How to Filter a Data Frame by Factor Levels in dplyr

Authored by

stats writer

January 31, 2026

dplyr is an essential **R package** renowned for streamlining **data manipulation** and analysis tasks. It forms a core component of the modern R workflow, providing a grammar for data transformation that is both intuitive and highly efficient. A fundamental operation within any analysis workflow is the ability to subset data based on specific criteria, and **dplyr** excels at this through its powerful **filter** function.

When working with survey results, experimental data, or any dataset containing categorical variables, these variables are often stored as **factors** in R. Factors represent a specific challenge because they possess two distinct properties: the human-readable text labels (e.g., 'A', 'B', 'C') and the underlying numerical integer levels (e.g., 1, 2, 3). Understanding how the **filter** function interacts with these dual properties is crucial for accurate and efficient data subsetting.

This comprehensive guide explores the two primary, expert-level techniques available within **dplyr** for filtering a **data frame** based on a factor variable. We will demonstrate how to target specific factor labels directly using logical operators and, alternatively, how to leverage the underlying numerical levels for filtering based on order or magnitude.

Advanced dplyr Techniques: How to Filter Based on a Factor

Understanding the Dual Nature of Factors in R

Before diving into the filtering syntax, it is vital to grasp how R manages **factors**. A factor is essentially an integer vector where each unique integer corresponds to a specific text label, or "level." For instance, if a factor variable represents 'High', 'Medium', and 'Low', these might be stored internally as 3, 2, and 1, respectively. This internal numerical representation is what allows R to perform certain statistical operations efficiently.

When you attempt to **filter** a column defined as a factor, the operation must explicitly address whether it is comparing the row value against the readable text label or against the underlying integer level. Failure to specify this distinction can lead to unexpected errors or incorrect subsetting results, particularly if the intention is to filter based on the inherent order of the categories (e.g., all categories ranked higher than 'Medium').

The two reliable methods presented below provide precise control over whether the filtering logic targets the categorical labels or the numerical levels, ensuring robustness in your **data manipulation** workflow using the **dplyr** **R package**. We will use the `%in%` operator for label matching and the `as.integer()` function for level matching.

Setting Up the Demonstrative Data Frame

To illustrate these filtering methods in practice, we will construct a small **data frame** containing information about basketball players. This dataset includes a team designation (a factor variable) and the points scored (a numeric variable). It is essential to explicitly define the `team` column as a **factor** during creation to accurately simulate the real-world scenario we aim to address.

The following code snippet creates and displays our base **data frame**, `df`, which we will use throughout the examples. Note that the default ordering of factor levels in R usually follows alphabetical order unless explicitly specified otherwise. In this case, 'A' maps to level 1, 'B' to level 2, 'C' to level 3, and 'D' to level 4.

#create data frame for demonstration
df <- data.frame(team=as.factor(c('A', 'A', 'A', 'B', 'B', 'C', 'C', 'D')),
points=c(12, 34, 20, 25, 22, 28, 34, 19))

#view the resulting data frame
df

team points
1 A 12
2 A 34
3 A 20
4 B 25
5 B 22
6 C 28
7 C 34
8 D 19

Method 1: Filtering Based on Factor Labels (Using the `%in%` Operator)

The most straightforward and often preferred technique for filtering **factors** in **dplyr** involves treating the factor column as a string vector and using the `%in%` operator. This method allows the analyst to select rows where the factor label matches one of a specified set of character strings. This approach is highly readable and less prone to errors related to changes in factor level order.

When employing the `filter()` function, the `%in%` operator checks for membership of the column value within a vector of desired values. By supplying a character vector--for example, `c('A', 'C')`--we instruct **dplyr** to return only those rows where the `team` factor label is either 'A' or 'C'. This method directly addresses the human-readable aspect of the categorical data.

This technique is superior when the desired subsetting criteria relate strictly to the names of the groups rather than any inherent numerical ordering. It preserves the integrity of the selection even if the underlying integer levels associated with 'A' or 'C' were to change due to reordering or dropping levels later in the **data manipulation** pipeline.

The following syntax demonstrates how to filter the **data frame** `df` to include only rows where the **team** column has the factor label 'A' or 'C':

library(dplyr)

#filter rows where team column is equal to factor label 'A' or 'C'
df %>%
filter(team %in% c('A', 'C'))

team points
1 A 12
2 A 34
3 A 20
4 C 28
5 C 34

As evident in the output, the resulting **data frame** successfully contains only those observations corresponding to teams A and C, effectively isolating the desired subsets based on their assigned categorical labels.

Method 2: Filtering Based on Underlying Factor Levels

While filtering by label is generally safer, there are specialized scenarios where filtering based on the underlying numerical levels of a **factor** is necessary. This is especially true when the factor has been explicitly ordered (an **ordered factor**) or when the criteria are based on a hierarchical or numerical ranking implicit in the level definitions. To achieve this, we must first convert the factor column to its integer equivalent using the `as.integer()` function.

The `as.integer()` function in **R** returns the integer representation of the factor levels. For our example **data frame** `df`, where levels are ordered A, B, C, D, applying `as.integer(team)` will map these labels to 1, 2, 3, and 4, respectively. Once converted to integers, standard numerical comparison operators, such as greater than (>) or less than (<), can be applied within the **filter** statement.

It is crucial to be fully aware of the factor level ordering when using this method. If the factor levels are reordered (e.g., manually defined or changed by a function), the numerical mapping will also

change, potentially breaking the filtering logic. Therefore, this technique is typically reserved for instances where the level ordering is guaranteed and the goal is to filter based on position within that defined hierarchy.

The following demonstrates how to filter the **data frame** to retain only those rows where the underlying numerical level of the **team** column is greater than 2:

library(dplyr)

#filter rows where factor level of team column is greater than 2
df %>%
filter(as.integer(team)>2)

team points
1 C 28
2 C 34
3 D 19

In this outcome, the rows corresponding to teams A (level 1) and B (level 2) have been excluded, leaving only teams C (level 3) and D (level 4). The use of the `as.integer` function successfully accessed the positional data of the factor levels.

Detailed Mapping of Factor Levels to Integers

To provide complete clarity on the mechanism used in Method 2, it is helpful to visualize the conversion performed by the `as.integer` function on our specific **factor** variable, `team`. Since the levels were not explicitly defined, **R** ordered them alphabetically based on the labels present in the data.

This conversion ensures that when we apply numerical logic (like `> 2`), we are querying the position of the factor label rather than its text value. Understanding this explicit mapping prevents misinterpretation of the filtering results.

Factor level 'A' becomes **1**.

Factor level 'B' becomes **2**.

Factor level 'C' becomes **3**.

Factor level 'D' becomes **4**.

Choosing the Optimal Filtering Approach

The decision between filtering by factor labels (Method 1) and filtering by underlying factor levels (Method 2) should be driven by the specific analytical goal and the robustness required of the code. Method 1, using `%in%` with character strings, is generally recommended for most standard **data manipulation** tasks because it is insensitive to changes in the factor level ordering.

If the goal is simply to subset based on group membership (e.g., "include all observations from the Northeast region or the West region"), using the labels provides clear, self-documenting code. This is the safest approach for collaborative projects or scripts that might be run on slightly varied datasets where level ordering might not be identical.

Method 2, leveraging `as.integer()`, is best suited when the factor represents inherently ordered data (e.g., severity ratings: Low, Medium, High) and the filtering criterion is intrinsically positional (e.g., "select all ratings higher than Medium," which translates to levels greater than 2). If you choose this method, always include checks or explicit definitions of the factor levels early in your script to prevent silent errors caused by unexpected level reordering.

Further Exploration in Data Manipulation with dplyr

Mastering factor handling within the **filter** function is a critical step in advanced **dplyr** proficiency. Whether you rely on the stable reference of factor labels or the precise control offered by underlying integer levels, both techniques provide powerful tools for analysts to effectively subset complex categorical data.

The principles demonstrated here extend readily to other **data manipulation** functions in **R**, reinforcing the importance of understanding the data type before applying transformation logic. Continued practice with these methods ensures clean and reproducible code for all your statistical and analytical projects.

The following tutorials explain how to perform other common functions in **dplyr**: