How to Easily Extract a String Between Two Characters in R

Authored by

stats writer

November 21, 2025

Extracting specific substrings from larger character sequences is a fundamental task in data cleaning and analysis using R. While basic indexing functions like `substr()` allow extraction based on fixed character positions, they become highly impractical when dealing with variable-length strings where the desired content is delimited by specific boundary characters. For dynamic extraction, we must rely on powerful pattern-matching techniques.

The most effective approach for achieving dynamic text isolation in R utilizes regular expressions (regex). Regular expressions provide a sophisticated language for describing patterns within text, enabling the identification and isolation of text segments that reside precisely between two known markers, regardless of the length of the intervening content. Mastering this technique is essential for advanced string manipulation.

This guide details two primary, robust methodologies available within the R ecosystem for string extraction between delimiters. We will explore leveraging the powerful built-in functions of Base R, specifically `gsub()`, and utilizing the highly streamlined functions provided by the popular Tidyverse component, the stringr package. Both techniques hinge upon the concept of capturing groups within regular expressions to isolate the target text efficiently.

Understanding the Role of Regular Expressions

Regular expressions are crucial because they allow us to define a search pattern that encompasses the entire string but specifically marks the segment we wish to keep. When extracting strings between two delimiters (e.g., 'char1' and 'char2'), the regex must perform three tasks: match everything before the first delimiter, capture the content between the delimiters, and match everything after the second delimiter. The captured content is then retrieved, discarding the rest.

Key components in this pattern include the dot (`.`), which matches any character (except newline by default); the asterisk (`*`) or plus (`+`), which denote repetition (zero or more, or one or more, respectively); and parentheses (`()`), which define the crucial capturing group. The content enclosed in parentheses is the substring that R will remember and allow us to reference later. Without a capturing group, the entire matched string would be returned or replaced, defeating the purpose of targeted extraction.

When working with Base R functions like `gsub()`, the ability to reference these captured groups using backreferences (like `1`, `2`, etc.) is what makes extraction possible. In essence, we are using the replacement function not to replace text entirely, but to replace the entire matched pattern with only the specific portion we captured within the parentheses. This precise control over the pattern matching process ensures accurate isolation of the desired substring.

Method 1: Utilizing Base R Functions for Extraction

Base R provides robust functionality for pattern matching and substitution without requiring external package dependencies. The primary tool for string extraction in this context is the `gsub()` function, which stands for Global Substitution. While typically used for replacing patterns, it can be creatively employed alongside regular expressions to perform targeted extraction.

To utilize `gsub()` for extraction, the search pattern must be constructed to include parentheses around the desired inner string. The replacement argument is then set to `1` (or `2`, depending on which capturing group holds the target text). This tells R: "Match the entire string pattern, but replace that entire match with only the content of the first capturing group."

The general structure for this method involves using a pattern that consumes the text on both sides of the target. For instance, the pattern `.*char1 (.+) char2.*` first matches zero or more characters (`.*`) up to the starting delimiter (`char1`), then captures one or more characters greedily (`(.+)`) until the ending delimiter (`char2`) is reached, followed by matching the rest of the string (`.*`). The use of `.+` ensures that there is content between the delimiters.

Practical Implementation of Base R Syntax

The following syntax demonstrates the Base R approach using `gsub()`. This command is designed to extract the substring found between the delimiters **char1** and **char2** within a given string variable, **my_string**. The replacement argument `1` is the key to successfully performing the extraction.

gsub(".*char1 (.+) char2.*", "1", my_string)

In this specific regex pattern, the `.*` ensures that we match any characters before `char1` and after `char2`. The inner portion, `(.+)`, is the capturing group that targets the actual content we wish to extract. Because this capturing group is the first (and only) set of parentheses in the pattern, its contents are referenced by the backreference `1` in the replacement string. This effectively isolates the desired string and substitutes it in place of the original full string.

Method 2: Leveraging the stringr Package

For users who prefer a cleaner, more intuitive syntax common in the Tidyverse ecosystem, the stringr package offers specialized functions designed specifically for pattern matching and extraction. Unlike `gsub()`, which performs substitution, `str_match()` is explicitly designed to identify and return captured groups from a string, often simplifying the extraction process and improving code readability.

To use `stringr`, the package must first be loaded into the R session. The function `str_match()` is then applied, requiring the input string and the pattern. The regex pattern used here is subtly different from the `gsub()` pattern because `str_match()` is intended to return the capture directly, rather than using substitution. It is often beneficial to use non-greedy matching (e.g., `*?`) in `stringr` to prevent the pattern from consuming content beyond the immediate next delimiter, which can happen with greedy matching (`.+`) if multiple pairs of delimiters exist.

The crucial difference in the pattern structure for `str_match()` often involves ensuring the delimiters are explicitly matched and using non-greedy quantification for the content in between, such as `(.*?)`. The pattern below incorporates `s*` to account for potential whitespace surrounding the target substring, making the extraction more robust against minor data inconsistencies.

library(stringr)

str_match(my_string, "char1s*(.*?)s*char2")

Both of these methods achieve the same objective: they successfully extract the string located between the characters **char1** and **char2** within the input variable **my_string**. The choice between Base R and `stringr` often comes down to personal preference, project dependencies, and required output format.

Setting Up the Demonstration Data

To provide a clear, practical demonstration of these two methods, we will apply them to a simple data frame (a core R structure for storing tabular data). This scenario reflects common real-world data cleaning tasks where a specific component needs to be extracted from a character column and placed into a new variable for subsequent analysis.

We create a data frame named `df` that contains team names, where the actual team identifier (the content we want to extract) is nested between the delimiters 'team' and 'pro'. Our goal is to isolate 'Mavs', 'Heat', and 'Nets' into a separate column.

#create data frame
df <- data.frame(team=c('team Mavs pro', 'team Heat pro', 'team Nets pro'),
points=c(114, 135, 119))

#view data frame
df

team points
1 team Mavs pro 114

2 team Heat pro 135
3 team Nets pro 119

The column `team` is the target variable for our extraction. We must ensure our regex pattern correctly identifies 'team' as the starting marker and 'pro' as the ending marker, capturing only the content that falls strictly between them for all rows in the data frame.

Case Study 1: Base R Extraction Using gsub()

This example demonstrates how to efficiently apply the Base R `gsub()` function across an entire column of a data frame to create a new variable containing only the extracted substring. By applying the function directly to `df$team`, the operation is vectorized, performing the substitution across every element in the column simultaneously.

The regular expression pattern `.*team (.+) pro.*` is tailored specifically to the data structure: `.*team` matches everything up to and including the word 'team', `(.+)` captures the required team name (e.g., ' Mavs '), and `pro.*` matches the trailing delimiter and the rest of the string. The replacement value `1` ensures that only the captured team name replaces the original full string content.

#create new column that extracts string between team and pro
df$team_name <- gsub(".*team (.+) pro.*", "1", df$team)

#view updated data frame
df

team points team_name
1 team Mavs pro 114 Mavs
2 team Heat pro 135 Heat
3 team Nets pro 119 Nets

The resulting data frame now includes a new column called `team_name`, which cleanly contains the isolated team names (Mavs, Heat, Nets). This successful extraction demonstrates the power and efficiency of using Base R functions coupled with strong regular expressions for data cleaning tasks.

Case Study 2: stringr Package Approach

For those utilizing the stringr library, the `str_match()` function provides an equally effective, and perhaps syntactically clearer, solution. This function is specifically optimized for extracting matched

groups, returning them in a matrix format which requires subsequent column selection. We must first ensure the stringr package is loaded.

The regex pattern `teams*(.*?)s*pro` used in this context uses non-greedy matching (`*?`) to ensure accurate extraction, especially in complex strings. The `s*` component is crucial here, as it matches any zero or more whitespace characters before and after the captured group, thereby automatically trimming any leading or trailing spaces from the extracted team names (e.g., removing the space before 'Mavs' and after 'Mavs' that was present in the original string).

library(stringr)

#create new column that extracts string between team and pro
df$team_name <- str_match(df$team, "teams*(.*?)s*pro")

#view updated data frame
df

team points team_name
1 team Mavs pro 114 Mavs
2 team Heat pro 135 Heat
3 team Nets pro 119 Nets

Once again, the new column `team_name` successfully populates with the desired substrings. The consistency in the final output across both methods highlights the flexibility R offers in handling string manipulation tasks, allowing the user to select the method best suited to their workflow or project dependencies.

Understanding str_match() Output Format

A significant distinction of the `str_match()` function, compared to substitution functions like `gsub()`, is its output structure. The `str_match()` function always returns a matrix, regardless of the number of input strings processed. This matrix is structured in a specific way to provide comprehensive results regarding the matches found.

The matrix produced by `str_match()` has the following column organization: the first column (index 1) contains the entire matched pattern (including the delimiters, 'team' and 'pro', in our example), and subsequent columns (index 2 onwards) contain the contents of the capture groups defined by the parentheses in the regular expressions.

Since we are only interested in the content *between* the delimiters--the text contained within the first (and only) capturing group--we must specifically extract the second column of the resulting

matrix. This is why the indexing is appended to the `str_match()` function call. Failure to include would result in the entire matrix being assigned, which would not align correctly with a standard vector column in a data frame, and would include the unnecessary full match column.