Learning VBA: How to Check if a String Contains Another String

Authored by

Mohammed looti

February 16, 2026

Mastering String Searches with the VBA InStr() Function

In the realm of data manipulation and automation within **Microsoft Excel**, the ability to efficiently search for specific text patterns within larger **strings** is an indispensable skill. Whether you are tasked with parsing complex user input, performing rigorous data cleaning, or filtering extensive record sets, knowing how to programmatically determine if one string contains another is crucial for robust application development. **VBA**, or **Visual Basic for Applications**, provides a highly effective built-in **function** specifically engineered for this purpose: the **InStr()** function.

The **InStr()** function, short for "In String," serves as the primary tool for locating the precise starting position of a specified substring within a host string. This capability is foundational for implementing complex conditional logic and validation routines in your automation scripts. This comprehensive guide will thoroughly explore the mechanics and arguments of **InStr()**, demonstrate its practical application through a detailed, real-world example, and offer insights into best practices for maximizing search efficiency in your **VBA** projects. By the conclusion of this tutorial, you will possess the expertise required to leverage this function effectively, significantly enhancing your data processing and filtering capabilities.

Understanding the Syntax and Parameters of InStr()

The core utility of the **InStr() function** lies in its ability to pinpoint the existence and location of a character sequence within a larger body of text. It returns a numeric **integer** value that represents the starting position of the first occurrence of the search string. Critically, if the target substring is not found anywhere in the main string, the function returns the value 0. This simple return logic makes **InStr()** perfect for integration into control flow structures like `If...Then` statements used for data validation and conditional execution.

The **syntax** for the **InStr()** function is flexible, designed to accommodate both basic searches and more nuanced, case-insensitive operations. A thorough understanding of each argument is essential for fully harnessing its power. The standard form of the function call is structured as follows:

InStr(, string1, string2,)

We must carefully examine each component of this function signature:

start (Optional): This numeric argument designates the character position within **string1** where the search should commence. If this argument is omitted, the search automatically begins at the first character (position 1). Specifying a start position can significantly optimize searches when you know the substring must appear later in the text.
string1 (Required): This is the primary **string** that is being analyzed. It represents the larger text

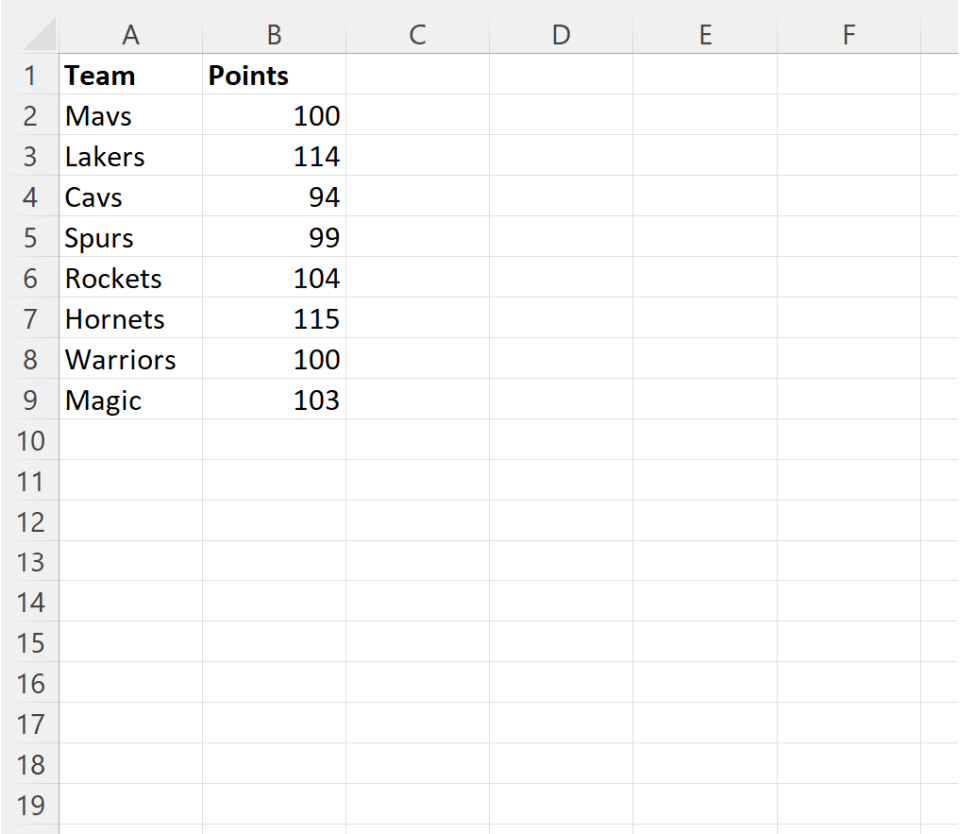
block within which the presence of **string2** is being checked.
string2 (Required): This is the target **substring** that the function attempts to locate within **string1**.
compare (Optional): This argument dictates the type of string comparison to be executed. It can be set to **vbBinaryCompare** (the default setting, which enforces case-sensitivity) or **vbTextCompare** (which performs a case-insensitive search). Utilizing this parameter is the cleanest method for flexible searches that disregard capitalization.

The return value is an **integer**. Any value greater than 0 precisely identifies the starting character position of **string2** within **string1**. If **string2** is not found, or if **string1** has zero length, the function consistently returns 0. Furthermore, special rules apply to edge cases: if **string2** is an empty string, **InStr()** returns the value specified in the **start** argument. These operational details are vital for designing robust and error-resistant search algorithms.

Practical Application: Filtering Data in an Excel Worksheet

To fully appreciate the efficiency and simplicity of the **InStr() function**, let us analyze a common task within **Excel** data management. Suppose you are managing a large spreadsheet containing sports team data, and your primary goal is to quickly identify and extract all rows where the team name contains a specific, shared sequence of characters, such as "avs". This type of targeted filtering is extremely valuable for generating specialized reports or conducting focused data analysis.

Our sample dataset, shown below, includes various team names and their corresponding points. Our objective is to write a **VBA** script that systematically scans the "Team" column (Column A) for entries containing "avs" and then copies those complete records (Columns A and B) to a new, dedicated output area (Columns D and E) on the same worksheet.



	A	B	C	D	E	F
1	**Team**	**Points**				
2	Mavs	100				
3	Lakers	114				
4	Cavs	94				
5	Spurs	99				
6	Rockets	104				
7	Hornets	115				
8	Warriors	100				
9	Magic	103				
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						

We seek to automate this filtering process using a **VBA macro**. This script will iterate through the dataset, applying the **InStr()** function to each entry to check for the target substring. Upon a successful match, the script will execute the necessary action--copying the relevant data row. Adopting this automated approach drastically reduces the manual effort and time required, especially when dealing with datasets that contain thousands of rows, while simultaneously guaranteeing the accuracy of the filtering operation.

Deconstructing the VBA Code for String Containment

Let's meticulously examine the **VBA** code snippet designed to execute our string containment check and data extraction. This code is structured to be both efficient and highly adaptable, allowing for straightforward modification of the search criteria, the target range, or the output destination. A line-by-line understanding of this script is key to customizing and extending its capabilities for your specialized data tasks.

```
Sub StringContains()
Dim i As Integer, i_num As Integer

For i = 2 To 9
If InStr(1, LCase(Range("A" & i)), "avs") <> 0 Then
```

```
i_num = i_num + 1
Range("D" & i_num & ":E" & i_num) = Range("A" & i & ":B" & i).Value
End If
Next i
End Sub
```

Below is a detailed walkthrough of the script's logic and functionality:

`Sub StringContains()`: This command initiates the subroutine, declaring the beginning of our **VBA macro** named `StringContains`.
`Dim i As Integer, i_num As Integer`: Two variables are declared as **Integer** data types. `i` serves as the primary loop counter, controlling the iteration through the source rows (A2 to A9). `i_num` is the output row counter, ensuring that every matched record is copied sequentially into the target area (Columns D and E) without leaving any blank rows.
`For i = 2 To 9`: This establishes the primary loop, which iterates from row 2 (skipping the header) up to row 9, covering the entire example dataset.
`If InStr(1, LCase(Range("A" & i)), "avs") <> 0 Then`: This line contains the crucial conditional logic.

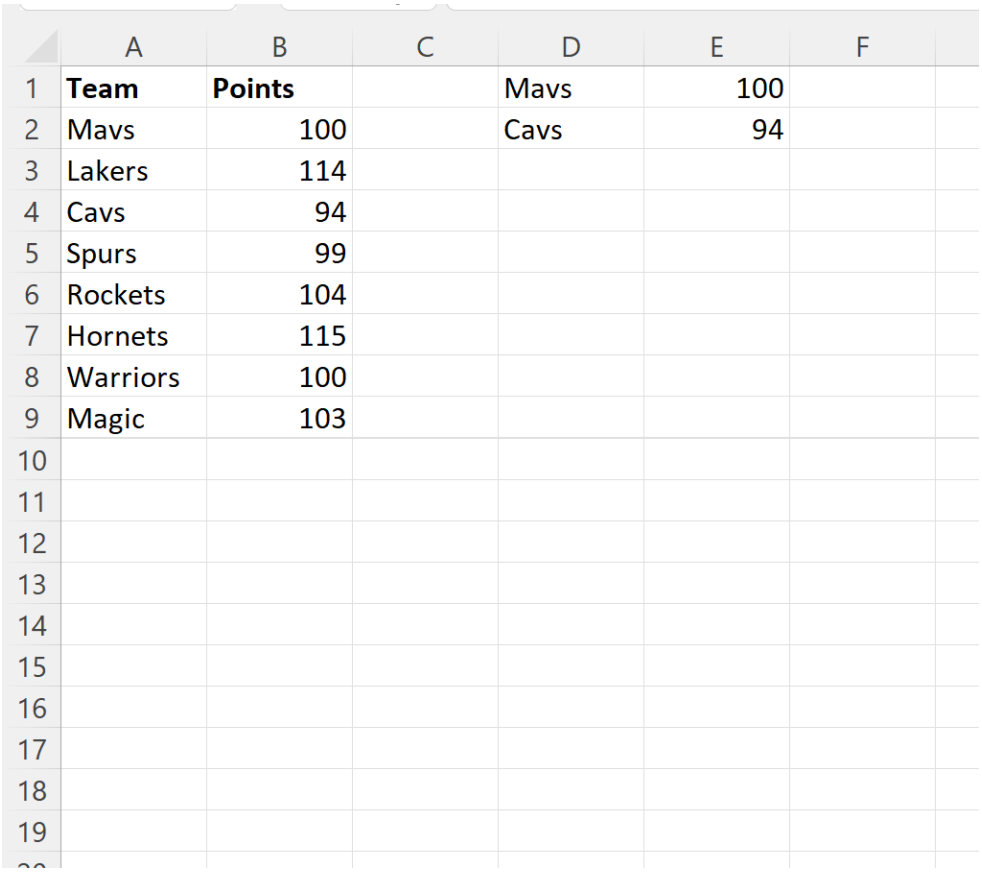
`Range("A" & i)`: Dynamically references the cell in Column A corresponding to the current loop iteration (e.g., A2, A3, etc.).
`LCase(...)`: The **LCase function** converts the cell content to lowercase before searching. This is essential because the search string "avs" is lowercase, and applying `LCase` guarantees a case-insensitive match against names like "Mavs" or "CAVS".
`InStr(1, ..., "avs")`: The **InStr()** function searches for "avs" within the modified (lowercase) team name, starting from character position 1.
`<> 0`: This comparison checks if the return value is non-zero, confirming that the substring "avs" was successfully located.

If the condition evaluates to true, the subsequent extraction steps are executed.
`i_num = i_num + 1`: Upon a successful match, the output row counter is incremented, preparing the script to write the data to the next available row in the output range.
`Range("D" & i_num & ":E" & i_num) = Range("A" & i & ":B" & i).Value`: This line handles the data extraction and copying process. It retrieves the `.Value` of the entire matched row (Columns A and B) and assigns these values to the designated output cells (Columns D and E) in the row specified by `i_num`.
The loop concludes with `End If` and proceeds to the next iteration with `Next i`, before finally terminating the subroutine with `End Sub`.

Analyzing the Output and Interpreting Results

Upon the successful execution of the **VBA macro** detailed above, the results of the filtering operation are instantly visible within your **Excel** worksheet. The objective was to isolate and copy only those rows where the "Team" column contained the substring "avs," and the resulting output perfectly confirms this precise filtering criterion.

The image below illustrates the final state of the worksheet after the script has run. It clearly shows that columns D and E now contain a subset of the original data, consisting only of the records that satisfied our string search condition defined by the **InStr()** function.



	A	B	C	D	E	F
1	**Team**	**Points**		Mavs	100	
2	Mavs	100		Cavs	94	
3	Lakers	114				
4	Cavs	94				
5	Spurs	99				
6	Rockets	104				
7	Hornets	115				
8	Warriors	100				
9	Magic	103				
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						

As the visual evidence demonstrates, only the records for "Mavs" and "Cavs" were copied to the output section. This outcome occurred because these were the only two team names in the source data that contained the required character sequence "avs". The macro correctly identified both occurrences, regardless of the original capitalization, thanks to the tactical inclusion of the **LCase() function**, which ensured a consistent case-insensitive comparison against the search term. This example powerfully illustrates the efficiency of combining **InStr()** with looping structures for high-volume data processing tasks.

Advanced Considerations and Best Practices

While the fundamental application of the **InStr() function** is highly effective, integrating several advanced considerations and best practices can optimize your string searches further, leading to more resilient and efficient **VBA** solutions.

A primary consideration is the handling of **case sensitivity**. In our earlier example, we used **LCase()** to force a case-insensitive search. A cleaner, alternative method is leveraging the optional `compare` argument of the **InStr()** function itself. By setting this argument to **vbTextCompare**, you achieve case-insensitivity without altering the content of the string being searched. For instance, the improved syntax would be `InStr(1, Range("A" & i).Value, "avs", vbTextCompare)`. While both methods yield the same result, **vbTextCompare** is often preferred by professional developers as it keeps the core string value intact and improves code readability.

Another crucial practice involves proper handling of **empty cells or potential errors**. If the target **Range** cell could be empty, contain an error value, or hold non-string data, attempting to access its `.Value` property for string manipulation can trigger a runtime error. To prevent script failure, it is advisable to incorporate explicit error trapping or, preferably, use validation checks. You could verify the cell state using functions like `IsEmpty(Range("A" & i).Value)` or confirm the data type before processing: `If VarType(Range("A" & i).Value) = vbString Then ...`. This proactive validation ensures robust execution across varied datasets.

Finally, for scenarios involving complex pattern matching--such as validating strings that must conform to a specific format like email addresses or structured identification numbers--the simple substring search provided by **InStr()** may be insufficient. In these complex cases, developers should consider utilizing the **Like operator** for pattern wildcard searches, or, for maximum flexibility and intricacy, implementing **Regular Expressions**. Nevertheless, for the common task of simply checking if one string contains another, **InStr()** remains the most efficient, fast, and straightforward tool available in **VBA**.

Conclusion and Further Learning

The **InStr() function** is unequivocally a cornerstone of effective **VBA** programming, providing a powerful yet remarkably simple mechanism for determining the presence and location of a target **string** within a larger body of text. By thoroughly grasping its essential syntax, understanding its optional parameters, and utilizing its distinctive return values, you can substantially improve your capacity to programmatically manipulate and analyze text data residing within **Excel**. Our detailed, practical example demonstrated how to leverage this function to automate data filtering, transforming a potentially tedious manual effort into a swift and error-free automated process.

When developing real-world applications, always prioritize best practices, particularly addressing

case sensitivity (via **LCase()** or **vbTextCompare**) and implementing robust error handling to account for empty or non-string data types. While **InStr()** excels at direct substring checks, remember that tools like the **Like operator** or **Regular Expressions** offer greater flexibility for highly complex pattern detection. Integrating these concepts will ensure you build reliable and efficient **VBA** solutions tailored to advanced data analysis needs.

To continue expanding your **VBA** proficiency, we encourage you to explore the following authoritative resources and related tutorials. Consistent practical application is the most effective path to mastering these fundamental programming concepts.

Official **InStr() function** documentation on Microsoft Learn.
Tutorials on using the **LCase() function** for case conversion.
Guides on working with the **Range object** in **Excel VBA**.
Understanding **Integer** and other data types in **VBA**.