How to Easily Compare Strings in VBA Using StrComp

Authored by

stats writer

November 19, 2025

Comparing strings is a fundamental operation in programming, and VBA (Visual Basic for Applications) provides robust tools for this purpose. The primary tool utilized for evaluating text values is the
StrComp function. This powerful function is designed specifically to analyze two textual inputs and determine their relative order or equality, making it indispensable for tasks ranging from data validation to complex sorting algorithms within Excel and other Office applications.

The core utility of the StrComp function lies in its ability to return a numeric value indicating the relationship between the two input strings. It accepts two required string parameters--the first string and the second string--along with an optional parameter that defines the type of comparison to be executed. Understanding the return values is critical for effective usage: a return value of 0 signifies that the two strings are exactly equal; a return value of -1 indicates that the first string is lexicographically "less than" the second string; and conversely, a return value of 1 shows that the first string is "greater than" the second. This numeric output allows developers to easily integrate string evaluations into conditional logic (e.g., If/Then statements) or loops.

For instance, if we execute the simple comparison StrComp("Apple", "Banana"), the function evaluates the alphabetical order. Since "Apple" comes before "Banana," the function returns the value -1. This behavior aligns with standard ASCII or Unicode sorting rules applied by VBA. However, it is essential to remember that without specifying a comparison type, VBA defaults to a binary, or case-sensitive comparison, meaning that "apple" and "Apple" would not be considered equal, leading to a non-zero return value.

Understanding the StrComp Function Parameters and Returns

The full syntax of the StrComp function is `StrComp(string1, string2, )`. The first two arguments, `string1` and `string2`, are mandatory and represent the text values being evaluated. The optional argument is what determines the method of evaluation--specifically, whether the comparison should be case-sensitive (binary) or case-insensitive (textual). If this argument is omitted, VBA uses the default comparison method established by the `Option Compare` setting defined at the module level, which typically defaults to binary comparison if not explicitly set.

The return values are meticulously defined to cover all possible relationships between the two input strings. If `string1` is lexicographically identical to `string2`, the result is 0. If `string1` precedes `string2` alphabetically or numerically (based on character codes), the result is -1. Conversely, if `string1` follows `string2`, the result is 1. Furthermore, if either input string is `Null`, the function returns `Null`, requiring robust error handling in production code. Understanding these precise outputs allows developers to handle sorting, filtering, and exact matching requirements with accuracy.

For operations involving complex datasets within VBA, utilizing StrComp is generally more performant and explicit than relying solely on standard comparison operators like `=` or `<>`, especially when case sensitivity must be precisely controlled. When comparing strings directly using the equals operator (=), VBA defaults to the module's `Option Compare` setting, which can sometimes lead to unexpected results if the setting is changed or misunderstood. StrComp bypasses this ambiguity by allowing the comparison mode to be explicitly passed in the function call, ensuring consistent behavior regardless of module settings.

The Mechanics of Case-Sensitive String Comparison (Method 1)

When performing a case-sensitive comparison, VBA executes a binary comparison. This means that the function compares the actual underlying binary values (ASCII or Unicode character codes) of each character in the strings sequentially. Because uppercase letters and lowercase letters have different character codes (e.g., 'A' has a different code than 'a'), they are treated as entirely distinct characters. This method is often required when precision is paramount, such as comparing passwords, file names on certain operating systems, or unique identifiers where case integrity must be maintained.

To explicitly force a case-sensitive comparison using the StrComp function, you can either omit the optional argument (relying on the default binary setting) or, for maximum clarity, use the constant `vbBinaryCompare` as the third argument. In this mode, the strings must match exactly, including the capitalization of every single character, to return 0 (equal). If even one character's case differs, the strings are considered unequal, and the function will return 1 or -1 based on the character code difference detected first.

The following example demonstrates how to implement a case-sensitive check across a range of cells in Excel using VBA. This macro iterates through rows 2 to 10 and compares the strings found in Column A and Column B. The result of the comparison is then translated into a boolean output (TRUE or FALSE) placed in Column C, indicating whether the strings are an exact match, case included.

The following method illustrates the implementation of a **case-sensitive string comparison** across a cell range:

```
Sub CompareStrings()
Dim i As Integer

For i = 2 To 10
Range("C" & i) = StrComp(Range("A" & i), Range("B" & i)) = 0
Next i
End Sub
```

This macro executes a **case-sensitive string comparison** between the corresponding strings located in the ranges **A2:A10** and **B2:B10**. The output in the range **C2:C10** will be either **TRUE** or **FALSE**, explicitly indicating whether the strings are identical in content and capitalization. Since no comparison method is specified in the `StrComp` call, the default `vbBinaryCompare` (case-sensitive) mode is automatically used.

Implementing Case-Insensitive String Comparison (Method 2)

In many practical scenarios, such as searching user input or validating general data entries, comparing strings without regard to capitalization is necessary. This is known as a case-insensitive, or textual, comparison. When performing a textual comparison, VBA internally converts both strings to a common case (usually uppercase or lowercase) before executing the comparison of character codes. This ensures that "Data" is treated identically to "data" or "DATA."

To enable this textual comparison mode, we must explicitly pass the constant vbTextCompare as the third argument to the StrComp function. This constant overrides the module's default comparison setting and guarantees that the check is case-insensitive. Using vbTextCompare is crucial when dealing with user-generated data where inconsistent capitalization is common, ensuring robust and flexible matching logic.

The following example demonstrates the minor but critical modification required in the VBA code to switch from binary comparison to textual comparison. By adding the `vbTextCompare` argument, we instruct the function to ignore case differences while evaluating equality, thus widening the scope of what constitutes a match.

The following method details the implementation of a **case-insensitive string comparison**:

```
Sub CompareStrings()
Dim i As Integer

For i = 2 To 10
Range("C" & i) = StrComp(Range("A" & i), Range("B" & i), vbTextCompare) = 0
Next i
End Sub
```

This modified macro performs a **case-insensitive string comparison** across the same range. The addition of vbTextCompare ensures that if the strings in columns A and B are identical in spelling but differ only in capitalization, they will still be evaluated as equal, resulting in a **TRUE** output in Column C.

Visualizing Comparison Results in Practice

To fully grasp the difference between case-sensitive and case-insensitive comparisons, it is beneficial to analyze a concrete dataset. Consider the following example data setup in an Excel worksheet, where strings in Column A are compared against corresponding strings in Column B. Notice how several pairs are identical except for capitalization, while others have substantive differences, providing a perfect test scenario for our VBA macros.

The differences between the columns highlight the necessity of choosing the correct comparison method. For instance, Row 4 contains "Apple" and "apple". A binary comparison will fail this pair, while a textual comparison will succeed. Similarly, Row 6 ("Banana" vs. "BANANA") tests the handling of all-caps variations. This preparation allows us to clearly observe the impact of the comparison type parameter on the final results returned to Column C.

This table serves as our baseline data for the following examples:

	A	B	C	D	E
1	**String 1**	**String 2**			
2	Duck	Duck			
3	rooster	Rooster			
4	Turtle	Turtle			
5	elephant	elephant			
6	pig	PIG			
7	horse	Horse			
8	COW	cow			
9	Ant	Ant			
10	Chicken	Human			
11					
12					
13					
14					
15					
16					
17					
18					

Example 1: Case-Sensitive String Comparison in VBA

We begin by executing the macro designed for **case-sensitive** evaluation. This involves using the

StrComp function without the optional comparison argument, defaulting to the stringent binary comparison mode. The objective is to determine absolute equality, where both content and capitalization must be identical for a match to occur. This mode is the most restrictive and is essential when precision is the highest priority.

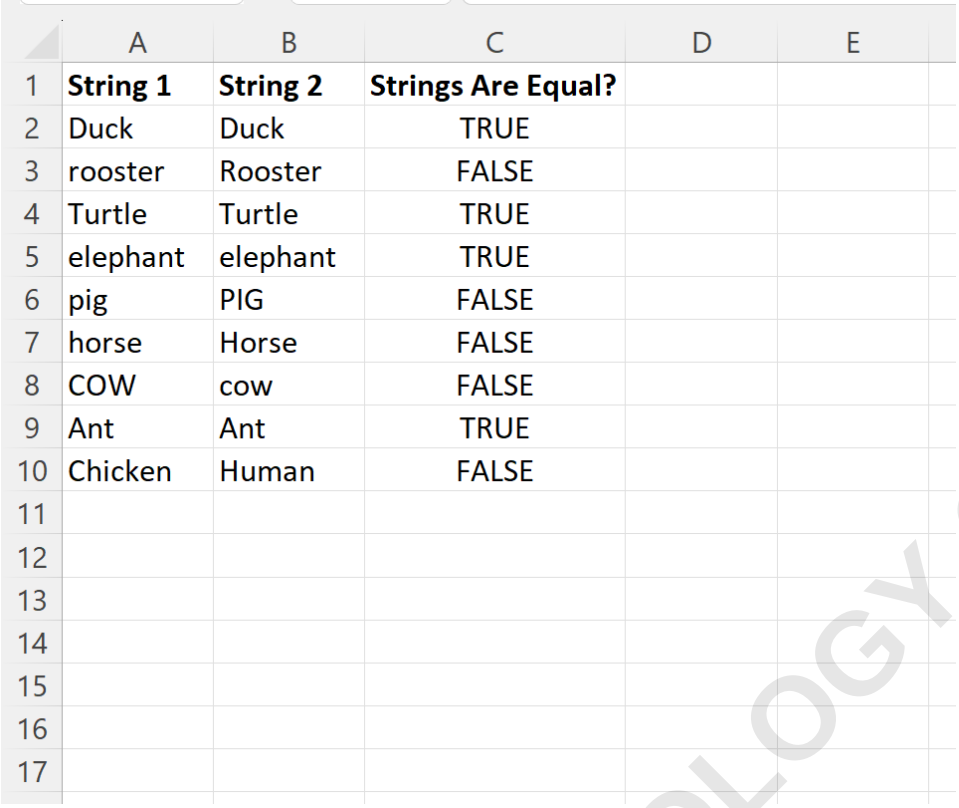
We utilize the following code block, which iterates from row 2 to 10 and assigns the result of the equality check (`StrComp(...) = 0`) to the corresponding cell in column C:

```
Sub CompareStrings()
Dim i As Integer

For i = 2 To 10
Range("C" & i) = StrComp(Range("A" & i), Range("B" & i)) = 0
Next i
End Sub
```

Upon execution of this macro against the sample data, the output clearly demonstrates where the case-sensitive requirement causes discrepancies. Notice that rows where capitalization differed, such as Row 4 ("Apple" vs. "apple") and Row 6 ("Banana" vs. "BANANA"), now return **FALSE**, even though the words themselves are spelled the same. Only rows where the strings match byte-for-byte, including case, return **TRUE**.

When we run this macro, we receive the following output illustrating the results of the binary comparison:



	A	B	C	D	E
1	**String 1**	**String 2**	**Strings Are Equal?**		
2	Duck	Duck	TRUE		
3	rooster	Rooster	FALSE		
4	Turtle	Turtle	TRUE		
5	elephant	elephant	TRUE		
6	pig	PIG	FALSE		
7	horse	Horse	FALSE		
8	COW	cow	FALSE		
9	Ant	Ant	TRUE		
10	Chicken	Human	FALSE		
11					
12					
13					
14					
15					
16					
17					

Column C returns **TRUE** only if the strings are equal and have the **exact same case**.

Otherwise, column C returns **FALSE**, indicating that either the content or the capitalization (or both) of the corresponding strings differed.

Example 2: Case-Insensitive String Comparison in VBA

In contrast to the previous example, we now apply the **case-insensitive** comparison using the vbTextCompare constant. This textual comparison method is significantly more lenient regarding capitalization, focusing solely on the sequence and identity of the characters themselves, regardless of whether they are upper or lower case. This is ideal for scenarios where the underlying meaning of the text is what matters most, rather than its formatting.

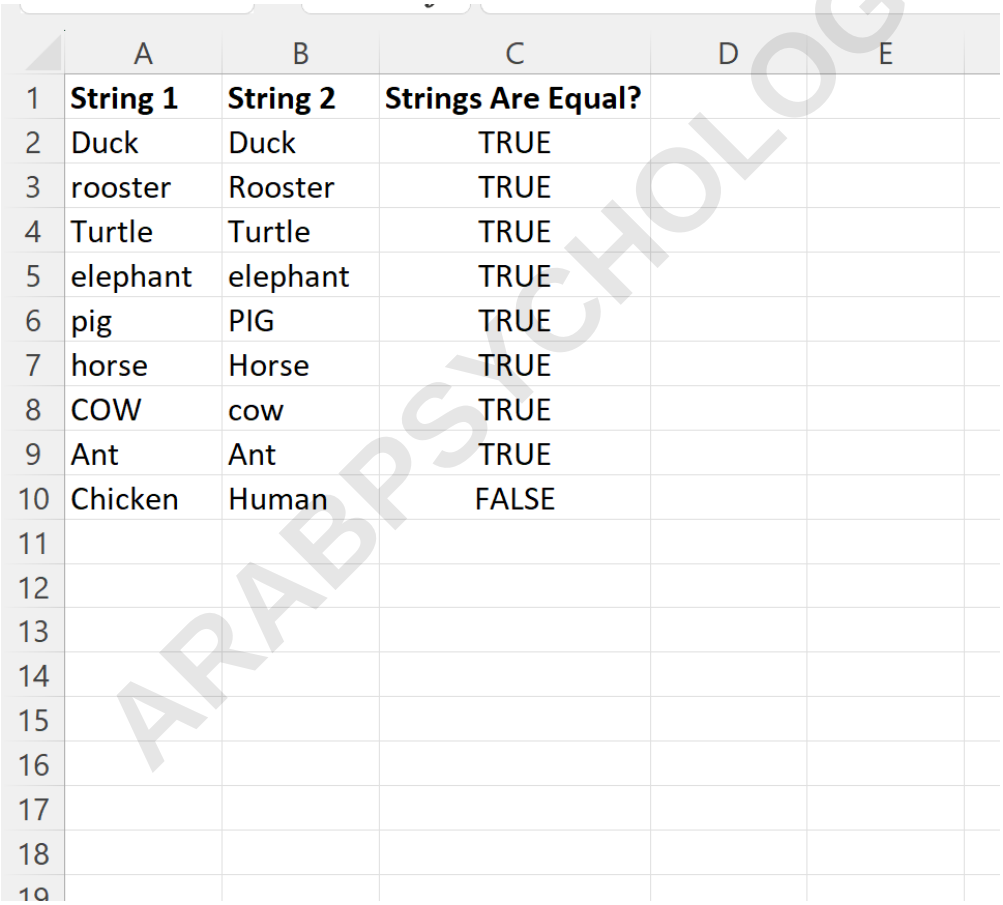
The macro implementation remains almost identical, with the crucial addition of the comparison constant as the third argument to the StrComp function. This addition is the directive that tells VBA to normalize the case internally before comparing the character codes, thereby treating 'a' and 'A' as equivalents.

```
Sub CompareStrings()
Dim i As Integer
```

```
For i = 2 To 10
Range("C" & i) = StrComp(Range("A" & i), Range("B" & i), vbTextCompare) = 0
Next i
End Sub
```

Running this revised macro yields a different outcome, particularly for the rows that previously failed due to case mismatch. Rows 4 and 6, which resulted in **FALSE** during the case-sensitive test, now correctly return **TRUE** because their textual content is identical. Only the rows where the actual spelling or the number of characters differed, such as Row 9 ("Grape" vs. "Grapes"), still return **FALSE**.

When we run this macro, we receive the following output, demonstrating the flexibility of textual comparison:



	A	B	C	D	E
1	**String 1**	**String 2**	**Strings Are Equal?**		
2	Duck	Duck	TRUE		
3	rooster	Rooster	TRUE		
4	Turtle	Turtle	TRUE		
5	elephant	elephant	TRUE		
6	pig	PIG	TRUE		
7	horse	Horse	TRUE		
8	COW	cow	TRUE		
9	Ant	Ant	TRUE		
10	Chicken	Human	FALSE		
11					
12					
13					
14					
15					
16					
17					
18					
19					

Column C returns **TRUE** if the strings are equal, **regardless of the case**.

Column C returns **FALSE** only if the underlying textual content of the strings is not equal, meaning there is a difference in spelling, characters, or length.

Alternative String Comparison Techniques in VBA

While the StrComp function is the dedicated tool for complex string comparisons and determining alphabetical order, VBA offers other methods, particularly for simple equality checks, which developers should be aware of. The most common alternative is the direct use of the equality operator (=). When using `string1 = string2`, the result is a Boolean `TRUE` or `FALSE`. However, the case sensitivity of this operation is dictated by the `Option Compare` statement at the top of the module. If `Option Compare Binary` is set (the usual default), it performs a case-sensitive check; if `Option Compare Text` is set, it performs a case-insensitive check.

Forcing case-insensitive comparison without modifying the module-level setting or using vbTextCompare can be achieved by converting both strings to a uniform case before comparison. This typically involves using the built-in functions `UCase()` or `LCase()`. For example, `If UCase(string1) = UCase(string2) Then...` guarantees a case-insensitive check, regardless of the `Option Compare` setting. While effective, this method requires function calls on both sides of the equality operator, which can be slightly less efficient than using StrComp with the comparison mode defined.

Developers often prefer `StrComp` because it provides more information than just a Boolean true/false result. Since it returns -1, 0, or 1, it allows for sophisticated sorting and ordering logic within a single function call, eliminating the need for separate checks like `If string1 > string2 Then...`. Moreover, `StrComp` is standardized across all versions of VBA and provides explicit control over the comparison type, making the code more portable and easier to debug than relying on implicit module settings.

Summary of Comparison Modes and Best Practices

Choosing the correct string comparison method is vital for data integrity and accurate application logic in VBA. The decision hinges entirely on the nature of the data being evaluated and whether capitalization holds inherent meaning. If you are dealing with identifiers, passwords, or data where 'A' must be distinct from 'a', the case-sensitive `vbBinaryCompare` mode (or default `StrComp` usage) is mandatory. If you are dealing with general text, names, or user input where case differences are irrelevant, the textual `vbTextCompare` mode offers the necessary flexibility.

To summarize the key takeaways regarding string comparison in VBA:

Always use the `StrComp function` when you need to determine the lexicographical order (i.e., whether one string is greater than, less than, or equal to another).
Use `vbBinaryCompare` (the default) for strict, case-sensitive checks where capitalization matters.
Use `vbTextCompare` for flexible, case-insensitive checks of textual content.
For simple equality checks where you only need a `TRUE` or `FALSE` result, you can append `= 0` to

the `StrComp` function call, as demonstrated in the examples, to quickly convert the numeric result into a Boolean value.

Note: You can find the complete documentation for the StrComp function in VBA on the official Microsoft Developer Network (MSDN) website, which provides further technical details on its parameters and error handling.