# STATS 266 Handout - Functions

Qi Wang

2025-04-14

## Contents

## 1 Introduction

Welcome to **STATS 266: Introduction to R**. This handout provides an introduction about functions in R. By the end of this document, you should be able to:

- Define a function that takes arguments.
- Return a value from a function.
- Set default values for function arguments.
- Explain why we should divide programs into small, single-purpose functions.

For this part, valuable materials to refer to include https://www.dataquest.io/blog/write-functions-in-r/ and https://swcarpentry.github.io/r-novice-gapminder/10-functions.html.

## 2 Definition of Functions

Functions gather a sequence of operations into a whole, preserving it for ongoing use. Functions provide:

- a name we can remember and invoke it by
- relief from the need to remember the individual operations
- a defined set of inputs and expected outputs
- rich connections to the larger programming environment

Why should we define functions?

- **Modularity:** Functions break complex tasks into smaller, manageable pieces, making code easier to understand and maintain.
- **Reusability:** Once a function is defined, it can be reused in different parts of a script or project without rewriting the same code.
- **Avoid Repetition:** Functions help avoid repeating code.
- **Improved Debugging:** Errors are easier to isolate and fix when logic is encapsulated in individual functions.
- **Parameterization:** Functions allow for flexible inputs, making it easy to apply the same logic to different data or arguments.
- **Documentation and Clarity:** Well-named functions with comments serve as documentation, improving readability and collaboration.
- **Testing and Validation:** Functions can be individually tested for correctness, which helps ensure code reliability.

As the basic building block of most programming languages, user-defined functions constitute "programming" as much as any single abstraction can. If you have written a function, you are a computer programmer. The basic format of a function looks like this:

```r
my_function <- function(parameters) {
  # perform action
  # return value
}
```

For example, if we want to create a function that transform the Fahrenheit to Kelvin, in a mathematical function, we use:

$$K = \frac{(F - 32) \times 5}{9} + 273.15.$$

In an R function:

```r
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
fahr_to_kelvin(95)
```

```
## [1] 308.15
```

In this function, `temp` is the input argument, the part inside {} is the main body of the function. One of the most important thing is the `return()` here, which tells the computer which argument will it return to you, i.e., which variable is the output of the function.

## 3 Function Scoping and Environment

What is Scoping? Scoping rules determine where and how variables are found. In R, function variables only exist inside the function unless explicitly returned.

```r
x <- 10
add_five <- function() {
  x <- x + 5
  return(x)
}
add_five()
```

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

The $x$ in the above function is a global variable. If a function cannot find a variable in its local environment, it will go to find the global variable. But if the function can find a local variable, it will not look for a global variable. Here is another example

```
y <- 1
add_five_2 <- function(){
  y <- 5
  y <- y + 5
  return(y)
}
add_five_2()
```

```
## [1] 10
```

Note: Avoid using global variables inside functions. Always pass variables explicitly as function arguments.

## 4 Set Default Values

As we see from the previous function, if we don't specify a value for the argument, the function will return an error. However, we can set the default values for the argument of the function.

```
fahr_to_kelvin <- function(temp = 95) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
fahr_to_kelvin()
```

```
## [1] 308.15
```

For the previous example, we set the default value of the `temp` to be 95. Therefore, although we didn't specify the value of `temp` when we use the function, it takes 95 as its default value.

## 5 Loop and Function

We use function typically to reduce repetitive coding chunks. Therefore, there are some cases that we need to put loop inside a function. For example, let's write a function to calculate the row sums of a matrix:

```
row_sum <- function(mat){
  d1 <- nrow(mat)
  d2 <- ncol(mat)
  result <- rep(0, d1)
  for (i in 1:d1) {
    for(j in 1:d2){
      result[i] <- result[i] + mat[i,j]
    }
```

```r
  }
  return(result)
}
M <- matrix(1:16,4,4)
row_sum(M)
```

```
## [1] 28 32 36 40
```

```r
rowSums(M)
```

```
## [1] 28 32 36 40
```

Note: Please be always careful even when you don't use a function, why do I define d1 and d2? Since if we change the input, the dimensions will also change. These types of code will not be sensitive to the input, and further will be more adaptive. Take this example:

```r
M <- matrix(1:16,4,4)
result <- rep(0, 4)
  for (i in 1:4) {
    for(j in 1:4){
      result[i] <- result[i] + M[i,j]
    }
  }
result
```

```
## [1] 28 32 36 40
```

The previous code is correct, but not so adaptive:

```r
M <- matrix(1:25,5,5)
result <- rep(0, 4)
  for (i in 1:4) {
    for(j in 1:4){
      result[i] <- result[i] + M[i,j]
    }
  }
result
```

```
## [1] 34 38 42 46
```

```r
rowSums(M)
```

```
## [1] 55 60 65 70 75
```

The return of the previous code is definitely not what we want, because when we change the dimension of the matrix, we also need to change the loop index and the result length. That would be too complicated when we have a long program. Therefore, avoid using detailed information about data, instead, define some argument to bring in the information of the data.

## 6 Recursive Functions

What Are Recursive Functions? A recursive function calls itself to solve a smaller subproblem. Useful for problems like factorial, Fibonacci sequences, and tree traversal.

```r
factorial_recursive <- function(n) {
  if (n == 1) {
    return(1)
  } else {
    return(n * factorial_recursive(n - 1))
  }
}

factorial_recursive(5)  # Output: 120
```

```
## [1] 120
```

## 7 Higher Order (Composite) Functions

We can take the output of one function as the input of another function. In math, it's like $f(g(x))$. Let's make an example:

```r
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}

kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

What if we want to transform the temperature from Fahrenheit to Celsius? We first transform the temperature from Fahrenheit to Kelvin and then from Kelvin to Celsius:

```r
fahr_to_kelvin(95)
```

```
## [1] 308.15
```

```r
kelvin_to_celsius(fahr_to_kelvin(95))
```

```
## [1] 35
```

Recall the pipe function, we can use pipe function to pass the previous value to the next function:

```r
library(dplyr)
fahr_to_kelvin(95) %>% kelvin_to_celsius()
```

```
## [1] 35
```

For more technical details about a function, refer to: https://drive.google.com/file/d/1fA1d_G0-w2UsODAbRB6EIJ6zzapJKXod/view. This course is an introduction level of basic R skills, and will move to data visualization quickly.

## 8 An Example: Bisection Method

### 8.1 Introduction to bisection method

In this example, we use a loop inside a function to implement the **bisection method**, which numerically solves equations of the form $f(x) = 0$. We will apply it to the equation:

$$f(x) = x^3 - x - 2$$

The **bisection method** is a simple and reliable numerical technique for solving equations of the form $f(x) = 0$. It belongs to the class of *bracketing methods*, which require the initial interval $[a, b]$ to satisfy $f(a) \cdot f(b) < 0$, meaning the function changes sign in the interval — and thus, by the Intermediate Value Theorem, has at least one root there.

It is particularly useful when an analytical solution is difficult or impossible to find.

### 8.2 Bisection Method Algorithm

### 8.3 Description of the Algorithm

The bisection method works as follows:

- Start with a closed interval $[a, b]$ such that $f(a) \cdot f(b) < 0$.
- Compute the midpoint $c = \frac{a+b}{2}$.
- Evaluate $f(c)$:
  - If $f(c) = 0$, then $c$ is the root.
  - If $f(c) \cdot f(a) < 0$, the root lies in $[a, c]$; set $b = c$.
  - Otherwise, the root lies in $[c, b]$; set $a = c$.
- Repeat this process until the interval is sufficiently small or $f(c)$ is close enough to zero.

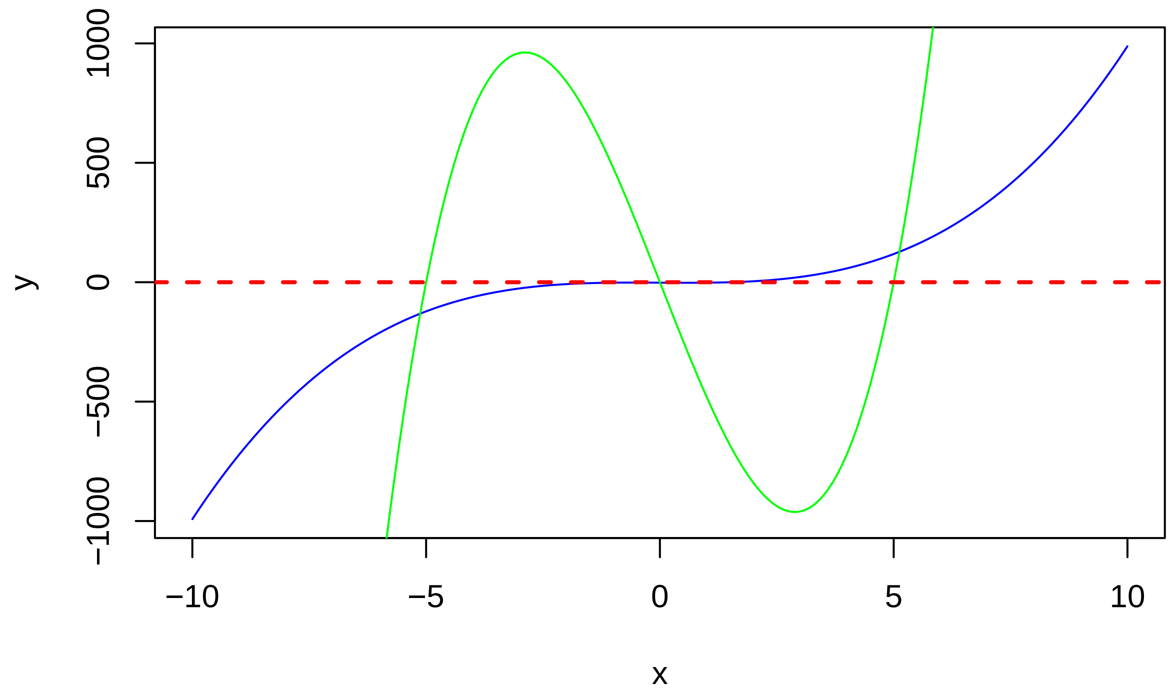
This method converges linearly and guarantees success if the function is continuous on $[a, b]$ and $f(a) \cdot f(b) < 0$.

An interesting fact: bisection method only needs $f(a) \cdot f(b) < 0$. However, one drawback is that it's hard to detect multiple roots. A lot of variants of bisection method are available to these cases. One brute force way is to visualize the function, and use bisection method multiple times to find all possible roots.

```r
f <- function(x) {
  x^3 - x - 2
}

f2 <- function(x){
  20*x^3 - 500*x
}

x <- seq(from = -10, to = 10, by = 0.1)
y <- f(x)
y2 <- f2(x)
plot(x,y, type = 'l', col = 'blue')
lines(x,y2, type = 'l', col = 'green')
abline(h = 0, col = "red", lty = "dashed", lwd = 2)
```



## 8.4 Bisection Method Implementation

```r
# Define the target function
f <- function(x) {
  x^3 - x - 2
}

# Bisection method function
bisection_method <- function(f, a, b, tol = 1e-6, max_iter = 100) {
  if (f(a) * f(b) > 0) {
    stop("Function must have opposite signs at endpoints a and b.")
  }

  for (i in 1:max_iter) {
    c <- (a + b) / 2  # Midpoint
    if (abs(f(c)) < tol || (b - a)/2 < tol) {
      cat("Converged in", i, "iterations\n")
      return(c)
    }
    if (f(c) * f(a) < 0) {
      b <- c
    } else {
      a <- c
    }
  }

  warning("Maximum iterations reached without convergence.")
  return((a + b) / 2)
}
```

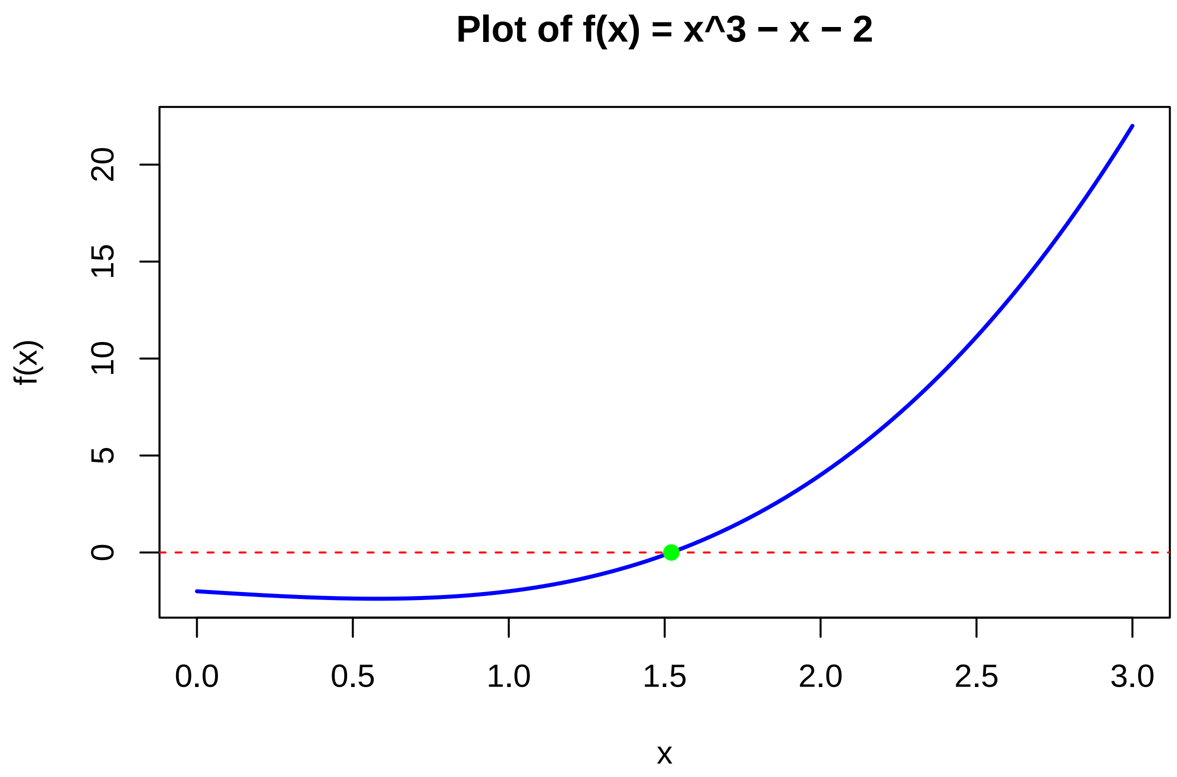
```r
# Use the function
root <- bisection_method(f, a = 1, b = 2)
```

```
## Converged in 20 iterations
```

```r
cat("Estimated root:", root, "\n")
```

```
## Estimated root: 1.52138
```

```r
# Visualize the function and the root
curve(f, from = 0, to = 3, col = "blue", lwd = 2,
      ylab = "f(x)", main = "Plot of f(x) = x^3 - x - 2")
abline(h = 0, col = "red", lty = 2)
points(root, f(root), col = "green", pch = 19)
```



## 9 Ackowledgement

This teaching material is adapted from the previous material of this course made by Marcela Alfaro-Córdoba and Sheng Jiang.