Writing Functions with R

Dr. Saad Laouadi

The Table of Contents

Different Types of Programming

Functional Programming:

It is also called **Procedural Programming**, which is way write well-structured (modular) code. Everything you do, like calculations, is built on functions. Functions let you reuse the same computational building block in different parts of a program or script.

Spaghetti Code:

Writing one line after another, a command after another, without a clear structure. This kind of code is hard to read, hard to debug and understand. Preferably avoid this kind of programming.

Object Oriented Programming **OOP**

More General concept of writing codes in programming languages. R is an OOP programming Language, which makes it very powerful programming language.

Motives for Writing Functions:

Why would I ever want to write functions in R while I can use others' functions?

I can answer this question by giving some reasons:

Shift from being an R user to an R developer

I do not want to be a developer, I am just a user.

Use and Understand others' functions effectively.

- Understand the mechanism of how functions perform to to use them efficiently.

Motives for Writing Functions:

Why would I ever want to write functions in R while I can use others' functions?

I can answer this question by giving some reasons:

Shift from being an R user to an R developer

I do not want to be a developer, I am just a user.

Use and Understand others' functions effectively.

- Understand the mechanism of how functions perform to to use them efficiently.
- What arguments the functions have

Motives for Writing Functions:

Why would I ever want to write functions in R while I can use others' functions?

I can answer this question by giving some reasons:

Shift from being an R user to an R developer

I do not want to be a developer, I am just a user.

Use and Understand others' functions effectively.

- Understand the mechanism of how functions perform to to use them efficiently.
- What arguments the functions have
- How to pass thes arguments

Motives for Writing Functions:

Why would I ever want to write functions in R while I can use others' functions?

I can answer this question by giving some reasons:

Shift from being an R user to an R developer

I do not want to be a developer, I am just a user.

Use and Understand others' functions effectively.

- Understand the mechanism of how functions perform to to use them efficiently.
- What arguments the functions have
- How to pass thes arguments
- Understand the function's documentation

Motives for Writing Functions:

Why would I ever want to write functions in R while I can use others' functions?

I can answer this question by giving some reasons:

Shift from being an R user to an R developer

I do not want to be a developer, I am just a user.

Use and Understand others' functions effectively.

- Understand the mechanism of how functions perform to to use them efficiently.
- What arguments the functions have
- How to pass thes arguments
- Understand the function's documentation
- If something goes wrong, you can solve the problem

When consider to write functions

1. Whenever you find yourself copying and pasting, there is a good chance to write a function
2. Sharing your code with others
3. writing functions make the code less error prone.
4. Uphold the programming principle **DRY**. **Don't Repeat Yourself**

Your First Functions

There are countless written functions out there by the **R** community. We write our first own function:

```
hello <- function(){
  print("Hello R programmers!")
}
hello()         # Here we call the hello() functions
```

```
## [1] "Hello R programmers!"
```

Some R base Functions

```
# mean()
# sort()
# print()
# ls()
# data()
# str()
```

Functions Definition

The General Function Template

```
my_func <- function(arg1, arg2){
  # Do some calculations
}
```

The Anatomy of Functions

- You give a name to the function: here is my_func

The Anatomy of Functions

- You give a name to the function: here is my_func
- use the assignment operator '<-'

The Anatomy of Functions

- You give a name to the function: here is my_func
- use the assignment operator '<-'
- Use the keyword 'function'

The Anatomy of Functions

- You give a name to the function: here is my_func
- use the assignment operator '<-'
- Use the keyword 'function'
- Pass some arguments to the function if there are any between parentheses

The Anatomy of Functions

- You give a name to the function: here is my_func
- use the assignment operator '<-'
- Use the keyword 'function'
- Pass some arguments to the function if there are any between parentheses
- function(arg1, arg2) ==> This is called the function signature

The Anatomy of Functions

- You give a name to the function: here is my_func
- use the assignment operator '<-'
- Use the keyword 'function'
- Pass some arguments to the function if there are any between parentheses
- function(arg1, arg2) ==> This is called the function signature
- open curly braces {}

The Anatomy of Functions

- You give a name to the function: here is my_func
- use the assignment operator '<-'
- Use the keyword 'function'
- Pass some arguments to the function if there are any between parentheses
- function(arg1, arg2) ==> This is called the function signature
- open curly braces {}
- Inside the curly braces is the body of the function

Remark

- Variables are objects, so they are nouns

Examples

Remark

- Variables are objects, so they are nouns
- functions perform some actions, thus they are verbs so preferably name your functions as "verbs" or has a verb in the name

Examples

Remark

- Variables are objects, so they are nouns
- functions perform some actions, thus they are verbs so preferably name your functions as "verbs" or has a verb in the name

Examples

- In `dplyr` package we have `select()`, `filter()` and `mutate()` functions. all are verbs

Remark

- Variables are objects, so they are nouns
- functions perform some actions, thus they are verbs so preferably name your functions as "verbs" or has a verb in the name

Examples

- In `dplyr` package we have `select()`, `filter()` and `mutate()` functions. all are verbs
- A function has a verb: **extract_coefficients()**

Remark

- Variables are objects, so they are nouns
- functions perform some actions, thus they are verbs so preferably name your functions as "verbs" or has a verb in the name

Examples

- In `dplyr` package we have `select()`, `filter()` and `mutate()` functions. all are verbs
- A function has a verb: **extract_coefficients()**
- Bad named functions `lm()` function

Three Necessary Functions

before starting writing functions you have to know that a function has three parts:

Parts of functions

1. Arguments or called formals

Checking the parts of functions

Three Necessary Functions

before starting writing functions you have to know that a function has three parts:

Parts of functions

1. Arguments or called formals
2. Body

Checking the parts of functions

Three Necessary Functions

before starting writing functions you have to know that a function has three parts:

Parts of functions

1. Arguments or called formals
2. Body
3. environment: or the location of the function's variables

Checking the parts of functions

Three Necessary Functions

before starting writing functions you have to know that a function has three parts:

Parts of functions

1. Arguments or called formals
2. Body
3. environment: or the location of the function's variables

Checking the parts of functions

1. `body()` function for the body

Three Necessary Functions

before starting writing functions you have to know that a function has three parts:

Parts of functions

1. Arguments or called formals
2. Body
3. environment: or the location of the function's variables

Checking the parts of functions

1. `body()` function for the body
2. `formals()` or `args()` for the arguments

Three Necessary Functions

before starting writing functions you have to know that a function has three parts:

Parts of functions

1. Arguments or called formals
2. Body
3. environment: or the location of the function's variables

Checking the parts of functions

1. `body()` function for the body
2. `formals()` or `args()` for the arguments
3. `environment()` for the environment

Checking Some Basic Functions

```
args(mean)
```

```
## function (x, ...)
## NULL
```

```
formals(mean)
```

```
## $x
##
##
## $...
```

```
body(mean)
```

```
## UseMethod("mean")
```

```
environment(mean)
```

```
## <environment: namespace:base>
```

Start Writing Functions

a function without arguments

```
print_my_name <- function(){
  print("My name is Dr. Saad")
}
```

```
print_my_name()
```

```
## [1] "My name is Dr. Saad"
```

```
# you have to use the parentheses for calling
```

Calling without parentheses and Body function

```
print_my_name
```

```
## function(){
##   print("My name is Dr. Saad")
## }
```

```
body(print_my_name)
```

```
## {
##     print("My name is Dr. Saad")
## }
```

Getting the source code of functions

You can check the source code of R functions easily by calling the function without **parentheses** or use `View()` function

```
head(cor)
```

```
##
## 1 function (x, y = NULL, use = "everything", method = c("pearson",
## 2     "kendall", "spearman"))
## 3 {
## 4     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete
## 5         "everything", "na.or.complete"))
## 6     if (is.na(na.method))
```

Functions in R Parlance

```
my_func <- function(){}
is.function(my_func)
```

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

```
class(my_func)
```

```
## [1] "function"
```

```
typeof(my_func)
```

```
## [1] "closure"
```

pay attention to the type of the function ## The NULL

If calling a function returns a NULL, it means that this function is empty.

```
my_func()
```

```
## NULL
```

Taking Input from the User

```r
define_me <- function(){
  Name <- readline(prompt = "What is Your name: ")
  paste("Hello Mr (Mrs): ", Name)
}
```

```r
define_me()
```

```
## What is Your name:
```

```
## [1] "Hello Mr (Mrs):  "
```

A more general function

```
def_me <- function(){
  name <- readline("Enter your Name: ")
  age <- readline("Enter your age: ")
  address <- readline("Your Address: ")
  field <- readline("What is your field: ")
  # Displaying Information
  int_age <- as.integer(age)  # Convert to integer
  cat("Hello Mr (Mrs):", name,"\n",
      "I am ",int_age, "years old\n" ,
       "I am in the", field, "business\n",
       "I Live in the wonderful", address)
}
```

More functions without arguments

```
add_1 <- function(){
  12 + 10
}
add_1()
```

```
## [1] 22
```

```
print_happy <- function(){
  print("I am happy that I am learning R")
}
```

```
print_happy()
```

```
## [1] "I am happy that I am learning R"
```

Functions that do not have arguments will always return the same value(s) in its body. It seems that this kind of functions are not useful. We will see a useful case later.

Passing Arguments to functions

```
add_num <- function(x, y){
  return(x + y)
}
```

Note that this function takes two required arguments; in other words, **if you do not give default values to them, they are required arguments** , calling a function without them will throw an error.

```
# add_num() this will throw an error
```

```
add_num(2, 3)
```

```
## [1] 5
```

Note

If a function is one line we could write it without curly braces{}

```
power_3 <- function(x) x^3
power_3(2)
```

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

Arguments with default values (Optional Arguments)

```
power <- function(x, y){
  x**y
}
```

Wouldn't be a good idea to give the argument y a default value. It would and doing so is easy.

```
power <- function(x, y = 2) {
  x ** y
}
```

Calling the function with the required argument

```
power(3)
```

```
## [1] 9
```

```
#Or you can change the default argument by give it a new value
power(3, 4)
```

```
## [1] 81
```

```
power(3, 9)
```

```
## [1] 19683
```

A more General Template of Functions

```r
my_func <- function(arg1, agr2, arg3 = 'default value'){
  # Do some calculation
}
```

Let us call the required arguments **Data arguments** and, the arguments that take default values, **Optional** or **Detail arguments**.

Calling functions

When calling a function in R,specifying the arguments can be in three ways

```
div <- function(first, second){first / second}
```

1. Exact names: in this case the order of arguments is not important

```
div(second = 12, first = 3)
```

```
## [1] 0.25
```

2. Argument order: without names, the order is important, passed as they were given in the signature

```
div(1, 10)    # 1 means first, 10 is second
```

```
## [1] 0.1
```

A rule of thumb use positional arguments with familiar and clear arguments. For unusual argument use full names, even it does require a little more typing, but it will make your code easy to read and understand.

The return keyword

Usually in R, the last computed value is returned implicitly, but we can use the keyword **return** explicitly. for example

```
power_n <- function(x, y){
  pow <- x**y
  return(pow)
}
```

is the same as

```
power_n <- function(x, y){
  pow <- x**y
  pow
}
```

Practice with writing functions

Write a function that calculates the Present Value

$$PV = FV/(1+r)^n$$

```
pv <- function(fv, r, n) {
  pv <- fv/(1+r)^n
  pv
}
```

Call the function with fv =1000, r = 0.05, n = 5

```
# call the arguments with their names(order not important)
pv(fv = 1000, r = 0.05, n = 5)
```

```
## [1] 783.5262
```

```
pv(r=0.05, n = 5, fv = 1000)      # I do not recomment this
```

```
## [1] 783.5262
```

```
# Call the arguments without names (they must in the same order),
# called positional argument
pv(1000, 0.05, 5)
```

```
## [1] 783.5262
```

Exercises:

give default values to the previous function

1. fv $= 2000$, r $= 0.04$, n $= 5$
2. Give only two arguments default arguments, r $= 0.04$, n $= 5$
3. give only one argument default value r $= 0.04$ or n $= 5$

Execrices: Continue

Write a function that converts Fahrenheit to celsius. the forumla is like this

$$Ctemp = (Ftemp - 32) * 5/9$$

Write the reverse function, Celsius to Fahrenheit

$$Ftemp = (Ctemp + 32) * 9/5$$

Write a function to calculate the circumference of a circle with radius r

$$circum = 2 * pi * r$$

$pi = 3.14$, r is the radius

Define a function to calculate the circle area

$$area = pi * r^2$$

Pythagorean theorem: Define a function to calculate the hypotenuse

$$a^2 + b^2 = c^2$$
$$c = \sqrt{(a^2 + b^2)}$$

a: Side of right triangle

b: Side of right triangle

c: Hypotenuse

Writing more functions

1. Convert from meter to millimeter, centimeter and decimeter
2. convert from kilometer to miles (1 mile $=$ 1.609 km)
3. convert from inches to centimeter (1 inch $=$ 2.54 cm)

Passing other Data types to functions

R is very powerfull programming language. Thus, we are limited to passing only a single value, we can pass other data types as well.

Let us write a function that add 155 to its input

```
add_100 <- function(x) x + 100
add_100(233)
```

```
## [1] 333
```

let us pass a vector

```
vec <- sample(1:99, 5)
vec
```

```
## [1] 29 98 45 64  5
```

```
add_100(vec)
```

```
## [1] 129 198 145 164 105
```

Pass a matrix

```
mat <- matrix(1:8, nrow = 2)
add_100(mat)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  101  103  105  107
## [2,]  102  104  106  108
```

Exercise: pass vectors and matrices to the previously written functions

More Practice

Write a function to sample coin flips. The function takes the number of flips, the probability of having Heads

```
toss_coin <- function(n_flips, p_head) {
  coin_sides <- c("head", "tail")
  # Define a vector of weights
  weights <- c(p_head, 1 - p_head)
  # Modify the sampling to be weighted
  sample(coin_sides, n_flips, replace = TRUE, prob = weights)
}
```

```
# Generate 5 coin tosses
toss_coin(5, 0.80)
```

```
## [1] "head" "head" "head" "head" "head"
```

retype the previous function by giving p_head a default value of 0.05

```
toss_coin <- function(n_flips, p_head = 0.05) {
  coin_sides <- c("head", "tail")
  # Define a vector of weights
  weights <- c(p_head, 1 - p_head)
  # Modify the sampling to be weighted
  sample(coin_sides, n_flips, replace = TRUE, prob = weights)
}
```

Condtionals: if-else statement

The general syntax is

```
if(TRUE){          # if this is true, first part will be the result
  ## do something
} else if (TRUE){ # if the first condition failed this will execute
  ## Do another thing
} else {   # after all conditions fail
  ## The last chance
}
```

```
## NULL
```

Example of if-else

```
if(runif(1, 0, 25) > 20){
  print("This number is big")
} else if(runif(1, 0, 25)  < 10){
  print("This is less than 10")
} else {
  print("I think this my lucky day")
}
```

```
## [1] "This is less than 10"
```

Types of default Arguments

1. Numeric (An integer of vector)
2. Categorical (A vector or character elements)
3. Logical (TRUE or FALSE)
4. NULL: Very special, which needs to be handled in the body of the function.
5. The ellipsis (...) Examples:

```
args(median)
```

```
## function (x, na.rm = FALSE, ...)
## NULL
```

```
args(cor)
```

```
## function (x, y = NULL, use = "everything", method = c("pearson",
##     "kendall", "spearman"))
## NULL
```

The NULL: Case of variables

NULL is a special value that represents an empty variables. It is different than zero. and it is NA which is a scalar. NULL takes no space in memory at all.

```
length(NA)
```

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

```
length(NULL)
```

```
## [1] 0
```

```
is.null(NULL)
```

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

```
is.null(NA)
```

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

Note: NULL can also be used to remove a variable from a dataset

The NULL: case of function argument

Sometimes some functions can take a NULL default. Like

```
func <- function(x, y = 2, z = NULL){}
```

When an argument takes NULL default, it means that there is no value is assigned to the argument. Also, it means that there is special handling of the argument that is too complicated to include in the function signature. In other words, the function should behave differently in this case.

So what should we do to understand the NULL: You have to read the documentation.

Example of NULL

see the arguments of different functions

```
# args(lm)
# args(set.seed)
# args(rank)
# args(cor)
# args(print.Date)
# args(prop.test)
# args(prop.table)
```

NULL is useful in many cases, so pay attention to this special **keyword**

A function example using NULL

```
ex <- function (x, max = NULL, ...)
{
  if (is.null(max))
    max <- getOption("max.print", 9999L)
  if (max < length(x)) {
    print(format(x[seq_len(max)]), max = max + 1, ...)
    cat(" [ reached 'max' / getOption(\"max.print\") -- omitted",
      length(x) - max, "entries ]\n")
  }
  else if (length(x))
    print(format(x), max = max, ...)
  else cat(class(x)[1L], "of length 0\n")
  invisible(x)
}
```

A naive example of NULL argument

```
func <- function(x, y = 2, z = NULL){
  if (is.null(z)){
    print("I am z and I have a NULL argument")
    return(x + y)
  } else if (is.numeric(z)){
    print ("I am z and I am not null")
    return(x + y + z)
  } else {
    return(paste(x, y, z, sep = " - "))
  }
}
```

Logical Argument TRUE or FALSE

We can pass a logical default `TRUE` or `FALSE` to control some arguments in the function. Many statistical functions have an argument called **na.rm** for dealing with missing values. For example

```
mean(c(1, 3, 6, 7))
```

```
## [1] 4.25
```

```
mean(c(1, 3, 6, 7, NA))
```

```
## [1] NA
```

```
args(mean.default)
```

```
## function (x, trim = 0, na.rm = FALSE, ...)
## NULL
```

```
mean(c(1, 3, 6, 7, NA), na.rm = TRUE)
```

```
## [1] 4.25
```

The . . . argument (ellipsis)

In R, or any other programming language, the body of a function contains other functions - which in turns takes arguments-. Wouldn't it be nice if we have a way to pass arguments **between functions**. This is exactly what the **ellipsis or . . .** for. It allows us to pass on arguments to inner functions. This is also indicates a variable number of arguments.

We will use a very simple example to illustrate how this works in practice. The formula of **geometric mean** is given as

$$
\begin{aligned}
GM =& (x_1 \times x_2 \times \ldots \times x_n)^{\frac{1}{n}} \\
LogGM =& \frac{1}{n} \log(x_1 \times x_2 \times \ldots \times x_n) \\
LogGM =& \frac{1}{n} (\log x_1 + \log x_2 + \ldots + \log x_n) \\
GM =& Antilog \frac{\sum \log x_i}{n}
\end{aligned}
$$

literally mean, we have a vector `X`, then we take the `log()` of each element of the vector, then we compute the **arithmetic mean**, last we exponentiate the results

We will write a function in R to calculate the Geometric mean as follows:

```
gm <- function(x){
  exp(mean(log(x)))
}
```

Not very helpful right!!! We try a more clear function

```
library(magrittr)
gm <- function(x){
  x %>%
    log() %>%
    mean() %>%
    exp()
}
```

Example

```
x <- sample(1:39, 5)
gm(x)
```

```
## [1] 14.50735
```

What if x has a missing value

```
x[3] <- NA

gm(x)           # Result is NA.
```

```
## [1] NA
```

How can we solve this problem?

We try to find the problem first

```
log(x)   # it has no problem whatsoever
```

```
## [1] 2.833213 3.401197       NA 2.484907 1.945910
```

```
mean(x)  # The problem happens here then
```

```
## [1] NA
```

Thus, we should be focusing on this function

```
args(mean)
```

```
## function (x, ...)
## NULL
```

```
args(mean.default)   # it has an argument called na.rm
```

```
## function (x, trim = 0, na.rm = FALSE, ...)
## NULL
```

na.rm is a conventional name for removing missing values. we rewrite the function again

```
gm <- function(x){
  x %>%
    log() %>%
    mean(na.rm = TRUE) %>%
    exp()
}
gm(x)
```

```
## [1] 14.38674
```

This is not efficient code, as you have to know about the arguments of the inner functions

We can add an **arg** to remove missing values (na.rm) and set it to false in the signature and we have to **change the body** of the function as well

```
gm <- function(x, na.rm = FALSE){
  x %>%
    log() %>%
    # change the body of this function
    mean(na.rm = na.rm) %>%
    exp()
}
gm(x)
```

```
## [1] NA
```

- The question, why an argument in the function signature?
- Answer: Simply to deal with the issue before any calculations occur. *There is still one problem with the code. It is a bit complicated
- Usually, we need a simplified code

Use ... to simplify it The function will be write again

```
gm <- function(x, ...){
  x %>%
    log() %>%
    mean(...) %>%
    exp()
}
```

This means accept any other argument in gm() function and pass them to mean() function.

- One of the main drawbacks of using ... is the user has to read the documentation of the inner functions

. . . as a first function argument

The ellipsis can be passed as first argument when the number of arguments is not known in advance. For example

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

Arguments Coming after the ... Argument

The arguments that come after ... **must be named** explicitly. For example

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
paste(letters[1:4], 1:4, sep = "_", collapse = NULL)
```

```
## [1] "a_1" "b_2" "c_3" "d_4"
```

```
# paste(letters[1:4], 1:4,  "_")  # this will not work properly
```

Final Note Always use ... when writing your R functions, it won't harm.

Categorical defaults

Another type of arguments a function can take is a character vector, which is called **categorical**.

When a function has this type of argument in the signature, it will have a line of code in the body contains the function `match.arg()`. The general syntax will be like this

```
func <- function(cat_arg =c("choice1", "choice2")){
  cat_arg <- match.arg(cat_arg)
}
```

A Naive Example of Categorical defaults

```
func <- function(cat_arg= c("choice1", "choice2", "choice3")){
  cat_arg <- match.arg(cat_arg)
  if(cat_arg=="choice1")print("I am the first choice")
  if(cat_arg=="choice2") print("I am the second choice")
  if(cat_arg=="choice3") print("I am the last choice")
}
func("choice1")
```

```
## [1] "I am the first choice"
```

```
func("choice2")
```

```
## [1] "I am the second choice"
```

```
func("choice3")
```

```
## [1] "I am the last choice"
```