R-StatProgamming with functions in R

Timothée Bonnet, BDSI

March 19, 2020

On Friday 20/03/2020 I will be presenting this tutorial live on Zoom. (ask by email/Slack if you do not know how to use Zoom.)

If you have any trouble going through this tutorial then or at a different time you can chat about it on Slack (rsb-r-stats-biology.slack.com , if you are not a member but would like to be, drop me an email) or email me at timotheebonnetc@gmail.com.

If you do not attend the Zoom meeting but would like to receive credit through the COS Career Development Framework program I need you to complete three exercises of your choice. Send me your answers via Slack or email. It does not have to be correct on the first try and you are welcome to get in touch if you are completely stuck. I will provide feedback to help you complete exercises you want to do.

In this tutorial you will learn:

- Why is it useful to make my own functions?
- How do I make a function?
- How can I test my functions?
- How should I document my code?

Contents

1 What is a function?

A function is a piece of code, possibly long and complex, that is saved as an object in R, so that you can access this code using a single word.

You can see the code invoked by a function by running the function name without parentheses. For instance try to run:

```
tapply
```

So, over 30 lines of complex code are run when you call this function.

In R, most **functions take some arguments (=input objects) and return one output**. For instance, the function `mean()` takes a numeric vector as arguments and return a number as output:

```
input <- c(1,3,5,10) # create a vector of numbers
mean(input) # output the mean of that vector

## [1] 4.75
```

It is much more efficient to call the function mean than to write a piece of code calculating the mean of that vector using addition and division.

If you want to know what a function does and what type of input it requires look at its help page (every function coming from R-base or CRAN packages should have a help page). You can look for this page on a web browser (for instance type `mean() R`), search the function name in the `Help` tab of R-Studio, type `?mean`, or start press `tab` when your cursor is between the parentheses of `mean()`.

Many functions can take more than one arguments. How does the function what is what? There are two complementary ways:

1. By order: the help page of each function shows you in which order arguments are expected. For instance `mean(1:4, 0, FALSE)`.

2. By name: direct a value to a specific argument using `name = value`. When you call some arguments by name, the order in which they appear within the function call does not matter. For instance `mean(trim=0, x=1:4, na.rm=FALSE)`

You can mix the two, for instance: `mean(1:4, na.rm=FALSE, trim=0)`. It is safer to use argument names and I recommend you use this approach unless you are just writing quick code that you will not reuse or you know the functions you are using very well.

One last thing: functions can have default values for their arguments; that is, values that will be use if you do not provide a value for that argument. For instance `mean()` has a default value for the arguments `trim` and `na.rm` but not for `x`. You absolutely need to provide a value for `x` or the function will return an error message. You can find which arguments have default values on the function help page, in the section Usage: arguments with default values are followed by `=` and the default value in the function call, for instance `mean(x, trim = 0, na.rm = FALSE, ...)`.

If you are curious, there is a more advanced description of what functions really are in R at https://adv-r.hadley.nz/functions.html (you probably do not need this information, unless you want to become an R wizard.).

2 Why and how make your own functions?

There are lots of functions available in R-base and R-packages, so why bother? If you think you are going a piece of code only once and no one else will ever use your code, then don't bother. But if there is a chance you need to run the same series of commands multiple times, it very quickly becomes faster, cleaner and safer to turn your code into a function.

For instance, imagine you have 20 data sets on which you want to apply some data transformation, run some statistical tests, plot model predictions and write that plot to a .png file to use in a report. That series of operations may be 15 lines long. If you copy paste the code for each data set you will need 300 lines of code and are likely to make typos that may be difficult to find. If you write your code as a function, you may need 20 lines of code (in general it will), but then you can analyse each data set in 1 line. In total you will need 40 lines instead of 300. Further, if you decide to change the statistical model later on, you need to modify your code in only one spot, inside the function. You probably will not need to edit the 20 function calls for each data set. Finally, writing functions will force you to make your code more robust (because it must work with slightly different input) which makes it less error prone and more easy to re-use and share with others.

To keep your scripts clean, you can save all the functions you create for a given project into a single .R file, separate from your script doing your actual analyses. At the start of your analysis script you can source the function script in a way similar to loading packages using library. For instance, if you save all your functions for a given project in the file MyFunctions.R, start the script where you will do your analyses by `source(MyFunctions.R)`.

NB: Building your own functions and sourcing them is similar to building your own package. In fact, once you have functions in a file it is quite simple to create an R-package (but we will see that another time).

2.1 How to define a function

To create a function, you need to assign the output of the function `function()` to an object... yes that is confusing, so here is an example. Let's start by defining a function fahrenheit_to_celsius that converts temperatures from Fahrenheit to Celsius:

```
fahrenheit_to_celsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5 / 9
  return(temp_C)
}
```

After running this code, the function fahrenheit_to_celsius exists in your environment and you can use it:

```
fahrenheit_to_celsius(99)

## [1] 37.22

fahrenheit_to_celsius(451)

## [1] 232.8
```

Look again at the code defining the function. Everything within {} is the code you run. Notice how the name of the argument (`temp_F`) is used for a variable inside the function. Arguments are assigned to variables and that is how the code inside the function returns different values when you use different argument values. What is inside `return()` is your function output. In R, if you do not write a `return()` in your function, the last command within your function will be returned as the output. I recommend you always use `return()` to be sure you know what your function outputs.

* Exercise 1 — Build a first function

Write a function that return the product of three arbitrary numbers together (x*y*z) provided by the user.

The `return()` function (yes, it is a function itself!) can output a single object, but often we want to calculate many things at once, not just a single number! So how do you do if you want two or more values out of your function? Remember that in R, objects can be vectors, matrices, lists, data-frames... all of which can contain many values, possibly of different types. So you could very well write on object containing several values within your function and include that object within `return()`.

** Exercise 2 — Multiple values in output

Write a function that return the product of three arbitrary numbers together (x*y*z) as well as their sum (x+y+z). Remeber you can create vectors of several values using the function `c(x,y)`.

** Exercise 3 — Write a data-frame of random values in output

Write a function that return a data-frame containing 3 columns of randomly generated normal numbers (use the function `rnomr`). Your function should have an argument to control how many rows the data-frames has. As a bonus, create arguments to control the mean and standard deviation of each column.

2.2 Combining functions

The real power of functions comes from mixing, matching and combining them into ever-larger chunks to get the effect we want.

Let's define two functions that will convert temperature from Fahrenheit to Kelvin, and Kelvin to Celsius:

```
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)
}
```

We can define a function to convert directly from Fahrenheit to Celsius, by reusing these two functions above:

```
fahr_to_celsius <- function(temp) {
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}
```

** Exercise 4 — Combine several functions together

Let's say you want to better understand the statistical properties of models you are using on real data. You would like to simulate data of known properties, fit models on these simulated data and visualize if you recover the truth in your data (match between expected and estimated values). That is all a bit complicated so you split the process in several functions. Below I give you working functions for each sub-task. Reuse these functions to create a short function running the whole process at once. Then play with parameter values (beta1 and beta2) and see what happens.

First you simulate data:

```r
simul_function <- function(nbdatasets = 10, nbsamples= 500, alpha=1, beta1=0.5, beta2
{
  alldat <- list()
  for (i in 1:nbdatasets)
  {
    x1 <- rnorm(n = nbsamples)
    x2 <- x1 + rnorm(n = nbsamples)
    y <- alpha + beta1*x1 + beta2*x2 + rnorm(n = nbsamples)
    alldat[[i]] <- data.frame(x1=x1, x2=x2, y=y)
  }
  return(alldat)
}#end simul_function()

datexmple <- simul_function()
```

Then you fit different models and extract parameter estimates.

```r
models_function <- function(dat)
{
  res <- data.frame(model1_alpha=rep(NA,times=length(dat)),
                          model1_beta1=NA, model1_beta2=NA,
                          model2_alpha=rep(NA,times=length(dat)),
                          model2_beta1=NA, model2_beta2=NA)
  for(i in 1:length(dat))
  {
    model1 <- lm(y ~ x1 + x2, data = dat[[i]])

    model2 <- lm(y ~ x1, data = dat[[i]])

    model2bis <- lm(residuals(model2) ~ dat[[i]]$x2)

    res[i,] <- c(coef(model1), coef(model2), coef(model2bis)[2])
  }
  return(res)
}#end models_function()

modelresults <- models_function(datexmple)
```

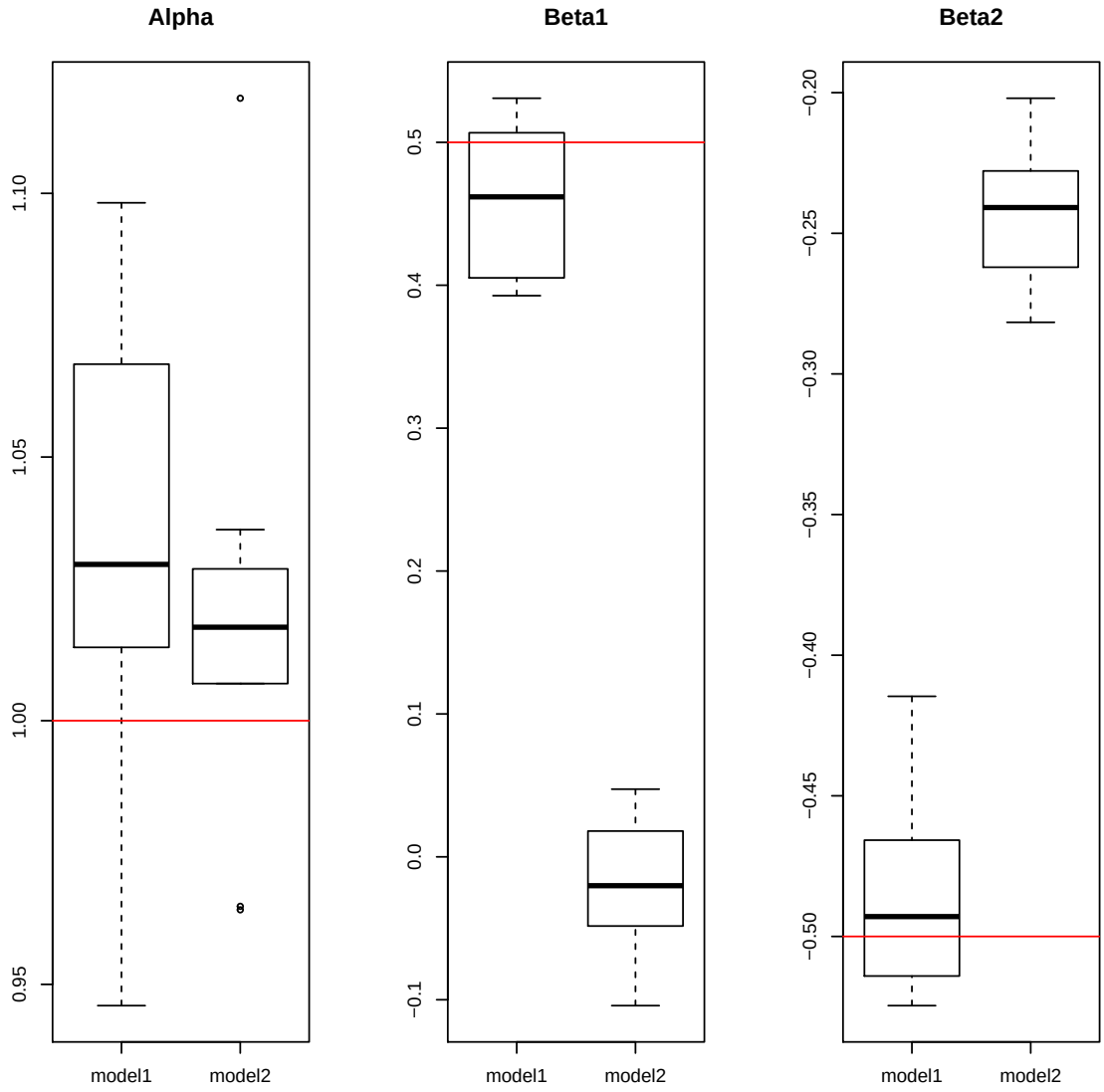
Finally you make boxplots out of pairs of parameter estimates and draw a red line showing the true value used during the simulation.

```r
visu_function <- function(modelsoutput, truevalues = c(1, 0.5, -0.5))
{
  par(mfrow=c(1,3))
  boxplot(modelsoutput[,c(1,4)], main= "Alpha",
          names = c("model1", "model2"))
  abline(h=truevalues[1], col="red")

  boxplot(modelsoutput[,c(2,5)], main= "Beta1",
          names = c("model1", "model2"))
  abline(h=truevalues[2], col="red")

  boxplot(modelsoutput[,c(3,6)], main= "Beta2",
          names = c("model1", "model2"))
  abline(h=truevalues[3], col="red")
}#end visu_function

visu_function(modelresults)
```



2.3 Turning long complicated code into a function

**** Exercise 5** — **Turning a piece of code into a function**

Tthe following piece of code simulate p-values for a random effect in a mixed model:

```
RandomVariance <- 0
sampsize <- 500
nbblocks <- 30
pvals <- vector(length = 1000)
altpvals <- vector(length = 1000)
for (i in 1:1000)
{
x <- rnorm(sampsize,mean = 4, sd=0.25)
block <- sample(x = 1:nbblocks, size = sampsize, replace = TRUE)
blockvalues <- rnorm(n = nbblocks, mean = 0, sd = sqrt(RandomVariance))
y <- 8 - x + blockvalues[block] + rnorm(sampsize,0,1)
dat <- data.frame(response = y, predictor = x, block=block)
lm0 <- lm(response ~ 1 + predictor, data=dat)
lmm0 <- lmer(response ~ 1 + predictor + (1|block), data=dat )
(LRT0 <- anova(lmm0, lm0)) #mixed model must come first!
pvals[i] <- LRT0$`Pr(>Chisq)`[2] # the p-value
altpvals[i] <- 1-pchisq(LRT0$Chisq[2],0.5) # a better p-value
}
pvals
altpvals
```

Turn this into a function were the arguments will let you control:

- The value of the simulated variance (RandomVariance)
- The number of simulations (above it is fixed to 1000)
- (optionally) The sample size and number of random levels (blocks)

And output:

- A vector of standard p-values from the LRT
- (optionally) a vector of "alternative" p-values (from the hand-made Chi-square)
- (optionally) histograms of the distribution of p-values

3 Funny things

3.1 The dot-dot-dot

*** Exercise 6 — Understand the . . .**

Consider these two functions:

```r
halfmean1 <- function(x)
{
  mean(x)/2
}
halfmean2 <- function(x,...)
{
  mean(x,...)/2
}
```

They behave the same way in the first case, but differently in the second:

```r
somedata <- c(10, 25, NA)

halfmean1(somedata)

## [1] NA

halfmean2(somedata)

## [1] NA

halfmean1(somedata, na.rm=TRUE)

## Error in halfmean1(somedata, na.rm = TRUE): unused argument (na.rm = TRUE)

halfmean2(somedata, na.rm=TRUE)

## [1] 8.75
```

What does . . . do?

3.2 The <<- operator

* Exercise 7 — Understand the <<-

We create to almost identical functions, f0 and f1 and add their output to an object `x`. Compare the output and explain what happens. What does `<<-` mean?

```
f0 <- function(x=2){
  x <- x
  y <- x+2
  return(y)
}

f1 <- function(x=2){
  x <<- x
  y <- x+2
  return(y)
}

x <- rnorm(1000)
f0()+x

x <- rnorm(1000)
f1()+x
```

3.3 Recursive functions

** Exercise 8 — What does this function do?

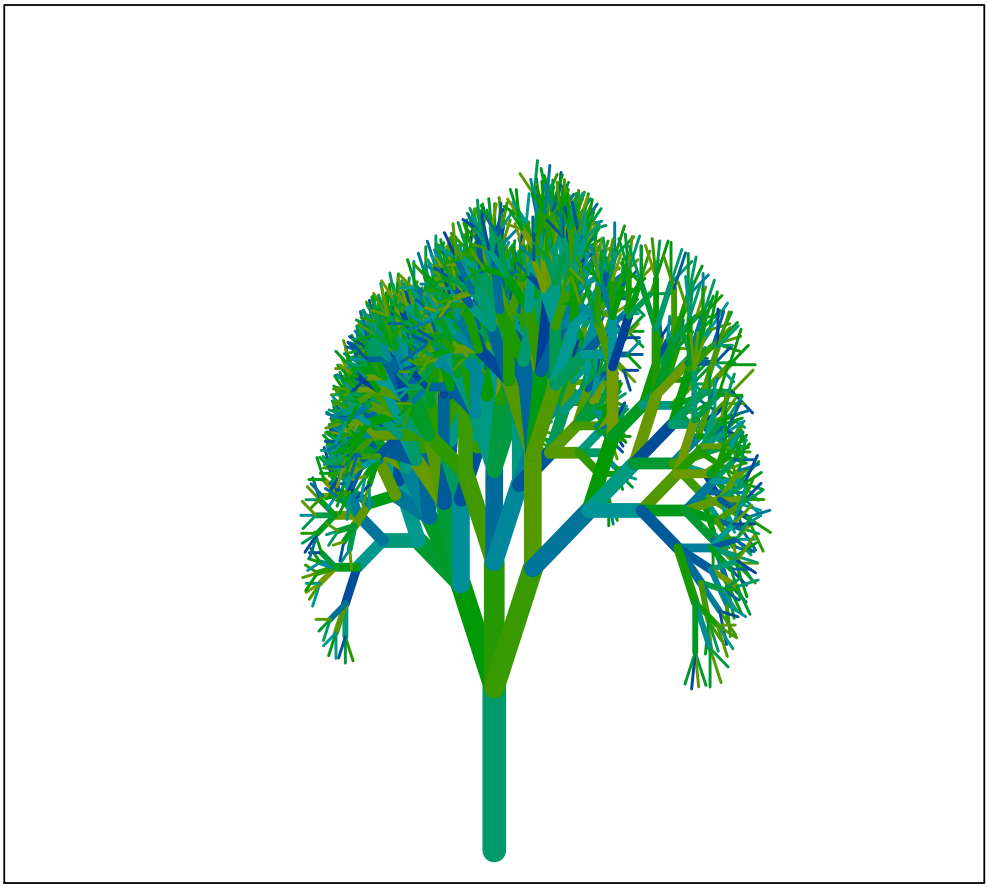
Consider the following function and try to understand what it does. (code by Dr. Koen van Benthem)

```
tree<-function(x,y,l,dir,n,nmax){
if(n==0){
return()
}
pos<-round(runif(1,1,199))
colour=rainbow(200,start=0.2,end=0.6,v=0.6)[pos]
lines(c(x,x+l*sin(dir)),c(y,y+l*cos(dir)),
      lwd=20*(n/nmax),col=colour)
branches<-round(runif(1,2,4))
for(i in 1:branches){
angle<-dir+(-pi/6)+(pi/3)*(i-1)/(branches-1)
l2<-runif(1,0.7,0.85)*l
tree(x+l*sin(dir),y+l*cos(dir),l2,angle,n-1,nmax)
}
}
```

You can run the function and see if you had guessed right:

```r
plot(0,0,type="n",xlim=c(-10,10),ylim=c(0,10),
xaxt="n",yaxt="n",ylab="",xlab="")

tree(0,0,2,0,8,12)
```



What is special about this function? Why should you be careful not to give large values to the parameters n?

*** Exercise 9 — Find a mistake in a family tree (Challenging!)

For this exercise you will need to download a file as follows:

```
download.file(url="https://raw.githubusercontent.com/timotheenivalis/RSB-R-Stats-Biol
destfile = "wrongpedigree.csv")

ped <- read.csv("wrongpedigree.csv", stringsAsFactors = FALSE)
```

The file `wrongpedigree.csv` contains a pedigree (a family tree) containing a mistake: a function we tried to apply on this data set informed us that the family tree was impossible without more information. We suspect that we have assigned an individual as its own ancestor (this can happen with genetic reconstruction of parentage). Use a self-referencing function to scan the tree and find where the problem is.