1\_functions

Paul Whitfield

2022-06-01

## functions in R

* **Everything that exists in R is an object**
* **Everthing that happens is a function call**

— John Chambers

Functions are “self contained” modules of code that accomplish a specific task. Functions usually take in some sort of data structure (value, vector, dataframe etc.), process it, and return a result.

What are functions? Why should you use them?

base R is a collections of functions ‘packages’ such as `CSHShydRology is a collection of functions

# Why should I write my own functions?

Most tasks can be done by functions that do one operation, but it is possible to group a series of related tasks together to conduct a more complex task. These combine many operations, and often provide related options.

Many people write code to do one series of tasks. Then edit that code to tweak it based on feedback or relection.

The alternative is to think about the series of tasks as somethin you are likely to do more than once. Dan says “if you are going to do it twice is should be a function”. Kevin says “if you are going to do it three times it should be a function”. Paul says everything should be a function.

My view is that if I write a function I can build in the options that I think might be necessary when collaborating with others, and I know everything is going to change as I think about the analysis and the outputs. It is better to anticipate what options are needed than to edit them into code. Repeatedly.

# Questions

How do I make a function?

How can I test my functions?

How should I document my function?

A function has three components: body{}, formals(), and environment(),  
When you print a function in R these three are shown, if the function is created in the global environment that will not be shown.

my\_f <- function (x) x^2  
  
print(my\_f)

## function (x) x^2

body(my\_f)

## x^2

formals(my\_f)

## $x

environment(my\_f)

## <environment: R\_GlobalEnv>

## Primitive functions

Primitive functions are an exception as they do not have three components. They exist only in the base package. These are avoided if at all possible as they behave differently. Here is an example of one.

sum

## function (..., na.rm = FALSE) .Primitive("sum")

body(sum)

## NULL

formals(sum)

## NULL

environment(sum)

## NULL

## writing a function

Here is the structure of a simple function:

name\_of\_function <- function(argument1, argument2) { statements or code that does something return(something) }

Arguments will include the data and options, although there are many functions that do not have arguments e.g. getwd()

Also, not all functions need to return something, and return is not necessary in simple cases. But, I prefer including `return as a matter of practice.

fahrenheit\_to\_celsius <- function(temp\_F) {  
 temp\_C <- (temp\_F - 32) \* 5 / 9  
# return(temp\_C)  
 }  
  
  
fahrenheit\_to\_celsius(32)  
fahrenheit\_to\_celsius(212)

## testing the function

Once we start putting things in functions so that we can re-use them, we need to start testing that those functions are working correctly. Here is an example that takes a set of data and shifts it to a new midpoint.

center <- function(data, midpoint) {  
 new\_data <- (data - mean(data)) + midpoint  
 return(new\_data)  
 }  
  
  
z <- c(0, 0, 0, 0)  
center(z,13)

## [1] 13 13 13 13

That behaves as expected but is it an adequate test. It is never a good plan to just use one test.

z <- rnorm(10)  
mean(z)

## [1] -0.1245174

sd(z)

## [1] 1.044679

test <- center(z,13)  
length(test)

## [1] 10

mean(test)

## [1] 13

sd(test)

## [1] 1.044679

# another useful test  
  
all.equal(sd(z), sd(test))

## [1] TRUE

## documenting the function

Who hasn’t had to come back to code they wrote last year and wondered if it would not be easier to start over? Most of use do not document code as well as we should, but we all do have good intentions.

A common way to put documentation in software is to add R comments like this:

center <- function(data, midpoint) {  
 # return a new vector containing the original data centered around the  
 # midpoint.  
 # Example: center(c(1, 2, 3), 0) => c(-1, 0, 1)  
 new\_data <- (data - mean(data)) + midpoint  
 return(new\_data)  
 }

This would tell you what the function was to do and what a test result would look like.

There is a method for more formally documenting functions in a two step process. Roxgen code is added to the function. That code is used to generate a markup language that is used to produce the documentation that is in help and package descriptions.

*Formal documentation for R functions is written in separate .Rd using a markup language similar to LaTeX. You see the result of this documentation when you look at the help file for a given function, e.g. ?read.csv. The roxygen2 package allows R coders to write documentation alongside the function code and then process it into the appropriate .Rd files. You will want to switch to this more formal method of writing documentation when you start writing more complicated R projects.*

Roxygen code is placed before the function and looks like:

#' @title center   
#' @decription shift the input data to a new midpoint.  
#' Example: center(c(1, 2, 3), 0) => c(-1, 0, 1)  
#'   
#' @param data input vector  
#' @param midpoint number indicating new midpoint  
#' @author my name  
#'   
center <- function(data, midpoint) {  
 new\_data <- (data - mean(data)) + midpoint  
 return(new\_data)  
 }

## The ‘…’ argument

Frequently,one needs to allow one function to pass on argument settings to another. For example many graphics functions use the function par() and functions like plot() allow the user to pass on graphical parameters to par() to control the graphical output.

Ellipsis can be used to accomplish this. This allows passing arguments through your function to another called function. The function called inside must also have the … in the function call.

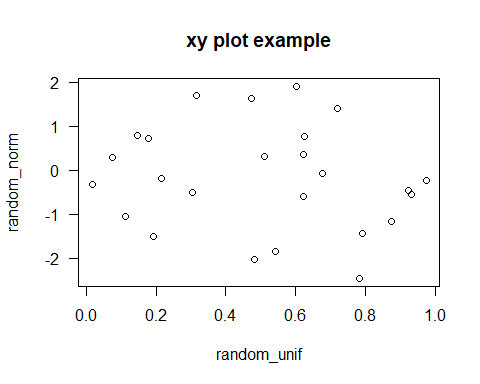
But, items used here will apply to any other called function where the argument applies should you call multiple functions where the argument would apply.

fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {  
#[omitted statements]  
if (graph)  
par(pch="\*", ...)  
#[more omissions]  
}

## Practice

Create a function that does the following: reads two arrays plots the two arrays x against y provide a summary of each variable provide the correlation between x and y allows the names of x and y on the plot to be set

x <- runif(25)  
y <- rnorm(25)  
   
 test <- my\_xyplot(x, y, xlab = "random\_unif", ylab = "random\_norm", main = "xy plot example")



str(test)

## List of 4  
## $ Number\_of\_points : int 25  
## $ summary\_of\_x : 'summaryDefault' Named num [1:6] 0.0166 0.2153 0.5429 0.5081 0.7192 ...  
## ..- attr(\*, "names")= chr [1:6] "Min." "1st Qu." "Median" "Mean" ...  
## $ summary\_of\_y : 'summaryDefault' Named num [1:6] -2.452 -1.049 -0.239 -0.18 0.728 ...  
## ..- attr(\*, "names")= chr [1:6] "Min." "1st Qu." "Median" "Mean" ...  
## $ correlation of xy:List of 9  
## ..$ statistic : Named num -0.784  
## .. ..- attr(\*, "names")= chr "t"  
## ..$ parameter : Named int 23  
## .. ..- attr(\*, "names")= chr "df"  
## ..$ p.value : num 0.441  
## ..$ estimate : Named num -0.161  
## .. ..- attr(\*, "names")= chr "cor"  
## ..$ null.value : Named num 0  
## .. ..- attr(\*, "names")= chr "correlation"  
## ..$ alternative: chr "two.sided"  
## ..$ method : chr "Pearson's product-moment correlation"  
## ..$ data.name : chr "x and y"  
## ..$ conf.int : num [1:2] -0.523 0.25  
## .. ..- attr(\*, "conf.level")= num 0.95  
## ..- attr(\*, "class")= chr "htest"

test

## $Number\_of\_points  
## [1] 25  
##   
## $summary\_of\_x  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.01662 0.21526 0.54295 0.50812 0.71917 0.97351   
##   
## $summary\_of\_y  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## -2.4515 -1.0487 -0.2385 -0.1804 0.7285 1.9042   
##   
## $`correlation of xy`  
##   
## Pearson's product-moment correlation  
##   
## data: x and y  
## t = -0.78374, df = 23, p-value = 0.4412  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## -0.5230782 0.2497662  
## sample estimates:  
## cor   
## -0.1612817

test <- my\_xyplot(x, y, xlab = "random\_unif", ylab = "random\_norm", main = "xy plot example",   
 pch = 19, col = "magenta" )  
 abline(h=0, lty = 4, col = "gray30")

