How to Extract Substrings Using grep() in R

Authored by

Mohammed looti

November 13, 2025

The statistical programming language R provides robust functionalities for manipulating and analyzing textual data. One of the most frequently used tools for searching text within a vector is the **grep()** function, which stands for Global Regular Expression Print. This function is instrumental in identifying elements that satisfy a specific criteria defined by a chosen pattern.

While **grep()** is excellent for filtering data based on the existence of a pattern, analysts often require a more surgical approach: finding elements that match a particular substring and subsequently returning only the residual text or the non-matching component of the original string. This task demands the combined power of **grep()** for selection and the **sub()** function for precise substitution and extraction.

To achieve this targeted extraction, we must chain these two functions. The core logic involves using **grep()** to identify the relevant entries and then passing that subset of strings to **sub()**. The **sub()** function is then configured to replace the identified pattern with a null value or an empty string, effectively isolating and returning the desired remaining substring. The following sections detail the precise syntax and provide a comprehensive, practical example of this powerful technique.

Prerequisites: Chaining grep() and sub() for Extraction

Understanding how **grep()** and **sub()** interact is crucial for successful string manipulation in R. The **grep()** function serves as a preliminary filter. When applied to a vector, it returns the indices or the values of the elements that contain the specified regular expression pattern. In the context of extraction, we typically use **grep()** to subset the original data, ensuring that only strings containing the pattern are processed further.

Once the subset is isolated, the **sub()** function takes over. **sub()** is designed to replace the first occurrence of a matched pattern within a string with a specified replacement string. By providing an empty string (`""`) or, as sometimes simplified in specific regex contexts, a backreference replacement (like `'1'` or `'2'`, depending on the pattern definition), we instruct **sub()** to effectively delete the matched pattern. If the pattern successfully matches the unwanted portion of the string, its removal leaves behind the desired substring.

This method offers an efficient way to clean or standardize textual data, such as removing known prefixes, suffixes, or codes embedded within identifiers. The general, powerful syntax structure for achieving this targeted removal and subsequent return of the remaining text is as follows:

sub('*avs*', '1', df$team)

This specific structure first identifies strings containing the pattern `"avs"` within the **team** column of the data frame **df** using **grep()**. It then uses **sub()** to substitute the matching pattern with a

placeholder (in this implementation, relying on the substitution logic to isolate the remainder), thereby returning only the characters that existed outside the matched pattern within the selected strings.

Example: How to Use grep() and Return Only Substring in R

To illustrate this technique, let us construct a sample data frame in R containing information about various fictional basketball teams. This data frame, named **df**, will serve as our working environment for demonstrating the string extraction process.

We will populate the data frame with a mix of team names, some of which contain the specific substring we intend to isolate and remove. The structure of the data includes columns for **team**, **points**, and **status**:

#create data frame
df <- data.frame(team=c('Mavs', 'Hawks', 'Nets', 'Heat', 'Cavs', 'Mavs2', 'Kings'),
points=c(104, 115, 124, 120, 112, 140, 112),
status=c('Bad', 'Good', 'Excellent', 'Great', 'Bad', 'Great', 'Bad'))

#view data frame
df

team points status
1 Mavs 104 Bad
2 Hawks 115 Good
3 Nets 124 Excellent
4 Heat 120 Great
5 Cavs 112 Bad
6 Mavs2 140 Great
7 Kings 112 Bad

Our objective is to use the combined functions to identify every string in the **team** column that contains the substring `"avs"` and then return only the remaining characters, effectively stripping away the `"avs"` pattern from the matched entries. This demonstrates how to extract prefixes or suffixes when the target pattern is located centrally within the string.

Executing the Substring Extraction Logic

We now apply the chained syntax to the newly created data frame. The **grep()** function first identifies the indices corresponding to `'Mavs'`, `'Cavs'`, and `'Mavs2'`. This subset is then passed to **sub()**, which performs the substitution:

```
#return each substring that matches 'avs' in team column
matches_avs <- sub('*avs*', '1', df$team)

#view matches
matches_avs
```

"M" "C" "M2"

The resulting vector, `matches_avs`, contains all of the non-matched substrings that remained after the `"avs"` pattern was stripped away from the relevant entries. This successful operation confirms that the filtering and substitution process worked as intended, providing a concise output of the residual strings.

Detailed Interpretation of the Extracted Values

It is important to understand the mapping between the original string and the extracted output to ensure data integrity and accurate interpretation of the results. The output vector contains three values, each corresponding directly to one of the matched entries in the **team** column:

The first value, `"M"`, is derived from the original team name `"Mavs"`. The **sub()** function located the substring `"avs"` and replaced it, leaving only the preceding character `"M"`.

The second value, `"C"`, originates from the team name `"Cavs"`. Similarly, the removal of `"avs"` leaves the remaining prefix `"C"`.

The third value, `"M2"`, corresponds to the team name `"Mavs2"`. In this instance, the characters surrounding the matched pattern `"avs"` were `"M"` (prefix) and `"2"` (suffix). Since both of these characters did not form part of the matched pattern, they were concatenated to form the resulting string `"M2"`.

This process demonstrates the precision of combining **grep()** and **sub()**: **grep()** selects the rows of interest, and **sub()** extracts the desired content by defining what needs to be removed based on the matching regular expression.

Handling Case Sensitivity in grep()

A critical consideration when performing string matching in R is the default behavior of the **grep()** function, which is inherently **case-sensitive**. This means that **grep()** will only recognize and match patterns where the case of every character is identical to the pattern specified in the function call. If your data contains variations in capitalization, strict case-sensitivity can lead to missed matches.

For instance, if we were to modify the search pattern in the **grep()** function call to look for `"AVS"` (all uppercase letters), the function would return an empty vector. This occurs because none of the strings in the **team** column--`'Mavs'`, `'Hawks'`, `'Nets'`, etc.--contain the exact sequence `"AVS"`.

To mitigate issues arising from inconsistent capitalization, users can incorporate the `ignore.case = TRUE` argument within the **grep()** function call. This ensures that the search performs a case-insensitive match, capturing variations like "avs," "Avs," or "AVS," thereby providing a more comprehensive result set for the subsequent **sub()** extraction step. Always keep the case-sensitivity default in mind when defining your regular expression patterns for filtering data.

R Resources and Further Reading

Mastering string manipulation is fundamental to data cleaning and preparation in R. The techniques demonstrated here using **grep()** and **sub()** form the basis for more complex text analysis tasks, including tokenization and complex regular expression parsing. Exploring the full capabilities of these functions, including their variants like **grepl()**, **gregexpr()**, and **gsub()**, will significantly enhance your analytical toolkit.

The following related tutorials explain how to perform other common tasks in R, providing context for advanced data handling:

Related: R: How to Use grep() to Find Exact Match

<!--

Featured Posts

-->