How to Easily Remove the Last Character from a String in R

Authored by

stats writer

November 27, 2025

Introduction: Mastering String Truncation in R

Manipulating text data is a fundamental requirement in data analysis, and one common task involves cleanly truncating or modifying strings. When working with the R programming language, removing the final character from a string--perhaps due to a trailing delimiter, newline character, or unwanted suffix--is a frequent necessity. Fortunately, R provides robust tools for this purpose, primarily through the base function **substr()** (or the highly similar **substring()** function) and the highly efficient **stringr** package function, **str_sub()**.

The mechanism for truncation relies on precise indexing. Using **substr()**, you must first determine the exact length of the string using the **nchar()** function, subtract one from that value to account for the character being removed, and then use that calculated length to define the end position of the desired substring. Conversely, using **str_sub()** simplifies this process through powerful negative indexing.

This guide provides a comprehensive overview of how to achieve precise string truncation, focusing specifically on removing only the last character. We will explore both methodologies: the classical Base R approach, which relies on calculation, and the modern Tidyverse approach using **stringr**, which offers a cleaner, more intuitive syntax for indexing strings from the end. Understanding both methods allows analysts to select the most appropriate tool based on their existing workflow and dataset size.

Key Methods for String Manipulation in R

To effectively remove the final character from a vector of strings in R, we employ functions that allow for the extraction of a substring based on defined start and end points. The core challenge lies in dynamically defining the end point so that it always excludes the last position, regardless of the individual string's length. This is particularly important when applying the function across a column in a **data frame** where string lengths are heterogeneous.

The two methods detailed below are the most common and robust ways to perform this operation. They offer equivalent results but differ significantly in their approach to string indexing and their reliance on external packages.

Method 1: Remove Last Character Using Base R

The Base R approach requires using the **substr()** function and dynamically calculating the necessary length using **nchar()**. We start at position 1 and end at the total length minus one.

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

In this syntax, `df$some_column` refers to the target vector of strings. The `1` specifies the starting position. The expression `nchar(df$some_column)-1` calculates the total number of characters in each string and then deducts one, ensuring the last character is omitted from the extracted result.

Method 2: Remove Last Character Using stringr Package

The Tidyverse approach utilizes the **stringr** package, which provides a more idiomatic way to handle string operations, especially through its support for negative indexing via **str_sub()**.

library(stringr)

str_sub(df$some_column, end = -2)

Here, the `end = -2` argument is highly intuitive: it instructs **str_sub()** to stop extraction at the second-to-last character. Since string indexing from the end uses -1 for the last character, stopping at -2 successfully excludes the final position. This eliminates the need for the explicit **nchar()** calculation.

Utilizing Base R Functions: `substr()` and `nchar()`

The Base R method for string truncation is essential knowledge for any R user, as it relies on functions that are guaranteed to be available without installing external packages. The core concept is mathematical: defining the precise range of characters to keep.

The **substr(x, start, stop)** function extracts substrings from character vectors. To remove the final character, the key is defining the `stop` argument correctly. Since strings are indexed starting at 1, we always set `start = 1` to capture the beginning of the string. The critical calculation is handled by **nchar()**. This function returns the number of characters in the string, which, when reduced by one, gives the index of the second-to-last character--precisely where we want our substring extraction to terminate.

This technique is robust and reliable, though it can become slightly verbose when embedded within larger data manipulation pipelines. While **substring()** is functionally identical to **substr()** in this context, **substr()** is generally the more conventional choice for defining start and end positions explicitly.

The Efficiency of the `stringr` Package

The **stringr** package, part of the Tidyverse, was developed to provide a consistent, cohesive, and efficient set of functions for working with strings. Its function **str_sub()** offers a compelling alternative to Base R by introducing clear support for negative indexing, a feature often missing or

poorly implemented in Base R string functions.

The use of `end = -2` transforms the task from a multi-step calculation (find length, subtract 1) into a simple, declarative instruction: "keep everything up to the second-to-last position." This significantly enhances code readability and reduces the likelihood of complex logic errors, particularly when performing multiple string manipulations sequentially.

Moreover, **stringr** functions are highly vectorized and often optimized for performance, making **str_sub()** the preferred option when dealing with millions of character strings, such as in large-scale text analysis or database processing tasks. By integrating **stringr**, analysts benefit from a concise syntax that is both powerful and computationally efficient.

Preparing the Sample Data Frame

To illustrate both methodologies, we need a common dataset to manipulate. We will create a small sample **data frame** named `df`. This structure simulates a common scenario in data analysis where a column contains text data that needs standardization or cleaning.

In this specific demonstration, we focus on the `name` column, where the goal is to remove the last letter of each name. This operation is analogous to removing unwanted trailing characters, such as spaces, periods, or delimiters, that frequently plague imported datasets.

The following example initializes the data frame in R:

#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

Before proceeding with the manipulation, it is helpful to note the current state of the `name` column. Our subsequent steps will show how this column is transformed in place using both the Base R

and the **stringr** methods.

Practical Application 1: Base R Implementation

This first practical example demonstrates how to apply the Base R solution using the combination of **substr()** and **nchar()** to modify the `name` column of our sample data frame. This technique underscores R's core functionality for vectorized string manipulation.

We apply **nchar(df$name)** to get the length of every name simultaneously, then subtract 1 from each length. This modified length vector is passed to the `stop` argument of **substr()**. The result, which is a new character vector containing the truncated names, is then assigned back to `df$name`, updating the data frame permanently.

The following code shows how to remove the last character from each string in the **name** column of the data frame:

#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

Notice that the last character from each string in the **name** column has been successfully removed. 'Andy' is now 'And', 'Bert' is 'Ber', and so on. This verifies the efficacy of using dynamic length calculation (`nchar() - 1`) to define the end boundary for extraction.

Practical Application 2: Using the `stringr` Package

The second example demonstrates the equivalent operation using the specialized **stringr** function, **str_sub()**. This method is preferred for its declarative nature and the clarity provided by negative indexing.

Before executing the truncation, the **stringr** package must be loaded using the `library()` command. Once loaded, the **str_sub()** function performs the operation seamlessly. By setting `end = -2`, we bypass the need for explicit length calculations. The function handles all internal logic, making the code cleaner and easier to maintain.

The following code shows how to remove the last character from each string in the **name** column of the data frame by using the function from the **stringr** package:

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

Notice that the last character from each string in the **name** column has been removed, producing results identical to those obtained using the Base R method. This confirms that both approaches are logically sound and yield the same desired output for data cleaning tasks.

Performance Considerations and Best Practices

When choosing between **substr()** and **str_sub()**, several factors beyond syntax should be considered, most importantly, computational performance and existing package dependencies.

The Base R method using **substr()** is advantageous because it requires absolutely no external dependencies. If you are writing a script that must run in an environment where installing additional packages is restricted, or if you need maximum portability, the Base R solution is the logical choice. However, for everyday use, especially within modern data workflows, **str_sub()** from the **stringr** package is often superior.

If you're working with an extremely large **data frame** (tens of thousands or millions of rows), **str_sub()** is generally implemented with underlying C code optimizations that make it faster and

more memory-efficient than the pure Base R implementation of **substr()** when performing large-scale vectorized operations. For smaller datasets, the performance difference is negligible, and the choice should prioritize readability.

Ultimately, for those already operating within the Tidyverse ecosystem, adopting **str_sub()** is the best practice due to its streamlined syntax (negative indexing) and robust performance characteristics, making complex string manipulations significantly simpler and more maintainable.

Summary

Successfully removing the final character from a string in **R** can be accomplished through two primary, reliable methods. The choice hinges on whether you prefer relying solely on Base R functionality or integrating the streamlined syntax of the **stringr** package.

The techniques demonstrated here provide deterministic, vectorized solutions for common data cleaning requirements in R, ensuring your character data is accurately prepared for subsequent analytical steps.

Base R Method: Requires combining **substr()** and **nchar()** to dynamically calculate the end position (`nchar() - 1`). This method is dependency-free but requires more complex syntax.

stringr Method: Utilizes **str_sub()** with negative indexing (`end = -2`). This provides cleaner syntax and often better performance for high-volume string operations.