How to Easily Select Columns by Index in a Pandas DataFrame

Authored by

stats writer

December 2, 2025

Working efficiently with data often requires precise methods for accessing subsets of information. In the context of Python's powerful data manipulation library, Pandas DataFrames, selecting columns based on their index position or label is a fundamental skill. This process allows data scientists and analysts to isolate specific features for analysis, visualization, or modification, streamlining complex workflows.

Pandas provides two primary methods for index-based selection: the iloc indexer and the loc indexer. Understanding the distinction between these two--one designed for position and the other for labels--is paramount for writing clean, predictable, and robust data manipulation code. While selecting columns by name (label) is common, selecting by numerical index position is often necessary when column order is consistent or when automating tasks across various datasets where column names might differ.

The goal of this guide is to demonstrate how to use both indexers to effectively retrieve columns, focusing on the syntax required for single column selection, multiple column selection, and range selection. We will emphasize the use of positional indexing (using integers) via **.iloc** and label indexing (using column names) via **.loc**, providing practical, executable code examples for each scenario.

Understanding Positional vs. Label Indexing

When working with a DataFrame, it is essential to distinguish between the two types of indexing employed by Pandas. Positional indexing, also known as integer indexing, relies purely on the zero-based order of rows and columns. This means the first column is at position 0, the second at position 1, and so forth. This method is deterministic and is exclusively handled by the **.iloc** accessor.

Conversely, label indexing uses the actual names assigned to the rows (index labels) and columns (column headers). This method is human-readable and generally preferred when writing scripts that need to be resilient to changes in column order. The **.loc** accessor manages all label-based selections, allowing users to select data based on explicit column names like 'points' or 'rebounds'.

The general structure for using both indexers is `df.indexer`. Since this tutorial focuses specifically on selecting columns, we will utilize the colon operator (**:**) in the row selection position to signify "select all rows," ensuring our selection applies across the entire vertical dimension of the DataFrame.

Using .iloc for Integer-Based Column Selection

The **.iloc** indexer is the go-to tool for selecting columns based on their position, regardless of their actual names. This is especially useful in situations where you might not know the column names

beforehand, or if you need to programmatically access columns based on a generated sequence of integers. Remember that Python and Pandas use zero-based counting, so the position index starts at 0.

To illustrate the functionality of **.iloc**, let's first establish a sample DataFrame representing statistical data, which we will use consistently across all upcoming examples. This structure allows us to clearly map index positions to column labels.

Selecting a Single Column by Position Index

To select a single column using its positional index, we specify the integer corresponding to that column in the second position of the **.iloc** accessor. For instance, to retrieve the fourth column (which is at index position **3**) from our example DataFrame, we use the syntax `df.iloc`. This returns a Pandas **Series** object containing all the values from that specific column.

import pandas as pd

#create DataFrame
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 with index position 3 (i.e., 'rebounds')
df.iloc

0 11
1 8
2 10
3 6
4 6

5 5
Name: rebounds, dtype: int64

As demonstrated, the output confirms that index position 3 corresponds to the 'rebounds' column. This positional approach provides reliable access even if the column names were numerically generated or obscured.

Selecting Multiple Non-Contiguous Columns using .iloc

When the requirement is to select multiple columns that are not adjacent to each other, .iloc requires a Python **list** of integers. This list specifies the exact positions of the columns you wish to extract. Unlike selecting a single column, selecting multiple columns, even just two, results in a new **DataFrame** being returned, not a **Series**.

Suppose we are only interested in the 'points' (index 1) and 'rebounds' (index 3) columns. We pass the list as the column selector. This powerful technique is central to feature engineering and statistical modeling, where analysts frequently need to work with specific, distributed features of the data.

#select columns with index positions 1 and 3
df.iloc]

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

Selecting Column Ranges with Slicing Syntax

For extracting a continuous block of columns, Python's native slicing syntax is utilized within the iloc indexer. Slicing with **.iloc** is defined by the syntax . Crucially, when using positional indexing, the **stop** index is always **exclusive**. This mirrors standard Python list slicing behavior.

If we want to select the first three columns, which include 'team', 'points', and 'assists', these correspond to index positions 0, 1, and 2. Therefore, we must specify a slice range that starts at 0 and stops just before 3, expressed as `0:3`. This method is highly efficient when dealing with datasets where features are grouped logically by proximity.

#select columns with index positions in range 0 through 3 (exclusive of 3)
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

Using .loc for Label-Based Column Selection

While **.iloc** is indispensable for positional queries, the **.loc** indexer offers a more intuitive and readable way to select columns using their actual names. This is typically the preferred method when building analytical pipelines that rely on stable column headers. Using column labels makes the code self-documenting and less susceptible to breaking if columns are reordered in the source data.

The basic syntax for .loc remains `df.loc`. For column selection, the column selector must be a string (for a single column), a list of strings (for multiple columns), or a slice composed of strings (for a range).

Selecting a Single Column by Name

To select a single column using **.loc**, we pass the column name as a string in the second position of the indexer. For example, to retrieve all data associated with the 'rebounds' column, we use `df.loc`. Similar to **.iloc**, this operation returns a Pandas **Series** object.

This method is generally safer and more explicit than relying on index positions, especially in collaborative environments where the data structure might evolve. Using labels ensures that even if a new column were inserted at the beginning of the DataFrame, the selection would still correctly target 'rebounds'.

import pandas as pd

#create DataFrame (Re-initializing for clarity)
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 with index label 'rebounds'
df.loc

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

Selecting Multiple Non-Contiguous Columns by Label

Similar to **.iloc**, when selecting multiple columns that are dispersed across the DataFrame, loc indexer requires a list of labels (strings). By passing as the column selector, we retrieve a new DataFrame containing only these two specified columns, maintaining the order in which they were listed in the input list.

This list-based selection capability is fundamental for constructing targeted data views, such as preparing data for visualization or running regression models that only utilize a subset of the available features. The clarity offered by label selection significantly enhances script maintainability.

#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

Selecting Contiguous Column Ranges with .loc

One unique and highly convenient feature of **.loc** is its handling of slicing for ranges. Unlike the positional indexer **.iloc**, when slicing with **.loc** using labels, the **stop** label is **inclusive**. This means that if you specify the range from 'team' to 'assists', Pandas includes both 'team' and 'assists', along with every column existing between them.

This inclusive slicing behavior is a key difference developers must internalize when switching between iloc indexer (exclusive stop) and loc indexer (inclusive stop). It simplifies code when selecting a block of adjacent columns defined by their start and end points.

#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

Summary of Selection Methods

To ensure clarity, here is a concise summary of the different methods demonstrated for selecting columns in a Pandas DataFrame:

Positional Selection (.iloc): Use integers (0-based) to define column location. The stop index in slicing is exclusive.
Label Selection (.loc): Use column names (strings) to define column location. The stop label in slicing is inclusive.
Single Column Output: Selecting one column (by position or label) results in a Pandas **Series**.
Multiple Column Output: Selecting two or more columns (using a list or a slice) results in a new **DataFrame**.

Mastering both the **.iloc** and **.loc** accessors is a vital step in becoming proficient with Pandas.

Choosing the correct indexer based on whether you need positional stability or label readability will dramatically improve the efficiency and maintainability of your data analysis scripts.

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