



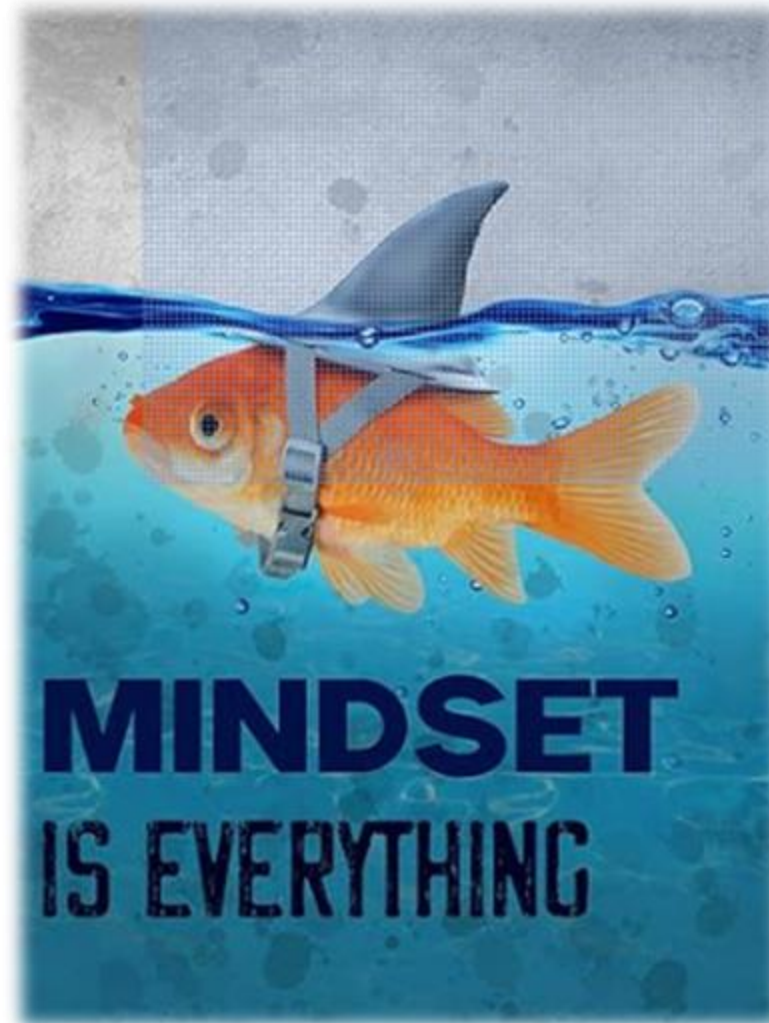
Bridging the Gap

**Creating and Debugging User-Defined R Functions from a
SAS Programmer's Perspective**

Hrideep Antony
Eli Lilly & Company, USA



- The R vs SAS Mindset!
- Operational Environments in R and SAS
- Introduction to R-Functions
- Understanding R Function Environments and Scoping
- Passing and Returning Arguments to R Functions
- Debugging Functions!



Key Conceptual Shift between SAS Macros and R Functions

Operational Environments in R and SAS

- SAS macros shares a global environment and perform text substitution
- Each function in R has its own environment.

(What happens in a function stays in the function, unless explicitly made know to the outside)

Returning results Values from the function/macro

- In SAS, macros returning values usually involves writing to a macro variable or dataset
- R functions return values are packaged explicitly using `return()`, but even without `return()`, the last evaluated expression is returned.

Operational Environments in R and SAS

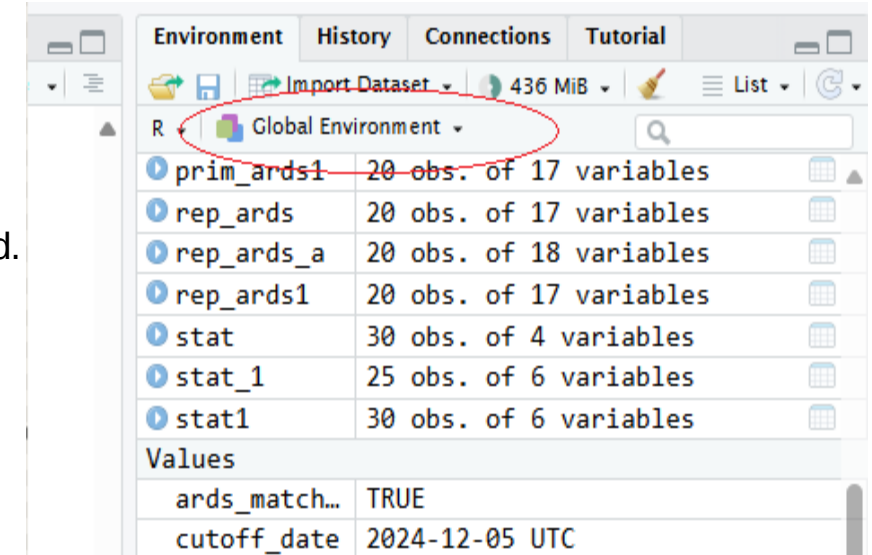
Environment defines the context in which a program or function operates and determines how variables are resolved and accessed.

R Environments (Function-Based Scope)

- ✓ A **new environment** is created each time a function executes.
- ✓ Variables inside the function **are not accessible outside** unless explicitly returned.
- ✓ Prevents **accidental modifications** to global variables.
- ✓ **Encapsulates logic**, improving modularity

SAS Environments (Global Scope in Macros)

- 🚀 **No hierarchical environments**—macro variables exist in a **global** scope.
- 🚀 Variables created inside a macro **can overwrite global variables**.
- 🚀 **Simpler structure**, making macros easy to use but **riskier** in terms of variable conflicts.
- 🚀 Requires **caution when handling variable scope** to prevent unintended modifications.



The screenshot shows the SAS Environment window. The 'Environment' tab is active, displaying a list of variables in the 'Global Environment'. A red circle highlights the 'Global Environment' header. Below it, a list of variables is shown with their attributes (observations and variables). At the bottom, a 'Values' section shows the values for 'ards_match...' and 'cutoff_date'.

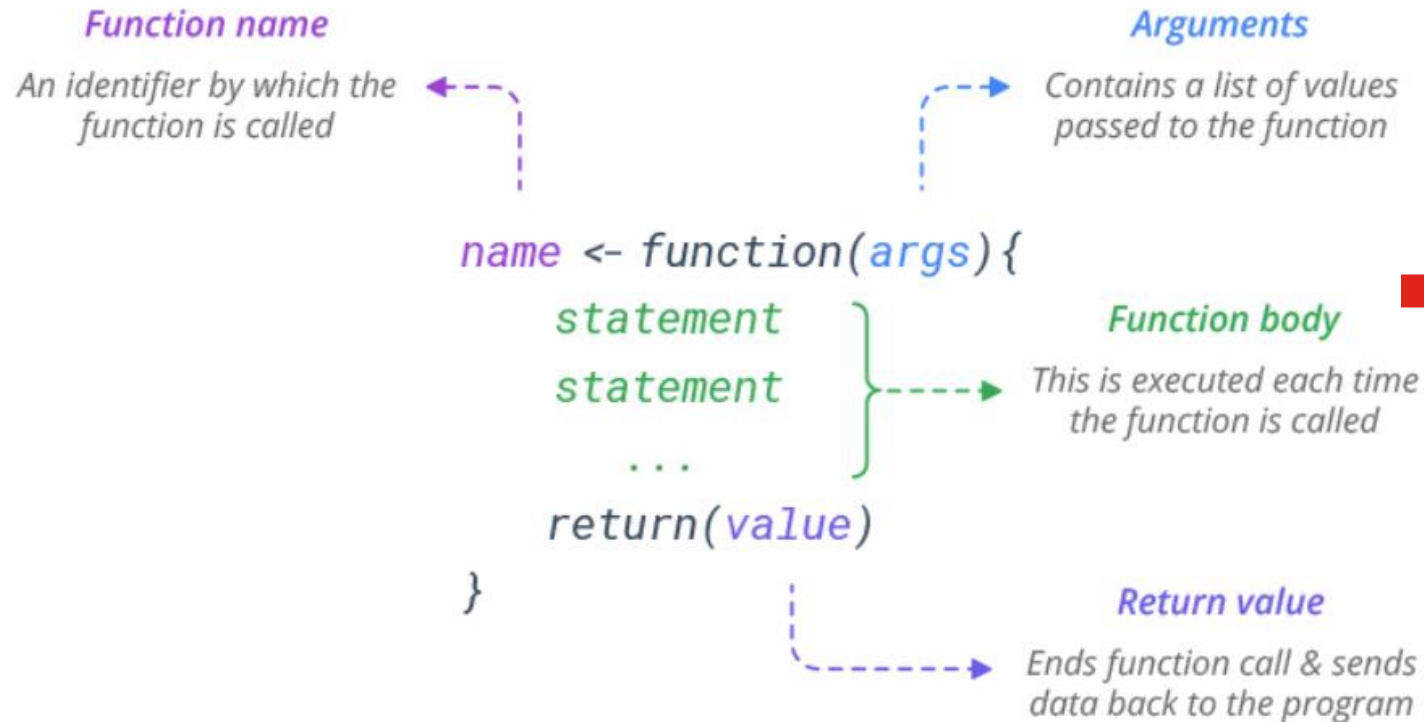
Variable	Observations	Variables
prim_ards1	20 obs.	17 variables
rep_ards	20 obs.	17 variables
rep_ards_a	20 obs.	18 variables
rep_ards1	20 obs.	17 variables
stat	30 obs.	4 variables
stat_1	25 obs.	6 variables
stat1	30 obs.	6 variables

Variable	Value
ards_match...	TRUE
cutoff_date	2024-12-05 UTC

Introduction to R-Functions

Syntax

Here's the syntax of a function in R:



```
# function to add 2 numbers  
add_num <- function(a,b)  
{  
  sum_result <- a+b  
  return(sum_result)  
}  
# calling add_num function  
sum = add_num(35,34)  
#printing result  
print(sum)
```

Understanding Scoping Rules in R

Determines **how variables are found and accessed** in different environments.

R vs. SAS Scoping

✅ **R:** Uses **lexical (static) scoping**

- Searches for variables in **local** → **global** environment.
- Functions operate within **isolated environments**, preventing unintended modifications.

✅ **SAS:** Uses **global macro variables**

- **Macro variables persist globally** unless explicitly defined as local.
- DATA step variables exist **only within execution blocks**.

◆ Key Differences

- 🚀 **R functions keep variables contained within functions**, reducing conflicts.
- 🚀 **SAS macro variables exist globally**, increasing the risk of unintentional overwrites.
- 🚀 **Understanding scoping is critical** for SAS programmers transitioning to R.

Scoping rule difference between SAS and R

- SAS relies on **global scope** by default.
- R enforces **local environments** with **lexical scoping**, for promoting modularity and reducing conflicts.

SAS

```
%macro example;  
%let x = 10; /* This variable remains accessible after the macro runs */  
%mend;  
  
%example;  
%put &x; /* Output: 10 (x is still available globally) */
```

R

```
example_function <- function() {  
  x <- 10 # This variable exists only inside the function  
}  
  
example_function()  
print(x) # Error: object 'x' not found
```

Hierarchical Environment Search in R

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. 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)
```

Global x Remains Unchanged: The value of the global variable x remains 10 because R functions do not modify global variables by default.

Passing Arguments to R Functions

- Case 1: Generally in R, the arguments are passed to the function in the same order as in the function definition.
- Case 2: If you do not want to follow any order what you can do is you can pass the arguments using the names of the arguments in any order.
- Case 3: If the arguments are not passed the default values are used to execute the function.

The `return()` function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it.

```
return(list(value1,value2,..value n))
```

```
# A simple R program to demonstrate  
# passing arguments to a function
```

```
Rectangle = function(length=5, width=4){  
  area = length * width  
  return(area)  
}
```

```
# Case 1:  
print(Rectangle(2, 3))
```

```
# Case 2:  
print(Rectangle(width = 8, length = 4))
```

```
# Case 3:  
print(Rectangle())
```

Returning arguments from R Functions

1) Simple Return:

This function multiplies a number by 2 and returns the result

```
multiply_by_two <- function(x) {  
  return(x * 2)  
}  
  
result <- multiply_by_two(5)  
print(result) # Output: 10
```

2) Implicit Return:

Here, the last expression is automatically returned without using the return() function

```
multiply_by_two <- function(x) {  
  x * 2 # Last evaluated expression is returned automatically  
}  
  
result <- multiply_by_two(5)  
print(result) # Output: 10
```

3) Conditional Return:

A function that returns different values based on conditions

```
check_value <- function(x) {  
  if (x > 0) {  
    return("Positive")  
  } else if (x < 0) {  
    return("Negative")  
  } else {  
    return("Zero")  
  }  
}  
  
print(check_value(10)) # Output: "Positive"  
print(check_value(-5)) # Output: "Negative"  
print(check_value(0)) # Output: "Zero"
```

Returning Arguments from Functions Cont..

4) Returning Multiple Values (Using Lists):

- R functions can only return one object
- you can return multiple values by packaging them into a list.



```
# Define a function that returns multiple data frames
my_function <- function() {
  df1 <- data.frame(Name=c( "Bob"), Age=c( 30))
  df2 <- data.frame(Name=c( "Sam"), Age=c( 35))

  # Use return() to return the list
  # In this example, the function my_function() returns a list containing two data frames (df1 and df2)
  return(list(first_df = df1, second_df = df2))
}

# Call the function
result <- my_function()

# You can then access each data frame using $ or by indexing the list.
first_df <- result$first_df
second_df <- result$second_df
```

Debugging Functions

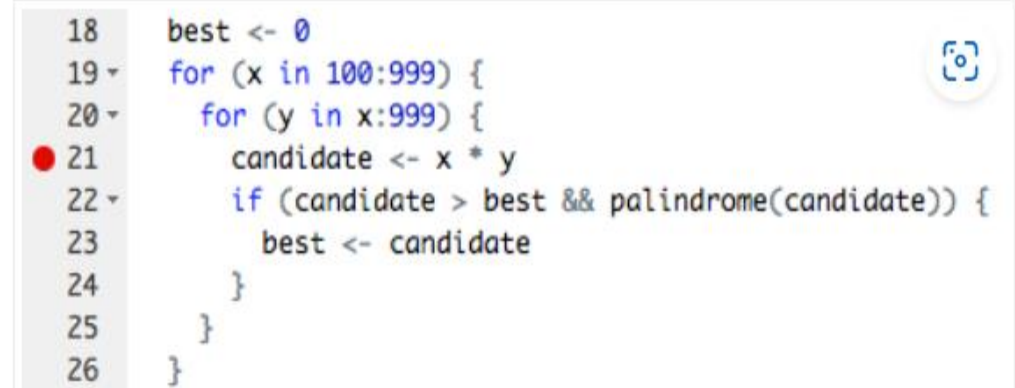
- **traceback()**: To see the call stack after an error.
- **browser()**: For interactive debugging during execution.
- **debug()**: To step through the function line by line.
- **recover()**: To inspect call stack after an error.
- **trace()**: To insert breakpoints without modifying the source.
- **print() statements**: For quick inspections.

Most useful Debugging tools

Super assignment operator <<-

```
update_value <- function() {  
  x <- 5 # Local assignment  
  y <<- 10 # Super assignment (global)  
}  
  
update_value()  
  
print(exists("x")) # FALSE (x is local to the function)  
print(y)           # 10 (y is created in the global environment)
```

Set a breakpoint



The screenshot shows a code editor with R code. A red dot on line 21 indicates a breakpoint. A magnifying glass icon is visible in the top right corner of the code block.

```
18 best <- 0  
19 for (x in 100:999) {  
20   for (y in x:999) {  
21     candidate <- x * y  
22     if (candidate > best && palindrome(candidate)) {  
23       best <- candidate  
24     }  
25   }  
26 }
```



```
x <- 3

mystery_function <- function() {
  y <- x + 2
  x <- 10
  return(y)
}

y<-mystery_function()
final_output<-x+y
```

What is the value assigned to final_output?

- (A) 14
- (B) 10
- (C) Error
- (D) 8



```
x <- 3

mystery_function <- function() {
  y <- x + 2
  x <<- 10
  return(y)
}

y <- mystery_function()
final_output <- x + y
```

What is the value assigned to final_output?

(A) 10

(B) 15

(C) Error

(D) 8

Conclusion

- Embrace R's approach to functions as objects.
- Understand how R handles scoping and closures, which differ from SAS macro processing.
- Recognize parallels between SAS procedures and R functions for statistical analysis.
- **Try R Functions with Confidence & Practice writing your own functions**