How to Filter a Pandas DataFrame by Column Values?

Authored by

stats writer

December 6, 2025

Filtering data is one of the most fundamental operations in data analysis. When working with Pandas, the powerful Python library for data manipulation, selecting specific rows based on column values is a daily necessity. A Pandas DataFrame is essentially a two-dimensional labeled data structure, and effective filtering allows data scientists to isolate subsets of interest, perform targeted calculations, and prepare data for visualization or modeling pipelines.

There are several robust methods for achieving precise filtering, including utilizing the boolean indexing technique, employing the explicit label-based indexing of the `.loc` accessor, or leveraging the intuitive syntax of the `.query()` function. Each method offers distinct advantages in terms of readability and performance, depending on the complexity of the conditional logic required. Mastering these techniques is crucial for anyone engaging in serious data processing within the Python ecosystem, ensuring that analysis is both efficient and reproducible.

This comprehensive guide will walk through the primary mechanisms available in Pandas for conditional row selection. We will start by establishing our sample dataset and then proceed through examples demonstrating simple filtering, complex multi-condition logic, and techniques for handling specific data types and missing values. Our goal is to provide a clear, step-by-step understanding, moving beyond simple examples to cover scenarios encountered in real-world data science projects. We will rely heavily on boolean expressions, which act as masks to select only the rows where the condition evaluates to **True**.

To demonstrate these filtering techniques, we will use a small, representative DataFrame tracking fictional sports team statistics. This example dataset provides numeric columns (points, assists, rebounds) and a categorical column (team), allowing us to illustrate various filtering conditions effectively. Before diving into the filtering syntax, let's establish this base DataFrame.

import pandas as pd

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

#view DataFrame
df

team points assists rebounds
0 A 25 5 11
1 A 12 7 8
2 B 15 7 10

3 B 14 9 6
4 C 19 12 6

Filtering Based on a Single Column Using Boolean Indexing

The most conventional and often fastest method for filtering in Pandas is through **boolean indexing**, also known as boolean masking. This technique involves generating a Series of **True** or **False** values--a mask--where the length of the Series matches the number of rows in the DataFrame. When this boolean mask is passed back to the DataFrame, Pandas automatically selects only the rows corresponding to the **True** values, effectively filtering the dataset according to the specified criteria. This direct method is highly readable for simple conditions and avoids the overhead associated with string parsing found in other methods like `.query()`.

To apply a simple filter, we first select the target column using standard bracket notation (e.g., `df`) and then apply a comparison operator (such as `>`, `<`, `==`, or `!=`). For instance, if we wanted to find all records where a team scored more than 15 points, the initial expression `df > 15` would return a boolean Series like . Passing this resulting Series directly back into the square brackets of the DataFrame, as in `df > 15]`, executes the filter, yielding a new DataFrame containing only the qualifying rows. This mechanism is foundational to data manipulation in Pandas and provides excellent performance.

Let's look at a practical example where we isolate rows for Team 'A' and another where we identify entries with points greater than 19. Notice how the syntax remains clean and concise, regardless of whether the filter targets numeric or categorical data. This method is particularly recommended when conditions are simple or when combining conditions programmatically, as it works directly with Pandas Series objects rather than relying on string expressions.

 # Filter 1: Rows where 'team' is 'A'
df == 'A']

team points assists rebounds
0 A 25 5 11
1 A 12 7 8

Filter 2: Rows where 'points' are greater than 19
df > 19]

team points assists rebounds
0 A 25 5 11
4 C 19 12 6

Leveraging the `.loc` Accessor for Conditional Selection

While standard boolean indexing is highly effective, many experts prefer using the `.loc` accessor for filtering, especially when simultaneously selecting specific columns or performing assignments. The `.loc` accessor is primarily designed for label-based indexing, taking two arguments separated by a comma: the row selector and the column selector. When filtering, we supply the boolean expression (the mask) as the row selector argument, making the intent explicit and preventing potential SettingWithCopyWarning issues that sometimes arise with chained indexing.

Using `.loc` is syntactically structured as `df.loc`. By placing our conditional filter--for example, `df > 8`--in the row position, we instruct Pandas to return only those rows that satisfy the condition. The true power of `.loc` emerges when we need to subset the columns of the filtered result. Instead of returning all columns by default, we can specify a list of desired column names in the column position, such as , dramatically cleaning up the output DataFrame.

Consider a scenario where we are only interested in the team name and the points scored for all entries where the number of rebounds is less than 10. By integrating the filtering condition and the column selection within a single `.loc` call, the operation becomes atomic and exceptionally clear. This approach ensures maximum clarity regarding which labels are being used for selection and which columns are being retained in the final, filtered result, making it a preferred pattern for complex extraction operations in production code.

 # Filter for rows where rebounds are less than 10, returning all columns
df.loc < 10]

team points assists rebounds
1 A 12 7 8
3 B 14 9 6
4 C 19 12 6

Filter for rows where assists are greater than 7, returning only 'team' and 'points'
df.loc > 7,]

team points
3 B 14
4 C 19

Combining Multiple Conditions (AND / OR Logic)

Real-world data filtering rarely relies on a single criterion. Data analysts frequently need to combine multiple conditional statements using logical operators to define precise subsets. In

Pandas boolean indexing, the standard Python keywords `and` and `or` cannot be used directly to combine boolean Series. Instead, we must use the bitwise operators: the ampersand (`&`) for the logical AND operation, and the vertical bar (`|`) for the logical OR operation. Critically, each individual boolean expression must be encapsulated within parentheses to ensure correct operator precedence, as bitwise operators have higher precedence than comparison operators, which can lead to unexpected errors if not properly grouped.

The logical AND (`&`) requires that both conditions evaluate to **True** for a row to be included in the filtered DataFrame. For example, if we want to find all players belonging to Team 'B' **AND** who scored more than 14 points, we would write `(df == 'B') & (df > 14)`. Conversely, the logical OR (`|`) includes a row if at least one of the conditions is **True**. If we wanted to see records for Team 'A' **OR** any record where assists exceeded 10, we would use the OR operator: `(df == 'A') | (df > 10)`. Understanding this distinction and correctly using the parentheses are paramount to writing accurate and bug-free filtering code.

Furthermore, it is often necessary to use the logical NOT operator, represented by the tilde (`~`) symbol. The NOT operator negates the result of a boolean expression, selecting all rows that **do not** meet the specified criterion. For instance, to select all rows where the team is **not** 'C', one would use `df == 'C')]`. By mastering the combination of AND, OR, and NOT operators, complex filtering logic can be constructed to isolate extremely specific subsets of data, which is essential for detailed segment analysis and outlier detection within large datasets. Below are examples demonstrating both the AND and OR logic in action.

 # Use AND (&): Find records where team is 'B' AND points are greater than 14
df == 'B') & (df > 14)]

team points assists rebounds
2 B 15 7 10

Use OR (|): Find records where points are less than 13 OR rebounds are greater than 10
df < 13) | (df > 10)]

team points assists rebounds
0 A 25 5 11
1 A 12 7 8

Advanced Filtering with the `.query()` Function

The `.query()` function offers an alternative, more SQL-like syntax for filtering DataFrames. Unlike boolean indexing, which relies on generating boolean Series and using bitwise operators,

`.query()` accepts a string expression that resembles natural code logic. This can significantly improve the readability of complex filtering operations, especially those involving multiple conditions or comparisons between columns, as it allows the use of standard Python logical operators like `and` and `or` directly within the string.

As seen in the original examples, `.query()` simplifies the syntax by automatically handling the comparison against the columns within the DataFrame. For instance, instead of writing `df > 13) & (df > 6)]`, you can simply write `df.query('points > 13 and rebounds > 6')`. This string-based approach is often preferred when the condition is dynamically generated or when dealing with highly complex expressions that would otherwise require deep nesting of parentheses in boolean indexing. However, it is important to note that `.query()` relies on evaluation through the underlying NumExpr engine, which introduces a slight overhead but often provides optimization benefits for extremely large datasets.

One powerful feature of `.query()` is its native support for evaluating variables defined outside the DataFrame context. To reference an external Python variable within the query string, you must prefix the variable name with the `@` symbol. This feature is particularly useful when the threshold for filtering is calculated dynamically or stored in a configuration setting. For example, if you define `min_points = 15`, you can filter using `df.query('points > @min_points')`. This integration is seamless and greatly enhances the flexibility and maintainability of filtering code, allowing for quick adjustments to thresholds without rewriting the core query structure.

 # Example 1: Filter Based on One Column Value (from original content)
df.query('points == 15')

team points assists rebounds
2 B 15 7 10

# Example 2: Combined conditions using query syntax
df.query('points > 13 and assists > 6')

team points assists rebounds
2 B 15 7 10
3 B 14 9 6
4 C 19 12 6

Filtering Based on Values in a List (Set Membership)

A common requirement in data filtering is checking whether the value in a specific column belongs to a predefined set of values, often stored in a Python list or tuple. This is known as checking for **set membership**. The most idiomatic way to achieve this using standard boolean indexing is

through the `.isin()` method, which is highly optimized for this purpose. The `.isin()` method is called on the target Series (the column), and it takes the list of desired values as its single argument. It then returns a boolean Series indicating **True** wherever the value matches any item in the provided list.

For instance, if we wanted to filter our sports data to include only the rows where the 'team' is either 'A' or 'C', defining a list and then using `df.isin()` provides an immediate and clean boolean mask. This approach is far superior to manually chaining multiple OR conditions (e.g., `(df == 'A') | (df == 'C')`), particularly when the list of values is extensive. Furthermore, the `.isin()` method integrates perfectly with the NOT operator (~) for identifying values that are **not** present in the list. By preceding the entire expression with a tilde, `~df.isin()`, we efficiently select all rows corresponding to teams other than 'A' or 'C'.

The `.query()` function also supports set membership checks using the dedicated keywords `in` and `not in`. As demonstrated in the original content, when using `.query()` for set membership, the list of values must be passed as an external Python variable, referenced using the `@` prefix. This method, `df.query('points in @value_list')`, provides a highly readable and self-documenting way to perform the same operation. While boolean indexing with `.isin()` is generally preferred for performance, `.query()` offers better clarity when dealing with complex, multi-faceted queries that mix set membership checks with other relational operators.

 # Method 1: Using the .isin() method with boolean indexing
 # Define list of target teams
target_teams =

Filter for rows where 'team' is in the target list
df.isin(target_teams)]

team points assists rebounds
0 A 25 5 11
1 A 12 7 8
4 C 19 12 6

Method 2: Using .query() with 'in' and '@' variable reference (from original content)
Define list of values for points
value_list =

# return rows where points is in the list of values
df.query('points in @value_list')

team points assists rebounds

0 A 25 5 11
1 A 12 7 8
4 C 19 12 6

# return rows where points is not in the list of values (using not in)
df.query('points not in @value_list')

team points assists rebounds
2 B 15 7 10
3 B 14 9 6

Filtering String (Text) Columns

Filtering on string or object columns introduces specific considerations, particularly when dealing with partial matches, pattern recognition, or case sensitivity. For exact matches, the techniques discussed previously (boolean indexing with == or `.isin()`) work perfectly fine, as demonstrated when filtering by 'team' names. However, when the requirement shifts to matching text based on substrings or regular expressions, Pandas provides access to powerful string methods through the `.str` accessor, which must be chained after selecting the target column (Series.str).

The most common string filtering methods accessible via `.str` include `.str.contains()`, `.str.startswith()`, and `.str.endswith()`. For instance, if our 'team' column contained longer names (e.g., 'Team A-1', 'Team A-2'), and we wanted all records that simply contained 'A' anywhere in the name, we would use `df.str.contains('A')`. Importantly, `.str.contains()`, by default, respects case sensitivity, but this behavior can be easily modified by setting the optional argument `case=False`, which is vital for robust, user-input driven searches.

When filtering requires the complexity of regular expressions--for example, matching records where a team name starts with 'B' followed by any digit--`.str.contains()` is particularly powerful because it natively supports regex patterns. Furthermore, if you are attempting to identify rows based on non-matches, the logical NOT operator (~) can be applied directly to the result of the string method, such as `df.str.contains('C')]`, to exclude all entries containing the letter 'C'. These specialized string methods ensure that filtering categorical and text data is just as flexible and powerful as filtering numeric data.

 # Filter for rows where 'team' starts with the letter 'B'
df.str.startswith('B')]

team points assists rebounds
2 B 15 7 10

3 B 14 9 6

Filter for rows where 'team' contains 'A' (case sensitive by default)
df.str.contains('A')]

team points assists rebounds
0 A 25 5 11
1 A 12 7 8

Filter for rows where team does NOT contain 'A' (using the NOT operator ~)
df.str.contains('A')]

team points assists rebounds
2 B 15 7 10
3 B 14 9 6
4 C 19 12 6

Filtering for Missing Data (NaN Values)

A critical aspect of data cleaning and preparation involves handling missing values, which are typically represented in Pandas by NaN (Not a Number). Filtering a DataFrame to include or exclude rows containing missing data is essential before conducting statistical analysis. Pandas provides two dedicated methods for this purpose that operate efficiently on Series objects: `.isna()` (or `.isnull()`) to identify missing values, and `.notna()` (or `.notnull()`) to identify present, non-missing values.

To demonstrate, let's assume we introduce a missing value into our sample DataFrame. If we wanted to locate every row where the 'assists' count is missing, we would apply the `.isna()` method directly to the 'assists' column and use the resulting boolean mask to filter the DataFrame: `df.isna()]`. Conversely, if our goal is to ensure data quality by only keeping rows that have complete information in a specific column--that is, rows where the value is present--we utilize the `.notna()` method. This is typically the operation performed before aggregating data, preventing NaN values from skewing averages or sums.

While these methods are applied to individual columns, it is also possible to filter based on missingness across multiple columns simultaneously using logical operators. For instance, to find rows where either 'points' or 'rebounds' are missing, one would combine the `.isna()` conditions using the OR operator (`|`): `df.isna() | df.isna()]`. This flexible handling of missing data is crucial for robust data pipelines, allowing analysts to selectively include or exclude rows based on the completeness status of the data they contain, ensuring that subsequent analytical steps are based on sound, reliable records.

 # Add a NaN value for demonstration
import numpy as np
df.loc = np.nan

Filter for rows where 'assists' is missing (NaN)
df.isna()]

team points assists rebounds
1 A 12 NaN 8

Filter for rows where 'assists' is NOT missing (present)
df.notna()]

team points assists rebounds
0 A 25 5.0 11
2 B 15 7.0 10
3 B 14 9.0 6
4 C 19 12.0 6

Performance Considerations for Filtering Methods

When selecting a filtering method in Pandas, especially for production environments or large datasets, performance considerations become highly relevant. Although boolean indexing, the `.loc` accessor, and the `.query()` function all achieve the same logical result, their underlying implementation and computational efficiency differ. Generally, **boolean indexing** (using the mask directly, e.g., `df > value]`) is often the fastest method for simple, single-condition filters because it operates directly on NumPy arrays, minimizing Python object overhead.

However, the `.query()` function, while introducing string parsing overhead, leverages the high-performance NumExpr engine. For complex queries involving multiple, chained conditions on very large DataFrames (typically over a million rows), NumExpr's optimization can sometimes make `.query()` faster than constructing the equivalent complex boolean mask manually using `&` and `|` operators. This efficiency gain stems from NumExpr avoiding the creation of intermediate boolean Series objects for each sub-condition, which reduces memory usage and improves cache locality.

For most day-to-day operations on medium-sized datasets, the difference in speed between these methods is negligible, and therefore, the choice should prioritize code readability and maintainability. Boolean indexing is syntactically clean for simple tasks, while `.loc` is superior when simultaneous column selection or mutation is necessary. When writing highly complex, multi-variable queries, `.query()` often provides the most readable solution, helping to prevent errors caused by mismatched parentheses in boolean indexing. Ultimately, an expert Pandas user should

be proficient in all three methods, selecting the one that best balances performance, clarity, and specific task requirements.