How to Easily Remove Non-Printable Characters from a String in VBA

Authored by

stats writer

November 20, 2025

The ability to manipulate textual data is fundamental in applications developed using VBA (Visual Basic for Applications). When importing data from external sources, developers frequently encounter challenges related to hidden formatting or unnecessary control codes, often referred to as non-printable characters. These invisible elements can disrupt calculations, comparisons, and display formatting, necessitating a robust mechanism for their removal or replacement. Fortunately, VBA provides the powerful built-in Replace function, which is the primary tool for executing sophisticated string sanitation and transformation tasks within the environment, whether working in Microsoft Excel, Access, or other supported Office applications.

The Replace function is exceptionally versatile, designed not only for simple substitution but also for complex removal operations. By leveraging its structure, we can target specific elements within a given text string--including those non-printable characters--and substitute them with an empty string, effectively deleting them without leaving any trace. This technique relies on knowing the specific ASCII character code associated with the unwanted control character, which is then fed into the "find" parameter of the function. Setting the "replace" parameter to an empty string (`""`) ensures that the unwanted character is removed completely, leading to clean, standardized data ready for further processing or analysis.

Understanding the VBA Replace Function Syntax

The Replace function is a core component of VBA's text processing capabilities. Unlike simple string concatenation or trimming, Replace allows for targeted, large-scale substitution based on defined patterns. Mastering its syntax is critical for efficient data manipulation, providing granular control over how and where replacements occur within a target string. This function is designed to search for a specified substring and replace all or a defined number of its occurrences with a new substring, returning the modified string as its result.

The complete method utilizes several parameters, some mandatory and others optional, allowing for diverse applications ranging from simple text updates to complex, positional filtering. The use of optional parameters greatly enhances its flexibility, enabling developers to define the starting point for the search and even limit the total number of replacements performed. Furthermore, while the function is case-sensitive by default, combining it with other intrinsic VBA functions can easily override this behavior, granting truly case-insensitive searching capabilities, which is often necessary when cleaning user-entered or externally sourced data.

The basic syntax for the Replace function is as follows, incorporating both required and optional arguments:

Replace(expression, find, replace, start, count, compare)

Each parameter serves a distinct purpose within the function's execution pipeline:

expression: This is the mandatory string expression within which you wish to perform the replacement operation. It represents the source data that will be searched and modified.
find: This is the mandatory string that the function searches for within the main expression. This could be a word, a phrase, or, crucially for sanitation, a specific ASCII control character.
replace: This is the mandatory string used to substitute any instances of the find parameter. To achieve character removal, this parameter is set to an empty string (`""`).
start (*optional*): An optional parameter specifying the character position in the expression where the search should begin. If omitted, the search starts at position 1. This is useful for optimizing performance when dealing with very long strings.
count (*optional*): An optional parameter defining the maximum number of substitutions to be made. If omitted, all occurrences of the find parameter are replaced.

Removing Non-Printable Characters from a String

The primary use case addressed by the title of this post is the removal of non-printable characters, such as tabs, line breaks, or other control codes (like carriage return/line feed), that often contaminate data extracted from text files or web sources. Since these characters cannot be typed directly into the `find` argument, we must use the VBA function Chr(), which converts a numerical ASCII value into its corresponding character representation. By iteratively replacing these specific control characters with an empty string, we can completely sanitize the input data.

Common non-printable characters and their corresponding ASCII codes that typically require removal include: the horizontal tab (Chr(9)), the line feed (Chr(10)), the carriage return (Chr(13)), and sometimes the non-breaking space (Chr(160)). A robust cleaning function would chain multiple Replace operations together, ensuring that the input string is scrubbed of all known contaminants. For example, to remove both carriage returns and line feeds (which together form a common newline sequence), you would nest two Replace calls or execute them sequentially against the target string.

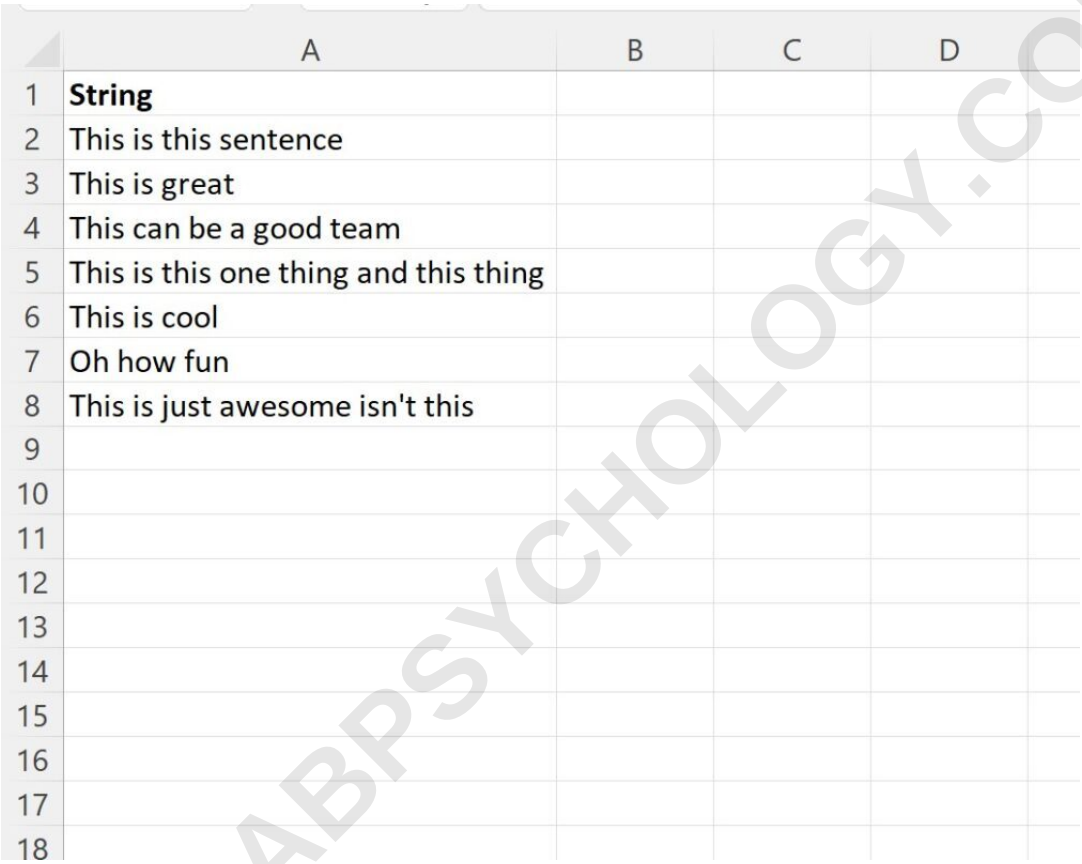
Although the original example code focuses on standard string substitution, the principle for removing non-printable characters remains the same: identify the unwanted character (via its ASCII code) and replace it with nothing. This process ensures data integrity, especially when data is destined for databases or further algorithmic manipulation where hidden control characters could lead to parsing errors or unexpected formatting issues. A function designed for this purpose would serve as a critical preprocessing step for virtually any imported textual data within Microsoft Excel.

Example Data Setup for Demonstrations

To effectively illustrate the capabilities of the Replace function, we will utilize a sample dataset within a spreadsheet environment. The following image displays a list of strings housed in Column

A of an Microsoft Excel worksheet. These strings contain various instances of the word "this," appearing in different cases. We will use VBA macros to process this data iteratively, placing the results of the replacement operation into Column B for direct comparison.

This setup allows us to clearly demonstrate the subtle yet crucial differences between case-sensitive and case-insensitive replacements, as well as the functionality of limiting replacements using the optional arguments. Pay close attention to how the variation in capitalization of the target word affects the output in the subsequent examples.



	A	B	C	D
1	**String**			
2	This is this sentence			
3	This is great			
4	This can be a good team			
5	This is this one thing and this thing			
6	This is cool			
7	Oh how fun			
8	This is just awesome isn't this			
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				

In all forthcoming examples, we utilize a simple iteration loop (`For i = 2 To 8`) to process rows 2 through 8, applying the logic of the Replace function to the content found in Column A and outputting the result into the corresponding cell in Column B. This structure is typical for batch processing tasks in VBA.

Example 1: Performing Case-Sensitive Character Replacement

The default behavior of the Replace function in VBA is to execute a case-sensitive search. This means that if you specify "this" as the item to find, the function will strictly only identify and replace instances that match this exact capitalization. It will ignore "This," "THIS," or any other variation. This precise matching capability is essential when data integrity depends on preserving original

capitalization or when differentiating between identical words that serve different grammatical or technical roles.

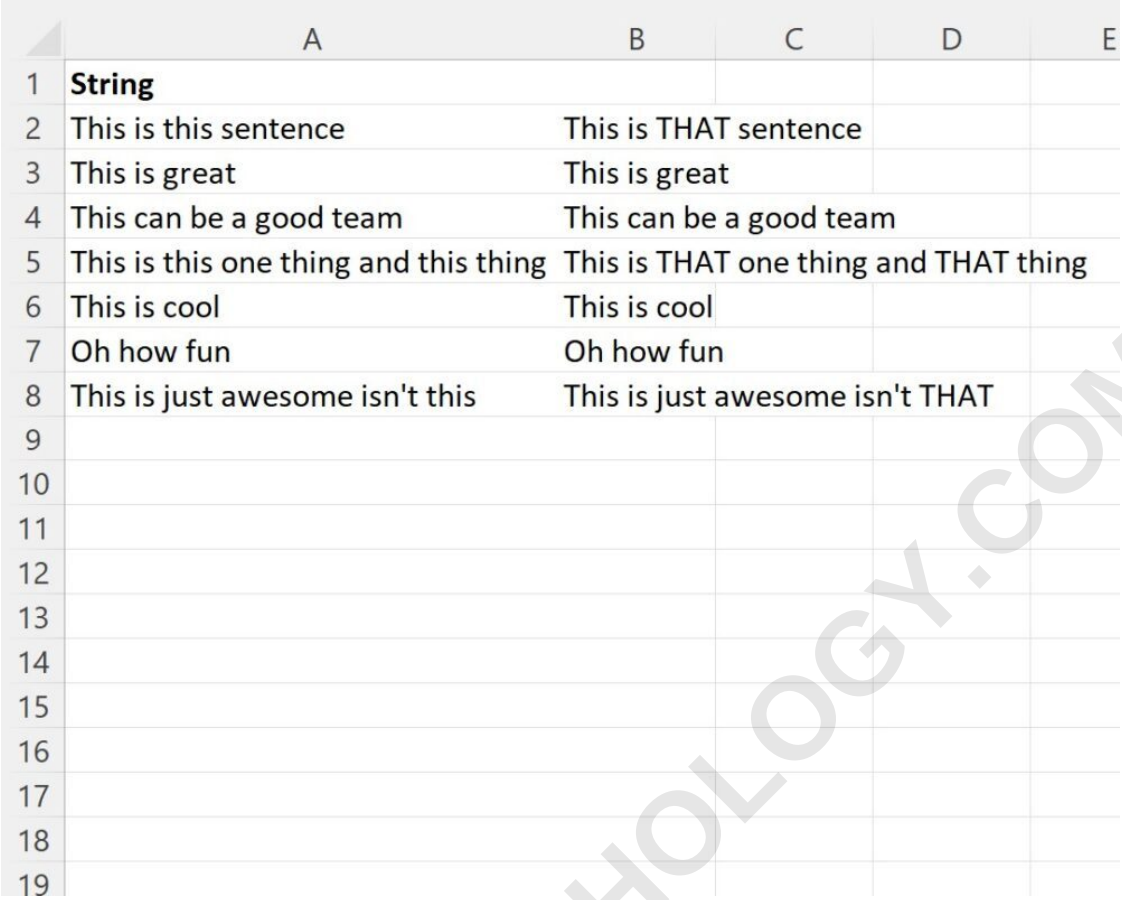
In this initial example, we aim to replace the lowercase sequence "this" with the uppercase sequence "THAT" across all defined strings in Column A. Because the replacement is case-sensitive, any instances where "this" starts a sentence (e.g., "This is a test") will remain unaltered, as the function only looks for the exact case specified in the `find` argument. This behavior is demonstrated clearly in the output, where only exact matches are successfully substituted.

The following macro efficiently loops through the specified Range and applies the strict, case-sensitive replacement:

```
Sub ReplaceChar()
Dim i As Integer

For i = 2 To 8
Range("B" & i) = Replace(Range("A" & i), "this", "THAT")
Next i
End Sub
```

Upon execution of this macro, observe the resulting data transformation:



	A	B	C	D	E
1	**String**				
2	This is this sentence	This is THAT sentence			
3	This is great	This is great			
4	This can be a good team	This can be a good team			
5	This is this one thing and this thing	This is THAT one thing and THAT thing			
6	This is cool	This is cool			
7	Oh how fun	Oh how fun			
8	This is just awesome isn't this	This is just awesome isn't THAT			
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					

The output in Column B confirms that only the lowercase occurrences of "this" were successfully replaced by "THAT." Any instance starting with a capital letter, such as "This," remained untouched, vividly demonstrating the strict nature of the default case-sensitive matching provided by the Replace function.

Example 2: Implementing Case-Insensitive Character Replacement

In many real-world data cleaning or transformation scenarios, strict case-sensitivity is undesirable. When standardizing data, users often need to replace all variants of a word, regardless of whether they are capitalized, lowercase, or mixed-case. To achieve this case-insensitive behavior in VBA using the Replace function, we must preprocess the target string before searching. This is achieved by nesting the Replace function within the LCase function.

The LCase function temporarily converts the entire input string to lowercase before it is passed to the Replace function. By performing the search and replacement on the standardized lowercase version, the replacement operation becomes inherently case-insensitive. It is crucial to note that while the search happens on the lowercase version, the substitution itself affects the original string's position, effectively replacing whatever capitalization existed there with the `replace`

parameter (in this case, "THAT").

We modify the previous macro by wrapping the expression argument (`Range("A" & i)`) within the LCase function. We continue to search for the lowercase "this" and replace it with "THAT":

```
Sub ReplaceChar()
Dim i As Integer

For i = 2 To 8
Range("B" & i) = Replace(LCase(Range("A" & i)), "this", "THAT")
Next i
End Sub
```

Running this revised macro yields the following output, demonstrating that all capitalized and non-capitalized occurrences have been targeted:

	A	B	C	D	E
1	**String**				
2	This is this sentence	THAT is THAT sentence			
3	This is great	THAT is great			
4	This can be a good team	THAT can be a good team			
5	This is this one thing and this thing	THAT is THAT one thing and THAT thing			
6	This is cool	THAT is cool			
7	Oh how fun	oh how fun			
8	This is just awesome isn't this	THAT is just awesome isn't THAT			
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					

As evident in Column B, every instance of "this" (including "This") has been successfully replaced by "THAT." This outcome confirms the successful implementation of the case-insensitive search by utilizing the LCase function to standardize the case of the input string before the Replace operation

is executed.

Example 3: Limiting Replacements to the First N Occurrences

While global replacement is the standard, scenarios often require replacing only the initial instances of a substring, leaving subsequent occurrences untouched. This is where the optional `Count` argument of the Replace function becomes invaluable. By specifying a numerical value for `Count`, we instruct VBA to stop searching and replacing once that limit is reached. If the `Count` argument is omitted, the function defaults to replacing all occurrences.

For this example, we aim to replace only the first occurrence of "this" (regardless of case) with "THAT" in each string. We will maintain the case-insensitive functionality achieved in Example 2 by retaining the LCase function wrapper, and additionally, we introduce the `Count:=1` parameter to restrict the replacement to the very first match found. This approach allows precise control over data modification, particularly useful in structured text where initial occurrences might hold specific significance.

We incorporate the optional `Count` argument into the macro:

```
Sub ReplaceChar()
Dim i As Integer

For i = 2 To 8
Range("B" & i) = Replace(LCase(Range("A" & i)), "this", "THAT", Count:=1)
Next i
End Sub
```

Executing this code demonstrates how the `Count:=1` argument successfully limits the modification:

	A	B	C	D	E
1	**String**				
2	This is this sentence	THAT is this sentence			
3	This is great	THAT is great			
4	This can be a good team	THAT can be a good team			
5	This is this one thing and this thing	THAT is this one thing and this thing			
6	This is cool	THAT is cool			
7	Oh how fun	oh how fun			
8	This is just awesome isn't this	THAT is just awesome isn't this			
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					

Column B now clearly shows that only the first occurrence of the target string (whether "this" or "This") has been replaced by "THAT," while any subsequent occurrences within the same cell remain unaltered. It is important to remember that by adjusting the value assigned to `Count` (e.g., `Count:=2`), you can similarly limit the replacement to the first two, three, or any desired number of matches, providing fine-grained control over complex text manipulation tasks in VBA.

Conclusion and Further VBA Resources

The Replace function in VBA is an essential tool for any developer working with textual data in Microsoft Excel. Whether the goal is to sanitize data by removing specific non-printable characters using the ASCII codes or to perform targeted substitutions across a dataset, understanding the nuances of its optional arguments--like `start`, `count`, and combining it with functions like LCase for case-insensitivity--is paramount. Effective string manipulation leads directly to cleaner data, more reliable calculations, and enhanced overall application performance.

For advanced scenarios involving more complex pattern matching (such as finding strings that fit a specific structure rather than an exact sequence), developers might explore using VBA's Regular Expression capabilities, though the native Replace function remains the fastest and simplest option for direct substitution and cleanup tasks illustrated in these examples. Always consult the official Replace documentation for the most comprehensive details regarding its behavior and

performance characteristics.

Reviewing these examples provides a strong foundation for handling virtually all common text cleaning requirements within the VBA environment. Utilizing the techniques demonstrated here will ensure your data remains accurate and consistent across all applications.

The following tutorials explain how to perform other common tasks using VBA: