Learning SAS: How to Extract Substrings Using the SUBSTR Function

Authored by

Mohammed looti

October 31, 2025

The ability to manipulate textual data, or **strings**, is a fundamental requirement in data processing and analysis. In **SAS** programming, one of the most essential functions for this purpose is the **SUBSTR function**. This powerful tool allows users to precisely extract a specific portion of a character string, facilitating tasks like data cleaning, parsing identifiers, and creating new categorization variables.

Understanding how to correctly implement **SUBSTR** is crucial for efficient data management within the **SAS Data Step**. Whether you need the first few characters of a product code or the middle segment of a long text field, the function provides a straightforward mechanism to achieve your goal.

Understanding the SUBSTR Syntax

The **SUBSTR** function adheres to a simple, three-part syntax structure, making it highly versatile yet easy to remember. When calling the function within a SAS program, you must specify the source string, the desired starting point, and the length of the substring to be extracted.

This function uses the following basic syntax structure:

SUBSTR(Source, Position, N)

Each component of the syntax plays a critical role in defining the desired output:

Source: This is the mandatory **string** variable or literal text from which you wish to extract characters.
Position: This integer specifies the starting position where the function begins reading the source string. The first character is always position 1.
N: This optional integer denotes the number of characters to read from the starting position. If this parameter is omitted, the function reads all characters from the specified position to the end of the source string.

By mastering these three parameters, SAS programmers can perform sophisticated character extractions that are necessary for complex data transformation tasks. The following sections demonstrate four common scenarios where **SUBSTR** proves indispensable.

Four Common Methodological Applications

While the syntax of **SUBSTR** remains constant, its application can be adapted to solve various common data manipulation challenges. The four methods outlined below represent standard ways to leverage this function for tasks ranging from simple prefix extraction to conditional variable creation.

These techniques often involve combining **SUBSTR** with other functions, such as **LENGTH**, or integrating it into conditional logic statements (**IF-THEN/ELSE**) to achieve precise results within the **SAS Data Step** environment.

Method 1: Extracting the Prefix (First N Characters)

This is perhaps the simplest use case, designed to capture the beginning segment of a **string**. By setting the starting position to 1, we guarantee the extraction begins at the start of the variable.

```
data new_data;
set original_data;
first_four = substr(string_variable, 1, 4);
run;
```

Method 2: Extracting Characters from a Specific Position Range

When dealing with standardized identifier formats, you often need to isolate a section that is neither the beginning nor the end of the string. This method requires careful counting to determine both the starting position and the exact number of characters to extract.

```
data new_data;
set original_data;
two_through_five = substr(string_variable, 2, 4);
run;
```

Method 3: Extracting the Suffix (Last N Characters)

Extracting the end of a string is slightly more complex, as it typically requires knowing the total length of the string first. This is achieved by nesting the **LENGTH** function within the **SUBSTR** call to dynamically calculate the correct starting position regardless of the string's overall size.

```
data new_data;
set original_data;
last_three = substr(string_variable, length(string_variable)-2, 3);
run;
```

Method 4: Conditional Variable Creation Based on Substring Content

Beyond simple extraction, **SUBSTR** is often used within conditional statements to check if a specific pattern exists at a known location in a string. This allows for the creation of flags or categorical variables based on the characteristics of the source data.

```
data new_data;
set original_data;
if substr(string_variable, 1, 4) = 'some_string' then new_var = 'Yes';
else new_var = 'No';
run;
```

Practical Demonstration: Setting Up the Sample Data

To illustrate these four powerful techniques, we will utilize a small, representative **dataset** containing names of various basketball teams. This sample data will serve as the source for all subsequent examples, allowing us to clearly see the effect of the **SUBSTR** function on a single variable, which is named 'team'.

The following **SAS Data Step** code is used to create and populate the initial **dataset** called **original_data**. We use the **DATALINES** statement to embed the team names directly into the program, ensuring reproducibility.

```
/*create dataset*/
data original_data;
input team $1-10;
datalines;
Warriors
Wizards
Rockets
Celtics
Thunder
;
run;

/*view dataset using PROC PRINT*/
proc print data=original_data;
```

After running this initial code, the **original_data** table is generated and displayed. It contains a single character variable, 'team', which is correctly formatted and ready for string extraction operations.

Obs	team
1	Warriors
2	Wizards
3	Rockets
4	Celtics
5	Thunder

Example 1: Isolating the First Four Characters

In this first example, we apply Method 1 to extract the initial four characters from the **team** variable. This is often useful for creating standardized team abbreviations or classifying data based on known prefixes. The starting position is fixed at 1, and the length parameter is set to 4.

We create a new variable named **first_four** within the new **dataset**, which clearly demonstrates the result of the substring operation alongside the original source data. This allows for immediate verification of the function's execution.

```
/*create new dataset*/
data new_data;
set original_data;
first_four = substr(team, 1, 4);
run;

/*view new dataset*/
proc print data=new_data;
```

Upon reviewing the output, it is evident that the **first_four** variable successfully contains the first four characters of the **team** variable, such as 'Warr' from 'Warriors' and 'Wiza' from 'Wizards'.

Obs	team	first_four
1	Warriors	Warr
2	Wizards	Wiza
3	Rockets	Rock
4	Celtics	Celt
5	Thunder	Thun

Example 2: Extracting a Mid-String Segment

This example demonstrates Method 2, where we focus on extracting a middle segment of the string. We aim to capture the characters spanning from position 2 through position 5. To achieve this, we set the starting position parameter to 2 and specify that we want to read 4 characters (positions 2, 3, 4, and 5).

The variable **two_through_five** is generated to hold this mid-string segment. This technique is often critical when dealing with fixed-width data files where meaningful codes are embedded in specific, non-starting positions.

```
/*create new dataset*/
data new_data;
set original_data;
two_through_five = substr(team, 2, 4);
run;

/*view new dataset*/
proc print data=new_data;
```

Observing the resulting table confirms that the extraction was successful. For instance, 'Warriors' yields 'arri', starting at the second letter 'a' and continuing for four characters.

Obs	team	two_through_five
1	Warriors	arri
2	Wizards	izar
3	Rockets	ocke
4	Celtics	elti
5	Thunder	hund

Example 3: Extracting the Final Three Characters

Implementing Method 3 requires dynamic calculation to find the correct starting point for the extraction. If we want the last 3 characters, the starting position must be calculated as: (Total Length - 3) + 1. Since **SUBSTR** uses the **LENGTH** function to determine the string's end, the calculation simplifies to `length(team) - 2` to start at the third-to-last character.

The following code creates the **last_three** variable, demonstrating how to handle variable-length strings efficiently. This is a common requirement when processing files where identifiers might vary in overall length but maintain a standard suffix structure.

```
/*create new dataset*/
data new_data;
set original_data;
last_three = substr(team, length(team)-2, 3);
run;

/*view new dataset*/
proc print data=new_data;
```

As shown in the output, the new variable successfully captures the final three characters of each team name. For example, 'Celtics' correctly yields 'ics', and 'Thunder' yields 'der'.

Obs	team	last_three
1	Warriors	ors
2	Wizards	rds
3	Rockets	ets
4	Celtics	ics
5	Thunder	der

Example 4: Using SUBSTR for Conditional Logic

The final application, Method 4, demonstrates how **SUBSTR** can be integrated into control flow statements. Here, we use the function to check if the first character of the team name is 'W'. If the substring extracted from position 1 for a length of 1 equals 'W', we assign the value 'Yes' to the new variable **W_Team**; otherwise, we assign 'No'.

This powerful use of **SUBSTR** allows programmers to create logical flags based on specific data patterns, which is essential for filtering, grouping, and statistical modeling. This code efficiently categorizes all records in the **dataset** based on a defined character criterion.

```
/*create new dataset*/
data new_data;
set original_data;
if substr(team, 1, 1) = 'W' then W_Team = 'Yes';
else W_Team = 'No';
run;

/*view new dataset*/
proc print data=new_data;
```

The final output confirms that 'Warriors' and 'Wizards' are correctly flagged as 'Yes', while the remaining teams are flagged as 'No', validating the conditional logic implemented using the **SUBSTR** function.

Obs	team	W_Team
1	Warriors	Yes
2	Wizards	Yes
3	Rockets	No
4	Celtics	No
5	Thunder	No

Additional Resources for SAS Programming

Mastering string manipulation is just one aspect of effective **SAS** programming. To further enhance your data processing capabilities, consider exploring tutorials on related functions and procedures.

The following tutorials explain how to perform other common tasks in SAS: