How to Use Variables in Pandas `query()` Function: A Simple Guide

Authored by

stats writer

November 20, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Use Variables in Pandas `query()` Function: A Simple Guide*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=98388

ARABPSYCHOLOGY.COM

Introduction to Variable Usage in the query() Function

The query() function, a powerful and highly efficient method provided by the Pandas library, allows users to filter the rows of a DataFrame using expressions written as strings. This approach offers significant advantages in terms of readability and execution speed, as it leverages the underlying architecture of highly optimized libraries like NumExpr. Crucially, the function is designed to seamlessly integrate external Python elements, such as objects or local variables, directly into the filtering logic.

Incorporating external Python variables into a query expression drastically improves the flexibility of data manipulation scripts. Instead of hardcoding values directly into the query string--a practice that makes code brittle and difficult to maintain--users can define dynamic parameters outside the query scope and reference them within. This separation of logic and parameterization is essential for creating reusable functions and sophisticated analytical pipelines where filtering criteria frequently change based on user input or iterative processes.

To successfully reference a local Python variable within the query string, Pandas utilizes a specific referencing mechanism. The variable must be prefixed with the at-symbol (@). For instance, if you have a Python variable named `target_value`, you would reference it in the query string as `@target_value`. This explicit annotation signals to the Pandas parsing engine that the subsequent identifier should be treated not as a column name within the DataFrame, but rather as an external object whose value should be substituted into the expression at runtime.

This functionality is particularly useful when constructing complex conditions involving various comparison operators or when the criteria itself is dynamic. It enables powerful, SQL-like filtering capabilities directly within the Python environment, allowing data scientists and analysts to write highly expressive and concise data manipulation code. The ability to use Python variables within the query string ensures that the logic remains consistent whether dealing with string, numeric, or boolean data types, making the query() function a cornerstone of advanced Pandas workflows.

Basic Syntax for Referencing Variables

The syntax for incorporating a local variable into a Pandas query is straightforward yet mandatory for proper execution. Failure to use the `@` prefix will result in the parser interpreting the variable name as an actual column name, invariably leading to a `NameError` or unexpected filtering behavior if a similarly named column exists. It is critical to establish the variable in the local scope before calling the query() function, ensuring that its value is readily available for substitution.

Consider a scenario where we wish to filter a DataFrame based on a specific team name that is stored in a Python variable called `team_name`. The structure of the query string will define the relationship between a column (e.g., `team`) and the external variable (`@team_name`) using an

ARABPSYCHOLOGY.COM

appropriate comparison operator, such as the equality operator (==). This method enhances code clarity significantly, especially when the value being filtered against is a long string or a derived metric.

The standard pattern utilizes method chaining on the DataFrame object, passing the required expression enclosed in quotation marks. The following demonstrates the fundamental syntax required to execute this type of variable-dependent filtration, assuming `df` is a defined Pandas DataFrame and `team_name` holds the criterion value.

You can use the following syntax to use the **query()** function in pandas and reference a variable name:

df.query('team == @team_name')

This particular query searches for rows in a Pandas DataFrame where the **team** column is equal to the value saved in the variable called **team_name**.

The following example shows how to use this syntax in practice.

Practical Demonstration: Filtering Data with a Single Variable

To illustrate the efficacy of using external variables, let us construct a sample DataFrame relevant to sports statistics. This DataFrame, which we name `df`, contains information regarding basketball players, including their team designation, primary position, and accumulated points. This initial setup establishes the foundation upon which all subsequent filtering operations will be performed, allowing us to clearly track how the query() function interacts with dynamically supplied parameters.

The structure of the data is crucial for understanding the filtering logic. We have three distinct columns: `team` (a categorical string), `position` (also a categorical string), and `points` (a numeric integer). Our goal is often to isolate subsets of this data--for instance, focusing only on players from a specific team. Utilizing a variable ensures that if we later decide to target a different team (e.g., switching from 'A' to 'C'), only the variable assignment needs modification, not the core query logic.

The code snippet below demonstrates the creation of our sample DataFrame using the Pandas library. Following its creation, we display the full contents to provide a clear reference point for verifying the results of our forthcoming filtering operations.

import pandas as pd

ARABPSYCHOLOGY.COM

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

#view DataFrame
print(df)
```

```
team position points
0 A G 22
1 A G 25
2 A F 24
3 B G 39
4 B F 34
5 B F 20
6 B F 18
7 C G 17
8 C G 20
9 C F 19
10 C F 22
```

Executing Dynamic Filtration using the @ Symbol

Suppose the requirement is to specifically extract all records pertaining to Team C. Instead of embedding the string literal 'C' directly into the query expression, which ties the logic to that specific value, we first instantiate a Python variable, `team_name`, and assign it the value 'C'. This approach maintains a **strong** separation between the data criterion and the filtering mechanics, adhering to best practices for maintainable code.

Once the variable is defined, we utilize the query() function and reference `@team_name` within the string expression. The Pandas engine interprets the `@` symbol as a directive to look up the value of `team_name` in the global or local scope and inject that value ('C') into the expression before evaluation. Effectively, the parser transforms `'team == @team_name'` into the literal expression `'team == "C"'`, ensuring highly efficient vector operations are performed.

The output clearly demonstrates that the query() function successfully utilizes the external variable's value to return a subset of the original DataFrame where the `team` column matches 'C'. This confirms the proper implementation of the `@variable_name` syntax for dynamic filtering based on a single criterion.

ARABPSYCHOLOGY.COM

#specify team name to search for
team_name = 'C'

#query for rows where team is equal to team_name
df.query('team == @team_name')

team position points
7 C G 17
8 C G 20
9 C F 19
10 C F 22

Notice that the **query()** function returns all rows where the value in the **team** column is equal to C.

Handling Multiple Variables in Complex Queries

A significant benefit of the query() function is its capacity to handle complex boolean expressions and reference multiple external variables simultaneously within a single string. When filtering data that requires matching against a set of discrete values, utilizing multiple variables connected by logical operators, such as OR (`|`) or AND (`&`), simplifies the filtering process considerably compared to indexing methods like `.loc`. This is especially true when the criteria set is derived dynamically or loaded from an external configuration file.

To demonstrate this advanced usage, consider the requirement to retrieve data for players belonging to either Team A or Team C. We must define two separate variables, `team_A` and `team_C`, each holding the respective team identifier. These variables are then integrated into the query string using the `@` prefix, and the separate conditional checks are joined using the bitwise OR operator (`|`), which serves as the logical OR within the query expression context. The resulting expression, `'team == @team_A | team == @team_C'`, is highly readable and efficiently executed.

This method scales effectively; whether filtering based on two variables or ten, the structural approach remains consistent. By maintaining the criteria externally, analysts can swap out complex lists or criteria sets without altering the underlying data filtering implementation. The following example illustrates how to define two separate team variables and subsequently leverage the logical OR operator to retrieve records matching either criterion.

#create two variables
team_A = 'A'
team_C = 'C'

ARABPSYCHOLOGY.COM

```
#query for rows where team is equal to either of the two variables
df.query('team == @team_A | team == @team_C')
```

```
team position points
0 A G 22
1 A G 25
2 A F 24
7 C G 17
8 C G 20
9 C F 19
10 C F 22
```

The query returns all of the rows in the DataFrame where **team** is equal to the values stored in one of the two variables that we specified. This result confirms that the expression was correctly parsed to include both 'A' and 'C' as valid team entries.

Distinguishing External Variables from Column Names

One of the most frequent sources of confusion when first utilizing the query() function is understanding the scope resolution mechanism employed by Pandas. When an identifier appears in the query string without the `@` prefix, Pandas assumes it refers to a column name within the target DataFrame. This assumption is fundamental to the function's design, allowing for simple, natural expressions like `'points > 20'`.

Conversely, the explicit use of the `@` symbol mandates that the parser look outside the DataFrame columns and search the local or global Python namespace for a corresponding object. If a variable named `points_threshold` is defined in the Python script, it must be referenced as `@points_threshold` in the query (e.g., `'points > @points_threshold'`). Attempting to use `'points > points_threshold'` when `points_threshold` is a local variable and not a column name will cause the query to fail, or worse, execute incorrectly if a column with the same name exists but holds unexpected values.

This clear distinction is intentional and provides control over expression evaluation. It prevents naming collisions, especially in complex environments where column names might accidentally match external parameters. For instance, if you define a numeric variable called `team` (not recommended, but possible) and try to use it without the `@`, Pandas will prioritize the actual `team` column within the DataFrame, ignoring your external variable. Always prefix external scalar values or objects with `@` to ensure proper value substitution.

ARABPSYCHOLOGY.COM

Using Variables for Numerical and Inequality Comparisons

While our previous examples focused on string equality matching (e.g., matching team names), the real power of variable substitution is realized when performing numerical filtering. Numerical criteria often involve thresholds, minimums, or maximums that are subject to frequent change during data exploration or modeling processes. Using a variable for these thresholds significantly enhances code maintainability and facilitates quick experimentation.

In our basketball dataset, suppose we want to identify players who scored more than a certain number of points. We can define a variable, say `min_points`, and use it in conjunction with various comparison operators (`>`, `<`, `>=`, `<=`). For instance, setting `min_points = 25` and running the query `df.query('points > @min_points')` instantly filters the DataFrame to show only players who exceeded that score.

This approach is particularly valuable when dealing with calculations or derived metrics. Imagine `min_points` is calculated dynamically as the average score plus one standard deviation. By storing this complex calculation result in a single variable and referencing it with `@`, the query string remains short and focused on the filtering logic, rather than containing complex arithmetic. This adherence to clean logic is a hallmark of efficient Pandas usage.

 # Example using numerical threshold
min_points_threshold = 20

# Query for players exceeding the threshold
high_scorers = df.query('points > @min_points_threshold')

print(high_scorers)

team position points
0 A G 22
1 A G 25
2 A F 24
3 B G 39
4 B F 34
10 C F 22

Advanced Usage: Leveraging Lists with the 'in' Operator

The utility of referencing external objects is not limited to scalar values (single numbers or strings). Pandas' query() function also supports passing iterable objects, such as Python lists or tuples, and

ARABPSYCHOLOGY.COM

using them in conjunction with the powerful `in` operator. This mechanism provides a highly concise and efficient way to filter records where a column value belongs to a predefined collection of acceptable criteria.

If we wanted to filter our data to include only specific positions--for example, only Guards ('G') and Forwards ('F')--we could define a list containing these specific string values: `target_positions =` . When this list is referenced within the query using the `@` prefix, the expression `'position in @target_positions'` effectively checks each row's `position` against every element in the list simultaneously. This is highly optimized and avoids the need to chain multiple OR conditions manually, which quickly becomes cumbersome.

This technique is particularly valuable for parameterizing data validation or inclusion criteria. Imagine loading a list of banned IDs or approved regions from a configuration file; by assigning this list to a Python variable and referencing it using `@list_variable`, the filtering logic adapts immediately without rewriting the query string. This is a crucial feature for data pipelines requiring dynamic exclusion or inclusion sets.

 # Define a list of accepted positions
target_positions =

# Query for rows where the position is contained within the list
all_players = df.query('position in @target_positions')

print(all_players)

team position points
0 A G 22
1 A G 25
2 A F 24
3 B G 39
4 B F 34
5 B F 20
6 B F 18
7 C G 17
8 C G 20
9 C F 19
10 C F 22

ARABPSYCHOLOGY.COM

Performance Considerations and Best Practices

The primary reason developers choose the query() function over traditional boolean indexing (e.g., `df == val`) is often related to performance, especially when handling massive DataFrame structures. The underlying implementation of `query()` leverages the NumExpr engine, which is optimized for fast, vectorized execution of string expressions. When external variables are referenced using `@`, their values are substituted before the expression is handed off to NumExpr, ensuring that the engine executes highly efficient, compiled operations.

To maximize performance and code quality when integrating variables, several best practices should be observed. First, always confirm that the data type of the variable matches the data type of the column being queried. While Pandas is robust, mismatches (e.g., comparing a string variable to an integer column) can lead to unexpected errors or silent failures. Second, strive to keep complex calculations outside the query string itself; calculate the desired threshold or list of values beforehand and store the result in a variable. This keeps the query string clean and ensures that the calculation is performed only once, rather than potentially being repeated during expression evaluation.

Finally, be mindful of the variable's scope. The `query()` function primarily searches the local scope where it is called. If the variable is defined in a global scope or a nested class environment, explicit scope resolution might sometimes be necessary, although the `@` symbol usually handles standard local and global variable access effectively. Utilizing variables not only enhances readability but also solidifies the foundation for highly maintainable and dynamically configurable data processing workflows in Pandas.

Note: You can find the complete documentation for the pandas **query()** function here.

ARABPSYCHOLOGY.COM