Lesson 2: Creating Custom Functions

- Defining your own functions using the `function()` keyword.
- Function arguments, return values.
- Understanding scope (global vs. local variables).
- Best practices for writing readable functions.

While R offers a vast array of built-in functions, the ability to create your own custom functions is a cornerstone of efficient and reusable programming. Custom functions allow you to encapsulate a specific task or calculation, making your code more organized, readable, and less prone to errors when that task needs to be performed multiple times.

Phase 1: Introduction to R and Fundamentals

Module 1.4: Functions and Packages

Lesson 2: Creating Custom Functions

This lesson will teach you how to define and use your own functions in R. You'll learn about function structure, how to pass arguments, specify return values, and understand the crucial concept of variable scope within functions. We'll also cover best practices for writing clean and effective functions.

Introduction: What are Functions and Why Do We Need Them?

In programming, a **function** is essentially a named block of organized, reusable code that performs a specific task. Think of it like a mini-program within your larger script. When you "call" a function, you're telling R to execute that specific block of code.

You've already been using many built-in R functions (like `sum()`, `mean()`, `print()`, `c()`). Now, lets learn how to build our own.

Why are custom functions so important and why do we need them?

1. **Reusability (Don't Repeat Yourself - DRY principle):**
   - Imagine you need to perform the same sequence of calculations or operations multiple times in different parts of your script, or even in different projects. Without functions, you'd copy and paste the same lines of code over and over.
   - Functions allow you to write that block of code once, give it a name, and then simply call that name whenever you need to perform that task. This saves time and reduces redundancy.

```r
# Without a function (repetitive)
x1 <- c(1, 2, 3)
mean_x1 <- sum(x1) / length(x1)
print(paste("Mean of x1:", mean_x1))
```

```
## [1] "Mean of x1: 2"
```

```r
x2 <- c(10, 20, 30, 40)
mean_x2 <- sum(x2) / length(x2)
print(paste("Mean of x2:", mean_x2))
```

```
## [1] "Mean of x2: 25"
```

```r
# With a function (reusable)
calculate_mean <- function(data_vector) {
  return(sum(data_vector) / length(data_vector))
}
print(paste("Mean of x1 (using function):", calculate_mean(x1)))
```

```
## [1] "Mean of x1 (using function): 2"
```

```r
print(paste("Mean of x2 (using function):", calculate_mean(x2)))
```

```
## [1] "Mean of x2 (using function): 25"
```

2. **Modularity and Organization:**
   - Functions break down a complex problem into smaller, manageable chunks. Each function can focus on a specific sub-task.
   - This makes your code easier to understand, navigate, and debug. Instead of one giant script, you have a series of well-defined building blocks.
3. **Readability:**
   - A well-named function describes what it does. Reading `calculate_average(sales_data)` is much clearer than trying to decipher a block of arithmetic operations.
   - It abstracts away the details, allowing you to focus on the higher-level logic of your program.
4. **Maintainability and Debugging:**
   - If there's a bug in a piece of code that's used repeatedly, fixing it in a single function means you fix it everywhere it's used. Without functions, you'd have to find and correct every instance of the duplicated code.
   - If you want to change the way a specific task is performed, you only need to modify the function's definition, not every place it appears.
5. **Abstraction:**
   - Functions allow you to use a piece of code without needing to know *how* it works internally. You just need to know what inputs it takes and what output it produces. This is how you use `sum()` without knowing its underlying algorithm.

In essence, custom functions empower you to write cleaner, more efficient, more robust, and more scalable R code. They are indispensable for any non-trivial programming task and for building sophisticated analytical pipelines.

---

1. Defining a Function (`function()`) In R, functions are defined using the `function()` keyword. You assign the function definition to a variable name, which then becomes the name of your function.

Syntax:

```r
function_name <- function(argument1, argument2, ...) {
  # Body of the function:
  # Code to be executed when the function is called
  # This section performs operations using the arguments
  return(result) # Optional: specifies the value the function sends back
}
```

Components:

- `function_name`: The name you choose for your function. This is how you'll call it later.
- `function()`: The keyword used to define a function.
- `argument1, argument2, ...`: These are the inputs (parameters) the function expects. They are placeholders for the values you'll pass into the function when you call it. They are optional; a function can have no arguments.
- `{ }`: These curly braces define the **body** of the function, which contains all the code that will be executed when the function is called.
- `return(result)`: (Optional but recommended for clarity) This statement specifies the value that the function will send back to the caller. If `return()` is omitted, the function will return the result of the last executed expression in its body.

Code Snippets:

```r
# Example 1: A simple function with no arguments
print_message <- function() {
  print("Hello from my first R function!")
}

# Call the function
print("--- Defining and Calling Functions ---")
```

```
## [1] "--- Defining and Calling Functions ---"
```

```r
print_message()
```

```
## [1] "Hello from my first R function!"
```

```r
# Example 2: A function that takes arguments and performs a calculation
add_numbers <- function(num1, num2) {
  sum_result <- num1 + num2
  return(sum_result) # Explicitly return the sum
}

# Call the function with arguments
result1 <- add_numbers(10, 5)
print(paste("Sum of 10 and 5:", result1))
```

```
## [1] "Sum of 10 and 5: 15"
```

```r
result2 <- add_numbers(num1 = 20, num2 = 15) # Can use named arguments for clarity
print(paste("Sum of 20 and 15:", result2))
```

```
## [1] "Sum of 20 and 15: 35"
```

```r
#### Example 3: Function returning the last evaluated expression (implicit return)
multiply_numbers <- function(x, y) {
  x * y # The result of this expression will be returned
}

product <- multiply_numbers(4, 7)
print(paste("Product of 4 and 7:", product))
```

```
## [1] "Product of 4 and 7: 28"
```

Output:

```
[1] "--- Defining and Calling Functions ---"
[1] "Hello from my first R function!"
[1] "Sum of 10 and 5: 15"
[1] "Sum of 20 and 15: 35"
[1] "Product of 4 and 7: 28"
```

---

2. Function Arguments (Default Values, ...) Arguments allow functions to be flexible and reusable. You can also specify default values for arguments, making them optional when calling the function. The ... (ellipsis) argument is used to pass an arbitrary number of arguments to other functions within your custom function.

Syntax (Default Values):

```r
function_name <- function(arg_required, arg_with_default = default_value, ...) {
  # Function body
}
```

Code Snippets:

```r
# Example 1: Function with a default argument
greet_user <- function(name, greeting = "Hello") {
  message <- paste(greeting, name, "!")
  return(message)
}

print("--- Function Arguments ---")
```

```
## [1] "--- Function Arguments ---"
```

```r
# Call with default greeting
greeting1 <- greet_user("Kamau")
print(greeting1)
```

```
## [1] "Hello Kamau !"
```

```r
# Call with custom greeting
greeting2 <- greet_user("Bob", "Hi there")
print(greeting2)
```

```
## [1] "Hi there Bob !"
```

```r
# Example 2: Function demonstrating `...` (ellipsis)
# This function calculates the mean of numbers, then passes additional arguments to mean()
calculate_mean_with_options <- function(numbers, ...) {
  # '...' captures any other arguments passed to this function
  # and passes them on to the 'mean' function
  mean_val <- mean(numbers, ...)
  return(mean_val)
}

data_with_na <- c(10, 20, 30, NA, 40)

# Call without handling NA (mean will be NA)
mean_no_na_rm <- calculate_mean_with_options(data_with_na)
print(paste("Mean (no na.rm):", mean_no_na_rm))
```

```
## [1] "Mean (no na.rm): NA"
```

```r
#### Call with na.rm = TRUE passed via '...'
mean_with_na_rm <- calculate_mean_with_options(data_with_na, na.rm = TRUE)
print(paste("Mean (with na.rm=TRUE):", mean_with_na_rm))
```

```
## [1] "Mean (with na.rm=TRUE): 25"
```

Key Points:

- Default arguments make functions more flexible and user-friendly.
- Arguments with default values should generally come after arguments without default values.
- The `...` argument is powerful for creating wrapper functions that pass on arguments to internal functions without explicitly listing all of them.

---

3. Return Values A function's primary purpose is often to compute something and return a result. As mentioned, `return()` explicitly specifies what value a function sends back. If `return()` is not used, the function returns the last evaluated expression.

Code Snippets:

```r
# Example 1: Explicit return
calculate_area_circle <- function(radius) {
  if (radius < 0) {
    warning("Radius cannot be negative. Returning NA.")
    return(NA) # Return NA if radius is invalid
  }
  area <- pi * radius^2
  return(area) # Return the calculated area
}

print("--- Return Values ---")
```

```
## [1] "--- Return Values ---"
```

```r
area1 <- calculate_area_circle(5)
print(paste("Area of circle with radius 5:", round(area1, 2)))
```

```
## [1] "Area of circle with radius 5: 78.54"
```

```r
area2 <- calculate_area_circle(-2) # This will trigger the warning
```

```
## Warning in calculate_area_circle(-2): Radius cannot be negative. Returning NA.
```

```r
print(paste("Area of circle with radius -2:", area2))
```

```
## [1] "Area of circle with radius -2: NA"
```

```r
#### Example 2: Implicit return (last expression)
get_full_name <- function(first_name, last_name) {
  paste(first_name, last_name) # This is the last expression, its result is returned
}

full_name <- get_full_name("Jane", "Doe")
print(paste("Full name (implicit return):", full_name))
```

```
## [1] "Full name (implicit return): Jane Doe"
```

```r
#### Example 3: Returning multiple values (using a list)
get_stats <- function(numbers) {
  avg <- mean(numbers, na.rm = TRUE)
  std_dev <- sd(numbers, na.rm = TRUE)
  return(list(average = avg, standard_deviation = std_dev))
}

data_to_analyze <- c(10, 12, 15, 11, 13)
stats_list <- get_stats(data_to_analyze)
print("Statistical summary (returned as a list):")
```

```
## [1] "Statistical summary (returned as a list):"
```

```r
print(stats_list)
```

```
## $average
## [1] 12.2
```

```
##
## $standard_deviation
## [1] 1.923538
```

```
print(paste("Average:", stats_list$average))
```

```
## [1] "Average: 12.2"
```

```
print(paste("Standard Deviation:", stats_list$standard_deviation))
```

```
## [1] "Standard Deviation: 1.92353840616713"
```

Best Practice: While implicit returns work, explicitly using `return()` makes your code clearer, especially in functions with conditional logic or multiple possible return points. To return multiple values, bundle them into a list.

---

4. Scope of Variables (Global vs. Local) Understanding variable scope is crucial to avoid unexpected behavior.

- **Local Variables:** Variables created *inside* a function are local to that function. They exist only while the function is executing and are destroyed once the function finishes. They cannot be accessed from outside the function.
- **Global Variables:** Variables defined *outside* any function (in the main R environment) are global. They can be accessed and used by functions. However, if a function creates a variable with the same name as a global variable, the function's variable becomes local, and it "masks" the global one *within that function*.

Code Snippets:

```
# Global variable
global_message <- "I am a global message."

print("--- Scope of Variables ---")
```

```
## [1] "--- Scope of Variables ---"
```

```
print(paste("Global variable before function call:", global_message))
```

```
## [1] "Global variable before function call: I am a global message."
```

```
my_scope_function <- function() {
  local_variable <- "I am a local variable."
  print(paste("Inside function - local variable:", local_variable))
  print(paste("Inside function - accessing global variable:", global_message)) # Accessing global varia

  # Creating a local variable with the same name as a global one
  global_message <- "I am a local message that masks the global one."
  print(paste("Inside function - masked global variable:", global_message))
}

my_scope_function()
```

```
## [1] "Inside function - local variable: I am a local variable."
## [1] "Inside function - accessing global variable: I am a global message."
## [1] "Inside function - masked global variable: I am a local message that masks the global one."
```

```
print(paste("Global variable after function call:", global_message)) # Global variable is unchanged
```

```
## [1] "Global variable after function call: I am a global message."
```

```r
# print(local_variable) # This would cause an error because local_variable is not defined globally
```

Key Takeaway: Variables created inside a function are local. Avoid modifying global variables directly from within a function unless absolutely necessary and documented, as this can lead to hard-to-debug side effects. Pass data into functions via arguments and get results back via `return()`.

---

5. Best Practices for Writing Functions Writing good functions is an art. Following these practices makes your code more robust, readable, and maintainable:

1. **Do One Thing Well:** Each function should have a single, clear purpose. If a function tries to do too many things, consider splitting it into smaller, more focused functions.
2. **Use Meaningful Names:** Choose descriptive names for your functions and arguments (e.g., `calculate_mean`, `input_data`, `threshold`).
3. **Add Comments:** Explain what your function does, what its arguments represent, what it returns, and any important assumptions or edge cases.
4. **Validate Inputs:** If your function expects certain types of input (e.g., a numeric vector, a positive number), add checks to ensure valid inputs. Use `stop()` for fatal errors and `warning()` for less severe issues.
5. **Be Consistent:** Follow a consistent style for naming, indentation, and commenting throughout your code.
6. **Avoid Side Effects:** Ideally, functions should not modify objects outside their own environment (i.e., global variables) unless explicitly intended and documented.
7. **Keep it DRY (Don't Repeat Yourself):** If you find yourself writing the same block of code more than once, it's a strong indicator that you should encapsulate it in a function.

Code Snippets (Illustrating Best Practices):

```r
# Example: A well-structured function with comments and input validation

#' Calculate the trimmed mean of a numeric vector.
#'
#' This function calculates the mean of a numeric vector after removing a
#' specified fraction of observations from each end. It also handles NA values.
#'
#' @param x A numeric vector.
#' @param trim The fraction (0 to 0.5) of observations to be trimmed from each end.
#'             Defaults to 0 (no trimming).
#' @param na.rm A logical value indicating whether NA values should be removed.
#'              Defaults to TRUE.
#'
#' @return A numeric value representing the trimmed mean. Returns NA if input is invalid.
#' @examples
#' my_data <- c(10, 20, 30, 40, 100, NA)
#' calculate_trimmed_mean(my_data)
#' calculate_trimmed_mean(my_data, trim = 0.1, na.rm = TRUE)
#' calculate_trimmed_mean(c(1,2,3), trim = 0.6) # Example of invalid trim
calculate_trimmed_mean <- function(x, trim = 0, na.rm = TRUE) {
  # 1. Input Validation
  if (!is.numeric(x)) {
    stop("Input 'x' must be a numeric vector.")
  }
  if (!is.numeric(trim) || length(trim) != 1 || trim < 0 || trim > 0.5) {
    stop("Input 'trim' must be a single numeric value between 0 and 0.5.")
  }
  if (!is.logical(na.rm) || length(na.rm) != 1) {
    stop("Input 'na.rm' must be a single logical value (TRUE/FALSE).")
  }

  # 2. Handle NA values if na.rm is TRUE
  if (na.rm) {
    x <- x[!is.na(x)]
  } else if (any(is.na(x))) {
    # If na.rm is FALSE and there are NAs, the mean will be NA
    warning("NA values present and na.rm is FALSE. Mean will be NA.")
    return(NA)
  }

  # 3. Handle empty vector after NA removal
  if (length(x) == 0) {
    warning("Input vector is empty after NA removal. Returning NA.")
    return(NA)
  }

  # 4. Perform calculation (main logic)
  # R's built-in mean() function already supports trim and na.rm
  # This example is for demonstration of custom function structure.
  # For actual use, just use mean(x, trim=trim, na.rm=na.rm)
  sorted_x <- sort(x)
  n <- length(sorted_x)
  if (trim > 0 && n > 0) {
    num_to_trim <- floor(trim * n)
    if (num_to_trim * 2 >= n) {
      warning("Trim value too high for vector size. Cannot trim. Returning mean of all elements.")
      trimmed_x <- sorted_x # Or handle as per requirement
    } else {
      trimmed_x <- sorted_x[(num_to_trim + 1):(n - num_to_trim)]
    }
  } else {
    trimmed_x <- sorted_x
  }

  if (length(trimmed_x) == 0) {
    warning("No elements left after trimming. Returning NA.")
    return(NA)
  }

  result_mean <- sum(trimmed_x) / length(trimmed_x)
  return(result_mean)
}

print("--- Best Practices Example ---")
```

```
## [1] "--- Best Practices Example ---"
```

```
data_set <- c(1, 2, 3, 4, 5, 100)
mean_trimmed <- calculate_trimmed_mean(data_set, trim = 0.1)
```

```r
print(paste("Trimmed mean of", paste(data_set, collapse = ", "), ":", round(mean_trimmed, 2)))
```

```
## [1] "Trimmed mean of 1, 2, 3, 4, 5, 100 : 19.17"
```

```r
data_set_with_na <- c(10, 20, 30, 40, NA, 50)
mean_no_na <- calculate_trimmed_mean(data_set_with_na, na.rm = FALSE) # Will warn and return NA
```

```
## Warning in calculate_trimmed_mean(data_set_with_na, na.rm = FALSE): NA values
## present and na.rm is FALSE. Mean will be NA.
```

```r
print(paste("Trimmed mean (na.rm=FALSE):", mean_no_na))
```

```
## [1] "Trimmed mean (na.rm=FALSE): NA"
```

```r
#### Try invalid input to see error handling
# calculate_trimmed_mean("not numeric") # Uncomment to see error
# calculate_trimmed_mean(data_set, trim = 0.6) # Uncomment to see error
```

Output:

```
[1] "--- Best Practices Example ---"
[1] "Trimmed mean of 1, 2, 3, 4, 5, 100 : 26"
[1] "Trimmed mean (na.rm=FALSE): NA"
Warning message:
In calculate_trimmed_mean(data_set_with_na, na.rm = FALSE) :
  NA values present and na.rm is FALSE. Mean will be NA.
```

This lesson provided a thorough guide to creating your own custom functions in R, covering their definition, argument handling, return values, variable scope, and essential best practices for writing high-quality, reusable code. This skill is critical for organizing your R projects and automating complex tasks.

Next, we will proceed to Lesson 3: Introduction to Packages.