Learning the SAS FIND Function: Locating Substrings Within Strings

Authored by

Mohammed looti

October 31, 2025

The ability to efficiently manipulate and analyze textual data is fundamental in modern data processing. In SAS (Statistical Analysis System), the powerful
FIND function is one of the most essential tools available for this purpose. This function allows users to search within a larger text string for the position of the first occurrence of a specified substring. Understanding how to properly implement
the **FIND** function is crucial for tasks ranging from data cleaning and validation to complex text mining operations, as it returns a numerical value indicating the starting index of the found pattern, or zero if the pattern is not present.

While string searching might seem straightforward, the **FIND** function offers important modifiers that dictate how the search is performed, particularly concerning case sensitivity and handling of whitespace. Mastering these options ensures that your search operations are precise and yield the intended results, regardless of variations in capitalization or formatting within your datasets. We will delve into the primary syntax methods used to leverage this function effectively within the DATA step environment.

The function's output is critical: it returns an integer representing the byte position (for single-byte character sets) or character position (for multi-byte character sets) where the substring begins. If the search is unsuccessful, the function simply returns 0. This integer output can then be used in subsequent conditional logic or further string manipulation functions, such as **SUBSTR**, providing a robust mechanism for controlling data flow based on text content.

Understanding the SAS FIND Function Syntax

The SAS **FIND** function generally requires two core arguments: the character expression (the string you are searching within) and the search string (the substring you are looking for). However, its full potential is unlocked through optional arguments that modify the search behavior, allowing for greater flexibility and control over how patterns are matched. The standard syntax is designed for maximum efficiency when executing iterative searches across large datasets within a DATA step loop.

The fundamental application of the function is built around performing a simple, exact match. This is the default behavior and is highly useful when you require high precision, such as searching for specific codes or tags where capitalization is invariant. When using the basic form, SAS executes a byte-by-byte comparison, stopping immediately upon finding the first match. This method is demonstrated below, representing the most common use case for rapid position identification.

When implementing the **FIND** function in a DATA step, you typically assign the returned position value to a new variable. If the variable specified is numeric, SAS automatically handles the assignment. If the search string is not found, the resulting variable will contain a value of 0, which is often used in subsequent conditional statements (e.g., an **IF-THEN-ELSE** block) to differentiate

between successful and unsuccessful searches. This structure makes the **FIND** function a cornerstone for complex data quality routines and text processing.

Method 1: Find Position of First Occurrence of String (Case-Sensitive)

The first and most direct method for utilizing the **FIND** function involves specifying only the source string and the target substring. By default, this execution is **case-sensitive**, meaning that the capitalization of the target substring must exactly match the capitalization within the source string for a positive match to occur. This strict matching is often necessary when dealing with data where case carries semantic meaning, such as proper nouns or technical identifiers.

```
data new_data;
set original_data;
first_occurrence = find(variable_name, "string");
run;
```

In this structure, the new variable `first_occurrence` will hold the starting position of the substring specified by `"string"` within the variable `variable_name`, but only if the capitalization is identical. If the target string appears in a different case (e.g., searching for "apple" but finding "APPLE"), the function will return 0, indicating no match was found. This strict requirement highlights why understanding your source data's formatting consistency is crucial before implementing any string search operation.

For large-scale data processing, the case-sensitive nature of this method can sometimes be restrictive, especially if data entry errors or inconsistent formatting are present. However, it is the fastest method computationally because it requires the least amount of preprocessing or modifier handling by the SAS engine. For situations where you need to locate patterns regardless of capitalization, the second method, which introduces optional arguments, becomes necessary.

Method 2: Handling Case Insensitivity with the 'i' Modifier

When the goal is to locate a substring irrespective of whether it is uppercase, lowercase, or mixed case, the **FIND** function offers an optional argument that modifies its search behavior. This is achieved by including a third argument, typically a character literal containing the modifier 'i' (for ignore case), or 't' (for trim), or a combination thereof. Utilizing the case-insensitive modifier ('i') significantly broadens the utility of the function, allowing analysts to identify patterns even in messy or inconsistent data sources.

```
data new_data;
set original_data;
```

```
first_occurrence = find(variable_name, "string", "i");
run;
```

The inclusion of the `"i"` argument tells SAS to perform a temporary internal conversion or comparison that disregards the case sensitivity during the search process. When this modifier is used, searching for "string" will successfully match "String," "STRING," or "sTrInG." This capability is invaluable in analytical tasks where the underlying meaning of the text is constant, but its presentation varies, ensuring that all relevant occurrences are captured. This is particularly important when dealing with unstructured data or user-generated input.

It is worth noting that while the 'i' modifier is the most common optional argument, the **FIND** function supports others, such as 't' (Trim), which instructs SAS to ignore trailing blanks in the source string, preventing potential false negatives when searching character variables that have been defined with fixed lengths. Combining modifiers, such as `"it"`, allows for sophisticated, robust searches that ignore both case and trailing whitespace, addressing two of the most frequent formatting inconsistencies encountered in real-world datasets.

Setting Up the Demonstration SAS Dataset

To fully illustrate the differences between case-sensitive and case-insensitive searching using the FIND function, we will utilize a small, custom-created dataset. This dataset, named `original_data`, contains several phrases designed to test the limits of case sensitivity, ensuring we have examples where the target substring appears in different capitalizations, or not at all. This preparation step is crucial for demonstrating how the presence or absence of the 'i' modifier fundamentally alters the search outcome.

The following DATA step code creates this sample data, defining a single character variable named `phrase`. This setup allows us to easily execute the **FIND** function against a consistent set of textual records.

```
/*create dataset*/
data original_data;
input phrase $1-25;
datalines;
The fox ran fast
That is a quick FOX
This fox is a slow fox
The zebra is cool
;
run;
```

```
/*view dataset*/
proc print data=original_data;
```

The resulting dataset, as visualized below, clearly shows the variations in capitalization, particularly the deliberate inclusion of "fox" in lowercase, "FOX" in uppercase, and two instances where the substring is present in the same record. The final record, "The zebra is cool," serves as a control, ensuring we test the function's ability to correctly return a zero when the target substring is absent.

Obs	phrase
1	The fox ran fast
2	That is a quick FOX
3	This fox is a slow fox
4	The zebra is cool

Example 1: Practical Application of Case-Sensitive Search

Our first practical example demonstrates Method 1: finding the position of the substring "fox" using the default, **case-sensitive** approach. In this scenario, we are explicitly looking for the exact three-character sequence "f-o-x" in lowercase. We assign the resulting position to a new numeric variable called `first_fox`.

```
data new_data;
set original_data;
first_fox = find(phrase, "fox");
run;
```

Executing this code against our sample data yields specific results based purely on exact character matching. The output clearly illustrates the limitation of the case-sensitive search: any variation in capitalization is treated as a complete failure to match. This precision is necessary in some contexts, but as the subsequent interpretation shows, it excludes potentially relevant observations if the data is not perfectly clean.

Obs	phrase	first_fox
1	The fox ran fast	5
2	That is a quick FOX	0
3	This fox is a slow fox	6
4	The zebra is cool	0

A careful analysis of the output clarifies the impact of case sensitivity:

The **f**ox ran fast (First occurrence is at position **5**). This is a perfect match for "fox".
That is a quick FOX (The lowercase string "fox" never occurs). The result is **0** because "FOX" (uppercase) does not match "fox" (lowercase).
This **f**ox is a slow fox (First occurrence is at position **6**). The function finds the first instance of the lowercase "fox" and stops searching.
The zebra is cool (The string "fox" never occurs). The result is correctly **0**.

Example 2: Practical Application of Case-Insensitive Search

To overcome the limitations observed in Example 1, we now implement Method 2, utilizing the 'i' modifier to perform a **case-insensitive** search. This modification ensures that we capture all occurrences of "fox," regardless of their capitalization within the `phrase` variable. This approach is generally preferred when the goal is pattern recognition rather than strict textual identity.

```
data new_data;
set original_data;
first_fox = find(phrase, "fox", "i");
run;
```

The key difference in the code above is the inclusion of the `", "i""` argument. By instructing the FIND function to ignore case, we expect to see successful matches in records that previously returned a zero. The resulting dataset confirms that the function successfully identified occurrences of the target substring even when capitalized differently, providing a more comprehensive result set for analysis.

Obs	phrase	first_fox
1	The fox ran fast	5
2	That is a quick FOX	17
3	This fox is a slow fox	6
4	The zebra is cool	0

Interpreting the new output confirms the effectiveness of the 'i' modifier:

The **f**ox ran fast (First occurrence is at position **5**). The match remains the same as in Example 1.
That is a quick **F**OX (First occurrence of "fox" is at position **17**). This time, the function successfully matches the uppercase "FOX" and returns the correct starting position, demonstrating the power of ignoring case.
This **f**ox is a slow fox (First occurrence is at position **6**). The first occurrence (lowercase) is identified.
The zebra is cool (The string "fox" never occurs). The result remains **0**, as expected.

Conclusion and Further Resources

The FIND function is a flexible and indispensable component of the SAS string manipulation toolkit. Whether requiring strict, case-sensitive identification or a broader, case-insensitive pattern search, the function's optional arguments provide the precision needed for rigorous data processing. By correctly identifying the position of substrings, analysts can reliably perform data validation, extraction, and standardization tasks within the DATA step.

Selecting the appropriate method--case-sensitive versus case-insensitive--depends entirely on the requirements of your analysis and the quality of your source data. For perfectly structured data, the default case-sensitive search is efficient and precise. However, for real-world data plagued by formatting inconsistencies, utilizing the 'i' modifier is essential for robust and comprehensive pattern detection.

To further enhance your proficiency in SAS string and data handling, consider exploring related functions and procedures that work in conjunction with **FIND**. Understanding functions like **INDEX**, **SCAN**, and **SUBSTR**, along with advanced regular expression capabilities found in the **PRX** family of functions, will significantly broaden your ability to manage complex textual information effectively.

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