# Writing Functions (the minimum)

Silvie Cinková

2025-08-14

## Table of contents

## 1 Why write your own

- avoid repeated copy-pasting and adjusting of code

    - laborious and error-prone

- you can keep a separate file with functions
    - have your private "library"/"package"

Functions in programming are what verbs are in natural language. You may want to write your own ad-hoc functions that are not published any library. There basically two scenarios where you hardly can avoid writing your own function:

1. You use a workflow repeatedly, especially within the same project or even script.
2. You use "official" functions that themselves require a function as an argument. These are e.g. most functions from the `purrr` library, or `mutate` to edit already existing columns (with `across()` and formula notation) in `dplyr`. Also, some `ggplot2` functions offer slots for your own functions in the formula notation.

## 2 Keep functions in a separate file

1. Write your long script.
2. For your future self, extract the functions from it and save them in a separate bare R file (`.R`).
3. In the main document with the script, apply the function `source` on that file. This will run the file and hence load all functions into your Global Environment.
    - `source(<"path/YourFileWithFunctions">.R)`

## 3 Syntax

<function_name)> `<- function(` arguments `) {`

function body (a piece of code using the arguments)

`}`

Now we are speaking about functions that you want to save as variables. The opposite is called *anonymous functions* and they behave slightly differently. We are ignoring these. So, each non-anonymous function has a **name** that you invent. A good practice is to name them as verbs, so you can immediately tell them apart from other objects. For instance `count_weird_cases` rather than `weird_case_finder`.

To this name, you assign the actual object of the function with the common assignment operator `<-`. To indicate that the object is a function, you must write `function(){}`. The parentheses enclose **arguments** of the function. They can remain empty, if the function does not take any arguments, but the parentheses are obligatory still.

Inside the curly braces comes the actual code (**body**).

Tip

When you want to transform a piece of code into a function, RStudio can help you. In the menu (top left pane by default), the `Code` tab, you will find an option called `Extract function`. Normally it is greyed out, but it will come alive (black) when you highlight a piece of code. It will enclose your code in the pair of curly braces and add the `function()` part. It will even try and identify arguments in the body to put in the parentheses. But obviously, you have to check and edit it.

## 4 Arguments

```r
print_something <- function(some_string_provided_by_user) {
  print(some_string_provided_by_user)
}
print_something(some_string_provided_by_user = "Good morning!")
```

```
[1] "Good morning!"
```

## 5 Arguments with suggested options

```r
print_two_options <- function(user_selected_string = c("hello", "hi")) {
  print(user_selected_string)
}
print_two_options(user_selected_string = "hello")
```

```
[1] "hello"
```

```r
print_two_options(user_selected_string = "bye")
```

```
[1] "bye"
```

You can set some arguments in the function that modify its behavior. Typically you offer the user a few options. For instance, ggplot2::geom_smooth has a method argument. You can choose the smoothing algorithm from several given options: gam, glm, lm , loess, or something you provide yourself, and you remember that it gives you a curve by default (provided by gam). The author must have placed inside the body a condition that tests which option the user selected, and cater for all possible scenarios. Imagine it like `if (…) {…} else if(…) {…} else if(…) {…}…. else {…}`. (In reality, the functions in the official packages are written in a much more sophisticated way and mostly not even in R but in C++, so allow some artistic license here.)

## 6 Limit user's options

```r
print_strictly_two_options <- function(user_selected_string = c("hello",
 ↪  "hi")) {
  if (user_selected_string %in% c("hello", "hi")) {
    print(user_selected_string)
  } else {print("This value is not allowed. Please choose `hello` or `hi`.")}
}
print_strictly_two_options("wow")
```

```
[1] "This value is not allowed. Please choose `hello` or `hi`."
```

## 7 Arguments with default argument values

```r
print_in_case <- function(string_from_user, convert_q_to_upper = FALSE){
  if (require(stringr) == FALSE) {
    library(stringr)
  }
  if (convert_q_to_upper == TRUE) {
   string_from_user <- str_replace_all(string =
          string_from_user, pattern = "q",
          replacement = "Q")
   print(string_from_user)
  } else  {print(string_from_user)}
}
print_in_case("Basque")
```

```
Loading required package: stringr
```

```
[1] "Basque"
```

```r
print_in_case("Basque", convert_q_to_upper = TRUE)
```

```
[1] "BasQue"
```

```r
print_in_case("Basque", convert_q_to_upper = TRUE)
```

```
[1] "BasQue"
```

This function converts the input string to the upper case by default. When the user overrides this default, you test whether there are any letters that are lower case

## 8 What functions return

result: the output of the last line

- do not assign it to variable
- or use `return(variable)` as last line

## 9 The last line: spit out the result

```r
give_me_that <- function(string){
  toupper(string)
}
give_me_that(string = "hello")
```

```
[1] "HELLO"
```

```r
result_gimmethat <- give_me_that(string = "hello")
```

```r
result_gimmethat
```

```
[1] "HELLO"
```

## 10 The last line: return result

```r
result_return <- function(string){
  my_func_result <- toupper(string)
  writeLines(my_func_result, con = "myresultstring.txt")
}
```

Writes a file in the working directory, but will not save anything to a variable.

```r
myresult <- result_return(string = "hohoho")
myresult
```

```
NULL
```

```r
list.files(pattern = "myresultstring", full.names = TRUE)
```

```
[1] "./myresultstring.txt"
```

```r
file.remove("myresultstring.txt")
```

```
[1] TRUE
```

## 11 Visibility of outputs

- normally a result is visible but can be made invisible
  - (e.g.`print` , `readr::write_lines` unlike `writeLines` returning `NULL`)

```r
a <- print("hello")
```

```
[1] "hello"
```

That printed “hello” but also saved it into the variable:

```r
a
```

```
[1] "hello"
```

## 12 Allow for arguments of functions inside your function

- this is called argument forwarding
- `sample()` returns the whole sample by default.

```
allows_arguments <- function(vector, ...) {
  sample(x = vector, ...)
}
```

```
allows_arguments(vector = c(1:10), size = 3)
```

```
[1] 10  5  9
```

```
allows_arguments(vector = c(1:10))
```

```
 [1]  1  7  6  5  2  3  4  9  8 10
```

## 13 Scope fundamental

- Variables defined inside a function do not exist outside the function!!!

```
# computes mean of a numeric vector
mean_func <- function(numvec) {
  all_summed <- sum(numvec)
  divideby = length(numvec)
  sum(numvec)/length(numvec)
}
```

```
mean_func(numvec = c(3,2,4))
```

```
[1] 3
```

```
all_summed
```

```
Error in eval(expr, envir, enclos): object 'all_summed' not found
```