Functions and programming paradigm of R

Antonio Gasparrini

01 November 2023

Contents

This session provides a more technical introduction to the R software, presenting it as a *functional programming* language. From a practical perspective, this means that everything in R is obtained by the application of pre-defined functions to objects, thus creating new objects. The analysis is therefore carried out by a series of steps, flexible and not predetermined, entirely planned by the user. The results can be considered as a side effect of this process. The first section reveals how, in R, functions are objects themselves, and illustrates the operational procedures of this software. The next sections discuss the concepts of function call and evaluation, describe how a function actually works, and then introduce the concept of object *class*. The last section loosely illustrates the idea of *object-oriented programming* and its advantages.

Function objects

The R software largely follows the paradigm of *functional programming*, where programs are defined by composing and applying functions. Here functions are pre-defined mathematical statements that do not change and are not influenced by the environment and data. Specifically in R, *everything is an object*, and functions are no exceptions, being objects identified by names. For example:

```
rnorm
```

```
## function (n, mean = 0, sd = 1)
## .Call(C_rnorm, n, mean, sd)
## <bytecode: 0x000001743d342438>
## <environment: namespace:stats>
```

```
mode(rnorm)
```

```
## [1] "function"
```

The function `rnorm()` is an object of mode `function` available in the package STATS. Packages are in fact coherent environments providing a collection of pre-defined functions and data.

Everything in R is carried out by evaluating functions to objects, deriving other objects. Simple arithmetic expressions seem to represent an exception, as apparently they do not involve functions. In fact, operators such as '+', '/' or ':' are themselves functions with special names. For example, the simple computation `3 + 2/3` can be written as:

```
'+'(3, 2/3)
```

```
## [1] 3.666667
```

where `3` and `2/3` are simply arguments of the function `'+'()`. Note the use of quotes, as the plus symbol has been assigned to a function and works as an operator. A function object is not different from other types of objects: it can be printed, re-assigned, or used in the arguments of other functions.

The operational procedure of R is probably easier to understand now: objects produced by the applications of functions are used as arguments of other functions, producing new objects. In this scheme, results are simply intermediate products of this process, and can be printed or saved as external files, or used in further computations. This flexible approach underpins a computational scheme which is not pre-determined in a series of commands. The same results can be obtained by alternative steps, which can be tailored to your knowledge of R functions and of R in general.

Function call and evaluation

Functions are used within commands through *function calls*, namely the definition of expressions such as those seen before, where the name of the function is associated with arguments provided in round brackets. A function call is an object itself (of course): you can think of it as a statement which is used in the *evaluation* of the function.

The process of function evaluation can be clarified with an example. Let's see what happens with the function `rnorm()`, seen earlier:

```
z <- 3
obs <- rnorm(mean=z, 5)
```

Here R first matches the arguments of the function with the values provided by the user, or with default values. The argument `mean` is matched first with the value `3`, returned by the object `z`. Then, the constant value `5` is matched with the first unmatched argument, as no name is provided. In this case, the first argument `n` of `rnorm()`. The last argument `sd`, not provided in the call, is given the default value `1`. Now, R evaluates the call, and returns as a final result a numeric vector, which is assigned to the name `obs`.

A function call can be used directly as an argument of another function, without the need to save intermediate objects. For example:

```
mean(rnorm(10))
```

```
## [1] 0.4797248
```

The possibility of defining expressions where calls to functions are nested each other in such a way provides a flexible and efficient programming approach, which is one of the main advantages of using R.

All the objects used as arguments in a function call need to be present in some of the environments, for example, the global environment or one of the packages loaded in the session. The matching of each argument follows the *search path*, provided by the function `search()`:

```
search()
```

```
##  [1] ".GlobalEnv"        "package:foreign"   "package:boot"      "package:rmarkdown" "package:tools"
##  [7] "package:Epi"       "package:patchwork" "package:ggplot2"   "package:knitr"     "tools:rstudio"
## [13] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"  "package:methods
## [19] "package:base"
```

The R evaluator searches the environment at position 1 (the global environment), and if the object associated with an argument in the call is not there, it then searches the environment at the next position in the hierarchy, and so on. The package BASE is the last to be searched. If the object is not found, the evaluator returns an error. Packages loaded in the session are by default given position 2. The method outlined above prevents conflicts between objects belonging to different environments, for example functions with the same name in different packages.

Although what given here is just a rough (and not entirely formally correct) description of function evaluation, it should help you understand the use and capability of the software.

The class

All objects in R belong to a *class*. The class is a generic definition of the object that identifies it in terms of a common set of attributes, independently from its pure data structure. More broadly, the class provides a blueprint for objects that defines a set of properties and computational methods.

As an illustrative example to clarify the concept of class, let's create a factor from a corresponding numeric vector:

```
myvec <- c(3,1,3,-2,7,1,1)
myfct <- factor(myvec)
```

Let's now check first the mode of these two objects:

```
mode(myvec) ; mode(myfct)
```

```
## [1] "numeric"
```

```
## [1] "numeric"
```

and then their class:

```
class(myvec) ; class(myfct)
```

```
## [1] "numeric"
```

```
## [1] "factor"
```

Here, the function `mode()` simply shows that both objects are composed of numeric elements, while the class also indicates the specific nature of the object, namely how the elements are combined. Specifically, the class `factor` is defined, among others, by the attribute `levels` identifying the categories. The class is a generic definition: objects can be thought of as instances of different classes. New classes can be created and assigned to specific objects.

Object-oriented programming

R is also described as an *object-oriented* language, a feature which complements its functional style. Such a programming approach, consistently with what was described earlier, emphasizes the key role played by objects in R.

Roughly speaking, we can say that in the object-oriented paradigm, the program can be described as a collection of interacting objects of different types. Each type of object not only defines a data structure but also establishes the operations which can be applied to such structure. In R, the type of object is determined by its class. Following its programming style based on function evaluation, this means that a function call may generate different computations depending on the class of the objects it is applied to.

Let's see an example using the objects `myvec` and `myfct` created in the previous section. As discussed, they have the same mode (numeric) but different classes, as factors are numeric vectors with a `levels` attribute. Now, let's try to apply the function `summary()` to both of them:

```
summary(myvec)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      -2       1       1       2       3       7
```

```
summary(myfct)
```

```
## -2  1  3  7
##  1  3  2  1
```

Surprisingly, while the objects are composed of the same underlying data, the results are different. A call to `summary()` produces a vector with statistics related to the distribution of the numeric vector `myvec`, but when applied to the factor `myfct` it returns a table with the number of occurrences of each level. The same function applies therefore different computations and produces different outputs depending on the classes of the objects used as it arguments.

The concept of class becomes less obscure now. The class is an abstract definition of the object, and such definition pertains to its interaction with objects of different classes and the computations with specific functions. From a practical perspective, this approach simplifies your life with computing in R: you can simply apply the same function to different objects, and expect different results coherently with the definition of the function and the objects themselves. This feature makes things easier, in particular with software based on a command-line interface, as the syntax becomes clearer and conceptually simpler.