How to Easily Filter DataFrames with isin() and query()

Authored by

stats writer

November 20, 2025

The process of filtering large datasets efficiently is fundamental to data analysis. When working with the powerful Python library, Pandas, two essential methods often come into play for sophisticated filtering operations: the **`isin()`** function and the **`query()`** method. While **`isin()`** is typically applied directly to a Series object to check for membership against a collection of values, the **`query()`** method provides a highly readable and often optimized way to filter an entire DataFrame using string expressions. Combining the functionality of value list membership checking with the elegance of the **`query()`** syntax is a common requirement for data scientists seeking cleaner, more concise code. This guide details exactly how to integrate list-based filtering logic using the **`query()`** method, specifically demonstrating the appropriate syntax needed to check if values are "in" a specified list.

The core concept revolves around leveraging the string evaluation capabilities of **`query()`**. Instead of calling the functional method **`df.isin()`**, which returns a boolean mask, the **`query()`** method allows us to embed this logic directly into a single string expression. This approach not only streamlines the code but can also lead to performance improvements, particularly when dealing with massive datasets, as Pandas optimizes the execution of the query string internally. Understanding the subtle syntax differences--specifically the use of the **`in`** keyword--is crucial for successful implementation.

Understanding the Pandas query() Method

The **`query()`** method is a powerful utility within the Pandas library designed to filter rows from a DataFrame based on an evaluation expression. It accepts a single string argument that represents a boolean condition, similar to what you might place inside an SQL `WHERE` clause. This method shines because it allows users to write filtering logic using standard Python syntax (like comparison operators and logical operators) without needing to manually create boolean masks and pass them back to the DataFrame using bracket notation.

One of the primary advantages of using **`query()`** is enhanced readability, especially when complex filtering involving multiple conditions across several columns is necessary. It eliminates the visual clutter associated with nested brackets and repetitive DataFrame indexing calls. For instance, filtering rows where column X is greater than 10 and column Y is less than 5 becomes exceptionally clean: `df.query('X > 10 and Y < 5')`. When dealing with membership checking--that is, checking if a value in a column exists within a predefined collection of values--the **`query()`** method offers a special keyword derived from the **`isin()`** function's behavior.

Implementing List Filtering Logic Using 'in'

Often you may want to use the filtering logic equivalent to the **`isin()`** function within the **`query()`** method in Pandas to efficiently filter for rows in a DataFrame where a specific column contains a

value present in a list of target values. While the standard Pandas function is named `isin()`, the string expression accepted by `query()` requires the simple keyword `in` to perform this membership check.

You can use the following syntax structure to perform this list-based filtering. The syntax is highly intuitive, reading almost like natural language: select rows where the specified column is `in` the provided list of values. Note the square brackets surrounding the list of values within the query string itself, indicating the collection to be checked against.

df.query('team in ')

This particular query string instructs Pandas to scan the entire **DataFrame**, filtering for rows where the value in the `team` column is exactly equal to one of the specified elements: A, B, or D. This approach is highly efficient for selecting subsets of data based on categorical criteria, provided those criteria are finite and known beforehand.

Important Syntax Note: When constructing the expression string for the Pandas `query()` method, we must explicitly use the keyword `in` instead of the function name `isin`. Using `isin` will result in a `NameError` or unexpected behavior, as the query engine looks for the specific operator `in` for list membership checks.

Example: Setting Up the Sample DataFrame

To illustrate how this syntax works in a practical context, we will utilize a sample Pandas DataFrame containing hypothetical statistics for several basketball players. This example will clearly demonstrate the initial data structure before applying the powerful `query()` filter.

We begin by importing the Pandas library and defining the data structure. Our sample data includes columns for **team**, **points**, **assists**, and **rebounds**, providing a diverse set of values across different categorical groups. The creation of this DataFrame is the crucial first step before any filtering operations can be performed.

import pandas as pd

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

#view DataFrame

print(df)

team points assists rebounds
0 A 18 5 11
1 A 22 7 8
2 B 19 7 10
3 B 14 9 6
4 C 14 12 6
5 C 11 9 5
6 D 20 9 9
7 E 28 4 12

As shown in the output, we have eight rows of data, spanning five unique teams (A, B, C, D, E). Our goal in the subsequent steps will be to precisely isolate the rows belonging only to teams A, B, and D, effectively demonstrating how the `in` operator within the **`query()`** method achieves the desired filtering outcome without requiring verbose boolean indexing logic.

Filtering Using a Literal List in query()

Now suppose that, based on our analysis requirements, we need to query the DataFrame for all rows where the value in the **`team`** column is equal to either A, B, or D. This is a classic case where list membership checking is required. Using the **`query()`** method provides the most concise way to achieve this result compared to writing chained logical OR conditions (e.g., `(df == 'A') | (df == 'B') | (df == 'D')`).

We can use the following straightforward syntax, embedding the list of target values directly into the query string expression. Remember that since the values (A, B, D) are strings, they must be enclosed in quotes within the list structure , which is itself contained within the single quotes defining the overall query string.

#query for rows where team is in list of specific teams
df.query('team in ')

team points assists rebounds
0 A 18 5 11
1 A 22 7 8
2 B 19 7 10
3 B 14 9 6
6 D 20 9 9

Careful inspection of the resulting output confirms the successful filtering operation. The `query()` function returns only those rows where the value in the `team` column matches A, B, or D. Rows corresponding to team C (index 4 and 5) and team E (index 7) are successfully excluded from the result set, demonstrating the powerful and succinct capability of the `in` operator within the query expression. This method ensures high performance, especially beneficial when running complex filters on extensive datasets.

Advanced Filtering: Referencing External Variables with @

While hardcoding the list of values directly into the query string is suitable for simple, one-off operations, it becomes impractical and difficult to maintain when the list of values changes frequently or is dynamically generated elsewhere in the Python script. Thankfully, Pandas provides a highly flexible mechanism for referencing external Python variables directly within the `query()` string: the `@` operator.

By preceding the variable name with an `@` symbol inside the query string, the Pandas query engine knows to look up the value of that variable in the local environment and substitute it into the expression before evaluation. This significantly improves code modularity and readability, adhering to best practices by separating data definitions from filtering logic. This technique is particularly valuable when the list of values is derived from user input, configuration files, or the result of a prior calculation.

To apply this advanced technique, we first define a standard Python list containing the target team names. We then use this variable, prefixed by `@`, in place of the literal list in the query string:

#create variable to hold specific team names
team_names =

#query for rows where team is equal to a team name in team_names variable
df.query('team in @team_names')

team points assists rebounds
0 A 18 5 11
1 A 22 7 8
2 B 19 7 10
3 B 14 9 6
6 D 20 9 92

The query executes by effectively substituting the list contents of `team_names` into the expression, performing the same filtering operation as the previous example. The resulting DataFrame subset is identical, reinforcing that this method is an equally valid, and often preferred, way to handle list

membership checks when the filtering criteria are dynamic or stored outside of the immediate query call.

Comparison: query() versus Boolean Masking with isin()

While the focus here is on leveraging the **`query()`** method, it is valuable to briefly compare its syntax for list filtering against the traditional Pandas method using boolean indexing and the dedicated **`isin()`** function. Traditional boolean masking requires two distinct steps: first generating the mask, and then applying it.

To demonstrate the structural differences, consider the implementation steps for filtering based on our target list of teams:

Traditional Boolean Masking: To filter by a list of teams, one would typically write: `target_teams = `, and then `filtered_df = df.isin(target_teams)]`. This method is explicit and clear, relying heavily on the **`isin()`** function applied directly to the target column (Series).

`query()` Method: The equivalent operation using **`query()`** is achieved in a single command: `filtered_df = df.query('team in @target_teams')`. This approach condenses the operation into a single method chain, enhancing functional style programming and often offering superior performance due to internal optimizations within the Pandas engine which processes the query string as a whole.

For simple list checks, the difference in performance is often negligible. However, when complex filtering involves combining list checks with other numerical or relational comparisons (e.g., `team in and points > 15`), the **`query()`** method offers superior syntactic brevity and readability, making the code much easier for others (or your future self) to interpret and maintain. The string-based syntax allows for a highly declarative style of data manipulation.

Summary of Best Practices for in/query()

Successfully utilizing the **`query()`** method for list membership checks requires adherence to a few key best practices to ensure code correctness, readability, and optimal performance. Firstly, always remember the specific syntax requirement: use the keyword **`in`** within the query string, not the functional name **`isin`**. This is the single most common source of error for newcomers to the **`query()`** method, resulting from conflating the Python function name with the necessary string operator.

Secondly, when the list of values is long, dynamically generated, or used across multiple queries, it is strongly recommended to define the list as a separate Python variable and reference it using the `@` operator. This practice avoids embedding large data structures directly into string literals, which

significantly improves code clarity and maintainability. For example, replacing a lengthy query string containing dozens of quoted items with `'col in @long_list'` makes the code modular and much easier to debug.

Finally, remember that the DataFrame column name must be provided directly (e.g., `team`), and the list of values must match the column's data type. If the column contains strings, the list must contain strings, properly quoted inside the string definition. By following these guidelines, you can effectively harness the combined power of **`query()`** and the **`in`** operator for highly efficient and clean data manipulation in Pandas.