How to Filter Rows in `dplyr` Based on the Beginning of a String

Authored by

stats writer

January 31, 2026

The ability to precisely subset datasets is fundamental to data analysis in **R**. The **dplyr** package, a cornerstone of the Tidyverse ecosystem, provides powerful and intuitive tools for robust data manipulation. Among these, the `filter()` function stands out as the primary method for selecting specific rows based on defined criteria. This article focuses on an advanced application of `filter()`: identifying and extracting rows where the values in a designated column begin with a specific character or string pattern.

While standard filtering handles exact matches or numerical ranges efficiently, matching based on the start of a string requires integrating specialized functions from the **stringr** package, specifically `str_detect()`, alongside the logic of **Regular Expressions** (regex). Mastering this technique allows for highly granular data extraction, streamlining the process of cleaning and analyzing complex datasets where categorical variables might contain subtle string variations that need precise identification.

The Essential Syntax for 'Starts With' Filtering

To effectively filter rows based on the initial characters of a column value, we must combine the pipeline operator (`%>%`) provided by **dplyr** with string detection capabilities offered by `str_detect()` from the **stringr** package. This combination allows us to apply a pattern matching logic directly within the data manipulation flow, making the code both readable and powerful.

The core requirement for this operation is the use of **Regular Expressions**. Specifically, the caret symbol (`^`) plays a crucial role in the pattern, indicating that the match must occur precisely at the beginning of the string--it acts as an anchor for the prefix. Without this anchor, the function would simply search for the substring anywhere within the column value, defeating the purpose of a 'starts with' filter.

The following snippet illustrates the basic syntax used to achieve this focused filtration. We assume the installation and loading of both necessary packages before execution:

library(dplyr)
library(stringr)

df %>%
filter(str_detect(position, "^back"))

In this fundamental example, the code is instructing **R** to take the data frame named `df` and retain only those rows where the values in the `position` column begin with the exact string "back". The strict use of `^back` is essential, ensuring that strings like "halfback_role" are correctly excluded if the goal is only to match strings that truly start with the prefix "back".

Setting Up the Demonstration Data Frame

To demonstrate the practical application of string-based filtering, we will establish a simple data frame in **R** containing fictional data related to basketball players and their respective positional roles. This data frame, named `df`, provides varied strings in the `position` column, allowing us to clearly illustrate how the 'starts with' filter operates on messy or complex categorical data.

The dataset deliberately includes different positional roles such as 'starting_guard', 'backup_center', and 'starting_forward', presenting a common scenario where data cleaning or focused analysis requires isolating specific categories based purely on string prefixes (e.g., cleanly separating all 'starting' players from all 'backup' players).

The following code chunk creates and displays the raw data frame that will be used for all subsequent filtering demonstrations:

#create data frame
df <- data.frame(player=c('A', 'B', 'C', 'D', 'E', 'F'),
position=c('starting_guard', 'starting_center', 'backup_guard',
'backup_center', 'starting_forward', 'backup_forward'))

#view data frame
df

player position
1 A starting_guard
2 B starting-center
3 C backup_guard
4 D backup_center
5 E starting_forward
6 F backup_forward

Filtering for Specific String Prefixes (e.g., "back")

Our primary objective is now to isolate all players in the dataset whose positional description begins with the prefix "back". This action is crucial for analysis focused solely on backup roles, automatically excluding all starting players. This task demonstrates the precise power of combining **`filter()`** with prefix pattern detection.

By specifying the regex pattern `"^back"` within the `str_detect()` function, we instruct **R** to evaluate the `position` column for an exact match at the start of the string. Only rows satisfying this stringent condition--having "back" as the literal beginning of the string--are retained by the

`filter()` function.

The execution of the following code yields a concise subset of the original data frame, showcasing only the backup players:

library(dplyr)
library(stringr)

#filter data frame to only contain rows where position column starts with "back"
df %>%
filter(str_detect(position, "^back"))

player position
1 C backup_guard
2 D backup_center
3 F backup_forward

The resulting output confirms the successful application of the filter, leaving us with a data frame containing only three rows (players C, D, and F), all of whom occupy a position starting with "back". This verification step confirms that `str_detect()` accurately identified the target prefix when anchored to the beginning of the string using the caret symbol (^).

Filtering Using a Single Character Prefix

The pattern matching technique is highly flexible and is not limited to multi-character strings; it is equally effective when filtering based on a single initial character. This is particularly useful when analyzing datasets where the first letter of a category provides significant meaning, such as identifying all records beginning with 's' (for 'starting') or 'm' (for 'male').

In our current basketball dataset, we might want to isolate all players whose position starts with the letter 's'. We achieve this by simply modifying our pattern within `str_detect()` to `"^s"`. This precise regex pattern ensures that only strings initiating with a lowercase 's' are matched. Remember that, unless specified otherwise, regex is typically case-sensitive.

Executing the filtration process with the single character constraint provides the following results:

library(dplyr)
library(stringr)

#filter data frame to only contain rows where position column starts with "s"
df %>%

filter(str_detect(position, "^s"))

player position
1 A starting_guard
2 B starting-center
3 E starting_forward

As expected, the resulting data frame successfully isolates players A, B, and E, all of whom hold positions starting with the lowercase letter 's'. This flexibility demonstrates that the 'starts with' methodology is adaptable to prefixes of any length, provided the caret anchor is correctly utilized to denote the beginning of the string.

Deep Dive into Regular Expressions: The Caret Anchor (^)

Understanding the underlying mechanism of string detection is vital for advanced data manipulation. When using `str_detect()`, the pattern argument relies on **Regular Expressions** (regex). Regex provides a powerful, standardized way to search, match, and manipulate text strings based on specific patterns across various programming languages.

The most critical component in achieving the 'starts with' filter is the caret symbol (`^`). In standard regex syntax, the caret acts as an anchor, asserting that the position being matched must be the absolute beginning of the string. Without this anchor, `str_detect()` would return `TRUE` if the substring existed anywhere within the column value.

For instance, if we incorrectly omitted the caret and used `filter(str_detect(position, "center"))`, this would return rows for both "starting_center" and "backup_center" because "center" exists within the string. Conversely, `filter(str_detect(position, "^center"))` would only return a match if the entire string began with "center". This strict reliance on the anchor ensures that only true prefixes are captured by the filter.

Alternative Approaches: Base R and Other String Functions

While the **dplyr** and **stringr** combination offers the most legible and idiomatic solution within the Tidyverse framework, it is worth noting that similar functionality can be achieved using base **R** or other specialized string manipulation packages. Knowing these alternatives can be useful when working in different project environments:

Base R Approach: The native `grepl()` function in base **R** performs pattern matching and returns a logical vector. For example, the equivalent base R code to filter the data would be: `df`. While functional and often faster for single operations, many analysts prefer the pipe-based syntax of

dplyr for improved readability and ease of chaining complex data transformation steps.

The `startsWith()` Function: For simple prefix checks (where no complex regex is needed, i.e., just checking if a string starts with "ABC"), the base **R** function `startsWith(x, prefix)` is highly efficient. However, integrating this function directly within **dplyr**'s `filter()` sometimes requires less intuitive syntax compared to the standard `str_detect()` pattern. The Tidyverse approach using `str_detect()` is generally considered more robust and versatile, as it allows for immediate scaling to more complex regex requirements.

Ultimately, the method involving **`filter()`** and `str_detect()` provides a clean, consistent, and powerful way to handle complex string operations directly within the data transformation pipeline, fitting seamlessly into modern **R** scripting practices and promoting code maintainability.

Conclusion and Next Steps

Filtering data based on precise string prefixes is a mandatory step in many data cleaning and feature engineering tasks. By leveraging the combined strengths of the **`filter()`** function from **dplyr** and the pattern matching capabilities of **stringr**'s `str_detect()`, analysts can efficiently isolate rows that meet specific initial character criteria. We reiterate that the correct application of the caret symbol (`^`) in your Regular Expression is the key technical detail that ensures the match occurs exclusively at the start of the string.

Mastering these precise filtering techniques enhances your ability to manipulate and analyze structured data effectively in **R**. We encourage readers to explore further advanced functions within the **dplyr** and **stringr** packages, such as `str_ends()` for suffix matching or complex character classes for more nuanced pattern detection.

Related Data Manipulation Tutorials

The following resources explain how to perform other common data manipulation and transformation functions using the Tidyverse ecosystem:

How to use `str_ends()` to perform 'ends with' matching in the **dplyr** pipeline.
Advanced uses of `mutate()` for conditional column creation based on string logic.
Techniques for joining multiple data frames using `left_join()` and `inner_join()` for data aggregation.