Learning VBA: Mastering Functions That Return Arrays

Authored by

Mohammed looti

November 14, 2025

The Power of Array Functions in VBA Development

Developing sophisticated applications in VBA (Visual Basic for Applications) often demands functions capable of handling and returning complex, multi-element data structures rather than simple scalar values. While conventional functions are perfectly suited for returning single results (like a specific calculation or a Boolean status), the capability to return an entire array is a fundamentally powerful technique. This approach is essential for scenarios requiring batch processing, aggregate data retrieval, or structured data manipulation within Microsoft Excel.

When a Function delivers a collection of related values, it provides significant advantages over older methods, such as passing numerous variables by reference. By encapsulating related outputs into a single variable, the calling procedure (whether a Sub routine or another Function) can efficiently access a structured dataset, thereby streamlining the code. This practice not only enhances the clarity and maintainability of the codebase but also drastically improves performance when dealing with large volumes of data that need to be processed or returned simultaneously.

Mastering this technique allows developers to create reusable components that aggregate complex results--for instance, returning a set of calculated statistics, a list of filtered records, or, as we will demonstrate, a collection of generated random numbers. This shift towards returning structured data is a hallmark of robust and modern VBA programming.

Defining the Array-Returning Function: Syntax and Declaration

To successfully implement a function that returns a collection of values in VBA, the function's signature must explicitly signal its intent to return a dynamic collection rather than a single value. This critical declaration is achieved by appending empty parentheses, `()`, immediately after the specified data type in the function definition. This syntax informs the Function that its output will be a data structure of variable size.

Consider the following structure for our demonstration function, `GenerateRandom()`, which is designed to output a structured collection of Integer values. Internally, the process involves three key steps: first, declaring a local array variable within the function body; second, populating this local array with the required data elements; and third, assigning the contents of this fully populated internal array to the function's name just prior to its completion. This final assignment is the mechanism that transfers the entire data structure back to the calling scope.

The code below illustrates the standard definition of a fixed-size array function. Note how the return type is defined as `As Integer()`, signaling the return of an array of Integers. The internal array, `RandValues`, is dimensioned to hold three elements (indices 0, 1, and 2).

Function GenerateRandom() As Integer()

```
Dim RandValues(2) As Integer

'generate three random integers and store them in array
RandValues(0) = Int(Rnd * 100)
RandValues(1) = Int(Rnd * 100)
RandValues(2) = Int(Rnd * 100)

'return array as a result of the function
GenerateRandom = RandValues

End Function
```

This specific implementation generates three pseudo-random Integer values, each ranging from 0 to 99, utilizing the `Rnd` function. These values are stored within the three-element array `RandValues`. Crucially, by assigning `RandValues` to `GenerateRandom`, the entire collection is packaged and delivered as the function's output. Once defined, this utility function becomes a highly versatile tool, ready to be called by any procedure needing a dynamic set of data.

Handling Array Return Values in the Calling Procedure

Once an array-returning function, such as `GenerateRandom()`, has completed execution, the calling procedure must be prepared to receive and process the structured output. This usually involves a `Sub` routine that first executes the function and captures the result in a suitably declared variable, and then iterates through that resulting collection. We will explore two primary, practical methods for handling the output, demonstrating the versatility of array-returning functions in diverse application contexts:

Displaying the returned values instantly using a modal dialog box, specifically the MsgBox function. Writing the collected data elements sequentially and directly into specific cells within an Excel Worksheet, a technique essential for reporting and data population tasks.

These methods showcase how the retrieved array can be seamlessly integrated into the Excel environment. The developer must ensure that the receiving variable in the calling procedure is also declared as an array (e.g., `Dim RandomValues() As Integer`) to avoid type mismatch errors when capturing the function's output. The following detailed examples will illustrate the necessary code for both common uses.

Example 1: Analyzing and Displaying Results via MsgBox

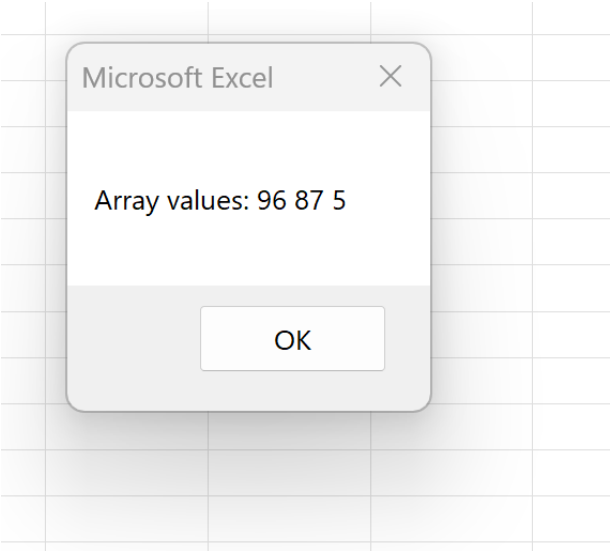
For rapid development, debugging, or providing quick user notifications, presenting function results in a modal dialog box is highly effective. This demonstration integrates the `GenerateRandom()`

function with a new `Sub` routine named `DisplayRandom()`. The primary role of `DisplayRandom()` is to execute the function, capture the resulting array, and then process its elements for textual output.

Within `DisplayRandom()`, the receiving variable, `RandomValues()`, is dimensioned as an array of Integer type to correctly store the function's output. Following the array capture, a standard `For...Next` loop is initiated. This loop iterates from the lower bound (0, as VBA arrays are zero-indexed by default) up to the upper bound (2). In each iteration, the loop concatenates the current array element into a string variable, `j`, ensuring all elements are collected before a final display. This collected string is then presented to the user via the MsgBox function.

```
'define function to generate array of three random integers
Function GenerateRandom() As Integer()

Dim RandValues(2) As Integer

RandValues(0) = Int(Rnd * 100)
RandValues(1) = Int(Rnd * 100)
RandValues(2) = Int(Rnd * 100)

GenerateRandom = RandValues

End Function

'define sub to display values from function in a message box
Sub DisplayRandom()

Dim WS As Worksheet
Dim RandomValues() As Integer
Dim i As Integer

Set WS = Worksheets("Sheet1")
RandomValues = GenerateRandom()

j = "Array values: "
For i = 0 To 2
j = (j & RandomValues(i) & " ")
Next i
MsgBox j

End Sub
```

Upon execution, the `DisplayRandom` macro first initiates the data generation process via the function, successfully retrieves the three random numbers, and then presents them to the user via the MsgBox dialogue. The following image demonstrates a typical sample execution, clearly confirming the successful retrieval and display of the structured array contents:



As illustrated, the function successfully generated and returned the random Integer values, which were then handled, concatenated, and displayed precisely as intended by the calling `Sub` routine.

Example 2: Efficiently Writing Data to an Excel Worksheet

The most frequent and arguably most useful application of array-returning functions within the Excel environment is utilizing the structured result to populate cells directly onto a Worksheet. This capability is foundational for tasks such as automated report generation, performing complex multi-output calculations, or systematically importing and structuring external data. This example employs the identical `GenerateRandom()` function but strategically modifies the `DisplayRandom()` procedure to write each element of the returned array sequentially into column A, commencing at cell **A1**.

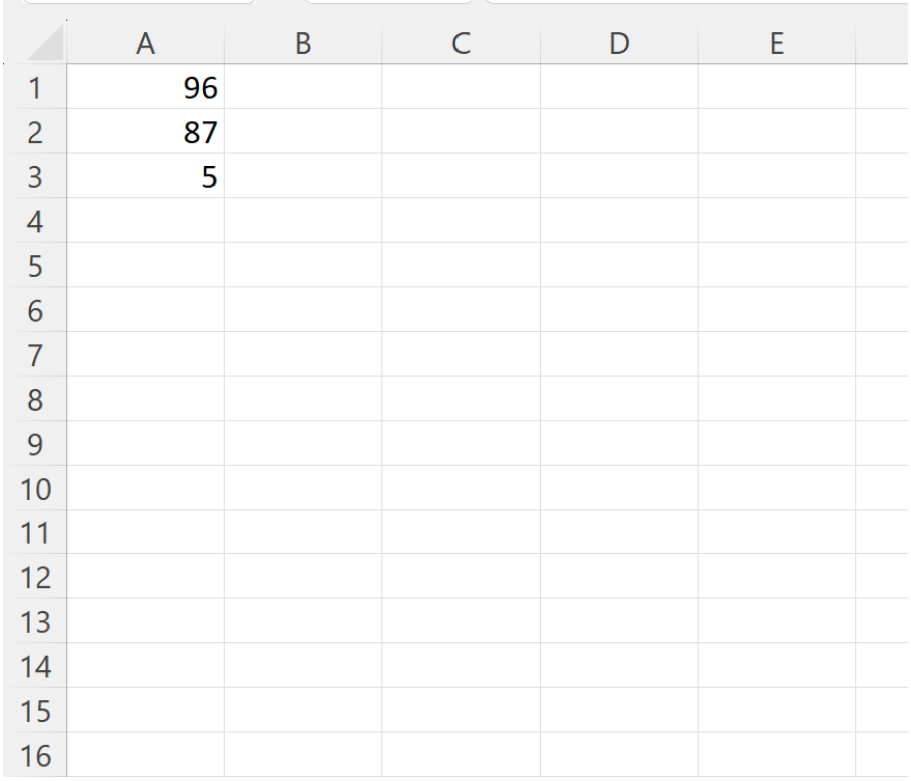
To manage the sequential placement of data, we first initialize a Worksheet object. Within the iteration loop, we leverage the powerful `Offset` property in conjunction with the loop counter (`i`). The command `WS.Range("A1").Offset(i, 0)` is crucial: it instructs the macro that in each iteration, it must move down `i` rows (and 0 columns) relative to the starting reference point (A1). This efficiently places each successive array element into a new cell vertically down the sheet (A1, A2, A3, etc.).

'define function to generate array of three random integers

Function GenerateRandom() As Integer()

```
Dim RandValues(2) As Integer

RandValues(0) = Int(Rnd * 100)
RandValues(1) = Int(Rnd * 100)
RandValues(2) = Int(Rnd * 100)

GenerateRandom = RandValues

End Function

'define sub to display values from function starting in cell A1
Sub DisplayRandom()

Dim WS As Worksheet
Dim RandomValues() As Integer
Dim i As Integer

Set WS = Worksheets("Sheet1")
RandomValues = GenerateRandom()

For i = 0 To 2
WS.Range("A1").Offset(i, 0).Value = RandomValues(i)
Next i

End Sub
```

Executing this macro results in the three randomly generated values being instantly written directly into cells A1, A2, and A3 of the specified Worksheet. The resulting output in the Excel interface is displayed in the image below:



	A	B	C	D	E
1	96				
2	87				
3	5				
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					

A significant benefit of utilizing the `Offset` property is the inherent flexibility it offers in controlling data placement. Should the requirements change, and the data need to be directed to a new starting point--such as cell C5--the developer only needs to modify the starting range reference within the loop line. By changing `WS.Range("A1")` to `WS.Range("C5")`, the macro will automatically begin writing the array values from C5, C6, and C7, respectively, without requiring changes to the array handling logic itself.

Advanced Considerations and Best Practices for Robust Code

While returning arrays from Functions in VBA offers immense power, adopting specific best practices ensures that the code remains robust, adaptable, and free of common runtime errors. These considerations are vital for transitioning from simple examples to production-level code.

Firstly, maintaining the correct syntax is non-negotiable. Always use the empty parentheses `()` in the function declaration (e.g., `As String()` or `As Variant()`) to explicitly signal that an entire structured collection is being returned, not merely a single element of that type. Failure to include these parentheses will result in a compile-time error, preventing the code from running entirely. This declaration is the compiler's assurance that the function's output will match the expected data structure.

Secondly, for code that must handle variable numbers of data elements, reliance on fixed-size

arrays (like `Dim RandValues(2)`) is discouraged. Instead, employ **dynamic arrays** by declaring them with empty dimensions (`Dim MyArray() As Data_Type`). When the exact size of the output is determined during runtime, utilize the `ReDim Preserve` statement within the function to correctly size the array. This allows the function to adapt dynamically to different input parameters or fluctuating data sizes, ensuring maximum flexibility and adaptability.

Finally, when iterating through the returned array in the calling procedure, developers should leverage the built-in `LBound` and `UBound` functions instead of relying on fixed indices (e.g., iterating from 0 to 2). Using `For i = LBound(RandomValues) To UBound(RandomValues)` provides a future-proof solution. If the size of the array returned by the function ever changes, the calling procedure will automatically adjust its loop bounds, preventing potential runtime errors such as "Subscript out of range." This practice is a cornerstone of professional VBA development.

Summary and Further Learning

Returning arrays from functions in VBA is a highly effective technique for consolidating related data and improving the flow of information between different procedures within your Excel application. By following the clear syntax guidelines for function declaration and adopting best practices like using dynamic sizing and boundary functions, you can write powerful, flexible, and maintainable code.

Mastering the nuances of VBA programming involves not only understanding how to handle various data structures but also how to optimize interactions with the Excel object model. We encourage continued exploration of advanced VBA topics to further enhance your automation and reporting capabilities.