How to Filter Rows in R

Authored by

stats writer

December 23, 2025

Data manipulation is a fundamental skill in statistical computing, and learning how to efficiently filter rows in R is essential for any serious data analyst. Row filtering, often referred to as subsetting, allows users to isolate and focus on specific records within a data frame based on defined criteria. This capability is crucial for cleaning data, exploratory analysis, and preparing specific subsets for modeling.

While the base R installation offers functions like `subset()` using logical expressions, modern data science in R heavily relies on the Tidyverse suite of packages. Specifically, the dplyr package provides highly optimized and readable functions for data wrangling. Among these, the powerful filter() function stands out as the standard tool for selecting rows that satisfy a given set of conditions. This tutorial will focus exclusively on mastering the `dplyr::filter()` function due to its superior performance and ease of use in complex filtering scenarios.

To begin utilizing these powerful tools, we must first load the necessary package. If you do not have it installed, you can use `install.packages("dplyr")`. Once loaded, we will use the built-in `starwars` data frame, provided directly within the dplyr package, to demonstrate various filtering techniques. This dataset contains information on characters from the Star Wars universe and provides excellent examples of both numeric and character variables suitable for subsetting.

library(dplyr)

Understanding the Data Structure and Pipe Operator

Before diving into specific examples, it is helpful to understand the structure of the data we are working with and the syntax we will employ. The data frame `starwars` is a typical tabular dataset. We will primarily use the pipe operator (`%>%`), which is a core concept in Tidyverse programming. The pipe takes the output of one function and passes it as the first argument to the next function, making complex data chains remarkably clear and readable.

Let's examine the initial structure of the `starwars` data to familiarize ourselves with the column names and data types, which is essential for constructing accurate logical expressions for filtering. Note the presence of missing values, represented by `<NA>`, which we will address in later examples, highlighting the importance of data quality checks during the filtering process.

#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

Example 1: Filtering Rows Based on Exact Categorical Match

One of the most frequent filtering operations involves selecting rows where a specific categorical variable is equal to a target value. In R, this is achieved using the double equals sign (==), which tests for strict equality. It is crucial to remember that R is **case-sensitive**, so specifying the value exactly as it appears in the data is paramount for accurate subsetting. If the target value is a character string, it must be enclosed in quotation marks.

To illustrate this, we will isolate all characters from the `starwars` data whose `species` variable is precisely equal to 'Droid'. We utilize the pipe operator (%>%) to feed the dataset directly into the filter() function, simplifying the syntax significantly compared to nested base R calls. The resulting output, an object of class `tibble`, confirms the number of records that satisfy this strict condition, showing the selected rows that meet the criteria.

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 <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 BB8 NA NA none none black NA none <NA>
... with 4 more variables: species , films , vehicles ,
starships

The output confirms that 5 rows met the specified equality condition. This simple application demonstrates the efficiency of the filter() function for quick subsetting based on character strings. It is important to note that when filtering numeric data, you would simply omit the quotation marks around the target value, for example, `filter(height == 172)`. This method provides the most direct way to select records belonging to a single, clearly defined category.

Example 2: Combining Multiple Logical Conditions (AND and OR)

Real-world data filtering rarely relies on a single condition; analysts often need to satisfy multiple criteria simultaneously. The filter() function seamlessly handles complex compound conditions using standard logical operators: the **AND operator (&)** and the **OR operator (|)**. Understanding the distinction between these two operators is fundamental to precise data subsetting, as they define whether all conditions must be true, or if only one condition suffices.

When using the **AND operator (&)**, a row is included in the resulting subset only if **every single condition** specified within the logical expression is met. For instance, if we want to find Droids who also have red eyes, both conditions must evaluate to `TRUE` for a record to be selected. The use of `&` tightens the selection criteria, typically resulting in a smaller, more focused subset of the data. It is good practice to explicitly state the column name for each condition, even if they are logically linked, such as `filter(species == 'Droid' & eye_color == 'red')`.

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

Conversely, the **OR operator (|)** significantly broadens the selection criteria. A row is included if **at least one** of the specified conditions is met. Using the same variables, if we filter for characters who are either a 'Droid' **OR** have 'red' eyes, the resulting dataset will contain all Droids (regardless of eye color) and all characters with red eyes (regardless of species). This operation is useful when exploring heterogeneity or when combining different categories of interest into a single analysis pool. As demonstrated below, using `|` results in a larger subset (7 rows) compared to the restrictive `&` (3 rows).

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

Example 3: Efficient Filtering Using Vector Matching (the %in% Operator)

When you need to filter a variable based on membership within a collection of multiple potential values, using a sequence of OR operators (`|`) becomes cumbersome and difficult to maintain. A much cleaner and more efficient approach is to use the dedicated vector matching operator in R, known as `%in%`. This comparison operator tests whether the values in the variable on the left-hand side are present within the vector or list provided on the right-hand side.

The `%in%` operator is mathematically equivalent to writing several OR conditions, but it significantly streamlines the code, especially when dealing with dozens of potential matches. For instance, instead of writing a long chain of equality checks separated by `|`, we can concisely define the target list using the concatenation function `c()` and apply the `%in%` operator. This technique is invaluable for subsetting data based on specific categories where a row qualifies if its value is one of several possibilities.

Here, we aim to select all characters whose `eye_color` is one of 'blue', 'yellow', or 'red'. Notice how the `c()` function creates a vector of strings that `%in%` checks against every entry in the `eye_color` column of the `starwars` dataset. This approach not only enhances readability but also often results in faster computation compared to chaining many logical operations.

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 resulting `tibble` contains 35 records, confirming that the vector matching successfully combined all rows where the eye color matched any element in our defined list. Mastering the `%in%` operator is key to writing clean, scalable filtering code, especially when the number of target categories is large, providing a powerful shorthand for inclusive filtering.

Example 4: Filtering Numeric Data Using Comparison Operators

When working with quantitative data, filtering often requires the use of comparison operators to define ranges, thresholds, or inequalities. R supports standard operators such as **greater than (>)**, **less than (<)**, greater than or equal to (`>=`), and less than or equal to (`<=`). These operators are vital for identifying outliers, selecting data within specific quartiles, or segmenting based on physical measurements like height or mass in the `starwars` data.

We begin by identifying characters whose height strictly exceeds 250 units. This is a straightforward application of the greater than operator. Only one character, Yarael Poof, satisfies this extreme condition, demonstrating how simple filtering can highlight unusual observations or potential data entry errors that warrant further investigation.

#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

For more nuanced selection, we often need to define a specific range. To find characters with heights between 200 and 220 (exclusive of the endpoints), we combine two separate logical expressions using the AND operator (`&`). This bounding technique is essential for subsetting data into meaningful intervals required for subsequent analysis. Five characters fall within this specified height range, showcasing a powerful method for isolating records within a continuous variable.

#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

The flexibility of the filter() function extends to dynamic calculations. To find all characters whose height is above the overall average height of the population, we calculate the mean of the `height` column directly within the filter call. It is absolutely necessary to include the argument `na.rm = TRUE` within the `mean()` function to instruct R to ignore missing values (`NA`), as the presence of even one `NA` without this argument would cause the `mean()` calculation to return `NA`, resulting in an empty filtered subset. This dynamic filtering capability is essential for comparative analysis.

#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

Example 5: Addressing Missing Data with is.na()

Missing values, represented as `NA` (Not Available) in R, are a common challenge in real-world datasets. Standard comparison operators, such as `height == NA`, will not work as expected because `NA` represents an unknown value, and comparing an unknown value to anything else, even another `NA`, yields `NA`. Since the filter() function only keeps rows where the condition evaluates to `TRUE`, rows resulting in `NA` are automatically dropped. To specifically target or exclude missing data, we must use the dedicated function `is.na()`.

If the primary objective is to analyze only those characters whose height is known, we use the negation operator (`!`) in combination with `is.na()`. The expression `!is.na(height)` evaluates to `TRUE` only for rows where the `height` value is present, thus effectively removing all records with missing height data from the resulting data frame. This is a crucial step in preparing data for many statistical models that require complete observations.

 # Filter out rows where height is missing (NA)
starwars %>% filter(!is.na(height))

A tibble: 82 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 Leia~ 150 49 brown light brown 19 female
6 Owen~ 178 120 brown, gr~ light blue 52 male
7 Beru~ 165 75 brown light blue 47 female
8 R5-D4 97 32 <NA> white, red red NA <NA>
9 Bigg~ 183 84 black light brown 24 male
10 Obi-~ 182 77 auburn, w~ fair blue-gray 57 male
... with 72 more rows, and 5 more variables: homeworld , species ,
films , vehicles , starships

Alternatively, if the goal is specifically to identify and subset only the records that contain missing data in a certain column--perhaps for inspection or targeted imputation--we simply use `is.na()` without the negation operator. For example, `filter(is.na(mass))` would return all characters whose mass measurement is unknown. This dedicated function ensures that analysts can reliably control for data completeness, which is an important part of ensuring the validity of analytical results.

Example 6: Negating Conditions to Exclude Specific Values

The negation operator (`!`) provides a highly readable way to define conditions that must **not** be true. Instead of trying to list every acceptable outcome, sometimes it is easier and more reliable to list the outcomes that should be explicitly excluded. This is particularly effective when dealing with long lists of categories where only one or two need to be removed, simplifying complex logical expressions.

For instance, if we want to select all characters who are **not** human, we can use the negation operator in front of the list matching operation. We define a set of homeworlds (Tatooine and Naboo) that we wish to exclude from our analysis. By placing the negation operator immediately before the vector matching expression, we invert the result of the entire condition. If `homeworld %in% c('Tatooine', 'Naboo')` returns `TRUE`, the `!` converts it to `FALSE`, and the row is dropped, resulting in a cleaner subset containing only characters from other planets.

 # Filter out rows where homeworld is Tatooine OR Naboo
starwars %>% filter(!(homeworld %in% c('Tatooine', 'Naboo')))

A tibble: 62 x 13
name height mass hair_color skin_color eye_color birth_year gender

1 Leia~ 150 49 brown light brown 19 female
2 Ob~ 182 77 auburn, w~ fair blue-gray 57 male
3 Ack~ 180 83 none brown orange 41 male
4 Pal~ 170 75 grey pale yellow 82 male
5 Bib~ 175 NA none light pink NA male
6 Wec~ 129 NA none white blue NA male
7 Mon~ 170 NA auburn fair blue 48 female
8 Adi~ 184 NA none dark blue NA female
9 Kit~ 196 87 none green black NA male
10 Mace~ 188 84 none dark brown 72 male
... with 52 more rows, and 5 more variables: homeworld , species ,
films , vehicles , starships

This method provides clarity and robustness, ensuring that the defined exclusion list is respected across the entire dataset. Utilizing negation effectively allows analysts to quickly pivot from inclusive filtering (selecting what you want) to exclusive filtering (excluding what you do not want), adding another layer of control over the data subsetting process in R and ensuring that the final data frame is tailored precisely to analytical needs.

Conclusion: Mastering the Art of Data Subsetting in R

The ability to efficiently subset and filter data is paramount to reproducible data science workflows. By leveraging the dplyr package and its versatile filter() function, analysts gain precise control over their datasets, enabling them to move beyond simple selection toward complex, multi-conditional filtering.

We have demonstrated how to use basic equality checks, combine multiple criteria using `&` (AND) and `|` (OR), efficiently match against lists using `%in%`, define numeric thresholds, and reliably manage missing values using `is.na()`. These techniques form the backbone of exploratory data analysis and preparatory work for statistical modeling, ensuring data integrity and relevance.

By integrating the pipe operator (`%>%`) with the `filter()` function, R programmers can write data manipulation code that is not only powerful and fast but also highly expressive and easy to read. For those seeking deeper insights or comprehensive technical specifications on function arguments and advanced usage, the official documentation remains the definitive source.

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