How do I Get Row Numbers in a Pandas DataFrame?

Authored by

stats writer

December 23, 2025

When working with data in Pandas DataFrames, a frequent requirement is retrieving the numerical identifiers--often referred to as row numbers or indices--for rows that satisfy specific criteria. Although standard Python lists use zero-based integer positions, Pandas introduces a more flexible concept: the index. Understanding how to efficiently access and manipulate the DataFrame.index attribute is key to performing advanced data manipulation and subsetting operations in Python. This guide delves into the powerful combination of Boolean filtering and the `.index` attribute to accurately pinpoint row locations.

Data science workflows frequently demand the ability to isolate and identify the physical location of records based on their content. Whether you are debugging, creating complex masks, or preparing data for visualization, being able to quickly obtain the index labels corresponding to specific values is a foundational skill in data analysis using the Pandas library.

Fortunately, Pandas provides an extremely streamlined method for this using **Boolean Indexing** coupled with the powerful `.index` property. This approach is significantly faster and more "Pythonic" than traditional looping methods when dealing with large datasets.

This detailed tutorial will walk you through multiple practical examples, illustrating how to harness this technique to retrieve, count, and manipulate row indices efficiently, ensuring your code remains clean and performant.

Understanding Indices vs. Row Numbers in Pandas

It is essential to clarify the difference between the physical row position (the implicit row number, 0, 1, 2, ...) and the index label used by Pandas. While often they are identical (especially in DataFrames created without explicit indexing), the DataFrame.index attribute provides the actual label used to reference the rows. This label can be any hashable type--integers, strings, dates, or even tuples--offering flexibility that goes beyond simple numerical counting.

When we talk about "getting row numbers," we are usually referring to obtaining this index label. Pandas facilitates this retrieval by allowing us to create a Boolean mask--a Series of `True`/`False` values--that specifies which rows meet our condition. Applying this mask to the DataFrame and then accessing the `.index` property extracts only the labels associated with the `True` values. This process forms the basis of all index retrieval techniques discussed below.

The resulting object is typically an `Int64Index` or similar index type, which is optimized for fast lookups. This method is preferred over iterative approaches, as Pandas operations are optimized C implementations running much faster than native Python loops.

The Core Mechanism: Using Boolean Indexing and the .index Attribute

The primary method for filtering data and obtaining indices relies on Boolean Indexing. This involves generating a sequence of boolean values where `True` indicates that the corresponding row satisfies the specified condition, and `False` indicates otherwise. By passing this sequence back into the DataFrame, we effectively select a subset of rows.

Once the subset is selected, we simply append `.index` to the filtered DataFrame expression. This crucial step extracts the index labels of the rows that passed the filter. This method is exceptionally powerful because it handles complex filtering logic within a single, readable line of code, drastically simplifying what might otherwise require verbose looping structures.

Let us begin by establishing the sample data structure we will use throughout the examples. We define a DataFrame containing typical sports statistics, including points, assists, and team names.

Example 1: Identifying Rows Based on a Single Column Value

Our first task is to identify the indices of all rows where a specific column contains a predefined value. This is the most common filtering operation and perfectly illustrates the power of combining filtering with the `.index` attribute.

Suppose we have the following Pandas DataFrame (`df`):

import pandas as pd

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

#view DataFrame
print(df)

points assists team
0 25 5 Mavs
1 12 7 Mavs
2 15 7 Spurs
3 14 9 Celtics
4 19 12 Warriors

To determine the row indices where the 'team' column is exactly equal to 'Mavs', we construct a

Boolean Series `df == 'Mavs'`. When we apply this Series back to the DataFrame `df`, we get a subset of the DataFrame. Crucially, accessing .index on this subset yields the index labels 0 and 1.

The implementation is concise and highly effective, providing immediate feedback on the location of the matching entries:

#get row numbers where 'team' is equal to Mavs
df == 'Mavs'].index

Int64Index(, dtype='int64')

This output, an `Int64Index` object, confirms that the team name 'Mavs' is found at the row indices **0** and **1** of our original DataFrame. This method scales extremely well, even when dealing with millions of records, due to Pandas' underlying optimization strategies.

Advanced Filtering: Retrieving Indices Using Multiple Values (.isin())

Often, the filtering requirement is not based on a single strict equality but on membership within a predefined list of possibilities. For instance, we might want to find all rows belonging to either 'Mavs' or 'Spurs'. In Pandas, the `.isin()` method is specifically designed for this purpose, offering a much cleaner syntax than chaining multiple OR (|) conditions.

We first define a Python list containing all target values. We then apply the `.isin()` method to the relevant DataFrame column, which generates the required Boolean Indexing mask. Finally, we apply this mask to the DataFrame and extract the index labels using .index, just as before.

Consider the scenario where we need to find the indices for all rows associated with either the 'Mavs' or 'Spurs' teams. The necessary code defines the list and then performs the filtering operation:

#get row numbers where 'team' is equal to Mavs or Spurs
filter_list =

#return only rows where team is in the list of team names
df.index

Int64Index(, dtype='int64')

The resulting `Int64Index()` clearly shows that the team name is present in our filter list at row indices **0**, **1**, and **2**. This demonstrates the efficiency of `.isin()` for handling set membership queries, a common task in data preparation and filtering.

Extracting a Single Index (When Uniqueness is Guaranteed)

In certain analytical contexts, we might be searching for a unique record, such as finding the index associated with a specific ID or an entry we know appears only once. If you are confident that only a single row will match your search criterion, you can retrieve that scalar index value directly from the resulting index object.

If the Boolean filtering operation returns a list of indices, and we only expect one, we can use standard Python indexing () on the resulting index object to pull out the first--and presumed only--value. This simplifies the data type from an `Int64Index` array into a simple Python integer, which is often more convenient for subsequent operations.

Returning to our sample DataFrame structure:

import pandas as pd

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

If we specifically target the 'Celtics' team, which we know appears only once (at index 3), we can modify our previous approach by appending to the `.index` result:

#get the row number where team is equal to Celtics
df == 'Celtics'].index

3

The output is the integer **3**. It is critical to note that this technique relies on the assumption of uniqueness. If the filter condition were to match multiple rows, using would only return the index of the first match, potentially hiding other valid results. For cases where multiple matches are possible, it is safer to handle the result as a list or index object.

Calculating the Total Count of Matching Rows

In many scenarios, the actual index labels are less important than knowing how many rows satisfy the condition. For instance, we might want to quantify the frequency of a certain category or measure the size of a specific subset. Instead of retrieving the full index array, we simply need the count of elements in that array.

Since the result of `df.index` is an index object (which behaves like an array), we can wrap the entire expression in Python's native len() function. This is highly efficient because Pandas does not need to extract and store the underlying data, only the length of the resulting index array.

Using the same DataFrame initialization:

import pandas as pd

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

If we want to determine the total number of rows where the team is 'Mavs', we apply the length function to the filtered index:

#find total number of rows where team is equal to Mavs
len(df == 'Mavs'].index)

2

The output confirms that the team 'Mavs' appears in a total of **2** rows. While this specific example could also be solved using the `value_counts()` method, retrieving the length of the filtered index is a universal technique applicable to any complex filtering condition.

Alternative Method: Iterating Over Rows for Index Retrieval (.iterrows())

While Boolean Indexing is generally the fastest and most preferred method for data manipulation in Pandas, there are scenarios--particularly when performing row-by-row calculations or handling complex stateful logic--where iteration is necessary. For iterating over the rows and simultaneously accessing both the index and the data, the iterrows() method is available.

The `iterrows()` method yields pairs of (index, Series) for each row in the DataFrame. The index returned is the exact index label of the row. Although useful for specific tasks, this method is fundamentally slower than vectorized Boolean operations and should generally be avoided for simple filtering and index retrieval tasks.

Here is how one might use `iterrows()` to collect indices based on a condition, noting that this is often considered non-idiomatic for performance-critical code:

 # Initialize an empty list to store matching indices

matching_indices =

# Iterate through the DataFrame
for index, row in df.iterrows():
if row == 'Spurs':
matching_indices.append(index)

print("Indices found via iterrows():", matching_indices)

Indices found via iterrows():

As shown, iterrows() successfully retrieves the index, but requires several lines of Python code and incurs performance overhead. The vectorized approach using `df == 'Spurs'].index` remains the superior choice for simple filtering.

Handling DataFrames with Custom Indices

The examples provided so far used the default, sequential integer index (0, 1, 2, ...). However, Pandas is powerful because indices can be custom labels, such as dates, unique IDs, or names. Retrieving "row numbers" in this context means retrieving these custom labels. The core filtering mechanism remains exactly the same, which is a testament to the consistency of the DataFrame.index attribute.

If we redefine our DataFrame using descriptive names as the index, the index retrieval operation returns those names themselves, acting as unique identifiers for the filtered rows, rather than simple integers.

import pandas as pd

# Create DataFrame and set descriptive names as the custom index
df_custom = pd.DataFrame({'points': ,
'assists': },
index=)

# The index is now names, not sequential integers:
print(df_custom)

points assists
PlayerA 25 5
PlayerB 12 7
PlayerC 15 7
PlayerD 14 9

PlayerE 19 12

To demonstrate index retrieval on this custom structure, imagine we want the index labels (Player names) for rows where 'points' are greater than 15:

 # Get the custom index labels where points > 15
df_custom > 15].index

Index(, dtype='object')

In this context, the "row numbers" returned are the labels 'PlayerA' and 'PlayerE'. This reinforces the idea that the `.index` attribute retrieves the index labels, regardless of whether they are default integers or descriptive strings.

Summary and Best Practices for Index Retrieval

Mastering the retrieval of row indices is fundamental to effective data manipulation in Python with Pandas. The consensus best practice centers around vectorized operations using Boolean filtering because of their superior performance and clarity.

To ensure your code remains optimized and readable, always prioritize the following steps:

Vectorization First: Always use Boolean masking (e.g., `df == value`) rather than iterative methods like `iterrows()` for filtering tasks.

Use `.index`: Access the `.index` attribute immediately after filtering to extract the row identifiers without needing to materialize the subset DataFrame entirely.

Leverage `.isin()`: When filtering by multiple discrete values, use the built-in `.isin()` method for cleaner and faster code compared to chaining multiple OR conditions.

Calculate Length with `len()`: If only the count is needed, wrapping the filtered index expression in the len() function is the most efficient way to get the total number of matching rows.

By consistently applying these techniques, you can ensure that your Pandas code is not only accurate but also robust and scalable to handle large datasets effectively.

How to Find Unique Values in Multiple Columns in Pandas
How to Filter a Pandas DataFrame on Multiple Conditions
How to Count Missing Values in a Pandas DataFrame