Learning to Filter Data Frames in R Using dplyr’s filter() Function

Authored by

Mohammed looti

November 7, 2025

In the modern environment of R and the greater data science ecosystem, the ability to efficiently isolate specific observations is arguably the most fundamental skill a data analyst must possess. Analysts are routinely required to perform sophisticated **subsetting**, refining a large data frame to contain only the rows that meet precise, predefined logical criteria. Fortunately, what might appear to be a complex task is made exceptionally easy, robust, and highly readable using the powerful **`filter()`** function, a foundational "verb" provided by the essential dplyr package.

The `filter()` function is meticulously designed to select rows based exclusively on the evaluation of logical expressions. This targeted design makes it vastly superior in both clarity and computational efficiency when compared to legacy base R subsetting methods, which often lead to code that is difficult to read and maintain. This comprehensive tutorial will guide you through a series of practical, progressively complex examples that demonstrate how to harness the full potential of `filter()`. Before we begin applying these techniques, we must ensure the necessary library is loaded to access its core functionality.

library(dplyr)

The Foundation: Understanding the Data

For the purpose of illustrating these filtering concepts, this tutorial utilizes the readily available, built-in dplyr dataset known as **starwars**. This particular dataset is an ideal candidate for demonstration because it features a rich mixture of data types, including numeric (height, mass), character (name, hair_color), and categorical columns (species, gender). Successfully constructing accurate filtering logic is entirely dependent on a solid understanding of the column names and their underlying data types. We begin by examining the structure and content of the first few observations in the data set.

#view first six rows of starwars dataset
head(starwars)

A tibble: 6 x 13
name height mass hair_color skin_color eye_color birth_year gender homeworld

1 Luke~ 172 77 blond fair blue 19 male Tatooine
2 C-3PO 167 75 <NA> gold yellow 112 <NA> Tatooine
3 R2-D2 96 32 <NA> white, bl~ red 33 <NA> Naboo
4 Dart~ 202 136 none white yellow 41.9 male Tatooine
5 Leia~ 150 49 brown light brown 19 female Alderaan

6 Owen~ 178 120 brown, gr~ light blue 52 male Tatooine
... with 4 more variables: species , films , vehicles ,
starships

It is important to note that the dataset is outputted not as a traditional R data frame, but as a tibble. A **tibble** is a modernized, enhanced version of the data frame, offering significantly cleaner printing, smarter handling of column names, and generally more predictable behavior, which integrates seamlessly into the dplyr workflow. This streamlined output ensures that data analysis remains focused and manageable, even when dealing with very wide datasets.

Example 1: Filtering Rows Based on Exact Equality

The most straightforward application of filtering involves selecting all rows where a specific variable's value is precisely equal to a single, desired criterion. This fundamental operation is achieved using the double equals sign (`==`), which functions as a relational operator testing for strict equality. This technique is indispensable when working with categorical or character data where the objective is to isolate observations belonging to a single classification, whether it be a specific country, a defined gender, or, as demonstrated here, a particular species.

The following code snippet demonstrates how to filter the `starwars` dataset to include only rows where the `species` variable is strictly equal to the character string 'Droid'. We utilize the pipe operator (`%>%`) to clearly articulate the data transformation workflow: first, take the `starwars` data, and sequentially, filter it based on the specified condition. A critical consideration when working with character data in R is that all text comparisons are inherently **case-sensitive**; 'droid' would not match 'Droid'.

starwars %>% filter(species == 'Droid')

A tibble: 5 x 13
name height mass hair_color skin_color eye_color birth_year gender homeworld

1 C-3PO 167 75 gold yellow 112 Tatooine
2 R2-D2 96 32 white, bl~ red 33 Naboo
3 R5-D4 97 32 white, red red NA Tatooine
4 IG-88 200 140 none metal red 15 none
5 BB8 NA NA none none black NA none
... with 4 more variables: species , films , vehicles ,
starships

As indicated by the output *#A tibble: 5 x 13*, five rows in the dataset successfully met this condition. This confirms that `filter()` accurately evaluated the logical test for every single row and returned only those observations for which the condition evaluated to `TRUE`, effectively isolating the Droids.

Example 2: Combining Conditions Using the AND Operator

In real-world data analysis, **subsetting** rarely involves just one restriction; it typically requires imposing multiple constraints simultaneously. To ensure that a row is only included if it satisfies Condition A **and** Condition B, we must employ the logical AND operator, which is represented by the ampersand (`&`). This operator is inherently restrictive and demanding, meaning that all linked conditions must evaluate to `TRUE` for the row to successfully pass the filter and be retained in the resulting dataset.

We can refine our previous filter to select rows where the species is 'Droid' **and**, concurrently, the eye color is 'red'. This example powerfully illustrates the method for combining two distinct column criteria within a single, highly efficient `filter()` call, yielding a significantly more specific subset of the data than either condition alone would provide. This focused application of Boolean operators is absolutely critical for data analysts seeking to create precise and narrowly defined data segments.

starwars %>% filter(species == 'Droid' & eye_color == 'red')

A tibble: 3 x 13
name height mass hair_color skin_color eye_color birth_year gender homeworld

1 R2-D2 96 32 <NA> white, bl~ red 33 <NA> Naboo
2 R5-D4 97 32 <NA> white, red red NA <NA> Tatooine
3 IG-88 200 140 none metal red 15 none <NA>
... with 4 more variables: species , films , vehicles ,
starships

The output shows that only three rows in the dataset satisfied this stringent condition. By utilizing the `&` operator, the final result set is logically reduced from the five Droids identified in Example 1, confirming that only three of those characters also possess red eyes. This reduction clearly demonstrates the restrictive power inherent in the logical AND operation.

Example 3: Broadening Selection with the OR Operator

In stark contrast to the restrictive nature of AND, the logical OR operator (`|`, represented by the vertical bar) offers a powerful mechanism to broaden the scope of your filter, including any rows that satisfy at least one of the specified conditions. The OR operator is fundamentally inclusive, meaning that rows meeting Condition A, rows meeting Condition B, or rows meeting both conditions simultaneously, will all be retained in the resulting dataset.

We can now filter for rows where the species is 'Droid' **or** the eye color is 'red'. This operation will inclusively pull in all characters classified as Droids (irrespective of their eye color) and simultaneously pull in all characters who possess red eyes (irrespective of their species). This capability is incredibly valuable when performing data analysis where subjects or observations can rightfully belong to multiple, non-exclusive groups or categories.

starwars %>% filter(species == 'Droid' | eye_color == 'red')

A tibble: 7 x 13
name height mass hair_color skin_color eye_color birth_year gender homeworld

1 C-3PO 167 75 <NA> gold yellow 112 <NA> Tatooine
2 R2-D2 96 32 <NA> white, bl~ red 33 <NA> Naboo
3 R5-D4 97 32 <NA> white, red red NA <NA> Tatooine
4 IG-88 200 140 none metal red 15 none <NA>
5 Bossk 190 113 none green red 53 male Trandosha
6 Nute~ 191 90 none mottled g~ red NA male Cato Nei~
7 BB8 NA NA none none black NA none <NA>
... with 4 more variables: species , films , vehicles ,
starships

We observe that seven rows in the dataset satisfied this condition. As anticipated, the resulting set is larger than the restrictive AND example because it includes rows that satisfy only one of the two logical tests, demonstrating the inclusive nature of OR. The careful and deliberate use of `&` and `|`, often combined with parentheses for grouping complex expressions, allows data analysts to effectively translate highly specific research questions into executable R code, precisely defining the analytical boundaries.

Example 4: Checking for Vector Membership Using %in%

A common requirement in data filtering is matching a variable's value against a potentially long list of acceptable possibilities (e.g., filtering characters whose home planet is Alderaan, Coruscant, or Kamino). Writing this using multiple OR statements (`== 'A' | == 'B' | == 'C'`) quickly

becomes cumbersome, unwieldy, and highly susceptible to typographical errors. The elegant and efficient solution provided in R is the **`%in%`** operator, which is designed specifically to check for vector membership.

The `%in%` operator efficiently executes a check to determine whether the value contained in the column (which sits on the left side of the operator) is contained within the collection of specified values (which sits on the right side, typically constructed using the vector creation function `c()`). This specialized approach dramatically simplifies the filtering syntax when working with multiple target categories or labels, significantly improving code readability and minimizing the risk of errors associated with manual OR chaining.

starwars %>% filter(eye_color %in% c('blue', 'yellow', 'red'))

A tibble: 35 x 13
name height mass hair_color skin_color eye_color birth_year gender

1 Luke~ 172 77 blond fair blue 19 male
2 C-3PO 167 75 <NA> gold yellow 112 <NA>
3 R2-D2 96 32 <NA> white, bl~ red 33 <NA>
4 Dart~ 202 136 none white yellow 41.9 male
5 Owen~ 178 120 brown, gr~ light blue 52 male
6 Beru~ 165 75 brown light blue 47 female
7 R5-D4 97 32 <NA> white, red red NA <NA>
8 Anak~ 188 84 blond fair blue 41.9 male
9 Wilh~ 180 NA auburn, g~ fair blue 64 male
10 Chew~ 228 112 brown unknown blue 200 male
... with 25 more rows, and 5 more variables: homeworld , species ,
films , vehicles , starships

The result set confirms that 35 rows in the dataset featured an eye color of blue, yellow, or red. This efficient syntax vividly demonstrates why the `%in%` operator is the decisively preferred method for checking categorical membership within robust dplyr workflows, ensuring both conciseness and speed in data retrieval.

Related: How to Use %in% Operator in R (With Examples)

Example 5: Filtering Based on Quantitative Inequalities and Dynamic Comparison

When filtering data based on quantitative measurements, such as height, mass, or birth year, the focus shifts away from simple equality checks (`==`) toward the use of inequality operators. The `filter()` function seamlessly supports all standard numerical comparisons, allowing for precise control over ranges and thresholds. These operators include: **`>`** (greater than), **`<`** (less than), **`>=`** (greater than or equal to), and **`<=`** (less than or equal to).

The following set of examples is designed to illustrate three distinct and common scenarios encountered when working with the quantitative `height` variable. The first operation establishes a simple upper-bound check, identifying characters who are exceptionally tall. The second example demonstrates how to filter observations that fall within a precise numerical range, which necessitates the combination of two inequality conditions linked explicitly by the AND operator (`&`).

The final, and most advanced, example highlights the immense flexibility of `filter()`: its powerful ability to compare a specific variable against a calculated **summary statistic** directly within the function call. In this instance, we first calculate the average height of all characters in the dataset (using the `mean()` function and setting `na.rm = TRUE` to safely handle any missing values) and then immediately filter the data to retain all characters whose measured height exceeds that dynamically calculated average.

#find rows where height is greater than 250
starwars %>% filter(height > 250)

A tibble: 1 x 13
name height mass hair_color skin_color eye_color birth_year gender homeworld

1 Yara~ 264 NA none white yellow NA male Quermia
... with 4 more variables: species , films , vehicles ,
starships

#find rows where height is between 200 and 220
starwars %>% filter(height > 200 & height < 220)

A tibble: 5 x 13
name height mass hair_color skin_color eye_color birth_year gender homeworld

1 Dart~ 202 136 none white yellow 41.9 male Tatooine
2 Rugo~ 206 NA none green orange NA male Naboo
3 Taun~ 213 NA none grey black NA female Kamino
4 Grie~ 216 159 none brown, wh~ green, y~ NA male Kalee
5 Tion~ 206 80 none grey black NA male Utapau

```
# ... with 4 more variables: species , films , vehicles ,
# starships

#find rows where height is above the average height
starwars %>% filter(height > mean(height, na.rm = TRUE))

# A tibble: 51 x 13
name height mass hair_color skin_color eye_color birth_year gender

1 Dart~ 202 136 none white yellow 41.9 male
2 Owen~ 178 120 brown, gr~ light blue 52 male
3 Bigg~ 183 84 black light brown 24 male
4 Obi-~ 182 77 auburn, w~ fair blue-gray 57 male
5 Anak~ 188 84 blond fair blue 41.9 male
6 Wilh~ 180 NA auburn, g~ fair blue 64 male
7 Chew~ 228 112 brown unknown blue 200 male
8 Han ~ 180 80 brown fair brown 29 male
9 Jabb~ 175 1358 <NA> green-tan~ orange 600 herma~
10 Jek ~ 180 110 brown fair blue NA male
# ... with 41 more rows, and 5 more variables: homeworld , species ,
# films , vehicles , starships
```

The final dynamic example reveals 51 characters whose heights are statistically above the average population height. This sophisticated use case clearly demonstrates how the `filter()` function can dynamically and powerfully interact with other core R functions for complex and comparative data retrieval. Mastering these techniques ensures that analysts can execute precise data subsetting efficiently, forming the basis of all subsequent data manipulation and modeling.

You can find the complete official documentation for the ***`filter()`*** *function here.*