Remove Last Character from String in R (2 Examples)

Authored by

Mohammed looti

October 28, 2025

When performing data cleaning and preparation in **R**, the refinement of textual data is a routine necessity. One of the most common requirements in this process is precise **string manipulation**, which often involves adjusting the length or content of character sequences. Specifically, developers and analysts frequently encounter scenarios where they must eliminate the **last character** from every string residing within a **vector** or data column. This comprehensive guide details two distinct, yet equally powerful, methodologies available in R to achieve this specific truncation, allowing you to choose the approach best suited to your environment and stylistic preferences.

We will explore solutions using core **Base R** functions--leveraging their inherent availability without requiring external dependencies--and contrast this with the streamlined approach offered by the **`stringr` package**, a key component of the **Tidyverse** ecosystem. Each method provides a robust solution for trimming strings, and we will provide clear, executable examples to ensure you can seamlessly integrate these techniques into your larger data processing workflows, regardless of whether you prioritize performance or native functionality.

Method 1: Utilizing Base R's `substr()` Function for Truncation

The first strategy for accurately trimming the final character from a sequence utilizes the core capabilities of **Base R**. This means the required functions are intrinsically part of the R installation, eliminating the need to install or load any supplementary packages. This native approach centers around the powerful **`substr()`** function, which is designed to handle the extraction or substitution of substrings from character vectors based on defined start and end positions.

To successfully remove the **last character** using `substr()`, we must precisely define the range of characters we wish to keep. The function requires three critical inputs: the target **character vector**, the **starting position** for extraction (always `1` for the beginning of the string), and the **ending position**. The crucial step here is calculating this ending position, which must correspond exactly to the index of the second-to-last character.

This calculation is performed using another essential **Base R** tool: the **`nchar()`** function. The `nchar()` function dynamically returns the total number of characters in the specified string. By taking the result of `nchar()` and subtracting `1`, we determine the exact index of the second-to-last character. This combined operation instructs `substr()` to extract everything up to that calculated point, effectively excluding the final character and achieving the desired **string manipulation** without relying on external libraries.

substr(df$some_column, 1, nchar(df$some_column)-1)

Method 2: Leveraging the `stringr` Package with `str_sub()` and Negative Indexing

For users who prefer a more streamlined, modern, and often more readable syntax for **string manipulation**, the **`stringr` package** provides an excellent alternative. As a foundational element of the **Tidyverse**, `stringr` was specifically designed to standardize and simplify string operations in R, making complex tasks more intuitive and reducing boilerplate code compared to some native Base R solutions.

The primary function employed here is **`str_sub()`**, which serves as the powerful equivalent to `substr()` within the Tidyverse framework. The key advantage that makes `str_sub()` particularly efficient for this specific task--removing the **last character**--is its robust support for **negative indexing**. This feature allows positions to be referenced relative to the end of the string, significantly simplifying the required logic.

To drop the final character, we only need to specify the `end` argument as `-2`. In `str_sub()` syntax, `-1` refers to the last character, `-2` refers to the second-to-last character, and so on. By setting `end = -2`, we instruct the function to extract every character starting from position `1` (the default start) up to and including the second-to-last character. This elegant solution bypasses the explicit need to call `nchar()`, resulting in cleaner, more concise code. Remember that the **`stringr` package** must be explicitly loaded into your R session using `library(stringr)` before `str_sub()` can be used.

library(stringr)

str_sub(df$some_column, end = -2)

Setting Up and Executing the Base R Truncation Example

To provide a clear, practical demonstration of both string truncation methods, we must first establish a representative **data frame**. This structured example allows us to observe how these functions operate directly on data structures commonly encountered in R programming. Our sample data frame, named `df`, includes a `name` column containing character strings that we intend to shorten, and a secondary `sales` column (an integer vector) for context.

Below is the necessary code to initialize our sample data frame and display its original contents. Our objective is to apply the removal technique specifically to the `name` column, ensuring that the last character is systematically eliminated from each entry before updating the column in place.

#create data frame

```
df <- data.frame(name=c('Andy', 'Bert', 'Chad', 'Derrick', 'Eric', 'Fred'),
sales=c(18, 22, 19, 14, 14, 11))

#view data frame
df
```

```
name sales
1 Andy 18
2 Bert 22
3 Chad 19
4 Derrick 14
5 Eric 14
6 Fred 11
```

We now apply the **Base R** technique. By combining `substr()` and `nchar()`, we calculate the length of each string in `df$name`, subtract one, and extract the substring from position 1 up to that new calculated endpoint. This modified string **vector** is then directly assigned back to `df$name`, achieving the desired modification efficiently using only native R functionality.

```
#remove last character from each string in 'name' column
df$name = substr(df$name, 1, nchar(df$name)-1)

#view updated data frame
df
```

```
name sales
1 And 18
2 Ber 22
3 Cha 19
4 Derric 14
5 Eri 14
6 Fre 11
```

As clearly demonstrated by the output, the **last character** has been successfully and systematically removed from every entry in the `name` column using the robust combination of native **Base R** functions.

Practical Application: The `stringr` Package Example

Following our successful implementation with **Base R**, let us now replicate the exact same result

using the **`stringr` package**, showcasing the power and simplicity afforded by the **Tidyverse** philosophy. For this example, we assume we have reset our `df` **data frame** back to its original state or are working with a fresh copy of the data structure defined earlier.

The implementation is notably concise. After ensuring the `stringr` library is loaded, we utilize the **`str_sub()`** function. By leveraging its **negative indexing** capabilities and setting the `end` parameter to `-2`, we instruct R to extract the sequence starting from the beginning and stopping just before the final character. This single-function call encapsulates the logic that required two functions in the Base R approach.

Observe the code snippet below, which executes this operation and displays the final modified **data frame**. The critical takeaway is the identical result achieved compared to the previous method, confirming that both techniques are reliable for complex **string manipulation** tasks but offer different levels of coding clarity and dependency management.

library(stringr)

#remove last character from each string in 'name' column
df$name <- str_sub(df$name, end = -2)

#view updated data frame
df

name sales
1 And 18
2 Ber 22
3 Cha 19
4 Derric 14
5 Eri 14
6 Fre 11

The consistency in output confirms the effectiveness of `str_sub()`. This method often appeals to users who prioritize readable code and are already deeply integrated into the **Tidyverse** suite of tools for data science in **R**.

Performance and Readability: Choosing the Optimal Method

While both **`substr()`** (Base R) and **`str_sub()`** (from `stringr`) deliver identical, reliable results, the choice between them should be guided by considerations of dependency management, code readability, and performance scale. Understanding these trade-offs is crucial for developing efficient and maintainable R code.

For smaller scale projects or infrequent data cleaning operations, the performance delta between the two functions is typically negligible. In such cases, opting for the **Base R** solution is logical if you wish to minimize external dependencies, ensuring your script runs successfully in any standard R environment without requiring users to install the **`stringr` package**. The explicit calculation using `nchar()` also provides transparent logic regarding the index manipulation.

However, when transitioning to enterprise-level data processing involving millions of strings or high-frequency operations, performance efficiency becomes paramount. The `stringr` package benefits from highly optimized underlying C/C++ code, generally making **`str_sub()`** significantly faster than the combination of `substr()` and `nchar()` when applied across large **data frames**. Furthermore, many programmers find the syntax of `str_sub()`--especially its use of negative indexing--to be far more intuitive and expressive, which enhances code readability and speeds up debugging within a cohesive **Tidyverse** environment.

Summary of Methods

This guide has detailed two foundational methods for trimming the last character from strings in R: the native approach using **`substr()`** and **`nchar()`** from Base R, and the modern, concise alternative offered by **`str_sub()`** from the `stringr` package. Both are highly effective tools for **string manipulation**.

Base R Method: Utilizes `substr()` starting at 1 and ending at `nchar() - 1`. Pros: Zero external dependencies; native availability. Cons: Requires two functions for the length calculation, potentially less readable.
`stringr` Method: Utilizes `str_sub()` with the `end = -2` argument. Pros: Superior readability and conciseness due to negative indexing; often better performance on large datasets. Cons: Requires loading an external package.

By understanding the mechanics and comparative benefits of each technique, you are now equipped to select the most suitable method for your data cleaning requirements in R, optimizing your code for clarity, efficiency, or simplicity based on the specific project context.

Further Learning and Resources

To further expand your proficiency in R programming and master more complex **string manipulation** challenges, we recommend consulting the following authoritative resources. These links provide comprehensive documentation and detailed tutorials that will enhance your data preparation and transformation skills.

Official R Documentation for **Base R functions**, offering deep insights into native capabilities.
The **`stringr` package documentation**, providing a full reference guide to Tidyverse string

functions.

Comprehensive guides on **the Tidyverse** ecosystem for streamlining modern data science workflows.