Learning How to Return Multiple Values from R Functions

Authored by

Mohammed looti

November 13, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning How to Return Multiple Values from R Functions*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=24100

The Challenge of Returning Multiple Values in R

In the world of R programming language, a function is the fundamental building block used to encapsulate a sequence of operations designed to perform a specific task. By default, most standard programming environments, including R, are designed to return a single output object when a function completes its execution. This single-output constraint can often be restrictive, especially when developing complex analytical routines where a function might naturally generate several different results--such as a processed dataset, a statistical summary, and a derived model object--all of which need to be passed back to the main program flow.

When creating your own custom function in R, the need to return multiple, distinct values simultaneously is a frequent requirement for robust data science workflows. Unlike some languages which natively support tuple or multiple assignment returns, R requires the use of a consolidated data structure to bundle these individual results together. This method ensures that the function adheres to the single-output rule while still allowing the user to retrieve and utilize all necessary components.

The standard and most efficient way to overcome this limitation is by leveraging R's highly versatile list object. A list is a heterogeneous container capable of holding elements of different types, making it the perfect vehicle for bundling disparate results--be they numeric vectors, data frames, logical values, or even other functions--into one cohesive output object. By constructing a list within the function body and using the standard return function on this list, developers can effectively bypass the single-output restriction and streamline their code.

The Core Mechanism: Utilizing Lists for Consolidated Output

The technique for returning multiple values hinges entirely on the proper construction and utilization of the list data type. Within the body of the function, after calculating all necessary intermediate results (e.g., `result_A`, `result_B`, `result_C`), these elements are combined using the `list()` command. This action creates a single list object that contains all desired outputs, maintaining their distinct identity and allowing them to be indexed individually once the function call completes.

Crucially, the final step within the function is to use the `return()` statement, ensuring that this newly created list object is the specific item passed back to the calling environment. This approach centralizes the output process, simplifying how the results are managed externally. Since the calling environment receives a single object (the list), it can then access the components using standard list indexing methods, such as numeric indices or named elements, providing flexibility and clarity in subsequent analysis.

You can use the following basic syntax structure to implement this powerful multi-value return

mechanism in your custom R functions:

```
#define function
my_function <- function(x) {
first <- x*2
second <- x/2
output <- list(first, second)
return(output)
}

#use function with input value of 10
my_results <- my_function(10)

#print results of function
for (i in my_results) {
print(i)
}
```

Dissecting the Multi-Value Function Syntax

The syntax demonstrated above is highly effective because it treats the collection of results as a unified output container. Let's break down the role of each component within this standard structure to understand how the results are generated and retrieved. Understanding these steps is key to debugging and extending the functionality for more complex analytical tasks.

Specifically, the structure mandates three critical steps. First, the function is defined, in this case, named **my_function**. This function is designed to calculate two distinct outputs: the input value multiplied by two, and the input value divided by two. These two calculations are stored in temporary variables, `first` and `second`, respectively. These intermediate steps ensure clarity and modularity within the function body.

Secondly, the key to the multi-value return is the line `output <- list(first, second)`. This explicitly tells R to use the list() constructor to store the two calculated values as a single, indexed output object named `output`. This list is then returned. Finally, when the function is called and the results stored in `my_results`, we need a method to access the elements within that list. We typically use a for() loop to iterate through the `my_results` list, printing each individual element sequentially, thereby displaying all returned values clearly.

The function, named **my_function**, is established to return two derived values: the input multiplied by 2 and the input divided by 2.
Within the function definition, the **list()** command is utilized as a container, storing the two

calculated values as a single, cohesive output object.
To display the results in the console, a standard for() loop iterates through the list object returned by the function, printing each output value sequentially.

Using this method, we are able to return multiple values from one function. The following example shows how to use this syntax in practice.

Practical Example: Returning Calculated Values in R

To illustrate the efficiency of this method, consider a scenario where we need a single function in R that processes an input number and provides two specific derivatives crucial for a subsequent statistical model. Suppose our requirement is to create a function that accepts a single numeric value as input and then returns both the double and the half of that input value. This requires the function to successfully output two distinct, calculable results.

We can use the list-based syntax described previously to define this function, ensuring that both the multiplication and division results are correctly captured and returned. Once the function is defined, we call it with our input value (10) and then use an iterative approach to display the contents of the resulting list object. This demonstrates how the function handles the computation and how the calling environment handles the consolidated output.

The following syntax defines `my_function`, executes it with the value 10, and uses a loop to print the contents of the resulting list, confirming that both values were successfully returned:

```
#define function
my_function <- function(x) {
first <- x*2
second <- x/2
output <- list(first, second)
return(output)
}

#use function with input value of 10
my_results <- my_function(10)

#print results of function
for (i in my_results) {
print(i)
}
```

```
20
5
```

As confirmed by the output, this function successfully returns the following two values:

10 multiplied by 2 = **20**
10 divided by 2 = **5**

Advanced Retrieval: Indexing Specific Values

While iterating through the returned list using a for loop is useful for displaying all results, often in practical data analysis, we only require a specific component of the returned object. Because the output is a standard R list, we can use indexing notation to precisely extract the desired element without needing to process the entire container. This is particularly efficient when the function returns many complex objects (like large data frames or models) but only one specific summary statistic is needed immediately.

R uses double brackets (`]`) for extracting a single element from a list, treating it as the object itself rather than a sub-list. By appending `]` immediately after the function call, we instruct R to execute the function and then immediately extract the nth element from the resulting list, returning only that specific value to the console or assigning it to a new variable. Note that we can use double brackets with a specific number to only return the nth return value from the function.

For example, we can use the following syntax to define our function and then only return the first value from the output:

```
#define function
my_function <- function(x) {
first <- x*2
second <- x/2
output <- list(first, second)
return(output)
}

#use function with input value of 10 and return only first output
my_function(10)]
```

20

This function successfully returns the following value:

10 multiplied by 2 = **20**

Targeting Subsequent Values using Indexing

Following the logic of list indexing, retrieving the second or any subsequent element follows the exact same pattern, simply by changing the index number within the double brackets. This confirms the flexibility of the list structure in handling multi-value returns. If the list contained five elements, we could easily retrieve the fifth element using `]`, provided the index is within the defined bounds of the output data structure.

We could also use the following syntax to define our function and then only return the second value from the output:

```
#define function
my_function <- function(x) {
first <- x*2
second <- x/2
output <- list(first, second)
return(output)
}

#use function with input value of 10 and return only second output
my_function(10)]

5
```

This function returns the following value:

10 divided by 2 = **5**

By specifying **my_function(10)]** we are able to provide an input value of **10** to the function and then return the nth value from the output list. This syntax provides a clean and efficient way to access specific components of a multi-value return without needing intermediate variable assignment. Note: In each of these examples we used a function that returned two values but you can use similar syntax to create a function that returns as many values as you would like, simply by adding more elements to the internal list before the final return() statement.

Example: How to Return Multiple Values from Function in R

This section serves as a direct summary of the practical application demonstrated above. The ability to return multiple outputs efficiently is a cornerstone of advanced scripting in R. By leveraging the list container, developers can construct powerful analytical tools that perform several calculations simultaneously, reducing the need for multiple function calls and improving

overall code readability and performance. Always consider naming the elements within your output list (e.g., `output <- list(double = first, half = second)`) for better clarity, allowing users to access results by name (e.g., `my_function(10)$double`) rather than just by numerical index.

Related:

Additional Resources

The following tutorials explain how to perform other common tasks in R programming language:

<!--

Featured Posts

-->