How to Use isin() to Check Multiple Columns in a Pandas DataFrame

Authored by

stats writer

November 20, 2025

The ability to efficiently query and filter data is foundational to modern data analysis, particularly when working with large datasets in pandas. The native isin() method provides a powerful and vectorized way to perform membership checking across your data structure. While commonly used for single-column filtering, its true utility shines when applied to check for the existence of specific value combinations across multiple columns simultaneously within a DataFrame.

Unlike simple equality checks, the isin() method operates by taking a list or series of target values and returning a Boolean object (a DataFrame of True/False values). A value of `True` indicates that the corresponding element is present in the specified list of values, and `False` otherwise. This vectorized approach is highly optimized compared to traditional loop-based filtering mechanisms, making it essential for high-performance computing in Python.

When extending this functionality to multiple columns, we are typically looking for one of two logical outcomes: either all checked columns must meet their respective criteria (an **AND** condition), or at least one of the checked columns must meet its criterion (an **OR** condition). Achieving these complex filters requires combining `isin()` with other powerful aggregation methods like all() or any(), applied across the axis representing the columns (`axis=1`).

The Mechanism of pandas.DataFrame.isin()

Understanding how the `isin()` function works when applied to a subset of columns is key to mastering multi-criteria filtering. When you apply `df].isin()`, the method does not check if the entire row matches the list; rather, it broadcasts the list of values against every cell in the selected subset of columns. The result is a new Boolean DataFrame whose shape matches the subset of columns used in the initial selection.

For example, if you select two columns, 'team' and 'position', and check `isin()`, the resulting Boolean DataFrame will contain two columns. For any given row, the 'team' column will be `True` if its value is 'A' or 'Guard', and the 'position' column will be `True` if its value is 'A' or 'Guard'. This initial result only shows cell-level matches. To condense this result back down to a single Boolean Series that can be used for Boolean Indexing on the original DataFrame, we must aggregate the results across the rows (`axis=1`).

This is where the distinction between the **AND** logic (using `all()`) and the **OR** logic (using `any()`) becomes critical. If we require that *both* the team and the position must match one of the specified values, we use `all(axis=1)`. Conversely, if we accept a match as long as *either* the team *or* the position matches one of the specified values, we utilize `any(axis=1)`. This methodology provides extreme flexibility in defining complex filtering criteria based on membership testing.

Method 1: Applying the AND Condition using all(axis=1)

When analyzing structured data, we frequently encounter scenarios where we need rows that satisfy multiple, specific conditions simultaneously. This is the definition of an **AND** operation. In the context of the `isin()` method applied to multiple columns, achieving an AND condition requires every column in the selection to yield `True` for the given row. This is implemented using the all() function, which checks if all values along a specified axis are `True`.

By setting `axis=1`, we instruct pandas to evaluate the Boolean results horizontally, across the columns for each row. Only if every single column in the temporary Boolean DataFrame returns `True` will the final aggregated result for that row be `True`, allowing it to be included in the filtered subset. This is often the preferred method when attempting to isolate precise combinations of features, such as finding players who belong to Team 'A' **AND** play 'Guard' position.

The following syntax illustrates this precise filtering mechanism:

df = df].isin().all(axis=1)]

This particular example filters the DataFrame for rows where the **team** column is equal to 'A' **and** the **position** column is equal to 'Guard.' It is important to remember that the list is treated as a set of acceptable values for *all* selected columns. Since we use `all(axis=1)`, for a row to be kept, the value in the 'team' column must be 'A' or 'Guard' **AND** the value in the 'position' column must be 'A' or 'Guard'. In practice, if 'team' is 'A' and 'position' is 'Guard', both conditions are met, resulting in a successful AND condition.

Method 2: Applying the OR Condition using any(axis=1)

In contrast to the strict requirements of the AND condition, an **OR** operation allows for greater inclusiveness, keeping rows if at least one of the specified criteria is met. When applied to multi-column filtering using `isin()`, we achieve the OR logic using the any() function. The `any()` function returns `True` if even a single value along the specified axis is `True`.

By specifying `axis=1`, we tell pandas to check if any of the columns in the Boolean intermediate result satisfy the membership condition for that specific row. This method is exceptionally useful for broad filtering, such as identifying rows where the entity is associated with 'Team A' **OR** holds the 'Guard' position, regardless of the other column's value. If either criterion is met, the row is retained.

The corresponding syntax utilizes `any(axis=1)`:

df = df].isin().any(axis=1)]

This construction filters the DataFrame for rows where the **team** column is equal to 'A' **or** the **position** column is equal to 'Guard.' If a row has 'B' for team but 'Guard' for position, it passes the filter because the position condition is met. This inclusive nature of `any()` makes it ideal for maximizing the scope of the search criteria.

Preparing the Data: Setting up the Sample DataFrame

To demonstrate these powerful filtering methods, we will first create a sample DataFrame representing player statistics. This DataFrame contains categorical data (team and position) suitable for membership checking, alongside numerical data (points).

This setup allows us to clearly observe how the application of `all(axis=1)` versus `any(axis=1)` radically changes the resulting filtered dataset. We import the necessary `pandas` library and define the data structures below.

Review the initial data structure carefully. We have players split across two teams ('A' and 'B') and two positions ('Guard' and 'Forward').

import pandas as pd

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

#view DataFrame
print(df)

team position points
0 A Guard 11
1 A Guard 18
2 A Forward 10
3 A Forward 22
4 B Guard 26
5 B Guard 35
6 B Forward 19
7 B Forward 12

Deep Dive into Example 1: Filtering for Exact Matches (AND Logic)

Our goal in this first example is to isolate players who are strictly members of Team 'A' **and** play

the 'Guard' position. This requires a strict intersection of criteria, which is perfectly suited for the `isin()` method combined with all(axis=1). The list we pass to isin() is .

When this code executes, the intermediate Boolean DataFrame generated by `isin()` will look for 'A' or 'Guard' in both the 'team' and 'position' columns. For instance, Row 2 (A, Forward) would yield `True` for 'team' but `True` for 'position' (since Forward is not 'A' or 'Guard', wait, this is where the flexibility of isin applied to multiple columns can be confusing, the list is applied to all selected columns). Correction: If we use , we are checking if the value in 'team' is 'A' or 'Guard', AND if the value in 'position' is 'A' or 'Guard'.

Let's refine the logic: Row 2 has team='A', position='Forward'. The 'team' column check is `True` (A is in). The 'position' column check is `False` (Forward is not in). Since `all(axis=1)` requires both to be `True`, Row 2 fails. Only rows where both columns contain values from the list will pass.

#filter rows where team column is 'A' and position column is 'Guard'
df = df].isin().all(axis=1)]

#view filtered DataFrame
print(df)

team position points
0 A Guard 11
1 A Guard 18

Notice that only rows 0 and 1 satisfy the rigorous criteria: 'team' must be either 'A' or 'Guard', **AND** 'position' must be either 'A' or 'Guard'. Since both 'A' and 'Guard' are present in the corresponding columns for these two rows, they are retained. This demonstrates the effective use of Boolean Indexing combined with the `all()` aggregator to enforce multiple membership requirements across different columns.

Deep Dive into Example 2: Filtering for Partial Matches (OR Logic)

For our second example, we utilize the same list of desired values, , but this time we only require that **at least one** of the selected columns satisfies the membership check. We accomplish this by swapping `all(axis=1)` for any(axis=1). The use of `any()` makes the filter highly inclusive, capturing any row where the team is 'A' or 'Guard', OR the position is 'A' or 'Guard'.

Consider the original DataFrame again. Rows 0 and 1 (A, Guard) clearly pass. Row 2 (A, Forward) passes because the 'team' is 'A'. Row 4 (B, Guard) passes because the 'position' is 'Guard'. Even though 'B' is not in our list, the match in the 'position' column is sufficient for the OR condition to be met.

#filter rows where team column is 'A' or position column is 'Guard'
df = df].isin().any(axis=1)]

#view filtered DataFrame
print(df)

team position points
0 A Guard 11
1 A Guard 18
2 A Forward 10
3 A Forward 22
4 B Guard 26
5 B Guard 35

The resulting filtered DataFrame is significantly larger than in Example 1, demonstrating the inclusivity of the OR logic. We retained rows 0 through 5. Rows 0, 1, 4, and 5 were kept because the position was 'Guard'. Rows 2 and 3 were kept because the team was 'A'. The key takeaway is that using any() allows analysts to quickly gather all records that satisfy at least one criterion from a defined set of values across multiple attributes.

Summary and Best Practices for Boolean Indexing

The `isin()` method, when paired with the proper aggregation function (`all()` or `any()`), offers an elegant and highly efficient solution for complex, multi-criteria filtering in pandas. This technique is fundamentally based on generating a Boolean mask--a concept central to high-performance data manipulation.

When constructing these filters, always remember the distinction between the two aggregation functions: use `all(axis=1)` for strict **AND** requirements, ensuring every column meets the membership condition; use `any(axis=1)` for inclusive **OR** requirements, accepting rows if even one column meets the condition.

Furthermore, while the examples here used simple lists for the `isin()` argument, this method is flexible enough to accept dictionaries or other DataFrames as input, enabling even more sophisticated filtering where different columns must match different sets of values. Mastering this specific pattern--selecting columns, applying `isin()`, and aggregating with `all()` or `any()`--is a hallmark of efficient Boolean Indexing and crucial for scaling data processing workflows.