At the heart of R

R Programming - HAX815X

Jean-Michel Marin

January 2026

Faculty of Sciences, University of Montpellier

Introduction

To understand computations in R, two slogans are helpful:

Everything that exists is an object

Everything that happens is a function call

John Chambers

- R is, in essence, a functional programming language
- Using functions is therefore the preferred way of developing a program in R
- It is therefore essential to understand how functions work and interact with the user's environment in order to develop reliable and reproducible programs

Introduction

- Building a project around a sequence of functions is a practice that reduces the risk of bugs (by better identifying their origin) and, above all, ensures more flexible and reproducible code
- This goes back to the *do not repeat yourself* paradigm
- This approach is of particular interest to us because it makes the code clearer, more robust and easier to replicate and evolve

Introduction

Nevertheless, adopting a functional programming approach in R is a good start, but it is not enough

- Theoretical background on objects in R
- Building first functions in R with early adoption of good practice
- Understanding how R handles objects in the different working environments available

Reminders about objects in R

- There are two key points to remember about how R deals with objects
  - How R associates objects and names
  - How R modifies objects

Reminders about objects in R - Distinguishing between names and values

The first important distinction is between an object and its name; at this stage, you should be used to assignment

```
x <- c(1, 2, 3)
```

You might be tempted to read *I'm creating an object called* x *containing the values 1, 2 and 3*, in reality, R proceeds as follows

- R creates a vector of values c(1, 2, 3)
- R binds this vector to a name, in this case x

Reminders about objects in R - Distinguish between names and values

- In other words, the object/value doesn't have a name; it's the name that has a value
- The <- assignment operator therefore creates a link between a name and a value. A name is a reference to a value: this avoids the code below `y <- x` creates a copy (using memory) of an existing object (x)
- This assignment simply creates an additional reference to access the vector c(1,2,3), by linking this vector to the name y

Reminders about objects in R - Distinguish between names and values

On the other hand, R duplicates the object when you modify a name that refers to it. The code below links x to y and then modifies y

```
x <- c(1, 2, 3)
y <- x
y[[3]] <- 4
```

Reminders about objects in R - Distinguish between names and values

- You might think that this operation changed x, but it didn't:

```
x
## [1] 1 2 3
```

- While the value associated with y has changed, the original object has not
- R has simply created a new object the vector c(1,2,4), and given it the name y

Reminders about objects in R - Copying objects in R

In principle, R objects cannot be modified (immutable); it may seem surprising to talk about the immutability of R objects, since you can apparently add elements to an R object

```
a <- list()
a[[1]] <- 'I'm filling in my list
a
## [[1]]
## [1] 'fill in my list
```

Reminders about objects in R - Copying objects in R

In reality, R has temporarily duplicated a (copy) and then modified this object before reassigning it to the name a (modify)

This behaviour is called copy-on-modify

This approach makes the R language very flexible, allowing, for example, the class of an object to evolve (from a reassignment)

```
v1 <- c(1, 2, 3)
tracemem(v1)
v1[1] <- 3
v2 <- v1
v2[1] <- 0
```

Reminders about objects in R - Copy objects in R

For example, the following code transforms the object linked to vector name a into data.frame :

```
a <- c(1,2,3)
a <- data.frame('a' = a)
class(a)
## [1] 'data.frame'
```

Reminders about functions

A function can be broken down into three parts

- Arguments (ex: formals(lm)): the list of arguments that control the call to a function
- Body (ex: body(lm)): the function's internal commands
- Environment (ex: environment(lm)): the context which determines how the function finds values associated with the names mentioned

Reminders about functions

- In R, these names can refer to any type of object: data, variables, functions, packages, etc.

- While the arguments and the body of the function are defined explicitly, the environment is defined implicitly, depending on the context in which the function is invoked

Reminders about functions

```
f02 <- function(x, y) { # A comment
x + y
}
formals(f02)
## $x
##
##
## $y
```

Reminders about functions

```
body(f02)
## {
## x + y
## }
environment(f02)
## <environment: R_GlobalEnv>
```

Reminders about functions - Scoping a function

- As we have seen, assignment is the act of associating a name with a value
- Scoping is the action of finding a value associated with a name
- For example, what will the following code return: 10, 20, 25?

```
x <- 10
g01 <- function() {
x <- 20
x
}
g01()
## [1] 20
```

Reminders about functions - Hidden names

- The basic principle of scoping is that names defined inside a function mask names defined outside

- The following code shows an example of this rule:

```
x <- 10; y <- 20
g02 <- function() {
x <- 1; y <- 2
c(x, y)
}
g02()
## [1] 1 2
```

Reminders about functions - Hidden names

- If R can't find a name in a function, it will look for it in an environment one level above

```
x <- 2
g03 <- function() { y <- 1; c(x, y)
}
g03()
## [1] 2 1
# The command below does not change the previous value of y
y
## [1] 20
```

Reminders about functions - Hidden names

- This rule applies in the same way to nested functions: R searches first in the function's environment, then in the environment of the function above it, and so on up to the global environment

- If it still cannot find the name it is looking for (for example, a sum function), R will end up searching the list of loaded packages

Reminders about functions - Hidden names

- A good example of the scope of a function is as follows

```
# Take the vector a
a <- c(1,2,3)
g11 <- function() { if (!exists('a')) {
a <- 1 } else {a <- a + 1}
a
}
g11()
## [1] 2 3 4
g11()
## [1] 2 3 4
```

Reminders about functions - Hidden names

- Why does g11() always return the same value?
- Because every time a function is called, a new environment is created to host its execution
- A function is therefore incapable of knowing the names that may have been created by a function passed without return

Reminders about functions - Scope and lazy evaluation

- Scope allows objects to be linked together logically, but does not allow you to determine when this link will take place (this is known as lazy evaluation: R is given a plan for linking the objects, but is not yet told when the command will be executed)

- R associates names with values when the function is executed, not when it is created

Reminders about functions - Scope and lazy evaluation

In lazy evaluation, what allows R to bind objects and environments together is the promise.

This consists of three elements:

- An expression, such as `x + y`, which will lead to future calculations
- An environment, where the expression will be evaluated, i.e. the environment where the function is evaluated
- A value that is calculated and stored in memory within the environment (to avoid doing the same calculation twice if the same plan is called twice in the same environment)

Lazy evaluation has two major consequences for the way R works

Reminders about functions - Scope and lazy evaluation

Firstly, the output of a function can differ depending on the objects outside the function's environment

These objects are sometimes called global variables because they affect all functions, as opposed to local variables which are specific to a function, and only exist in the execution environment of that function

Reminders about functions - Scope and lazy evaluation

Here's a simple example

```
g12 <- function() x + 1
x <- 15
g12()
## [1] 16
x <- 20
g12()
## [1] 21
```

- Changing x (a global variable) therefore changes the function's output
- One way of quickly finding out which global variables can have an effect on the output is to use the `codetools::findGlobals()` function

This function lists all the external dependencies called in a function

Reminders about functions - Lazy scope and evaluation

- Secondly, the arguments to a function in R are only evaluated when it is necessary to know their value in order to continue executing the program

- For example, the following code will not cause an error because x is never used and is therefore never evaluated

```
h01 <- function(x) {
10
}
h01(stop('This is an error!'))
## [1] 10
```

Reminders about functions - Scope and lazy evaluation

This is very useful because it allows you to create functions that can perform very different operations depending on the type of argument, for example, create a function `add(x,y)` that

- adds x and y if x and y are numeric;
- concatenates x with y if x and y are strings;
- returns the error x and y are not of the same type' if x and y are not of the same type

Reminders about functions - Lazy scope and evaluation

```
add <- function(x,y){
if ((class(x) == 'numeric') & (class(y) == 'numeric')){
x + y
} else if ((class(x) == 'character') & (class(y) == 'character')){
paste0(x,y) } else {
stop(''x' and "y" are not of the same type') }
add(1,2)
## [1] 3
add('a', 'b')
## [1] 'ab'
add(1, 'b')
## Error in add(1, 'b'): 'x' and 'y' are not of the same type
```

Defining functions in R - A simple example

Suppose you have a data table that uses the -99 code to represent missing values. We want to replace all the -99s with NAs

```
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6] ; df
## a b c d e f
## 1 7 5 -99 2 5 2
## 2 5 5 5 3 6 1
## 3 6 8 5 9 9 4
## 4 4 2 2 6 6 8
## 5 6 7 6 -99 10 6
## 6 9 -99 4 7 5 1
```

Defining functions in R - A simple example

A first draft of code could take the following form

```
df2 <- df
df2$a[df2$a == -99] <- NA
df2$b[df2$b == -99] <- NA
df2$c[df2$c == -99] <- NA
df2$d[df2$d == -99] <- NA
df2$e[df2$e == -98] <- NA
df2$f[df2$g == -99] <- NA
```

Defining functions in R - A simple example

- The code is long and repetitive, which makes it difficult to read
- The code is highly dependent on the data structure (name and number of columns) and must be adapted as soon as the structure changes
- A human error has been introduced into the code, which is difficult to detect, in the instruction

```
df2$e[df2$e == -98] <- NA
```

Defining functions in R - A simple example

- As the task here is identical for all the columns, we'd like to set up a general structure that would make the code more concise
- To do this, we're going to build a function using R. This function will take the following form

```
fix_missing <- function(x) {
x[x == -99] <- NA ; x }
```

Defining functions in R - A simple example

- The apply family of functions is designed to apply the same function to several objects
- These functions are part of the basic R language and are known as functionals because they take a function as an argument
- The most powerful of the apply functions is the `lapply` function, which stores the result in the form of a list
- It is preferable to use `lapply` rather than the other apply functions (to avoid data type conversion problems)

Defining functions in R - A simple example

Here's a very simple example of using `lapply`

```
add1 <- function(x){ y <- x + 1 ; return(y) }
x <- list(1, 2, 3, 4)
lapply(x, add1)
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] 5
```

Defining functions in R - A simple example

- We're now going to use lapply to improve our code
- It's possible to apply lapply to a dataframe because a dataframe is actually a list of vectors (the columns of the dataframe)
- However, as lapply returns a list, we need to use a little trick to ensure that the function's output takes the form of a dataframe and not a list. Instead of assigning the result in the form `df <- lapply(.....)` we'll do `df[] <- lapply(...)`

Defining functions in R - A simple example

```
df2 <- df
fix_missing <- function(x) {
x[x == -99] <- NA ; x }
df2[] <- lapply(df2, fix_missing)
```

Defining functions in R - A simple example

- This approach to composing functions is very powerful and explains the great flexibility of the R language
- Writing simple functions and composing them into one or more master functions is also very useful in collaborative work because it allows you to break down complex processing into simple tasks and understand the flow better

Creating complex functions

- What do we do now if we want to authorise values other than -99?
- Our previous function assumed that the value -99 designated missing values

```
fix_missing <- function(x, na.value) {
x[x == na.value] <- NA ; x }
```

Creating complex functions

```
df2 <- df
df2[] <- lapply(df2, fix_missing, na.value = -99)
df2
## a b c d e f
## 1 7 5 NA 2 5 2
## 2 5 5 5 3 6 1
## 3 6 8 5 9 9 4
## 4 4 2 2 6 6 8
## 5 6 7 6 NA 10 6
## 6 9 NA 4 7 5 1
```

Creating complex functions

- Let's imagine that not all variables are coded in the same way
- To simplify things, let's reduce the problem to the variables a, b and c. Let's imagine that the missing values are indicated by the values 1 to 3 for variable a, by the values 4 to 6 for variable b and by the values 7 to 10 for variable c

Create complex functions

```
fix_missing2 <- function(x, na.value) { x[x %in% na.value] <- NA ; return(x) }
codes_na <- list('a' = 1:3, 'b' = 4:6, 'c' = 7:9)
df2 <- df
df2[names(codes_na)] <- lapply(names(codes_na), function(variable)
fix_missing2(df2[,variable], na.value = codes_na[[variable]]))
df2
## a b c d e f
## 1 7 NA -99 2 5 2
## 2 5 NA 5 3 6 1
## 3 6 8 5 9 9 4
## 4 4 2 2 6 6 8
## 5 6 7 6 -99 10 6
## 6 9 -99 4 7 5 1
```

Creating complex functions

- We have shown how the lapply function can be used to generalise the application of a function
- `purrr::map` behaves very similarly to lapply; we transform the data with reshape2 and apply one function per group.

```
df2_map <- df2
df2_map[] <- purrr::map(df2_map,fix_missing, na.value = -99)
```

Use Rstudio's debugging tools

- A function can return an error for two reasons, with different interpretations
- The error is foreseen by the function (bug by design): it is possible to introduce logical conditions into a function that return an error if the execution conditions of the function are not verified (stop instruction)
- Defensive programming is not a compulsory practice in a team, but rather a principle of caution

Use Rstudio’s debugging tools

- Errors are unintentional: unanticipated behaviour of a function, an error in the type of variable, and so on

These are frequent cases; in this case, the general principle is to try to identify the error from typical cases (the tests in a package are designed to automate this search) so that you can progressively improve a function

Use Rstudio's debugging tools

There are three tools you can use to debug

- Rstudio Error Inspector and use Show `Traceback()` to list the sequence of function calls that led to an error
- Use Rerun with Debug and the `options(error = browser)` command to execute the code that generated the error step by step and analyse the code execution environment
- Insert a breakpoint in a function or use the `browser()` command to open a session at a point determined by the function creator

Use Rstudio's debugging tools

This tool is sometimes called call stack. It shows the stack of function calls that generated the error

For example, here is a sequence that generated an error

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) 'a' + d
f(10)
```

Use Rstudio's debugging tools

Clicking on the Show Traceback button shows the sequence of functions executed by the `f(10)` command

This sequence is read from bottom to top: you start with the highest function call (`f(10)`) and work backwards through the sequence of functions called

This makes it possible to identify that the origin of the error is the `i(.)` function

If the code was read from the source() command, the traceback will also display the location of the function in the file

Use Rstudio's debugging tools

`traceback()` shows when and in which function the error occurred, but does not explain why the error occurred

Clicking on the *Rerun with debug* button opens an interactive window that allows you to pause the execution of a function and interactively explore its state

The debugger can be used to analyse the different environments, to ensure that the function execution environment is accessing the desired objects (packages, parameters, dataframe, functions, etc.), and that the nested environments are accessing the expected objects

Use Rstudio's debugging tools

There is also a bar in the console

- Next: executes the next step in the function
- Step into: works like next but if the next line is a function, it will send you into the function to progress line by line
- Finish: finishes executing the current loop or function
- Continue: returns you to the upper environment; this is useful if the problem with the function has been fixed and you want to test whether it has been corrected
- Stop: exit the debugger

Use Rstudio's debugging tools

You can insert a breakpoint in a function.

When R reaches the breakpoint, execution of the function will be stopped and the same debugger as in Rerun with debug will open at that point

The advantage of a breakpoint is that it allows the developer to take stock of the execution of a function at a specific point in time

There are two ways of introducing a breakpoint

- click to the left of the line number of a script - a red dot will appear
- enter the `browser()` command in the function code at the point where you want to stop it

Notions about environments

The environment is the structure that provides the scope

An environment is in fact a named list of objects with slightly more restrictive rules than a traditional list

- Each name must be unique
- Names in an environment have no order
- An environment has one parent
- Copying environments does not follow the copy-on-modify principle

Notions about environments

- An environment is used to link a set of names to a set of values

- Some functions in the rlang package are dedicated to environments. For example, let's create a new environment e1 containing 4 objects

```
e1 <- rlang::env(
a = FALSE,
b = 'a',
c = 2.3,
d = 1:3
)
e1
## <environment: 0x10f17b200>
```

Notions about environments

- If you want to know the characteristics of the environment, you can use the rlang::env_print function

```
rlang::env_print(e1)
## <environment: 0x10f17b200>
## Parent: <environment: global>
## Bindings:
## - a: <lgl>
## - b: <chr>
## - c: <dbl>
## - d: <int>
## names(e1)
## [1] 'a' 'b' 'c' 'd'
```

Notions about environments

- There are several environments in R which communicate with each other using hierarchical relationships
- The current environment, which can be displayed using `rlang::current_env()`, is the environment in which commands are currently being executed
- When you are experimenting with R, the execution environment is the global environment (.GlobalEnv)
- The global environment is the workspace because it is where all objects are stored

Notions about environments

- As every function is called with a new environment, the execution environment is distinguished from the global environment as soon as a function is called

- In the following code, we can see that the two calls to the `a()` function generate two different environments

```
print(rlang::current_env())
## <environment: R_GlobalEnv>
a <- function() print(rlang::current_env())
a()
## <environment: 0x10a3f4c00>
a()
## <environment: 0x10ae35108>
```

Notions about environments

- Every environment has a parent, which is itself an environment
- The parent is used to define the scope: if a name is not found in the current environment, R will search in its parent, then in its parent's parent, and so on
- The parent of an environment can be displayed using the `rlang::env_parent` function or the basic parent.env function

```
rlang::env_parent(e1)
## <environment: R_GlobalEnv>
parent.env(e1)
## <environment: R_GlobalEnv>
```

Notions about environments

- The set of parent environments of an environment are called ancestors
- The principle of name masking mentioned above means that the name of an object taken into account by `current_env` is always the first name found in the ancestor environments
- The traditional assignment, `<-`, always creates a variable in the current environment

Notions about environments

The super assignment, `<<-`, never creates the variable in the current environment but, instead, modifies the variable found in the most recent ancestor

```
x <- 0
f <- function() { x <<- 1 ; return(2) }
f()
## [1] 2
x
## [1] 1
```

If `<<-` does not find a variable existing under this name, it will be created in the global environment

Package environment and search path

- All packages attached by the `library()` or `require()` command become a parent environment of the global environment
- The immediate parent of the global environment is the last environment attached, the next parent is the one attached before that, and so on
- The principle of function inheritance applies in the same way as explained above for environments
- If a function name is not found in the global environment, R will search for this name up to the first parent where this name refers to an object
- This is a common source of error when sharing code

Package environment and search path

In the event of conflicts between function names, the priority will be Global environment > Last package loaded > Second last package loaded > ...

So, if two people have loaded the same packages but in a different order, they may not obtain the same result (an error vs. an output)

A typical case is the select function

An example of an environment problem with packages

The MASS and dplyr packages offer a function with the same name but a different purpose

```
data(iris) ; library(MASS) ; library(dplyr)
## Attaching package: 'dplyr'
## The following object is masked from 'package:MASS': select
## The following objects are masked from 'package:stats': filter, lag
## The following objects are masked from 'package:base': intersect,
## setdiff, setequal, union
iris %>% select(Sepal.Length) %>% head()
## Sepal.Length
## 1 5.1
## 2 4.9
## 3 4.7
## 4 4.6
## 5 5.0
## 6 5.4
```

Package environment and search path

```
detach('package:dplyr')
detach('package:MASS')
library(dplyr)
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats': filter, lag
## The following objects are masked from 'package:base': intersect,
## setdiff, setequal, union
library(MASS)
## Attaching package: 'MASS'
## The following object is masked from 'package:dplyr': select
iris %>% select(Sepal.Length) %>% head()
## Error in select(., Sepal.Length): unused argument (Sepal.Length)
```

How to use packages properly

While we regularly think about cleaning up our environment, or even starting from a clean environment with the `lm(list = ls())` command, we don't necessarily think about cleaning up the packages loaded

This is a real issue in terms of replicability and code robustness

- The best practice is to systematically use functions from packages in the form `pkg::function`, for example `dplyr::select`
- This removes any ambiguity from the function: R knows that it is necessary to fetch the select function from the `dplyr` namespace
- Regardless of the packages loaded or the functions in the environment, the `dplyr::select` command will always respond to the developer's intention

How to use packages properly

```
library(dplyr)
library(MASS)
iris %>% dplyr::select(Sepal.Length) %>% head()
## Sepal.Length
## 1 5.1
## 2 4.9
## 3 4.7
## 4 4.6
## 5 5.0
## 6 5.4
detach('package:MASS')
detach('package:dplyr')
```

How to use packages properly

- Note the difference between an attached package and a loaded package:
  - A package is loaded automatically as soon as one of its functions is accessed with `dplyr::select` (for example)
  - Loading is a flexible dependency: it does not cause the package to become a parent, but still makes it easy to find help for a function
- A package is attached when you use `library(pkg)` or `require(pkg)`

How to use packages properly

By going backwards through all the parent environments (the search path), you can see the order in which the packages were attached.

```
search()
## [1] '.GlobalEnv' 'package:stats' 'package:graphics'
## [4] 'package:grDevices' 'package:utils' 'package:datasets'
## [7] 'package:methods' 'Autoloads' 'package:base'
library(rlang) ; library(dplyr)
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats': filter, lag
## The following objects are masked from 'package:base': intersect,
## setdiff, setequal, union
search()
## [1] '.GlobalEnv' 'package:dplyr' 'package:rlang'
## [4] 'package:stats' 'package:graphics' 'package:grDevices'
## [7] 'package:utils' 'package:datasets' 'package:methods'
## [10] 'Autoloads' 'package:base'
```

Namespaces

- The aim of namespaces is to ensure that each package functions in the same way regardless of the other packages attached by the user
- Let's take an example using the `sd()` function to calculate standard deviations
- `sd()` is calculated from the `var()` function (variance)
- Is sd affected if a function is called var in the global environment or in the attached packages?
- R avoids this problem by taking advantage of the difference between the environment of a function and the environments linked to this function

Namespaces

Each function integrated into a package is associated with two environments:

- package environment: the external interface of a package; this is how R finds a function in an attached package or with `dplyr::select` for example
- namespace environment: internal interface of the package. It controls how a function finds internal objects (functions, variables...)

Namespaces

- Each namespace has the same set of ancestors
- Each package contains a `NAMESPACE` file which contains the link between the package and its dependencies
- Because importing all the `R` base functions would be tedious, the base functions namespace is a parent of any namespace

Namespaces

- The last parent, i.e. the first in the line of ancestors, is the global environment
- So, if a package requires a myvar variable and cannot find it in any parent, it will consider that it is a global variable and will therefore look in the global environment

Namespaces

- Let's go back to the `sd` example
- You can check the environment in which `sd` is found

```
environment(sd)
## <environment: namespace:stats>
# pryr::where('sd')
body(sd)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
## na.rm = na.rm))
```

Namespaces

- The definition of `sd()` therefore uses `var()`

- What happens when we create our own version of `var`

```
x <- seq_len(10)
sd(x)
## [1] 3.02765
var <- function(x) 'I do something really different'
sd(x)
## [1] 3.02765
var(x)
## [1] 'I do something really different'
```

- This does not affect `sd`