Paper SM07

Bridging the Gap: Creating and Debugging User-Defined R Functions from a SAS Programmer's Perspective

Hrideep Antony, Eli Lilly & Company, USA

ABSTRACT

As the field of data science continues to evolve, many SAS programmers are transitioning to R, attracted by its versatility, open-source advantages, and growing prominence in the pharmaceutical industry. However, this shift from SAS's procedural programming approach to R's functional paradigm introduces unique challenges, particularly in the creation and debugging of user-defined functions.

While R functions and SAS macros both serve as essential tools for modularizing and automating tasks within their respective programming environments, they differ substantially in their structure, execution, and underlying concepts.

INTRODUCTION

The knowledge and tools required to create and debug user-defined R functions effectively are essential for efficient programming in R. Many SAS programmers face a significant gap between how SAS macros are written and how R functions operate.

This paper bridges that gap by providing a detailed comparison of SAS macros and R functions, exploring R's robust debugging tools. By equipping programmers with these insights, the paper aims to streamline the transition from SAS to R and enhance overall programming efficiency.

When transitioning from SAS to R, it is crucial to shift thinking from text substitution to function-based programming. SAS macros primarily operate through text substitution, dynamically generating and inserting code without executing computations directly. They rely on macro variables and text-based logic, often modifying global variables. In contrast, R functions follow an object-oriented approach, where computations are executed within functions that return structured objects. This design allows for better encapsulation, ensuring that functions return values explicitly, which can then be assigned, manipulated, and reused in further operations. This paper will provide a detailed explanation of how functions are created, how values are passed to them, and how values are returned in R.

Operational Environments in R and SAS

In programming, an environment is a container that stores variable names (symbols) along with their associated values. It defines the context in which a program or function operates and determines how variables are resolved and accessed. This functionality differs significantly between R and SAS, making it a key concept for SAS programmers to understand when transitioning to writing functions in R.

When a function is executed in R, a new environment is created to store variables that are local to that function. This means that variables or data created within the function are not directly accessible outside of it and require explicit steps to be accessed. Similarly, the function does not have access to external inputs unless they are explicitly provided. This design prevents unintentional modifications to global variables, promotes modularity by isolating variable scope, and simplifies debugging by containing changes within the function's environment.

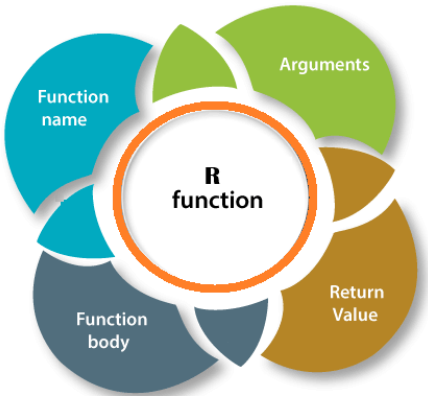
SAS, in contrast, does not employ the concept of hierarchical environments like R. SAS macros operate in a global environment, where variables created within a macro can overwrite global variables or be inadvertently modified. The absence of a strict environment hierarchy in SAS simplifies the functionality of macros, as users do not need to be overly cautious about the availability or scope of the variables created. This simplicity can make SAS macros more straightforward to use, though it can carry potential risks related to variable conflicts and unintended modifications. This fundamental difference requires SAS programmers to be more mindful of variable scope when transitioning to R. The concept of scoping will be explored further in the paper, providing a deeper understanding of how R manages variable environments.

Basic Syntax of R Function

Defining a function in R is straightforward and consists of the following components as shown below:

```r
function_name <- function(arg1, arg2 = default_value, ...) {
  # Function body
  result <- some_operation(arg1, arg2)
  return(result)
}
```

- **Function Name**: Serves as the identifier for the function.
- **Arguments**: A list of inputs enclosed within the "**function**" keyword, specifying the data the function will work with.
- **Body**: The block of code that performs the desired operations or computations.
- **Return Value**: The output generated by the function, explicitly defined using the **return**() function or implicitly as the last evaluated expression.



Understanding Scoping rules in R

Understanding scoping rules is essential for effectively writing and using R functions. In R, scoping determines how variables are found and accessed within different environments, similar to how SAS differentiates between macro variables (global scope) and DATA step variables (local scope). R follows lexical (static) scoping, meaning a function first looks for a variable within its own local environment before searching in its parent environment, and finally in the global environment if necessary.

This process mirrors SAS's approach, where macro variables persist globally, while variables created within a DATA step or procedure exist only within that specific execution block. However, unlike SAS macro variables, which reside in a single global symbol table, R functions operate within isolated environments, ensuring that variables remain contained within their respective functions.

In simple terms, this means that SAS macros expose all intermediate values globally, making them accessible outside the macro call. In contrast, R functions keep everything contained within the function, so intermediate calculations or variables inside the function are not available outside of it.

For example, in the SAS macro below, the macro variable x is stored globally, meaning it remains accessible even after the macro has executed.

```
%macro example;
%let x = 10;  /* This variable remains accessible after the macro runs */
  %mend;

%example;
%put &x;  /* Output: 10 (x is still available globally) */
```

In contrast, R functions do not store intermediate variables globally, as illustrated in the example below. Unlike SAS macros, which store variables globally by default, R functions follow lexical scoping, meaning that variables exist only within the function unless explicitly returned.

```
example_function <- function() {
    x <- 10  # This variable exists only inside the function
  }

example_function()
print(x)  # Error: object 'x' not found
```

Hierarchical Environment Search

When a function is executed, it searches for variables in a specific order:

1. Local Environment – First, the function looks for the variable inside its own environment.
2. Parent Environment – If not found, it looks in the environment where the function was defined.
3. Global Environment – If still not found, it searches in the global environment.

```
x <- 10  # Global variable

example_function <- function() {
  x <- 5  # Local variable (only inside the function)
  return(x)
}

print(example_function())  # Output: 5 (Local x is used)
print(x)  # Output: 10 (Global x remains unchanged)
```

Note that in the example above, the function creates its own local x and returns it. The global x remains unchanged with a value of 10 because functions in R do not modify global variables by default. Instead, variables declared inside a function exist only within that function's local environment.

Nested Function Scoping

Let's look at an example of a nested function to better understand the concept of Lexical Scoping as shown in the example below.

```
x <- 10  # Global variable

outer_function <- function() {
  x <- 20  # Local to outer_function

  inner_function <- function() {
    return(x)  # Searches for x in inner_function → outer_function → global
  }

  return(inner_function())
}

print(outer_function())  # Output: 20 (Finds x in outer_function, not global x)
```

In the example function above, inner_function() does not have its own local x, so it follows lexical scoping rules and searches for x in its parent environment (outer_function()) before checking the global environment. Since x <- 20 exists inside outer_function(), R finds and returns this value instead of the global x = 10. This behavior highlights how R functions inherit variable values from their defining environment, ensuring that nested functions can access parent function variables without modifying global variables.

Passing Arguments in R Functions

Positional and Named Arguments: In R, arguments can be passed to a function based on their order in the function definition. This is similar to how SAS macros pass parameters positionally, as demonstrated in the example below.

The subtract_numbers function in the example below has two parameters, x and y, where x has a default value of 10.

```
# Function with default values
subtract_numbers <- function(x = 10, y ) {
  return(x - y)
}
# 1. Passing parameters by position
result1 <- subtract_numbers(20, 5)  # 20 - 5 = 15

# 2. Passing parameters by name (order doesn't matter)
result2 <- subtract_numbers(y = 5, x = 20)  # Still 20 - 5 = 15

# 3. Passing parameters by name (takes default value for X)
result3 <- subtract_numbers(y = 5)  #  10 - 5 = 5
```

Note that when explicitly naming arguments while calling a function, we can pass them in any order, like how parameters are handled in SAS macros, as demonstrated in the result2 example. Additionally, if no value is assigned to x, it takes the default value of 10, as shown in the result3 example.

Returning Values from an R Function

While Passing arguments to R functions is like passing arguments in SAS macros, the mechanisms for retrieving results differ significantly due to how environments are managed in R. As mentioned earlier, each function in R operates within its own isolated environment, and these environments do not interact directly with one another. This design influences how results are handled and accessed.

Implicit Return

In R, the value of the last evaluated expression within a function is returned automatically, eliminating the need for an explicit return() statement. This behavior is illustrated in the example below.

```r
# Define a function
add_numbers <- function(a, b) {
  a + b  # No explicit return() needed
}

# Call the function
result <- add_numbers(3, 5)
print(result)  # Output: 8
```

Since a + b is the last evaluated expression, R automatically returns its result without requiring return() statement. It is important to note that while the last evaluated expression in a function is automatically returned, you must explicitly assign it if you intend to store or use its value later, as demonstrated in the example below.

```r
my_function <- function() {
  test <- 10
}

my_function()
test  # Error: object 'test' not found

test<-my_function()
test
print(test)  # Output: 10
```

Explicit Return

Although return() is not always necessary, using it can improve clarity, especially for complex functions where multiple operations are performed. An explicit return() statement ensures that the function's output is clearly defined. Below is an example of how an explicit return works:

```r
# Define a function
add_numbers <- function(a, b) {
  return(a + b)
}

# Call the function
result <- add_numbers(3, 5)
print(result)  # Output: 8
```

Returning Multiple Values with a List

In R, lists allow functions to return multiple values of different types as a single, unified object. This approach provides flexibility for organizing and structuring outputs, making it easy to handle diverse data types such as numbers, characters, and data frames within a single function.

Below is an example of an R function that calculates basic statistics using the calculate_stats() function and returns multiple values as a structured list:

```r
# Function to calculate basic statistics
calculate_stats <- function(x) {
  return(list(mean = mean(x), sd = sd(x), max = max(x)))
}

stats <- calculate_stats(c(1, 2, 3, 4, 5))
print(stats$mean)  # Output: 3
print(stats$sd)    # Output: 1.5811
print(stats$max)   # Output: 5
```

Returning Data Frames:

In R, data frames (analogous to SAS datasets) can be returned directly from functions, making R particularly effective for data manipulation tasks. This methodology of returning data frames is one of the most used approaches in data programming.

This concept is illustrated below with a function that creates and returns a data frame with three variables: ID, Name, and Score. This method is commonly used in data manipulation, reporting, and analysis in R.

```r
# Function to create a sample data frame
create_dataframe <- function() {
  df <- data.frame(
    ID = c(1, 2, 3),
    Name = c("Alice", "Bob", "Charlie"),
    Score = c(85, 90, 88)
  )
  return(df)  # Return the data frame
}

# Call the function and store the returned data frame
df_result <- create_dataframe()

# Print the returned data frame
print(df_result)
```

Conditional Returns: R functions can return different types of outputs based on specified conditions, making them highly flexible in handling varying input scenarios and logic paths. This feature allows functions to dynamically adapt their output based on user input, data properties, or logical evaluations.

```r
# Function to evaluate a student's score and return different outputs
evaluate_score <- function(score) {
  if (score >= 90) {
    return("Excellent")
  } else if (score >= 75) {
    return("Good")
  } else if (score >= 50) {
    return("Pass")
  } else {
    return("Fail")
  }
}

# Testing the function with different scores
print(evaluate_score(95))  # Output: "Excellent"
print(evaluate_score(80))  # Output: "Good"
print(evaluate_score(60))  # Output: "Pass"
print(evaluate_score(40))  # Output: "Fail"
```

The function evaluate_score() accepts a numeric input (score). Based on the value of score, it returns different categorical outputs:

- "Excellent" for scores 90 and above.
- "Good" for scores between 75 and 89.
- "Pass" for scores between 50 and 74.
- "Fail" for scores below 50.

The return() function is explicitly used to exit the function and return the appropriate result

Function Debugging Techniques in R

Debugging is an essential skill for identifying and fixing issues in functions. R provides several tools and techniques to assist in this process. However, debugging in R can sometimes feel more tedious compared to SAS, which offers a highly detailed log that simplifies the process by providing clear and comprehensive information about errors and execution steps.

In R, while the debugging tools are powerful, they often require more manual inspection and interactivity to trace issues effectively. An added challenge with R functions is that they operate within their own environments, meaning users must make additional effort to access intermediate information. In contrast, SAS macros function in the same global environment, allowing users to access intermediate data easily, which further simplifies the debugging process in SAS.

The simplest way to debug a function in R is by inserting print() or cat() statements to inspect intermediate values, as shown in the example below. This approach is useful for quickly checking intermediate results and validating the flow of logic within the function.

Example: Using print() for Debugging

```r
calculate_bmi <- function(weight, height) {
  print(paste("Input weight:", weight))
  print(paste("Input height:", height))

  bmi <- weight / (height^2)
  print(paste("Calculated BMI:", bmi))

  return(bmi)
}

# Call the function
result <- calculate bmi(70, 1.75)
```

Example: Using cat() for Debugging

```r
calculate_bmi <- function(weight, height) {
  cat("Debugging Info:\n")
  cat("  Input weight:", weight, "\n")
  cat("  Input height:", height, "\n")

  bmi <- weight / (height^2)
  cat("  Calculated BMI:", bmi, "\n")

  return(bmi)
}

# Call the function
result <- calculate_bmi(70, 1.75)
```

Interactive Debugging Tools

traceback(): Use traceback() after an error occurs to see the sequence of function calls leading to the error. This is particularly useful in understanding where an error originates in a nested call.

browser(): The browser() function pauses execution at a specific point in your code, allowing you to inspect variables and interact with the function environment. This feature is particularly useful for step-by-step debugging, as it enables you to examine the current state of variables and the function's environment during runtime. It is an especially valuable tool for accessing the local function environment, providing deeper insights into the function's behavior and aiding in efficient debugging.

```r
debug_function2 <- function(x) {
  browser()  # Execution pauses here
  result <- x + 2
  return(result)
}

# Call the function
debug_function2(5)
```

debug() and debugonce()

These functions allow for step-through debugging of an entire function. When a function is marked for debugging using debug() or debugonce(), R enters debug mode when the function is called, enabling you to step through each line of code interactively to inspect its execution and identify issues.

```
# Define a function
debug_function <- function(x) {
  result <- x + 2
  return(result)
}

# Enable debugging
debug(debug_function)

# Call the function
debug_function(5)
```

CONCLUSION

By understanding the differences between SAS macros and R functions, SAS programmers can seamlessly transition to R's functional programming paradigm. Mastering R's debugging tools not only facilitates the identification and resolution of issues but also ensures the creation of robust, efficient, and error-free code. As R continues to gain widespread adoption in the pharmaceutical industry, these skills will be indispensable for data analysts and programmers aiming to excel in clinical data analysis and remain competitive in the evolving landscape of data science.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my colleagues and management at **Eli Lilly and Company** for their unwavering support throughout this journey. A special thank you to **Scott Beattie** for his meticulous review and invaluable feedback, and to **Wei Zhou** for your exceptional guidance and assistance—your contributions have been instrumental to the success of this work. I deeply appreciate your encouragement and support.

8. References

For SAS programmers, it's time to learn R! by Sookie Kong, Sanofi

Pharmasug-China-2023-AP104.pdf

Clinical Trial Data Analysis Using R and SAS" by Ding-Geng Chen and Karl E. Peace

"Practical Data Science with R" by Nina Zumel and John Mount.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at

Hrideep Antony
Eli Lilly and Company
+1 (984)-301-3451
antony_hrideep@lilly.com | www.lilly.com