How to use Like Operator in VBA (With Examples)

Authored by

stats writer

November 18, 2025

Introduction: The Power of the VBA Like Operator

The **Like** operator is a fundamental tool within VBA (Visual Basic for Applications) used specifically for performing advanced pattern matching against textual data. Unlike simple equality checks, which demand an exact character-for-character match, the **Like** operator provides flexibility by allowing developers to determine if a given string conforms to a specified pattern. This functionality is invaluable when dealing with data validation, searching complex datasets, or extracting information based on partial or positional characteristics of a string. Mastery of this operator allows for highly efficient and dynamic handling of text comparisons within your automation scripts.

Effective utilization of the **Like** operator revolves around understanding the various wildcard characters that represent variable parts of the pattern. These wildcards enable you to specify conditions such as whether a string contains a certain substring, begins or ends with a specific sequence, or follows a predetermined structure (e.g., a fixed format like a phone number or part number). Because it is built directly into the VBA language, integrating **Like** operations into loops, conditional statements (such as **If...Then**), and data processing routines is straightforward, significantly enhancing the analytical capabilities of your macros.

For instance, a common application involves iterating through a range of cells in a spreadsheet and checking if the content of each cell meets a specific criterion. If you needed to identify all entries in the cell range **A2:A10** that contain the substring "hot," the **Like** operator provides a clean, readable syntax to accomplish this goal, subsequently outputting the logical results into the range **B2:B10**. This simple pattern search demonstrates the immediate utility of the operator in streamlining repetitive data analysis tasks within Excel.

Syntax and Fundamentals of the Like Operator

The basic syntax for the **Like** operator involves comparing a primary string expression against a pattern string expression. The structure generally follows the format: `Result = stringexpression Like patternexpression`. The result of this comparison is a Boolean value: **True** if the string matches the pattern, and **False** otherwise. Understanding this foundational structure is the first step toward building powerful conditional logic within your VBA procedures. The true power lies in the construction of the `patternexpression` using specialized wildcard characters.

When constructing the pattern, it is crucial to recognize that the comparison is case-sensitive by default, which is known as binary comparison mode. If you require a case-insensitive match (where "Hot" matches "hot"), you must declare `Option Compare Text` at the beginning of your module. This setting ensures that the pattern matching process treats uppercase and lowercase letters as equivalent, providing flexibility depending on the specific data requirements of your application.

Ignoring this distinction can lead to subtle bugs where data is missed due to capitalization differences.

Consider the following foundational example, which illustrates how to structure a basic VBA subroutine utilizing the **Like** operator to scan a range of cells. Note the use of the `*` wildcard, which is key to finding substrings regardless of their position within the text.

Sub CheckLike()

Dim i As Integer

For i = 2 To 10
If Range("A" & i) Like "*hot*" Then
Range("B" & i) = "Contains hot"
Else
Range("B" & i) = "Does not contain hot"
End If
Next i

End Sub

This implementation demonstrates the powerful combination of a looping structure (**For...Next**) with a conditional check (**If...Like...Then**) to automate data analysis across a defined range. It is a canonical example of how to leverage pattern matching capabilities to process cell data programmatically in VBA.

Wildcard Character Mastery: The Asterisk (*) and Question Mark (?)

The effectiveness of the **Like** operator relies heavily on the use of wildcard characters. The two most commonly used and essential wildcards are the asterisk (`*`) and the question mark (`?`). The asterisk is the most versatile, representing zero or more characters. Placing an asterisk at the beginning of a pattern (e.g., `"*text"`) signifies that the string must end with "text," preceded by any sequence of characters, including an empty sequence. Placing it at the end (e.g., `"text*"`) signifies that the string must start with "text," followed by zero or more characters. When placed around a substring (e.g., `"*text*"`), it performs a comprehensive "contains" search, matching "text" anywhere within the string.

In contrast, the question mark (`?`) is used to match exactly one single character. This is invaluable when searching for strings of a fixed length where only specific characters are known. For instance, the pattern `"A?C"` would successfully match "ABC," "A1C," or "AEC," but it would fail to match "AC" (too short) or "ABBC" (too long). By combining multiple question marks, you can

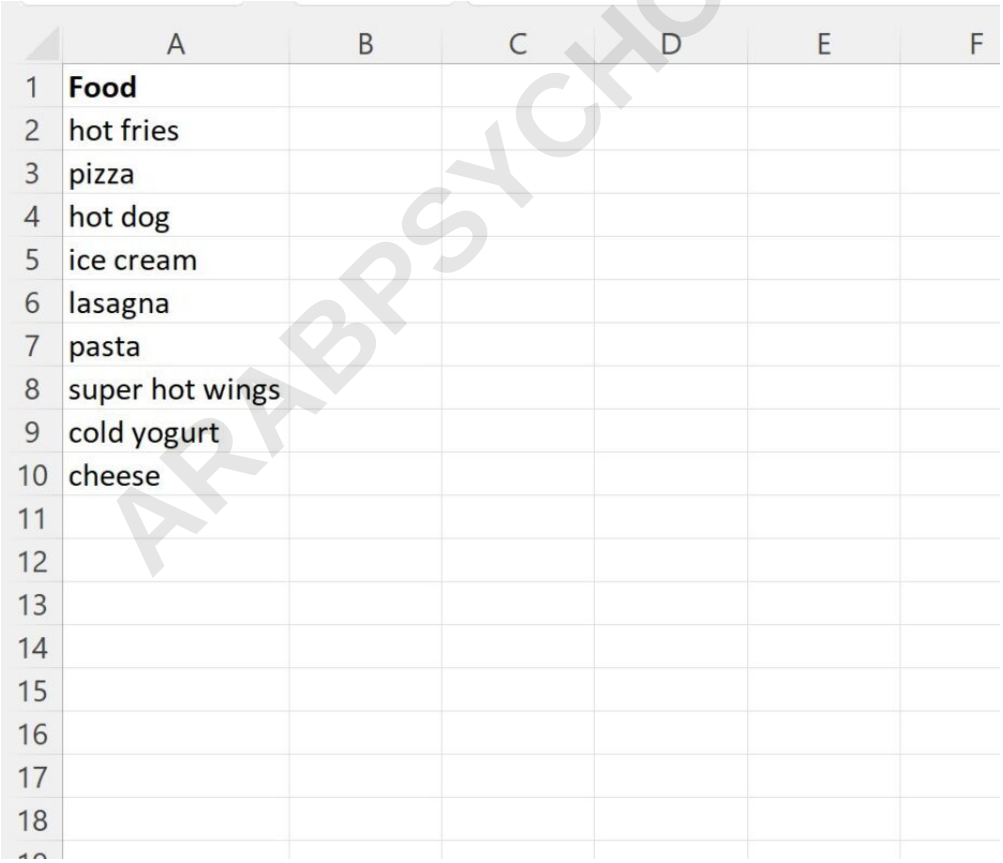
enforce specific string lengths. For example, a search for `"???-????-?"` could be used to validate or identify records that strictly adhere to a nine-character format with hyphens in specific positions.

Understanding the difference between these two wildcards is critical for precise string matching. The asterisk provides flexibility in length, while the question mark imposes strict length constraints on the variable parts of the pattern. By strategically placing these wildcards, you can craft highly specific patterns tailored to complex data formats encountered in real-world spreadsheets and databases.

Practical Application 1: Searching for Substrings (The "Contains" Check)

Let us delve into a practical demonstration of using the **Like** operator to check if cells contain a specific substring. Suppose we have a list of various food items in Column A of our Excel worksheet, and our objective is to quickly flag all items whose description includes the word "hot," regardless of where that word appears within the name.

The initial data setup often looks similar to this, illustrating a diverse list of entries ranging from simple names to more descriptive phrases:



	A	B	C	D	E	F
1	**Food**					
2	hot fries					
3	pizza					
4	hot dog					
5	ice cream					
6	lasagna					
7	pasta					
8	super hot wings					
9	cold yogurt					
10	cheese					
11						
12						
13						
14						
15						
16						
17						
18						

To perform this task, we construct a VBA macro that iterates from row 2 to row 10. Within the loop,

the core logic employs the pattern `"*hot*"`. The asterisks ensure that the match occurs whether the word "hot" is at the start, middle, or end of the food name. This is the definitive structure for an efficient "contains" search operation.

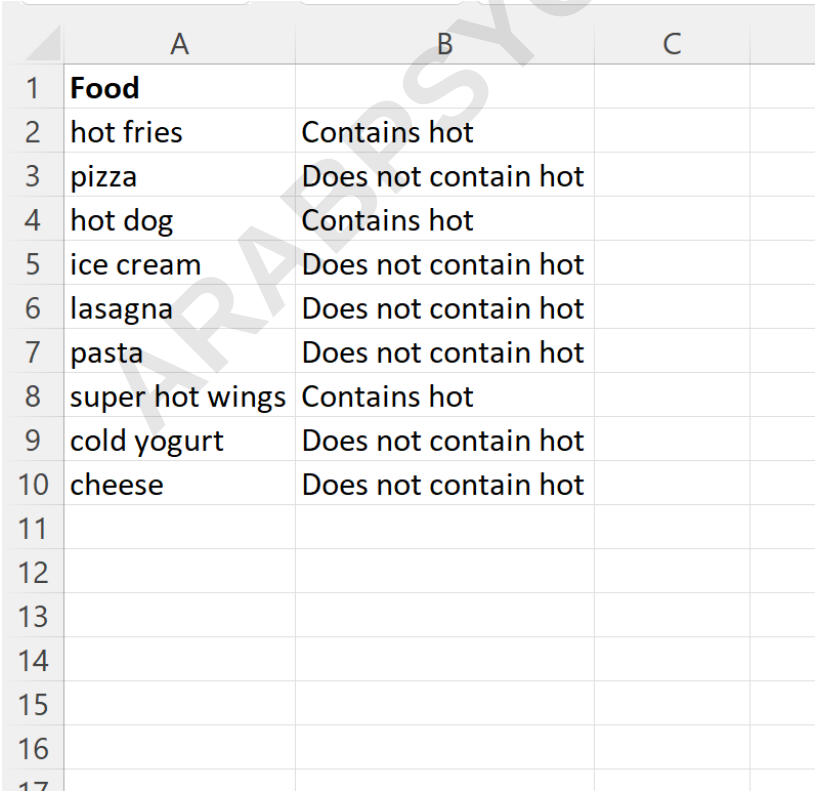
```
Sub CheckLike()

Dim i As Integer

For i = 2 To 10
If Range("A" & i) Like "*hot*" Then
Range("B" & i) = "Contains hot"
Else
Range("B" & i) = "Does not contain hot"
End If
Next i

End Sub
```

Executing this macro yields a definitive output in Column B, clearly indicating which food items meet the pattern criteria. The result visually confirms the powerful filtering capability provided by the bidirectional asterisk wildcard:



	A	B	C
1	**Food**		
2	hot fries	Contains hot	
3	pizza	Does not contain hot	
4	hot dog	Contains hot	
5	ice cream	Does not contain hot	
6	lasagna	Does not contain hot	
7	pasta	Does not contain hot	
8	super hot wings	Contains hot	
9	cold yogurt	Does not contain hot	
10	cheese	Does not contain hot	
11			
12			
13			
14			
15			
16			

As demonstrated by the output, Column B successfully verifies whether or not each corresponding cell in Column A contains the target substring "hot." The simplicity of the pattern `"*hot*"` belies its efficiency in handling flexible text searches across large datasets.

The Importance of Character Lists () and Ranges

Beyond the asterisk and question mark, the **Like** operator offers highly granular control through the use of character lists, denoted by square brackets (). A character list matches any single character enclosed within the brackets at that specific position in the string. For example, the pattern `"CT"` would match "CAT" and "CTT," but not "CBT." This feature is essential for ensuring strict adherence to allowed characters.

Furthermore, character lists can define a range of characters using a hyphen (`-`). This is particularly useful for matching all letters within a certain alphabetical range or all digits. Common ranges include for any uppercase letter, for any lowercase letter, or for any single digit. For instance, if you require a string to start with a capital letter followed by three lowercase letters, the pattern would be `"*"`.

To specify characters that must *not* be present, you can use the exclamation mark (`!`) as the first character inside the brackets. This creates a negative character list. For example, `"AC"` would match "ABC" or "A#C," but it would fail to match "A1C," effectively excluding any digits from that position. This level of precise exclusion makes the **Like** operator comparable in complexity and power to basic regular expressions for many common validation tasks in VBA.

Advanced Pattern Matching: Number Signs (#) and Special Characters

The final standard wildcard character available in the **Like** operator arsenal is the number sign (`#`). This character is specifically designed to match any single digit (0 through 9). It offers a concise alternative to using the character range when validating numerical patterns within a string. For example, if you need to verify an identification code that consists of three letters followed by exactly four digits, the pattern `"???####"` would be the most efficient representation.

Matching the wildcard characters themselves (`*`, `?`, `#`,) requires special handling. If you intend to search for a literal asterisk or question mark within a string, you must enclose the special character within square brackets. This process is known as escaping the character. For example, the pattern `""` matches a literal question mark, and `""` matches a literal asterisk. Similarly, if you want to match the literal square bracket characters, you must enclose them in their own set of brackets, such as `""` to match an opening bracket, or `"]"` to match a closing bracket.

This escape mechanism ensures that patterns containing otherwise reserved characters can be accurately processed. For example, matching a part number that is structured as `ABC*123` would

require the pattern `"ABC###"`. Understanding when and how to escape these reserved characters is vital for constructing robust and accurate pattern definitions, especially when processing structured data that includes delimiters or symbols commonly used as wildcards.

Practical Application 2: Checking for Prefixes and Suffixes

While the bidirectional wildcard search (e.g., `"*hot*"`) is useful for finding substrings, often we need to be more restrictive, perhaps checking only if a string begins or ends with a certain sequence. To check if each string in our dataset **starts with** "hot," we utilize a positional pattern matching technique by placing the asterisk only after the target prefix. This ensures that the string must begin immediately with the specified characters, followed by any number of other characters.

The pattern expression used for a prefix check is `"hot*"`. This instructs the **Like** operator to only return **True** if the string starts with 'h', followed by 'o', followed by 't', and then followed by zero or more characters. If we adapt our previous macro to incorporate this stricter prefix check, we dramatically alter the filtering results:

Sub CheckLike()

Dim i As Integer

For i = 2 To 10
If Range("A" & i) Like "hot*" Then
Range("B" & i) = "Starts with hot"
Else
Range("B" & i) = "Does not start with hot"
End If
Next i

End Sub

When this refined macro is executed against the same list of food items, the output clearly shows a reduction in matched items, as only those strings beginning with "hot" are flagged as matches. Items like "Chocolate" or "Chili dog hot," which contain "hot" but not at the beginning, are now correctly excluded.

	A	B	C	D
1	**Food**			
2	hot fries	Starts with hot		
3	pizza	Does not start with hot		
4	hot dog	Starts with hot		
5	ice cream	Does not start with hot		
6	lasagna	Does not start with hot		
7	pasta	Does not start with hot		
8	super hot wings	Does not start with hot		
9	cold yogurt	Does not start with hot		
10	cheese	Does not start with hot		
11				
12				
13				
14				
15				
16				

Conversely, to check for a specific suffix (i.e., if a string **ends with** a certain sequence), the asterisk must be placed at the beginning of the pattern. For example, `"*dog"` would match "Hot dog" and "Chili dog," but not "Dogfood." This precise control over positional matching is essential for tasks like categorizing files based on extensions or validating data integrity based on known terminal characters.

Choosing the Right Comparison Mode (Option Compare Text)

As previously mentioned, by default, VBA uses `Option Compare Binary` for all string comparisons, including those involving the **Like** operator. This means that string comparisons are case-sensitive. "APPLE" is not **Like** "apple" in binary mode, even with identical patterns. However, in many real-world data scenarios, particularly those involving user input or inconsistent legacy data, enforcing strict case sensitivity is counterproductive.

To perform case-insensitive comparisons, you must explicitly declare `Option Compare Text` at the very top of your module, before any procedures or declarations. When text comparison is enabled, the pattern matching process utilizes the system locale's collating sequence, treating 'A' and 'a' as identical characters. This is often the preferred mode when searching large datasets where capitalization is not guaranteed or relevant to the required match criteria.

It is important to understand that the comparison option must be set at the module level and affects

all string comparison operations within that module. If only a single comparison needs to be case-insensitive while the rest of the module requires binary comparison, the developer must employ creative workarounds, such as converting both the string and the pattern to a common case (e.g., using `UCase()` or `LCase()`) before executing the **Like** operation, effectively simulating a case-insensitive check without changing the module setting.

Conclusion: Enhancing Excel Automation with Like

The **Like** operator is far more than a simple search tool; it is a sophisticated method for establishing complex criteria based on the structure and content of strings. By mastering the available wildcard characters--the versatile asterisk (*), the single-character question mark (?), the numerical hash sign (#), and the powerful character list brackets ()--developers can create robust, efficient data processing macros.

Whether you are validating user-provided data formats, parsing complex log files, or simply filtering a large spreadsheet based on textual patterns, the **Like** operator provides the precision required for high-quality Excel automation. Its integration into conditional logic streamlines the process of data segmentation and retrieval, making your VBA code cleaner and significantly more maintainable.

For developers seeking to push the boundaries of their VBA projects, a deep understanding of pattern matching--especially how to effectively use escape characters and manage comparison modes--is indispensable. Utilizing these techniques ensures your macros can handle the variability of real-world data with accuracy and reliability.

Further VBA Learning Resources

The following tutorials explain how to perform other common tasks using VBA, building upon the foundational knowledge of string manipulation and conditional programming established here:

Explore techniques for manipulating cell data based on logical criteria.
Learn advanced looping structures to handle larger data ranges efficiently.
Review methods for handling errors gracefully during data processing.