How to Easily Return a Value from a VBA Function

Authored by

stats writer

November 19, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Return a Value from a VBA Function*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=97196

ARABPSYCHOLOGY.COM

One of the most essential concepts in mastering VBA (Visual Basic for Applications) programming is understanding how to correctly pass data back to the calling procedure or worksheet cell. This process, known as returning a value, is what differentiates a custom function from a standard Sub procedure. Unlike a Sub procedure, a function is specifically designed to compute a result and return a value.

While some older tutorials or specific dialects of Visual Basic might suggest using the explicit `Return` statement, the standard and most reliable methodology within Microsoft Excel VBA is to assign the result directly to the name of the function itself. This elegant approach ensures clarity and compatibility across various Excel environments. By assigning the final computation to the function's identifier, you are explicitly defining the output that will be passed back to the environment that invoked the code.

It is paramount for any developer to grasp this core principle: the value assigned to the function's name just before the `End Function` declaration is the value that the function returns. This mechanism provides immense flexibility, allowing the function to return various data types, from simple numerics like `Integer` or `Double` to complex objects like the Range object, which we will explore in detail later in this guide.

The Core Principle: Assigning the Output to the Function Name

The fundamental method for a VBA function to successfully return a value involves equating the function's name with the calculated result within the function body. This approach avoids the ambiguity associated with the deprecated or context-specific `Return` keyword and is universally accepted in Microsoft Excel environments. When the function hits the `End Function` line, it automatically checks the internal state of the variable sharing its name and returns that value.

Consider a scenario where we need a simple VBA utility to perform division. To ensure the result is returned, we must assign the computed division to the function's identifier, `DivideValues`. This process is straightforward and creates highly readable code, which is a key trait of professional VBA development. The result of `x / y` is temporarily stored in the function's namesake variable before the function concludes its execution.

The syntax below illustrates this standard assignment method. Notice how the line `DivideValues = x / y` is the pivotal step that defines the function's output. If this assignment step is omitted, the function will typically return the default value for its declared data type (e.g., zero for numeric types or an empty string for text types), leading to logical errors in the calling code. Therefore, ensuring this assignment is correctly executed is the primary responsibility of the function writer.

For example, we can create the following function to divide two values and then return the result of the division:

Function DivideValues(x, y)
DivideValues = x / y
End Function

The name of this function is **DivideValues**, so to return a value from this function we must assign the result of **x / y** to a variable with the same name of **DivideValues**.

Integrating Conditional Logic for Robust Functions

Real-world functions rarely consist of a single, simple calculation. Often, they must handle different scenarios, validate inputs, or branch their execution based on certain conditions. This is where conditional structures, particularly If Else logic, become indispensable. When incorporating conditional logic, the function's ability to return a value must be carefully managed across all possible execution paths.

The assignment rule remains constant: regardless of which branch of the If Else statement is executed, the function's name must be assigned the desired output for that specific case. This allows a single function to return vastly different results or even error messages, depending on the validity of the input data. For example, in a division function, the most critical validation is preventing division by zero, which results in an error. By using If Else logic, we can catch this error proactively and return a meaningful textual error instead of triggering a runtime exception.

Handling errors gracefully is a hallmark of high-quality code. When an invalid operation is detected (such as dividing by zero), the VBA function can be programmed to return a descriptive string (e.g., "Cannot divide by zero") instead of a numerical result. This flexibility means that the function name will be assigned different data types depending on the conditions met--a string in the error case and a number otherwise. VBA manages these internal data type changes seamlessly, provided the final calling context (e.g., the Excel cell) can interpret the returned result.

If your function involves **If Else** logic, you can assign the value to the function name multiple times.

For example, you can create the following function that returns "Cannot divide by zero" if you attempt to divide by zero or else simply return the result of the division:

Function DivideValues(x, y)
If y = 0 Then
DivideValues = "Cannot divide by zero"
Else
DivideValues = x / y
End If
End Function

ARABPSYCHOLOGY.COM

Practical Example: Implementing the Robust Division Function

To demonstrate the utility of these concepts, let us walk through a practical example of creating and using the robust `DivideValues` function directly within an Excel environment. This function will read values from two specific cells, perform the calculation, and write the result back to the cell where the VBA function is called. We will utilize the function both with valid inputs and with the zero-division scenario to observe the effectiveness of the If Else conditional logic.

Suppose our spreadsheet contains the dividend (50) in cell **A2** and the divisor (10) in cell **B2**. Our goal is to calculate A2 divided by B2 in cell C2. We first define our simple, functional version of `DivideValues` in a standard VBA module. This basic function assumes valid inputs and focuses purely on the arithmetic operation, assigning the outcome directly to the function name.

When the user enters `=DivideValues(A2, B2)` into cell C2, Excel calls the custom function, passing 50 and 10 as arguments `x` and `y`, respectively. The function executes the division, assigns the result (5) to `DivideValues`, and then exits. This assigned value is then seamlessly returned to and displayed in cell C2, completing the cycle of a successful User Defined Function (UDF) execution in Excel.

Suppose we would like to create a function in VBA to divide the value in cell **A2** by the value in cell **B2**:

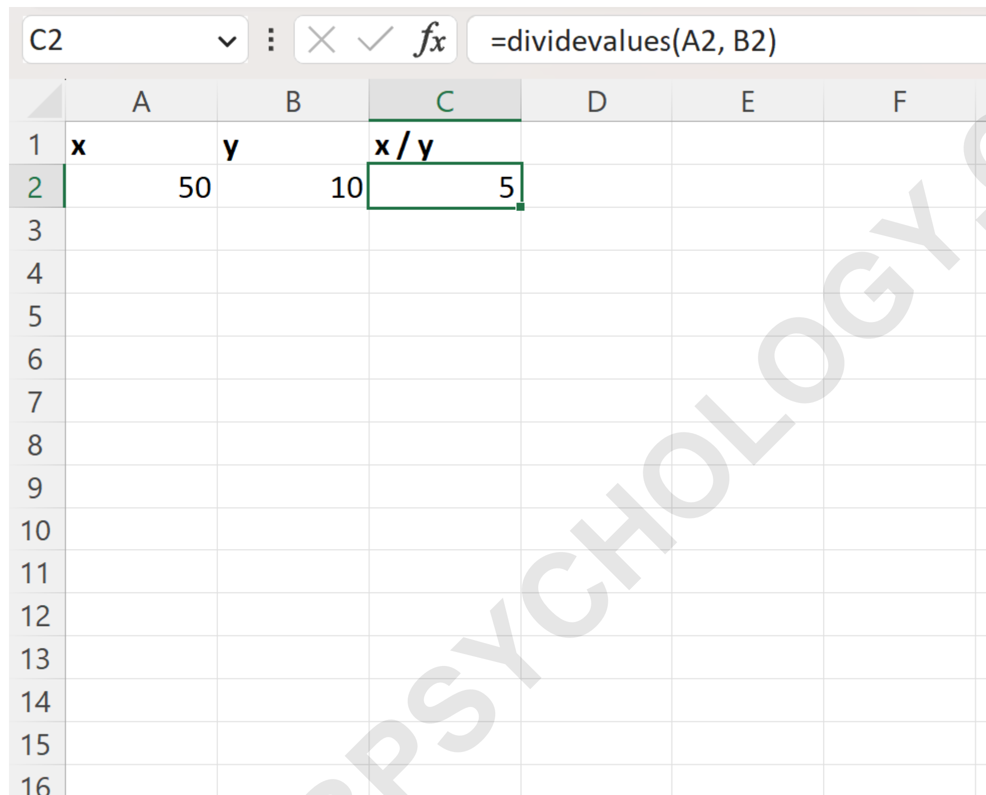
	A	B	C	D	E	F
1	**x**	**y**				
2	50	10				
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						

ARABPSYCHOLOGY.COM

We can create the following function to do so:

Function DivideValues(x, y)
DivideValues = x / y
End Function

When we run this macro, we receive the following output:



The function returns a value of **5**, which is the result of 50 divided by 10.

Handling Division by Zero Errors Effectively

The previous example works perfectly for valid numerical inputs. However, if the divisor (y) is zero, the basic function will fail, resulting in a `#DIV/0!` error in the spreadsheet or a runtime error in VBA. To create truly robust and user-friendly code, we must implement error-checking using the previously defined If Else structure. This allows us to intercept the potentially erroneous operation and provide a clear, non-crashing output.

By implementing the conditional check `If y = 0 Then`, we establish a safety net. If this condition is met, the function immediately assigns the error message string "Cannot divide by zero" to the function name `DivideValues`. Once the assignment is made, the function execution can proceed

ARABPSYCHOLOGY.COM

to `End If` and then `End Function`, ensuring that the descriptive error message is returned instead of an unhandled runtime exception. This is a critical practice for maintaining data integrity and improving the user experience in Excel.

When the input for cell **B2** is changed to 0, the robust version of the function demonstrates its value. It adheres to the logic, assigns the string literal, and this string is what is subsequently displayed in the calling cell. This mechanism provides clear feedback to the end-user without halting the macro or spreadsheet calculation process. This ability to return different data types (numerical result or descriptive string) based on internal logic highlights the power and flexibility of VBA functions when paired with appropriate conditional structures.

We could also create a function that uses **If Else** logic to first check if the value that we're dividing by is not equal to zero:

Function DivideValues(x, y)
If y = 0 Then
DivideValues = "Cannot divide by zero"
Else
DivideValues = x / y
End If
End Function

If we change the value in cell **B2** and then use this function to perform division, we'll receive the following output:

ARABPSYCHOLOGY.COM

C2 =dividevalues(A2, B2)

	A	B	C	D	E	F
1	x	y	x / y			
2	50	0	Cannot divide by zero			
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						

ARABPSYCHOLOGY.COM

Since we attempted to divide by zero, "Cannot divide by zero" is returned by the function.

Advanced Returns: Returning Range Objects

While most functions return a value of a simple data type (like `String`, `Integer`, or `Double`), VBA functions can also return complex objects, notably the Range object. Returning a Range is useful when a function needs to identify, process, or select a specific set of cells based on its input parameters, often used within larger macro procedures rather than directly in an Excel cell formula.

When working with objects in VBA, a crucial distinction must be made: objects must be assigned using the `Set` keyword. This rule applies equally when setting an internal object variable and when assigning the final object result to the function name for return. If the `Set` keyword is omitted when assigning a Range object to the function name, VBA will attempt to return the default property of the object (often its value), which is typically not the intended behavior when working with object references.

A function designed to return a Range object must declare its return type explicitly as `As Range` in the function signature. Inside the function, after processing inputs (e.g., performing calculations or determining cell addresses), the final Range object must be assigned to the function name using `Set FunctionName = RangeObject`. This allows the calling procedure (e.g., a Sub routine) to

receive a direct reference to the specified cells, which it can then manipulate further, such as changing formatting or extracting values.

To illustrate how a function can handle and return a value that is an object, consider a scenario where we want to populate a column of five cells with calculation results and return the reference to that block of cells.

```
Function AddNumsRange(num1 As Integer, num2 As Integer) As Range

Dim ResultRange As Range

Set ResultRange = Range("A1:A5")

Range("A1").Value = num1 + num2
Range("A2").Value = num1 - num2
Range("A3").Value = num1 * num2
Range("A4").Value = num1 / num2
Range("A5").Value = num1 ^ num2

Set AddNumsRange = ResultRange

End Function
```

In this example, the function `AddNumsRange` first sets up the five cells with various mathematical results and then, crucially, uses `Set AddNumsRange = ResultRange` to return the reference to the **Range** object containing those results.

Best Practices for Returning Values in VBA

To ensure that your VBA functions are efficient, robust, and maintainable, several best practices should be observed regarding how values are returned. Firstly, always explicitly declare the return type of your function (e.g., `As String`, `As Double`, `As Range`). While VBA can default to `Variant`, explicit declaration improves performance and prevents unexpected type coercion issues when the function is called.

Secondly, adopt the practice of assigning the final result to the function name only once, typically just before the `End Function` statement. While it is technically possible to assign the value multiple times within If Else blocks, consolidating the assignment at the end often improves readability, especially in complex functions. If early exit is required (e.g., an error is detected), ensure the function name is assigned the appropriate error value (like an error string) before using the `Exit Function` statement.

ARABPSYCHOLOGY.COM

Finally, always design your functions with error handling in mind. As demonstrated with the division by zero check, proactively anticipating invalid inputs and returning informative, non-crashing results (often strings like "Invalid Input" or "N/A") is far superior to letting the function fail. This combination of explicit type declaration, consistent assignment, and defensive programming ensures that your custom VBA functions are powerful, reliable tools within your Excel environment.

ARABPSYCHOLOGY.COM