# Learning Substring Extraction with the R substring() Function: A Tutorial with Examples

Authored by

**Mohammed looti**

October 30, 2025

In modern data science and programming, particularly within the environment of **R**, handling textual data efficiently is paramount. Raw text often requires cleaning, parsing, or standardization before analysis can begin. One of the most fundamental operations in this process is **substring** extraction--the ability to isolate specific segments of text from a longer string. The robust base **R** function designed for this precise task is `substring()`. This function allows developers and analysts to easily retrieve characters from a **character vector** by defining their exact positional indices.

Mastering the use of `substring()` is critical for numerous data preprocessing steps. For instance, you might need to extract identifiers embedded within coded strings, separate dates or timestamps, or simply standardize abbreviations by isolating the first few characters. When dealing with structured text data where elements are consistently located at specific positions, `substring()` becomes an invaluable tool for reliable and fast extraction.

The core purpose of the `substring()` function is straightforward: to extract a continuous sequence of characters from a given source string. This extraction is defined entirely by numerical inputs specifying the starting point and the ending point of the segment you wish to capture. This flexibility ensures that whether you need the beginning, the middle, or the end of a string, `substring()` can precisely isolate the required characters based on their position relative to the start of the string.

## Understanding the `substring()` Function Syntax

The structure, or **syntax**, of the `substring()` function is highly intuitive, adhering to simple R programming conventions. Its design clearly outlines the essential inputs required to perform a positional substring extraction, making it accessible even for those new to text manipulation in R.

The general **syntax** for utilizing the `substring()` function requires three main components, defined as follows:

**substring(text, first, last)**

Let us examine each critical **argument** required by the function to understand its role in the extraction process:

`text`**:** This first **argument** specifies the source data--the **character vector** from which characters will be extracted. This can be a single string, a dedicated column containing strings within a **data frame**, or any collection of text elements.
`first`**:** This numerical **argument** dictates the starting position of the desired substring. It is crucial to remember that R, like many programming languages, uses 1-based indexing for strings, meaning the very first character is located at position 1.

**`last`:** This numerical **argument** defines the ending position of the substring. The character residing at this specific index will be included in the final extracted segment. If the specified `last` position happens to exceed the actual length of the source string, **`substring()`** defaults to extracting all characters up to the string's end without raising an error.

## `substring()` vs. `substr()`: A Brief Comparison

When exploring string manipulation in R, users often encounter two functions that appear to perform the same task: **`substring()`** and **`substr()`**. It is important to clarify that these two functions are, in fact, aliases of one another in base **R**, meaning they execute the exact same underlying operation. This duality provides flexibility in code writing and can sometimes aid compatibility, though their functionality is perfectly identical.

While the core operation remains the same, the functions differ slightly in the naming convention of their **arguments**. The **syntax** for **`substr()`** uses slightly different positional labels:

**substr(x, start, stop)**

To maintain clarity, here is a quick mapping of the corresponding **arguments** between the two functions:

**`x`:** Equivalent to the `text` **argument** in **`substring()`**, representing the source **character vector**.
**`start`:** Equivalent to the `first` **argument** in **`substring()`**, indicating the starting index.
**`stop`:** Equivalent to the `last` **argument** in **`substring()`**, indicating the ending index.

Functionally, you can confidently use either **`substring()`** or **`substr()`** to achieve identical results in R. The preference between them usually comes down to the developer's personal coding style or the adoption of specific team conventions. For consistency throughout this tutorial, we will primarily utilize the **`substring()`** function in our practical examples.

## Setting Up Our Example Data in R

To provide a clear, practical demonstration of how **`substring()`** operates in a real-world context, we will first establish a simple **data frame** within R. This structure will hold textual data that is relatable--a list of sports team names--making the string manipulation operations easy to follow and visualize.

The following code snippet creates and displays this sample **data frame**, which will serve as the foundation for all subsequent extraction examples:

**# Create the example data frame**

**df <- data.frame(team=c('Mavericks', 'Hornets', 'Rockets', 'Grizzlies'))**

# View the resulting data frame structure
df

team
1 Mavericks
2 Hornets
3 Rockets
4 Grizzlies

The resulting object, named `df`, contains one column called `team`, populated with four distinct character strings. We will perform various **string manipulation** techniques on this `team` column, showcasing how **`substring()`** interacts dynamically with data structures typically encountered in R projects.

## Example 1: Extracting Characters Between Specific Positions

The most fundamental application of the **`substring()`** function involves defining a precise positional range (start and end indices) from which to extract characters. This method is essential when dealing with fixed-format strings, such as product codes or standardized identifiers, where the target information is consistently located in the middle of the string.

In this scenario, we aim to extract the segment of characters located between the 2nd and 5th positions, inclusive, for every team name in our `team` column. We achieve this by explicitly setting the `first` and `last` numerical **arguments**:

**# Create new column containing characters from position 2 to 5**
**df$between2_5 <- substring(df$team, first=2, last=5)**

# View the updated data frame
df

team between2_5
1 Mavericks aver
2 Hornets orne
3 Rockets ocke
4 Grizzlies rizz

The code above successfully instructed **`substring()`** to start at the second character (`first=2`)

and terminate after including the fifth character (`last=5`). The resulting column, `between2_5`, holds the isolated substrings. For "Mavericks," the segment "aver" is extracted, and for "Rockets," "ocke" is retrieved. This illustrates the power of **`substring()`** in accurately targeting and isolating specific internal segments of text based solely on positional logic.

## Example 2: Extracting the First N Characters (Prefixes)

A very common requirement in **string manipulation** tasks is the extraction of a fixed prefix--a set number of characters from the beginning of a string. This technique is frequently used to generate standardized abbreviations, short display names, or unique prefixes for indexing purposes.

To reliably extract the first N characters, the implementation is simple: we set the `first` **argument** to 1 (the start of the string) and set the `last` **argument** to the desired number N. Here, we demonstrate extracting the first three characters from all team names:

**# Create new column containing the first 3 characters**
**df$first3 <- substring(df$team, first=1, last=3)**

# View the updated data frame
df

team first3
1 Mavericks Mav
2 Hornets Hor
3 Rockets Roc
4 Grizzlies Gri

The output confirms that the new column `first3` now holds the three-character abbreviations for each team. This approach is highly efficient for data summarization, especially when creating consistent short forms is necessary across a large dataset of text entries.

## Example 3: Extracting the Last N Characters (Suffixes)

Extracting a fixed number of characters from the end of a string (the suffix) requires a slightly more advanced approach in **R** because the starting position is relative to the string's end, not its beginning. To calculate the correct starting index, we must first determine the total length of the string using the dedicated **`nchar()`** function.

To extract the last N characters, the `last` **argument** of **`substring()`** will be the total string length (given by **`nchar()`**). The `first` **argument** is calculated as: Total Length minus N plus 1. Let's extract the final three characters (N=3) from each team name:

```
# Create new column containing the last 3 characters
df$last3 <- substring(df$team, nchar(df$team)-3+1, nchar(df$team))

# View the updated data frame
df
```

```
team last3
1 Mavericks cks
2 Hornets ets
3 Rockets ets
4 Grizzlies ies
```

In this powerful example, `nchar(df$team)` dynamically calculates the length for each row. For the 9-character string "Mavericks," the calculation is 9 - 3 + 1 = 7, correctly starting the extraction at the 7th position and ending at the 9th, yielding "cks." This technique is essential when suffixes, regardless of the overall string length, need to be consistently isolated.

## Example 4: Replacing a Substring

The utility of the `substring()` function extends far beyond mere extraction; it can also be used as an assignment operator to perform powerful in-place replacements of **substrings**. This feature is particularly valuable as it allows you to modify targeted sections of a string without the complexity of rebuilding the entire text.

To execute a replacement, you assign the new character sequence directly to the result of the `substring()` call, specifying the range to be overwritten. A key consideration here is that R attempts to match the replacement string length to the length of the substring being replaced. If the lengths do not match, R will either truncate the replacement or pad it with spaces to fit the defined positional indices. Let's demonstrate this by replacing the first three characters of every team name with three asterisks ("***"):

```
# Replace the first 3 characters (positions 1 through 3) with asterisks
substring(df$team, first=1, last=3) <- "***"

# View the updated data frame
df
```

```
team
1 ***ericks
2 ***nets
3 ***kets
```

4 ***zzlies

The resulting output verifies that the initial three characters of each string have been overwritten successfully. This assignment capability is exceptionally useful for tasks such as data anonymization, masking confidential information, or precise **string manipulation** where alterations must be confined to specific positions within the text.

## Conclusion and Additional Resources

The `substring()` function stands as an essential and versatile component of the R base package for all text processing needs. It offers a clear, positional method for handling critical **string manipulation** tasks, whether your goal is to extract defined segments, isolate prefixes or suffixes, or execute in-place replacements within your data. Its consistent behavior and straightforward **syntax** make it a preferred tool for rapid data cleaning and text standardization in analytical workflows.

By effectively combining `substring()` with complementary functions, such as `nchar()` for dynamic length assessment, you gain precise control over the character data stored within your **data frame** objects. While `substring()` focuses on positional indexing, it is worth noting that R provides a rich ecosystem of tools--including base functions utilizing regular expressions and specialized packages like `stringr`--to handle more complex pattern-based string operations.

We encourage users seeking to further enhance their text processing skills in **R** to explore these related functions and dedicated packages. Continued practice with `substring()` and other string utilities will significantly boost your data preparation capabilities.

## Additional Resources

The following tutorials explain how to perform other common operations with strings in R:

How to Perform Partial String Matching in R