R403: Probabilistic and Statistical Computations with R

Topic 10: Writing R Functions

Kaloyan Ganev

2022/2023

Lecture Contents

1. Introduction

2. Notes on environments

3. Functions

Introduction

Introduction

- Functions are used in R all the time, and you already used a significant variety of them
- Of course, there cannot be a pre-programmed function for everything possible (actually not only in R but in any programming language)
- Therefore, it is sometimes very useful to create one or several functions for specific but recurring jobs
- Before that, because of the fact that functions use variables as their arguments, it is a good idea to have a quick discussion on environments in R

Notes on environments

What is an environment?

- Think back of how we defined variables: as containers for values
- Environments are quite similar: you can think of them as shelves for such containers (i.e. they are in a way containers for containers)
- More literally, environments are places to store variables (or, R objects in general)
- However, they are themselves treated as a special type of variable
- This means that environments can be created, manipulated, assigned to symbols, etc.

What is an environment? (2)

- The 'official' R definition for an environment:

> *Environments can be thought of as consisting of two things. A frame, consisting of a set of symbol-value pairs, and an enclosure, a pointer to an enclosing environment. When R looks up the value for a symbol the frame is examined and if a matching symbol is found its value will be returned. If not, the enclosing environment is then accessed and the process repeated. Environments form a tree structure in which the enclosures play the role of parents. The tree of environments is rooted in an empty environment, available through emptyenv(), which has no parent. It is the direct parent of the environment of the base package (available through the baseenv() function). Formerly baseenv() had the special value NULL, but as from version 2.4.0, the use of NULL as an environment is defunct.*

What is an environment? (3)

- Environments bear most similarities with lists in that they can hold various types of data
- It turns out that environments (logically) are handled through syntax similar to that for lists
- Also, R has capabilities to coerce lists to environments and vice versa

The global environment

- This is the environment that we naturally work with (even without thinking explicitly of it)
- It is also called the **user workspace**
- When a function is called, R automatically creates an environment where the variables specific to that function are stored
- Also, when R starts, it loads several packages by default (each package contains a number of functions) and the corresponding environments are thus also loaded
- You can check that by clicking on the drop-down list in the top-right panel of RStudio titled “Environment” (maybe you would also like to load an additional package to see what happens)

Creating new environments

- Most of the time you don’t do that deliberately
- Anyway, if you need to create one, you type:

```r
env1 <- new.env()
```

- Using list notation then, variables can be assigned to this environment
- The latter can be done in two ways, e.g.:

```r
env1[["v1"]] <- c(1,2,3)
env1$v1 <- c(1,2,3)
```

- List notation then can also be used to call variables
- You can list all objects in an environment by using an additional argument to `ls()`:

```r
ls(envir = env1)
```

- Conversion between lists and environments is carried out by means of `as.list()` and `as.environment()`

Environment hierarchy

- Environments obey a tree structure
- On top is a special environment called the **empty environment**
- Therefore, any other environment is nested within another environment
- More generally you can think of environments as library rooms where each shelving system has shelves, and each shell contains books, each book then contains chapters, etc.; the empty room would be the empty environment

Functions

What are functions?

- R functions perform operations on other objects
- Like other programming languages, R provides options to create user-defined functions
- Thus, the base capabilities of the software are extended to tasks which have not found a formal implementation (yet)
- Moreover, the language allows to create functions of other functions which provides enormous flexibility and versatility

How do functions work?

- Functions have arguments (just like in mathematics)
- Those arguments are R objects
- The statements that the function contains in its body carry out the desired operations on objects
- The function returns other R objects which can be of any data type
- We already used many built-in functions
- We can explore their structure by just typing their name in the console (without the parentheses and without any arguments)
- For example, typing

```
sd
```

outputs the structure of the function that calculates the standard deviation of a variable

Components of functions

- Each function has three components: body, formals, and environment
- The body contains the code that is executed within the function
- For a known function, this components can be explored via the `body()` command
- For example:

```r
body(rnorm)
```

- Formals is the list of arguments that the function can take and which control how the function can be called

```r
formals(rnorm)
```

Components of functions (2)

- Each function has its own environment
- This environment contains all the variables that are defined by the function
- In order to check a function's environment, type for example:

```
environment(mean)
```

- The function and its environment are collectively known as a function's **closure**
- Note that `body()`, `formals()`, and `environment()` can also be used to modify the respective components of functions (just for info, avoid it)

Primitive functions

- They are an exception in that they do not have three components
- Found only in the base package
- For such functions, `body()`, `formals()`, and `environment()` hold the NULL value
- Examples of such functions are `sum(), sin(), cos()`, etc.

Creating functions

- Functions are (of course) created by assignment with the `function` command
- Let's take an example to illustrate the process
- We will create a function that calculates the volume of a cylinder:

```
cyl_vol <- function(r,h){
  pi*r^2*h
  }
```

- You can see the function appearing in the top-right panel; you can click on it to view its structure
- It is then called by means of (using here 2 as the radius and 5 as the height):

```
cyl_vol(2,5)
```

- The example shows the typical features of functions

Creating functions (2)

- As it is obvious, the keyword `function` is followed by a list of arguments (separated by commas)
- Arguments can be supplied in three ways:
  - Through a symbol (symbols)
  - Through a `symbol = expression` statement
  - Through the special formal argument `...`
- What follows is the body of the function surrounded by curly braces
- The body may contain valid R expressions
- In this example the function is named but there can also be anonymous functions

Creating functions (3)

- Two more examples to illustrate the two other approaches used in creating functions
- In the first one, we create a function which finds finds the nth root of an arbitrary number

```
root_cplx <- function(x, root = 2){
  x <- as.complex(x)
  x^(1/root)
  }
```

- If you don’t specify a second argument, it will automatically calculate the square root of the number
- The function is called as follows

```
root_cplx(-2,4) # or:
root_cplx(-2, root = 4) # or:
root_cplx(root = 4, -2)
```

Creating functions (4)

- In the second example, we use the special argument `...`
- This argument stands for other arguments (can be any number), i.e. it allows creating a function that has an arbitrary number of arguments
- Can be used for example to absorb a subset of all arguments into an intermediate function
- The latter can then be passed on to functions which are called after the function in question
- The `plot(x,y,...)` function is often used as an example of such a function (plotting is discussed in a later topic)

Some more examples of functions

- Standardize a variable:

```
standardize <- function(x, m = mean(x), s = sd(x)){
  (x - m) / s
  }
```

- Mode (most frequent observation):

```
Mode <- function(x) {
  ux <- unique(x)
  ux[which.max(tabulate(match(x, ux)))]
  }
```

- Note however that the latter is far from perfect
- For proper mode estimation, use the **modeest** package (see below)

Some more examples of functions (2)

- In the latter example, `unique()` returns a vector, data frame or array like `x` but with duplicate elements/rows removed
- `which.max()` finds the index of the maximum of a numeric (or logical) vector
- `tabulate()` takes the integer-valued vector bin and counts the number of times each integer occurs in it
- `match()` returns a vector of the positions of matches of its first argument in its second.
- However, this code has some flaws (what are they?)

Some more examples of functions (3)

- A better way to find the mode:

```
library(modeest)
mlv(var1, method = "mfv")
```

- `mlv()` calculates the most likely value
- The method used is `mfv`, i.e. most frequent value
- Other methods can be looked up in the documentation

Storing user-created functions

- There are several ways to preserve your work on creating functions for future use
- One of them, of course, is to write them in an R script which is saved before quitting R
- Another possibility is to save function objects which you created and which reside in your global environment

```r
save(function1, function2, file = "MyFuncs.R")
```

- Note that this will not be the usual text file but a binary file
- Next time you need your functions, you can use them after issuing:

```r
load("MyFuncs.R")
```

Storing user-created functions (2)

- If you still need your functions in a text file that you can edit, use:

```r
dump(c("function1", "function2"), file = "MyFuncs.R")
```

- This time, the names of functions should be in quotes
- Later on, you can load the functions back to R by:

```r
source("MyFuncs.R")
```

References

- Cotton, R. (2013): *Learning R*, O'Reilly, Ch. 6
- Peng, R. (2016): *R Programming for Data Science*, Leanpub, Ch. 15