Learning SAS: A Comprehensive Guide to String Manipulation with the FIND and INDEX Functions

Authored by

Mohammed looti

November 14, 2025

Effective **string manipulation**--the processing of textual data--is a cornerstone skill in **SAS** programming. Data preparation, cleaning, and sophisticated text analysis frequently require locating specific patterns or identifying the precise location of text segments within character variables. To achieve this, SAS provides two critically important functions: the **FIND** function and the **INDEX** function. Both are designed to search for a specified **substring** within a larger target string and return an integer value corresponding to the **character position** of its first occurrence.

Although the primary goal of **FIND** and **INDEX** is shared--pinpointing text within text--their operational capabilities diverge significantly. These subtle, yet profound, differences determine which function is optimally suited for a given task. Failure to appreciate these nuances can result in code that is inefficient, overly complex, or, critically, returns incorrect results, particularly when dealing with non-standardized or messy data sources common in real-world environments. The choice between them often depends on specific search requirements, such as whether the search must be strictly case-sensitive or if the search logic needs to commence from a position other than the start of the string.

This comprehensive guide is dedicated to dissecting the functionalities of both the **FIND function** and the **INDEX function**, offering a detailed comparison of their syntax, behavior, and performance implications. We will specifically highlight the unique advantages of **FIND**, including its integrated support for **case-insensitive search** operations and its capacity to define an exact **starting position**. Mastering these technical distinctions is essential for writing robust, flexible, and accurate SAS code that stands up to demanding data processing challenges.

The **FIND** function is significantly more flexible, natively supporting advanced options like **case-insensitive search** capabilities, allowing programmers to ignore letter casing during the search process without the need for additional functions like UPCASE or LOWCASE.
The **FIND** function allows the user to specify a precise **starting position**, enabling searches to commence anywhere within the string, whereas **INDEX** is strictly constrained to searching from the first character.

Understanding the Core Functionality

Fundamentally, both the **FIND** and **INDEX** functions serve as indispensable tools for analyzing text within the SAS environment. Their primary and consistent output is an integer value that accurately identifies the numerical position where the targeted substring first begins within the target string. This returned position is invaluable for subsequent operations, such as extracting the text using the SUBSTR function or applying conditional logic based on the presence of a keyword. A key shared behavior, crucial for error handling, is the uniform return value when a match is absent: if the specified **substring** is not located within the target string, both **FIND** and **INDEX** reliably return a value of 0, clearly signaling a lack of occurrence.

In their most basic form--when the search is strictly case-sensitive and implicitly starts at position 1--the two functions are functionally interchangeable. For example, executing `INDEX(String, Substring)` yields the exact same numerical result as `FIND(String, Substring)`. This direct equivalence means that for quick, simple text checks in controlled environments, either function is technically sufficient. However, relying solely on this basic interchangeability overlooks the complexities of real-world data, where uniformity and cleanliness are rare commodities, necessitating more robust search methods.

Therefore, the selection between **FIND** and **INDEX** should be a strategic choice, heavily influenced by considerations of code maintainability and future flexibility. While **INDEX** is often favored for its inherent simplicity and fewer required arguments, **FIND** establishes a superior foundation for expansion and modification. If there is any foreseeable possibility that the search criteria might need to incorporate case insensitivity, directional searching, or a non-standard starting point, it is best practice to utilize **FIND** immediately. This proactive approach ensures the code remains efficient and easily adaptable to evolving data requirements without necessitating a complex structural rewrite.

To provide clear, practical context for our comparison, we first define a sample SAS dataset. This dataset, named `original_data`, contains various descriptive phrases that will serve as the target strings for demonstrating the search operations of both functions:

```
/*create dataset*/
data original_data;
input phrase $40.;
datalines;
A pig is my favorite animal
My name is piglet
Pigs are so cute
Here is a baby pig
His name is piggie
;
run;

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

Obs	phrase
1	A pig is my favorite animal
2	My name is piglet
3	Pigs are so cute
4	Here is a baby pig
5	His name is piggie

Example 1: Basic Search - Functional Equivalence

Our initial practical exercise is designed to establish a functional baseline, meticulously demonstrating how both the **FIND** and **INDEX** functions behave when invoked without any optional parameters. In this standard configuration, the search is inherently case-sensitive and always initiates from the first **position** of the string. The objective here is straightforward: to locate the numerical position of the first occurrence of the lowercase **substring** 'pig' across all phrases within our `original_data` table. Because we are limiting the search to this simple, strict criteria, the functions are expected to operate identically, yielding the exact same position values for every record.

The following **DATA step** code executes this precise comparison. We generate two new dedicated numeric variables, `find_pig` and `index_pig`, to store the output produced by each respective function call side-by-side. This dual variable creation allows for immediate and meticulous visual verification of the results, confirming their equivalence under these specific, fundamental conditions. Note that the syntax in this basic mode is exceptionally concise, requiring only the target string (`phrase`) and the search term ('pig').

```
/*find position of first occurrence of 'pig' in phrase column*/
data new_data;
set original_data;
find_pig = find(phrase, 'pig');
index_pig = index(phrase, 'pig');
run;

/*view results*/
proc print data=new_data;
```

Obs	phrase	find_pig	index_pig
1	A pig is my favorite animal	3	3
2	My name is piglet	12	12
3	Pigs are so cute	0	0
4	Here is a baby pig	16	16
5	His name is piggie	13	13

As confirmed by the resulting output table, the values populated in both the `find_pig` and `index_pig` columns are perfectly identical across all observations. This outcome firmly establishes that for a simple, case-sensitive search initiated from the start of the string, the **FIND** and **INDEX** functions are functionally equivalent. Crucially, they both return 0 for the record containing "Pigs are so cute" because the search term 'pig' (lowercase) does not exactly match 'Pigs' (capitalized). This shared adherence to strict case sensitivity sets the stage for exploring the differentiating features of the **FIND** function.

Example 2: Leveraging Case-Insensitive Search with FIND

The first truly significant capability that sets the **FIND** function apart is its integrated mechanism for performing a **case-insensitive search**. This feature is indispensable when analysts encounter real-world data plagued by inconsistent capitalization, which often arises from varied data entry standards or non-uniform source systems. In stark contrast, the **INDEX function** is fundamentally and strictly case-sensitive, meaning it mandates an exact match for the capitalization provided in the search term and offers no simple, built-in option to override this constraint.

To clearly demonstrate this critical disparity, we adjust our search term to 'PIG' (all uppercase) and apply it to both functions simultaneously. For the **FIND** function, we introduce the optional third argument: the 'i' **modifier**. This powerful modifier instructs **FIND** to disregard the case of characters in both the target string and the search term during the comparison process. The **INDEX** function, however, must continue its operation under its inherent strict case-sensitive mode, requiring an exact match of 'PIG' to return a valid position, which is unlikely given our source data's mixed casing.

```
/*find position of first occurrence of 'PIG' in phrase column*/
data new_data;
set original_data;
find_pig = find(phrase, 'PIG', 'i');
index_pig = index(phrase, 'PIG');
```

```
run;

/*view results*/
proc print data=new_data;
```

Obs	phrase	find_pig	index_pig
1	A pig is my favorite animal	3	0
2	My name is piglet	12	0
3	Pigs are so cute	1	0
4	Here is a baby pig	16	0
5	His name is piggie	13	0

The resulting table graphically illustrates the indispensable utility of the 'i' modifier. The `find_pig` column successfully returns the correct starting position for all rows containing 'pig', 'Pig', or 'PIG', effectively treating them as the same text entity. In contrast, the `index_pig` column returns 0 for every record because the exact string 'PIG' in all uppercase does not exist. This stark difference underscores a major functional limitation of **INDEX**: without manual preprocessing (such as converting the entire target string to uppercase via the UPCASE function before searching), it cannot handle case variations, solidifying **FIND** as the superior tool for flexible and robust text matching.

Example 3: Controlling the Search Start Position with FIND

The second crucial differentiating characteristic exclusive to the **FIND function** is its explicit ability to accept a numerical argument that dictates the precise **starting position** for the search operation. This functionality is absolutely vital in advanced **string** parsing scenarios, particularly when the requirement is to locate the second, third, or any subsequent occurrence of a pattern, or when the desired term is known to appear only after a certain fixed marker or text length. The **INDEX function**, lacking this parameter, is rigidly confined to always initiating its search from the very first **position** (position 1).

In this demonstration, we continue our search for the lowercase 'pig' within the `phrase` column. Critically, however, we use the **FIND** function to enforce a search starting point at the 5th **character position** onwards. This instruction means that if the substring 'pig' occurs at position 1, 2, 3, or 4, **FIND** will deliberately ignore it and return 0, unless another instance exists after position 4. For comparative purposes, the **INDEX** function executes its standard, full-string search, beginning at position 1 and capturing the very first occurrence irrespective of its location.

```
/*find position of first occurrence of 'pig' in phrase column starting at position 5*/
data new_data;
set original_data;
find_pig = find(phrase, 'pig', 5);
index_pig = index(phrase, 'pig');
run;

/*view results*/
proc print data=new_data;
```

Obs	phrase	find_pig	index_pig
1	A pig is my favorite animal	0	3
2	My name is piglet	12	12
3	Pigs are so cute	0	0
4	Here is a baby pig	16	16
5	His name is piggie	13	13

The visual output clearly highlights the functional difference created by controlling the starting position. Consider the first row: "A pig is my favorite animal." The substring 'pig' starts at position 3. Because the `find_pig` function was explicitly instructed to start searching only at position 5, it bypasses this legitimate occurrence and consequently returns 0. Conversely, `index_pig` correctly identifies the first occurrence at position 3. This controlled search capability transforms **FIND** into an invaluable tool for complex data extraction and validation, enabling developers to isolate targeted instances of a pattern without resorting to cumbersome manual string manipulation using functions like SUBSTR before searching.

Syntactic Flexibility and Advanced Modifiers in FIND

The extended utility of the **FIND** function is primarily derived from its robust support for various optional arguments, collectively known as **modifiers**, which grant nuanced and precise control over the search mechanism. This level of refinement is fundamentally absent in the **INDEX** function. While we have already examined the 'i' modifier for case insensitivity, other modifiers further enhance **FIND**'s versatility, positioning it as the definitive tool for advanced **string** processing tasks in SAS.

One critical modifier is 't' (for trim), which instructs the **FIND** function to ignore trailing blanks in the target string before performing the search. In the SAS environment, fixed-length character

variables often carry space padding up to their defined length. If this padding is not managed, it can sometimes skew search logic, particularly when searching for terms located near the end of a variable. By employing the 't' modifier, the search logically operates on a trimmed version of the string, guaranteeing that the function accurately reports the position of the **substring** without interference from extraneous whitespace. This capability simplifies code structure significantly, as it eliminates the necessity of manually wrapping the target string within the TRIM function prior to executing the search.

Furthermore, **FIND** provides powerful directional control through the 'b' (backward) modifier. While the default behavior for both **FIND** and **INDEX** is to scan from left to right (forward), the 'b' modifier allows **FIND** to initiate the search from the end of the string and proceed backward toward the beginning. This is extremely useful when the requirement is to locate the *last* occurrence of a **substring**, a common need when parsing complex file paths, URLs, or hierarchical identifiers. Achieving this same result using the **INDEX** function typically demands a multi-step workaround involving reversing the string, searching, and then performing complex recalculations based on the original string length. **FIND**'s ability to handle directional searching natively unequivocally establishes it as the more powerful and flexible choice for intricate character variable handling.

When to Choose FIND vs. INDEX

The strategic decision regarding whether to implement the **FIND function** or the **INDEX function** within your **DATA step** programming should always be driven by the complexity and specific constraints of the required string operation. While **INDEX** provides a simple, highly efficient solution perfectly suited for the most rudimentary searches, **FIND** is built with the enhanced capabilities necessary to manage the inherent variability of real-world data, offering superior control and adaptability.

You should confidently select the **FIND** function when your search parameters must extend beyond the fundamental, particularly in the following common scenarios:

When dealing with external or user-generated data that exhibits inconsistent capitalization, necessitating a **case-insensitive search** through the use of the 'i' **modifier**.
When your search logic requires locating an occurrence only after a specific point in the string, thereby utilizing the optional **starting position** argument to effectively segment or narrow the search area.
When you need to identify the last occurrence of a pattern (backward search using the 'b' modifier) or when you must ensure that trailing blanks do not interfere with the position calculation (using the 't' modifier).

Conversely, the **INDEX** function remains a perfectly appropriate and often simpler choice in straightforward situations. Use **INDEX** when the search must be strictly case-sensitive, when you

are guaranteed to search only from the beginning of the **string**, or when the priority is minimal syntax for a rapid, uncomplicated existence check. In these instances, **INDEX** delivers a concise and highly effective solution without the need to manage additional optional parameters.

In summary, **INDEX** serves as the reliable workhorse for standard, fixed-criteria substring searches, prized for its conciseness and speed in basic operations. However, **FIND** is the specialized, advanced tool, providing granular control over casing, starting position, padding management, and search direction. For any complex, mission-critical, or data quality-sensitive application, **FIND** is strongly recommended due to its versatility and superior ability to handle common data irregularities directly within the function call, thus reducing the need for extensive data preprocessing.

Additional Resources for SAS String Mastery

For programmers committed to advancing their expertise in **SAS** string functions and programming methodologies, continuous engagement with authoritative resources is essential. The official SAS documentation remains the definitive source, offering exhaustive detail on the full spectrum of modifiers available for the **FIND** function, as well as nuanced explanations of return values and performance considerations for both **FIND** and **INDEX**. Furthermore, actively participating in community forums and user groups provides invaluable insight into real-world applications and effective best practices for tackling complex data manipulation challenges.

By exploring tutorials and documentation on related SAS functions such as SCAN, SUBSTR, and UPCASE, you can significantly deepen your understanding of how **FIND** and **INDEX** integrate into a comprehensive toolkit. Mastering these fundamental building blocks is crucial for transitioning successfully from basic data retrieval tasks to sophisticated analytical programming within the SAS environment.