandas as pd

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

#view DataFrame structure (Indices 0, 1, 2, 3 correspond to team, points, assists, rebounds)
df

team points assists rebounds
0 A 11 5 11
1 A 7 7 8
2 A 8 7 10
3 B 10 9 6
4 B 13 12 6
5 B 13 9 5

#select column with index position 3 ('rebounds')
df.iloc

0 11
1 8
2 10
3 6
4 6
5 5
Name: rebounds, dtype: int64

In the code **df.iloc**, the colon (**:**) ensures all rows are included, and the integer index **3** specifies the column dimension. The resulting output is a Pandas Series containing only the data extracted from the 'rebounds' column.

Selecting Non-Contiguous Columns and Ranges with .iloc

The true utility of integer indexing emerges when there is a need to select multiple columns simultaneously, especially if they are not adjacent in the DataFrame structure. Instead of passing a single integer, we provide a Python list containing the desired integer positions to the column argument within the .iloc indexer.

For example, if our analysis requires the 'points' (index 1) and 'rebounds' (index 3) columns, we wrap these indices in a list like . Executing this operation returns a new DataFrame subset that maintains the original row indices but contains only the specified columns.

#select columns with index positions 1 ('points') and 3 ('rebounds')
df.iloc]

points rebounds
0 11 11
1 7 8
2 8 10
3 10 6
4 13 6
5 13 5

To select a continuous block of columns, standard Python slicing notation is used. A critical distinction to remember is that slicing with .iloc is **exclusive** of the stop index. If the range specified is **0:3**, the indexer will include indices 0, 1, and 2, but it will stop immediately before index 3. This behavior must be accounted for precisely when planning range-based positional extractions.

#select columns with index positions in range 0 through 3 (i.e., 0, 1, 2)
df.iloc

team points assists
0 A 11 5
1 A 7 7
2 A 8 7
3 B 10 9
4 B 13 12
5 B 13 9

Prioritizing .loc for Stable Label-Based Selection

While .iloc is indispensable for positional work, the .loc accessor is the designated and preferred method for selecting data using **explicit column labels**. Using labels instead of positions is highly recommended for production-level code and analysis scripts because it guarantees functional stability. If the internal order of columns shifts (e.g., due to a new data ingest step), label-based indexing will not break, unlike positional indexing.

To employ .loc for column selection, the exact column name (as a string) is passed to the column argument. For instance, to retrieve the 'rebounds' column, we use the literal string 'rebounds' rather than relying on its positional integer 3. This methodology connects the selection directly to the semantic meaning of the data.

import pandas as pd

#create DataFrame (Same setup as Example 1)
df = pd.DataFrame({'team': ,
'points': ,
'assists': ,
'rebounds': })

#view DataFrame
df

team points assists rebounds
0 A 11 5 11
1 A 7 7 8
2 A 8 7 10
3 B 10 9 6
4 B 13 12 6
5 B 13 9 5

#select column using index label 'rebounds'
df.loc

0 11
1 8
2 10
3 6
4 6
5 5

Name: rebounds, dtype: int64

Advanced Selection: Lists and Inclusive Slicing with .loc

Similar to .iloc, the .loc indexer allows for the selection of multiple, non-contiguous columns by simply supplying a list of column labels (strings). This technique is the most widely adopted method for extracting specific features necessary for downstream analysis, machine learning model training, or reporting.

To select both 'points' and 'rebounds', we pass a list containing these two labels: . This method highlights the robustness of label indexing, enabling users to select data based on logical names rather than being susceptible to changes in the physical column order.

#select the columns with index labels 'points' and 'rebounds'
df.loc]

points rebounds
0 11 11
1 7 8
2 8 10
3 10 6
4 13 6
5 13 5

A crucial distinction between the two indexers arises during range slicing. When slicing using .loc with labels, the selection is **inclusive** of both the starting label and the stopping label. For instance, if we slice from 'team' to 'assists' using `df.loc`, all three columns--'team', 'points', and 'assists'--are included in the resulting DataFrame subset. This inclusive behavior is different from standard Python list slicing and must be carefully noted.

#select columns with index labels between 'team' and 'assists' (inclusive)
df.loc

team points assists
0 A 11 5
1 A 7 7
2 A 8 7
3 B 10 9
4 B 13 12
5 B 13 9

Conclusion: Choosing the Right Indexer

Efficient column selection in Pandas is achieved through mastery of both integer position (via **.iloc**) and explicit column labels (via **.loc**). The choice between these two powerful tools should be dictated by the specific requirements of the data operation. Use **.iloc** when the column order is guaranteed to be stable or when iterating programmatically based purely on positional counts. Conversely, favor **.loc** for interactive analysis and, most importantly, for any production code where stability against structural changes is a critical requirement.

To summarize your understanding of these indexers, remember these two fundamental distinctions:

Slicing Behavior: Slicing with .iloc (integer positions) is exclusive of the stop value (standard Python behavior), whereas slicing with .loc (labels) is inclusive of the stop label.
Input Data Type: .iloc strictly requires integer indices, ranges, or lists of integers. **.loc** strictly requires string labels, label ranges, or lists of strings.

Mastering these indexing techniques ensures precise, efficient, and reproducible data analysis workflows within the Python environment.

Further Reading and Resources

To continue enhancing your data manipulation skills, review the following related tutorials on common Pandas operations:

How to Get Row Numbers in a Pandas DataFrame
How to Drop the Index Column in a Pandas DataFrame