# Learning Pandas: Mastering Row and Column Selection with the take() Function

Authored by

**Mohammed looti**

November 12, 2025

When performing intensive data manipulation using the Pandas library in Python, data scientists frequently require methods for selecting data based purely on its numerical position within a DataFrame. While familiar methods such as `.loc` (label-based indexing) and `.iloc` (integer position-based indexing) are widely used, the **take()** function offers a specialized, high-performance alternative designed exclusively for fast, positional extraction. This powerful utility enables users to return elements located at precise integer positions along a specified axis, providing a highly optimized approach for data subsetting.

The **take()** method proves particularly efficient when working with large datasets where execution speed is paramount. Unlike indexing methods that might require internal label lookups, **take()** interacts directly with the underlying array structure. This makes it an invaluable tool for complex operations like resampling, bootstrapping, or extracting specific, numerically defined data points where performance overhead must be minimized. A thorough understanding of its specialized syntax and application is essential for maximizing efficiency in advanced data processing within the Pandas environment.

## Deconstructing the take() Function Syntax

The fundamental purpose of the **take()** function is to retrieve data segments based solely on integer positions, intentionally disregarding any explicit row or column labels the DataFrame may possess. This strict reliance on physical order is what distinguishes it from label-aware methods. Applying this method directly to a Pandas DataFrame requires defining two main parameters to accurately specify the desired data slice.

The standard structure for invoking this function is robust and straightforward:

**pandas.DataFrame.take(indices, axis=0, ...)**

The parameters within the **take()** function are designed to offer precise control over which elements are selected and the dimension (row or column) along which the selection occurs. Mastering these arguments is key to accurate and targeted data extraction:

**indices**: This is the mandatory argument. It accepts a sequence (such as an array or list) of integers. These integers explicitly define the exact positional index locations that the function must retrieve from the DataFrame. It is critical that these indices correspond to valid positions within the designated axis.
**axis**: This optional parameter controls the dimension of the selection. By default, it is set to `0`, which specifies that the operation should occur along the rows (the index). Conversely, setting `axis=1` instructs the function to select elements along the columns.

A crucial convention to remember is that all positional indexing in Python, and consequently in

Pandas, is fundamentally **zero-based**. This mandates that the first row or column is always situated at position `0`. The **take()** function strictly adheres to this principle, requiring positional integers rather than relying on explicit row or column identifiers.

## Selecting Rows by Positional Index (Axis 0)

To demonstrate the practical utility of **take()**, let us consider a dataset detailing information about basketball players. We will define a sample DataFrame containing player teams and their average points scored per game. This example focuses on using **take()** to extract specific rows based on their physical, zero-based index position, utilizing the default setting of `axis=0`.

We begin by constructing the sample DataFrame using the Pandas library:

**import pandas as pd**

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

#view DataFrame
print(df)

team points
0 Mavs 29
1 Magic 34
2 Heat 12
3 mavs 15
4 Nets 22
5 hawks 40

If our objective is to isolate the record corresponding to the second entry in the dataset--the 'Magic' team record--we must remember that indexing starts at `0`. Therefore, the second row is located at index position `1`. We apply the **take()** function, embedding the desired position within a list of indices:

**#extract element located in index position 1 along rows**
**df.take()**

team points
1 Magic 34

This output confirms that the function successfully retrieves the entire row at position `1`. A key strength of **take()** is its efficiency in extracting multiple, non-contiguous rows in a single operation. For instance, if we need rows corresponding to positions 1, 2, 4, and 5, we simply supply all these indices as a list argument. This capability underscores the function's utility for complex, position-based data filtering:

**#extract element located in index positions 1, 2, 4, 5 along rows**
**df.take()**

team points
1 Magic 34
2 Heat 12
4 Nets 22
5 hawks 40

## Enhancing Data Selection with Negative Indexing

A significant operational benefit of employing the **take()** function is its robust, native support for **negative indexing**, a standard convention in Python for sequencing. When applied to a Pandas DataFrame, negative indices allow us to reference elements relative to the end of the sequence. Specifically, an index of `-1` targets the last element, `-2` targets the second-to-last, and so forth. This feature is exceptionally valuable when dealing with datasets of unknown or variable size, or when the explicit goal is to target the most recent or final records without calculating the total row count.

To illustrate this, imagine we need to retrieve the final row and the third-to-last row of our basketball player dataset. Rather than determining the absolute positive indices (which are `5` and `3`, respectively, in this small example), we can use the intuitive negative positions `-1` and `-3`. This practice results in cleaner, more scalable code that remains functional even if the dataset size changes dramatically.

We execute the selection by passing the negative integer array as the `indices` argument:

**#extract element located in index positions -1 and -3**
**df.take()**

team points
5 hawks 40
3 mavs 15

The resulting subset confirms the successful extraction of the last (index 5) and third-to-last (index 3) rows. Crucially, the output order of the rows precisely mirrors the order specified in the input `indices` list (). This demonstrates the precise control **take()** provides over the resulting structure, ensuring flexibility when focusing on the trailing segments of the data.

## Selecting Columns Positionally (Axis 1)

While the default mode of operation for **take()** is along the row axis (`axis=0`), its utility extends fully to column selection by simply modifying the `axis` parameter. Selecting columns based on their numerical position is achieved by explicitly setting `axis=1`. This technique is highly relevant when dealing with dataframes where column names are lengthy, repetitive, or unknown, but their sequential order remains reliable.

Considering our sample DataFrame, which contains two columns: 'team' (position 0) and 'points' (position 1). If the requirement is solely to extract the 'points' column, we must specify index position `1` and ensure the axis is set to `1`. We pass the index position `1` as the argument for `indices`.

The command structure for this column selection task is clean and unambiguous:

**#extract element located in index positions 1 along columns**
**df.take(, axis=1)**

points
0 29
1 34
2 12
3 15
4 22
5 40

The output is a Pandas Series containing all values exclusively from the **points** column, corresponding exactly to the column located at positional index 1. This method bypasses the reliance on column labels entirely, relying only on the zero-based order. This reinforces **take()** as a concise and powerful method for strict numerical column subsetting.

## take() vs. .iloc: Understanding the Nuance

Within the Pandas ecosystem, data retrieval is typically handled by methods like `.loc` (for label lookups) and `.iloc` (for integer positional lookups). While `.iloc` also relies on integer positions,

the **take()** function often provides distinct advantages, primarily concerning operational performance and specific behaviors required for array-style selection. The core difference lies in their architectural implementation and how they process the input index array, particularly concerning index repetition and ordering.

The `.iloc` indexer is designed to accept standard Python slicing conventions or lists of indices, functioning as the primary integer indexer for the DataFrame structure. In contrast, **take()** is specifically optimized for selecting elements from an array (or DataFrame) using an array of positional integers, often closely mimicking the highly efficient array selection capabilities found in NumPy. For computationally demanding tasks, such as generating bootstrap samples where indices are generated programmatically and may contain duplicates, **take()** can sometimes offer marginal performance benefits over the general-purpose `.iloc`.

It is vital to grasp the concept of **physical alignment** when utilizing these positional tools. Both `.iloc` and **take()** ignore any custom index labels of the DataFrame. If a DataFrame uses non-sequential identifiers (like dates, strings, or UUIDs) as its index, both methods still default to the internal, zero-based physical order of the rows. However, **take()** is uniquely engineered to select items based on their position in the underlying data structure, making it the most reliable choice when positional integrity and guaranteed output ordering are the primary concerns.

### Conclusion: Why take() Matters for Positional Indexing

The **take()** function stands out as an indispensable tool within the Pandas library, enabling exceptionally fast and precise positional indexing along both the row and column axis. By accepting an array of integer positions, and inherently supporting the flexibility of negative indices, it delivers a highly efficient mechanism for extracting specific subsets of data without dependence on index labels.

Regardless of whether the goal is to extract the initial few rows, target the final records, or isolate specific columns based solely on their order, **take()** provides a clean, specialized, and performance-driven syntax. For users requiring comprehensive documentation detailing all parameters, return values, and edge case behaviors, the authoritative source is the official Pandas documentation for the take() function.

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

## Featured Posts

5 Statistical Biases to Avoid

April 25, 2024

5 Free Statistics Courses for Beginners

April 19, 2024

5 MIT Statistics Courses That Are Free

April 18, 2024

5 Free Books to Learn Statistics

April 18, 2024

How to Use the info() Method in Pandas

April 12, 2024

How to Use pct_change() in Pandas

April 12, 2024