# Writing a Functions

## Function Creation

To define a function in R, you use the function keyword.The basicsyntax is:

```
function_name <- function(arg1, arg2, ...) {
  # Code to execute
  return(output)
}
```

1. function_name: The name you assign to your function.
2. arg1, arg2, …: The arguments (inputs) your function will take.
3. return(output): The value your function will return.

## function_1

```
a<-3
b<-5
sum_of_two <- function(a, b) {
  # Add the two numbers
  result <- a + b
  # Return the result
  return(result)
}
print(sum_of_two(a, b))
```

```
## [1] 8
```

## function_2

```
greet <- function(name = "World") {
  message <- paste("Hello,", name)
  return(message)
}
greet()
```

```
## [1] "Hello, World"
```

## function_3

```r
add_numbers <- function(a, b) {
  sum <- a + b
  return(sum)
}
add_numbers(3,5)
```

```
## [1] 8
```

## Returning Multiple Values function_4

To return multiple values, you can use a list:

```r
  calculate <- function(x, y) {
    sum <- x + y
    product <- x * y
    return(list(sum = sum, product = product))
  }

result <- calculate(4, 5)
result
```

```
## $sum
## [1] 9
##
## $product
## [1] 20
```

## Nested Functions_5

You can define functions within functions:

```r
  outer_function <- function(x) {
    inner_function <- function(y) {
      return(y * 2)
    }
    result <- inner_function(x)
    return(result)
  }

print(outer_function(3))
```

```
## [1] 6
```

## Anonymous Functions_6

Anonymous functions are useful for short, one-time tasks:

```
sapply(1:5, function(x) x^2)
```

```
## [1]  1  4  9 16 25
```

## Error Handling function_7

You can handle errors using try Catch:

```
  safe_divide <- function(a, b) {
    result <- tryCatch({
      a / b
    }, warning = function(w) {
      return(NA)
    }, error = function(e) {
      return(NA)
    })
    return(result)
  }
print(safe_divide(5, 0))
```

```
## [1] Inf
```

## Named Arguments function

When calling a function, you can specify arguments by name, which makes the code more readable:

```
  calculate_area <- function(length, width) {
    area <- length * width
    return(area)
  }

print(calculate_area(length = 5, width = 3))
```

```
## [1] 15
```

## Variable Number of Arguments

You can use the ... (ellipsis) to allow a function to accept a variable number of arguments:

```
  sum_all <- function(...) {
    numbers <- c(...)
    total <- sum(numbers)
    return(total)
  }

print(sum_all(1, 2, 3, 4, 5))
```

```
## [1] 15
```

## Helper function

Helper functions are designed to perform common tasks that you might need to use multiple times within your code.They help keep your main functions clean and focused on their primary tasks.

**Create a Helper Functions** To create a helper function, you define it just like any other function in R. Here's a simple example:

```r
# Helper function to check if a number is even
is_even <- function(x) {
  return(x %% 2 == 0)
}
# Using the helper function
print(is_even(4))  # Output: TRUE
```

```
## [1] TRUE
```

```r
print(is_even(5))
```

```
## [1] FALSE
```

**Use of Helper Function** to calculate the square of a number.

```r
square <- function(x) {
  return(x * x)
}
```

Main function that uses the helper function

```r
calculate_sum_of_squares <- function(a, b) {
  sum_of_squares <- square(a) + square(b)
  return(sum_of_squares)
}
print(calculate_sum_of_squares(3, 4))
```

```
## [1] 25
```

**Generating Summary Statistics**Helper function to generate summary statistics

```r
summary_stats <- function(x) {
  stats <- list(
    mean = mean(x, na.rm = TRUE),
    median = median(x, na.rm = TRUE),
    sd = sd(x, na.rm = TRUE)
  )
  return(stats)
}
print(summary_stats(c(1, 2, 3, 4, 5)))
```

```
## $mean
## [1] 3
##
## $median
## [1] 3
##
## $sd
## [1] 1.581139
```

## Recursive Functions

A recursive function is a function that calls itself in order to solve a problem .This technique is useful for problems that can be broken down into smaller, similar sub-problems. The basic structure of a recursive function includes:

1. A **base case** condition under which the recursion stops.
2. A **recursive call** function calls itself with modified parameters.

```r
recursive_function <- function(parameters) {
  if (base_case_condition) {
    return(base_case_value)
  } else {
    return(recursive_function(modified_parameters))
  }
}
```