

# Pandas: Use Group By with Where Condition

Authored by  
**Mohammed loot**

April 10, 2026

## RECOMMENDED CITATION

Mohammed loot (2026). *Pandas: Use Group By with Where Condition*. PSYCHOLOGICAL STATISTICS. Retrieved from <https://statistics.arabpsychology.com/?p=3411>

When performing data analysis, it is a common requirement to first filter a dataset based on specific criteria and then aggregate the filtered data. In the [Python](#) ecosystem, the [Pandas](#) library provides powerful tools for this, particularly through the combination of its filtering capabilities and the versatile `groupby()` method. This article will guide you through effectively applying a "where" condition before grouping your data in Pandas, focusing on the highly readable and efficient `query()` function.

The most straightforward approach to using a **group by** operation with a **where** condition in Pandas involves leveraging the `query()` function. This method allows you to filter rows using a string expression that resembles SQL syntax, making your code cleaner and more intuitive, especially for complex conditions.

Consider the following common pattern:

```
df.query("team == 'A'").groupby().mean().reset_index()
```

This powerful one-liner first filters a [Pandas DataFrame](#), selecting only rows where the `team` column is equal to 'A'. Subsequently, it calculates the [mean](#) value of `points` for these filtered rows, grouped by their respective `position`. Finally, `reset_index()` converts the grouped output back into a standard DataFrame. This combination ensures precise conditional aggregation, an essential technique in data analysis.

## Understanding GroupBy and Query in Pandas

Before diving into practical examples, it's crucial to understand the individual roles of the `groupby()` and `query()` functions within the Pandas library. These two methods, when combined, offer a highly efficient and readable way to perform conditional aggregations on your datasets. Mastering them is a cornerstone of effective data manipulation in Python.

The [groupby\(\)](#) function is central to many data analysis workflows. It implements the "split-apply-combine" strategy: it splits the data into groups based on some criteria, applies a function (like aggregation, transformation, or filtration) to each group independently, and then combines the results into a new DataFrame. This allows for complex analytical tasks such as calculating sums, means, counts, or other statistics for distinct categories within your data.

Conversely, the [query\(\)](#) method provides a concise way to filter the rows of a DataFrame using a boolean expression. Its primary advantage lies in its SQL-like syntax, which enhances readability, especially for complex filtering conditions involving multiple columns and logical operators. Instead of using bracket notation with [boolean indexing](#), you can express your conditions as a single string, which is then evaluated by Pandas.

The synergy between these two functions is evident when you need to perform aggregations on specific subsets of your data. By first using `query()` to precisely select the rows that meet your "where" condition, and then applying `groupby()` to these filtered rows, you ensure that your aggregations are performed only on the relevant data points, leading to accurate and focused insights. This approach is not only semantically clear but also often optimized for performance on larger datasets.

## Setting Up Our Example Data

To illustrate the practical application of combining `query()` and `groupby()`, let's create a sample Pandas DataFrame. This DataFrame will simulate a dataset containing information about various basketball players, including their team, playing position, and points scored. This relatable context will help in understanding the filtering and aggregation logic.

We begin by importing the Pandas library, typically aliased as `pd`. Then, we construct a DataFrame with three columns: `team` (representing the team the player belongs to), `position` (such as 'G' for Guard or 'F' for Forward), and `points` (the total points scored by the player).

```
import pandas as pd
```

```
#create DataFrame
```

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

```
#view DataFrame
```

```
print(df)
```

```
team position points
```

```
0 A G 22
```

```
1 A G 14
```

```
2 A F 15
```

```
3 A F 10
```

```
4 A F 8
```

```
5 B G 29
```

```
6 B G 33
```

```
7 B F 18
```

As shown in the output, our DataFrame now contains eight rows, each representing a player with their associated team, position, and points. This data will serve as the foundation for our conditional grouping examples, allowing us to demonstrate how to extract specific insights from a

larger dataset efficiently.

## Performing Conditional Aggregation with a Single Criterion

A frequent analytical task involves calculating aggregate statistics for a specific subset of data. For instance, in our basketball player dataset, we might want to determine the average points scored by players on 'Team A', broken down by their playing position. This requires applying a "where" condition (`team == 'A'`) before performing the grouping and aggregation.

The following code snippet demonstrates how to achieve this using the `query()` and `groupby()` functions in combination. We first filter the DataFrame to include only players from 'Team A', then group these filtered players by their 'position', and finally calculate the mean of their 'points'. The `reset_index()` call is crucial here to convert the grouped output from a Series (with 'position' as index) back into a more readable DataFrame format.

```
#calculate mean value of points, grouped by position, where team == 'A'  
df.query("team == 'A'").groupby().mean().reset_index()
```

```
position points  
0 F 11.0  
1 G 18.0
```

Upon executing this code, we receive a new DataFrame containing the average points. From this output, we can clearly observe two key insights:

The mean points value for players in the 'F' (Forward) position on Team A is **11.0**.

The mean points value for players in the 'G' (Guard) position on Team A is **18.0**.

This example highlights the elegance and efficiency of using `query()` for initial filtering before applying aggregations. It streamlines the process, making the code easier to write, understand, and maintain compared to chaining multiple boolean indexing operations. Furthermore, the `query()` method is often optimized internally for performance, especially on larger datasets.

## Handling Multiple Conditions with Logical Operators

Data analysis often requires filtering based on more than one condition simultaneously. For instance, we might want to narrow our focus even further, perhaps to analyze only players from 'Team A' who specifically play as 'Guards'. Pandas' `query()` function gracefully handles such scenarios by allowing the use of standard logical operators directly within its string expression.

To apply multiple conditions, you can use the `&` operator for an "AND" logic (meaning both

conditions must be true) or the `|` operator for an "OR" logic (meaning at least one condition must be true). In our case, we'll use `&` to specify that both the `team` must be 'A' AND the `position` must be 'G'.

**#calculate mean value of points by position where team is 'A' and position is 'G'**

```
df.query("team=='A' & position=='G'").groupby().mean().reset_index()
```

```
position points
```

```
0 G 18.0
```

The output from this query is more specific, revealing only one row. This shows that the mean points value for players in the 'G' (Guard) position on 'Team A' is indeed **18.0**. The `query()` function efficiently processed both conditions, ensuring that only rows satisfying both criteria were passed on to the `groupby()` operation.

This capability of `query()` to handle complex logical expressions within a single string significantly enhances code readability and maintainability. Instead of chaining multiple bracketed filtering operations, which can become cumbersome, you can express your filtering logic in a clear, SQL-like syntax, making your data analysis workflows much smoother.

## Why `query()` is Often Preferred

While Pandas offers several ways to filter DataFrames, including direct boolean indexing, the `query()` method stands out for specific use cases, particularly when dealing with conditional aggregations. Its advantages extend beyond mere syntax and contribute to more robust and readable code.

One of the primary reasons for preferring `query()` is its superior **readability**. The SQL-like syntax allows you to express complex filtering conditions in a natural language style, making the code easier to understand for anyone familiar with database querying. This is especially true when working with multiple conditions involving string comparisons, where traditional boolean indexing can lead to verbose and nested expressions.

Furthermore, `query()` can offer performance benefits, particularly for very large DataFrames. Pandas often optimizes the evaluation of the query string using a fast C engine, which can sometimes outperform equivalent boolean indexing operations, especially when intermediate DataFrame copies might be inadvertently created with the latter. This efficiency is a significant consideration in big data analytics.

Finally, `query()` provides excellent **flexibility**. It can effortlessly refer to variables in the local environment, allowing for dynamic filtering conditions without string formatting. It also supports a

wide range of operators and even allows for complex expressions, enabling sophisticated data subsetting with a concise and consistent approach. This combination of readability, performance, and flexibility makes `query()` an invaluable tool for conditional data manipulation in Pandas.

## Conclusion

Effectively combining filtering with aggregation is a fundamental skill in data analysis, and Pandas provides an elegant solution through its `query()` and `groupby()` functions. As demonstrated, the `query()` method offers a highly readable and efficient way to apply "where" conditions to your DataFrame, closely resembling familiar SQL syntax. This clarity is particularly beneficial when dealing with multiple or complex filtering criteria.

By first filtering your data using `query()` to select the precise subset of rows you need, and then applying `groupby()` to perform aggregations like calculating means, sums, or counts, you ensure that your statistical analyses are focused and accurate. The `reset_index()` function then neatly formats your results back into a DataFrame, ready for further exploration or visualization.

This technique not only streamlines your data manipulation workflows but also enhances the maintainability and understanding of your code. Whether you're working with small datasets or large-scale data, mastering the combination of `query()` and `groupby()` will significantly boost your productivity and the quality of your data insights.

## Additional Resources

To deepen your understanding of Pandas and its powerful features, consider exploring the following resources:

[Pandas User Guide: GroupBy](#) - Official documentation detailing the split-apply-combine strategy.

[Pandas User Guide: The .query\(\) method](#) - In-depth explanation of the query function and its capabilities.

[Pandas DataFrame Filtering](#) - A blog post offering various methods for filtering DataFrames.

[Wikipedia: Data Aggregation](#) - General overview of the concept of data aggregation in statistics and data science.