# How to Easily Split Strings by Delimiter in SAS

Authored by

**stats writer**

December 1, 2025



RECOMMENDED CITATION

stats writer (2025). *How to Easily Split Strings by Delimiter in SAS*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=103189

Mastering SAS requires proficiency in manipulating data, and few tasks are more fundamental than parsing complex data fields into distinct, usable components. When dealing with concatenated information stored within a single column--such as names, addresses, or codes separated by a specific character--the ability to reliably split a character string based on a defined separator is essential for effective data analysis and reporting. The primary and most efficient tool for this operation within the SAS environment is the **SCAN function**, a powerful utility designed specifically for tokenization.

The **SCAN function** simplifies the process of dissecting a string. It works by inspecting the input string and identifying specified tokens--or pieces of the string--that are separated by a designated delimiter. Unlike more cumbersome approaches involving combinations of **SUBSTR** and **INDEX** functions, **SCAN** streamlines the extraction process, returning the desired token based on its sequential position. This capability is critical for data cleaning, feature engineering, and preparing raw input for statistical modeling, ensuring that each data element resides in its own distinct variable.

This comprehensive guide will detail the structure and application of the **SCAN function**, providing practical examples within the Data Step environment. We will explore how to set up the data, execute the splitting logic, and handle common scenarios such as variable lengths and missing tokens, demonstrating why **SCAN** is the go-to choice for delimiting strings in SAS programming.

You can use the **scan()** function in SAS to quickly split a string based on a particular delimiter. The core benefit of this function is its resilience to varying token lengths, allowing for robust parsing with minimal coding overhead.

The following example shows how to use this function in practice, demonstrating the creation of a sample dataset and the subsequent splitting operation.

## Example: Split Strings by Delimiter in SAS

The efficiency of the **SCAN** function lies in its straightforward syntax, which requires three primary arguments to perform the desired split operation successfully. Understanding these components is key to accurately extracting the required segments from any character string. The fundamental structure is `SCAN(argument, n, delimiters)`, where each parameter serves a specific, vital role in the tokenization process.

Suppose we have the following dataset in SAS, where the names are joined by an underscore ('_'). This structure is common when importing data from external sources that use a single field to store multiple, related values. We define this initial structure using a Data Step with inline data specified by `DATALINES`.

The resulting dataset, `my_data1`, contains the raw data that requires tokenization. We use PROC PRINT to verify the structure of the source data before proceeding with the splitting operation.

```
/*create dataset*/
data my_data1;
input name $25.;
datalines;
Andy_Lincoln_Bernard
Barry_Michael
Chad_Simpson_Smith
Derrick_Parson_Henry
Eric_Miller
Frank_Giovanni_Goodwill
;
run;
```

```
/*print dataset*/
proc print data=my_data1;
```

| Obs | name |
|---|---|
| 1 | Andy_Lincoln_Bernard |
| 2 | Barry_Michael |
| 3 | Chad_Simpson_Smith |
| 4 | Derrick_Parson_Henry |
| 5 | Eric_Miller |
| 6 | Frank_Giovanni_Goodwill |



## Implementing the SCAN Function for Tokenization

We can use the following code to quickly split the name string into three separate strings. This operation is performed within a new Data Step (`my_data2`) that reads from the source (`my_data1`). The key is to call the **SCAN** function three times, incrementally changing the index number to extract the first, second, and third tokens.

The syntax `name1=scan(name, 1, '_');` explicitly tells SAS to look at the `name` variable, extract the token at position 1, using the underscore character ('_') as the separator. This is repeated for `name2` (index 2) and `name3` (index 3). This iterative use of **SCAN** is the standard method for parallel extraction when the number of desired tokens is known.

This implementation generates three new variables--`name1`, `name2`, and `name3`--which hold the parsed components. This method is highly flexible and automatically handles the varying lengths of the names in the input data, simplifying the overall parsing process significantly compared to traditional string manipulation techniques.

```
/*create second dataset with name split into three columns*/
data my_data2;
set my_data1;
name1=scan(name, 1, '_');
name2=scan(name, 2, '_');
name3=scan(name, 3, '_');
run;
```

```
/*view second dataset*/
proc print data=my_data2;
```

| Obs | name | name1 | name2 | name3 |
|---|---|---|---|---|
| 1 | Andy_Lincoln_Bernard | Andy | Lincoln | Bernard |
| 2 | Barry_Michael | Barry | Michael | |
| 3 | Chad_Simpson_Smith | Chad | Simpson | Smith |
| 4 | Derrick_Parson_Henry | Derrick | Parson | Henry |
| 5 | Eric_Miller | Eric | Miller | |
| 6 | Frank_Giovanni_Goodwill | Frank | Giovanni | Goodwill |

## Interpreting Results and Handling Missing Tokens

Notice that the string in the **name** column has been successfully split into three new columns. Observing the output of `my_data2` is crucial for understanding how the **SCAN** function manages records with inconsistent formatting, a common challenge in real-world data processing.

For the names where there was only one delimiter (e.g., 'Barry_Michael' and 'Eric_Miller'), which logically contain only two tokens, the value in the **name3** column is simply blank. This behavior is standard and desirable in SAS string functions: when a requested token index exceeds the actual number of tokens separated by the specified delimiter, SAS returns an empty string, preventing errors and maintaining consistent variable definition.

This automatic handling of missing tokens is a significant advantage of using **SCAN**. When designing data quality checks, analysts must account for these potential blank values, perhaps by using conditional logic or the **COALESCE** function if they need to substitute missing tokens with default values.

## Refining the Output: Using the DROP Statement

Note that we could also use the **DROP statement** to drop the original `name` column from the new dataset. While retaining the original composite variable can be useful for initial validation, removing redundant columns is essential for optimizing dataset size and simplifying subsequent analysis and reporting procedures.

The inclusion of the **DROP statement** within the Data Step ensures that the variable `name` is processed for the **SCAN** function's calculations but is never written to the final output file `my_data2`. This keeps the dataset clean and focused only on the parsed components.

This refined approach is highly recommended for production environments. It results in a final dataset that is perfectly tailored for downstream tasks, featuring only the essential `name1`, `name2`, and `name3` variables, as demonstrated by the output of the revised code below.

```
/*create second dataset with name split into three columns, drop original name*/
data my_data2;
set my_data1;
name1=scan(name, 1, '_');
name2=scan(name, 2, '_');
name3=scan(name, 3, '_');
drop name;
run;
```

```
/*view second dataset*/
proc print data=my_data2;
```

| Obs | name1 | name2 | name3 |
|---|---|---|---|
| 1 | Andy | Lincoln | Bernard |
| 2 | Barry | Michael | |
| 3 | Chad | Simpson | Smith |
| 4 | Derrick | Parson | Henry |
| 5 | Eric | Miller | |
| 6 | Frank | Giovanni | Goodwill |

## Advanced Usage: Utilizing SCAN Modifiers

The robustness of the **SCAN** function in SAS can be significantly augmented by specifying optional modifiers in the fourth argument. These modifiers dictate specific parsing rules, making the function highly versatile for handling complex or messy data inputs. Common modifiers include 'T' for trimming whitespace, 'D' for treating delimiters strictly, and 'S' for managing sequential delimiters.

For instance, using the 'T' modifier, such as in `SCAN(name, 1, '_', 'T')`, instructs SAS to automatically remove any leading or trailing blanks from the extracted token. This is invaluable when the source data might contain unintentional padding around the delimiter. If the data contains default delimiters (like spaces) mixed with the custom delimiter (like '_'), adding the 'D' modifier ensures that only the underscore is recognized as a separator, preventing unexpected splitting based on inherent whitespace.

Combining multiple modifiers, such as 'DT', allows for both strict delimiter recognition and automatic trimming, leading to cleaner and more predictable output variables, regardless of the quality of the raw input. SAS programmers rely on these modifiers to create robust and portable code that can handle a wide variety of data inconsistencies without needing extensive pre-cleaning steps.

## Conclusion and Best Practices

The **SCAN function** is the definitive tool in SAS for splitting a single character string into multiple variables based on a specified delimiter. Its simplicity of use, combined with its ability to reliably handle missing tokens and variable-length inputs, makes it superior to manual substring manipulation techniques.

When implementing string splitting in SAS, best practices suggest performing the operation within a new Data Step to maintain the integrity of the original dataset. Additionally, utilizing the **DROP statement** is recommended to exclude the original concatenated variable, minimizing memory usage and streamlining the resulting dataset for analytical work.

By effectively employing the **SCAN** function, SAS users can efficiently transform unstructured, delimited text fields into structured, analytical variables, preparing data for powerful procedures such as PROC PRINT or statistical modeling routines.

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