Programming with R

Antonio Gasparrini

01 November 2023

Contents

In this session, we introduce methods for using R more flexibly, illustrating how non-standard computational tasks can be addressed through *programming*. The idea of programming is to directly develop a set of instructions to enable the software to complete a certain task. In R, the user can easily program routines not implemented in existing functions, exploiting the flexibility of the language. High-level programming tools can be developed using the usual syntax and a limited set of specific language constructs. The topic of programming in R is vast and cannot be comprehensively introduced in these introductory notes. Here we provide a selection of the most important features. In particular, the first section highlights the blurred distinction between programming and standard computing in R. The next section introduces programming examples using matrix algebra, while the third section covers loops and conditional expressions. The last two sections introduce a fundamental topic of programming, which is how to create new functions and the rules governing them.

Programming vs computing in R

An important advantage of R, when compared with other software or languages, is that complex and sophisticated programming tasks can often be performed using a relatively simple code, and in fact, most of the times, adopting the same syntax used in standard computations. This aspect makes it difficult to clearly define and separate programming from more basic computing in R.

Loosely, we can define programming as writing a set of instructions to address specific computational tasks. More practically, programming involves developing routines to perform non-standard computations, not otherwise implemented in existing functions. Nonetheless, several common tasks involving simple data management and analysis steps already match this definition. For instance, when we create regular sequences and apply them as indexing vectors for subsetting objects, we can easily say that we are programming. As another example, we can write expressions to produce set of strings used to rename records or variables of a dataset, which again is programming. As a final example, we can exploit the flexibility of the functions

in the apply family to define complex computations across dimensions of matrices and data frames or categories of a variable. While involving complex programming aspects, these are common data management and analytical procedures

These examples emphasize how common data management steps often require complex computations, and that in R these can be programmed directly instead than requiring pre-implemented commands. In the rest of the chapter, we introduce some important programming topics, anticipating that the subject is too broad to provide a comprehensive illustration.

Programming with matrix algebra

R is an excellent platform for computational linear algebra and incorporates well-tested and fast routines for producing a wide range of algebraic computations with a simple and clear code. In particular, algebraic operations involving matrices are commonly required in several statistical programming tasks. Here we illustrate how matrix algebra can be used to directly program tasks that are normally produced using existing R functions.

As a first example, we perform a linear regression model performed using the dataset BIRTHS of the package EPI. We first load the package (which might also be installed, if not) and the dataset:

```
library(Epi)
data(births)
```

We can now fit a linear regression between birth weight as the response variable and maternal age and hypertension as predictors (see `(help(births)` for details), using the function `lm()`:

```
lmobj <- lm(bweight ~ matage + hyp, data=births)
```

The fitted model is saved in the object `lmobj`, from which is possible to extract the estimates of coefficients with related standard error as well as test statistics such as *t* and *p*-values. These are usually printed using the `summary()` function, but also stored in the component `coefficients` of the returned object:

```
summary(lmobj)$coef
```

```
##                 Estimate Std. Error      t value     Pr(>|t|)
## (Intercept) 3200.98765594 245.555197 13.035715357 1.284042e-33
## matage        -0.06104304   7.140662 -0.008548653 9.931827e-01
## hyp         -430.73946765  79.198327 -5.438744551 8.431311e-08
```

Below, we reproduce the results 'by hand', fitting the model using only matrix algebra. First, we need to recreate the response vector and design matrix. Usually, this is done internally in `lm()` by calling the functions, `model.frame()`, `model.matrix()`, and `model.response()`, but in this case it is straightforward:

```
y <- births$bweight
X <- cbind(int=1, matage=births$matage, hyp=births$hyp)
head(X, 3)
```

```
##      int matage hyp
## [1,]   1     34   0
## [2,]   1     30   0
## [3,]   1     35   0
```

Then, we can easily derive the estimates by programming the estimators used in linear models. Specifically, the estimates of the coefficients can be computed as $\widehat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$, and then compared to the output of the `lm()` call:

```
(beta <- drop(solve(t(X)%*%X) %*% t(X)%*%y))
```

```
##           int        matage           hyp
## 3200.98765594   -0.06104304 -430.73946765
```

```
coef(lmobj)
```

```
##   (Intercept)        matage           hyp
## 3200.98765594   -0.06104304 -430.73946765
```

The function `drop()` is called to drop the matrix dimensions and return the output as a vector. Although the estimation routine internal to `lm()` is based on a more general procedure, for instance involving more stable QR decompositions (see `help(qr)`), this example highlights the extreme flexibility and potential of R in programming regression routines using a short and simple code.

In a second example, we extract the standard errors and then the test statistics. We need first to derive the variance-covariance matrix of the estimated coefficients of the more complex model, following the formula $V(\hat{\beta}) = \sigma^2(\mathbf{X}^\intercal\mathbf{X})^{-1}$, where σ is the residual error. The latter is computed as the sum of the squared residuals divided by the residual degrees of freedom:

```
(df <- nrow(X)-3)
```

```
## [1] 497
```

```
(sig2 <- sum((births$bweight - X%*%beta)^2) / df)
```

```
## [1] 384976.2
```

```
(V <- sig2 * solve(t(X)%*%X))
```

```
##              int      matage         hyp
## int    60297.355 -1740.29927 -2142.35431
## matage -1740.299    50.98905    36.41508
## hyp    -2142.354    36.41508  6272.37495
```

These results are identical to those returned using `summary()` and `vcov()` functions with the fitted regression object. Now, we can extract the standard errors as the square root of the diagonal of `V`:

```
(stderr <- sqrt(diag(V)))
```

```
##        int     matage        hyp
## 245.555197   7.140662  79.198327
```

These are exactly the same as those shown from the model output above. Finally, we can extract the t test statistics and related *p*-values. In these simple tests, the former are the ratio between the estimated coefficients and standard errors, while the latter can be computed as the related probability of the *t* distribution given the statistics and the degrees of freedom (twice given the two-sided test) using the function `pt()` (see the related help page):

```
(tstat <- abs(beta / stderr))
```

```
##          int       matage          hyp
## 13.035715357  0.008548653  5.438744551
```

```
(pval <- pt(tstat, df, lower=F)*2)
```

```
##          int       matage          hyp
## 1.284042e-33 9.931827e-01 8.431311e-08
```

Again, the results are identical to the model regression output. This exercise exemplifies the flexibility of R in programming relative complex statistical tasks with only a few lines of code, using a simple and general syntax.

Loops and conditional expressions

Similarly to other programming languages, also in R it is possible to define *control statements* (also called *control-flow constructs*). These are used to create *loops* or *conditional expressions*, which are standard programming tools often used also in common computing and programming tasks.

Loops are used to specify a set of operations that must be repeated a certain number of times. The main loop construct in R is `for()`, with a syntax of the type `for(var in seq) exprs`. Here the argument `var` is a name assigned at each iteration of the loop to the elements of `seq`, which can be a vector/list or an expression evaluating to such objects. The length of `seq` defines a fixed number or iterations. The object `var` is usually included in the expression(s) `exprs`, evaluated in the loop. A simple example:

```
for(i in 1:3) {
  i2 <- i^2
  print(i2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
```

Here curly brackets '`{}`' are used to define multiple expressions within each iteration. However, they can be avoided if `exprs` can be written as a single expression. Modifying the previous example:

```
for(i in 1:3) print(i^2)
```

```
## [1] 1
## [1] 4
## [1] 9
```

In the examples above, the name `i` is assigned in a loop to the elements of the sequence 1 to 3. However, these elements can also be more complex objects in a list of names in a character vector.

Another important type of control statements are conditional expressions. These are formulated with the `if()` construct, with basic syntax `if(cond) exprs`. The logical element or expression `cond` defines a logical condition that, if `TRUE`, causes the expression(s) `exprs` to be evaluated. For instance:

```
x <- y <- -2
if(x<0) y <- 3
y
```

```
## [1] 3
```

Here the logical expression `x<0` returns `TRUE`, so the single expression `y <- 3` is evaluated and the value of `y` changed to 3. If the logical condition returns `FALSE`, the expression remains unevaluated and nothing changes.

The construct `if()` can be optionally used in combination with `else`. The syntax is extended to `if(cond) exprs1 else exprs2`, where the logical condition selects the evaluation of expression(s) `expr1` (if `TRUE`) or `expr2` (if `FALSE`). Modifying the previous example:

```
(y <- if(x<0) 3 else -2)
```

```
## [1] 3
```

However, it is important that the `else` construct is not included in a newline if the previous expression is syntactically complete. This is particularly relevant if `exprs1` contains multiple expressions. Again, the use of the curly brackets can help avoid errors. For example:

```
if(x>0) {
  y <- 4
```

```
  z <- 1
} else {
  y <- 5
  z <- 3
}
y ; z
```

```
## [1] 5
```

```
## [1] 3
```

There are other loop constructs available for programming with R. The `while()` construct is based on a condition, while `repeat` and `break` are frequently used together, possibly in combination with conditional expressions involving `if`. Some (silly) examples:

```
x <- 2
while(x<20) x <- x^2
x
```

```
## [1] 256
```

or similarly:

```
x <- 2
repeat {
  if(x>=20) break
  x <- x^2
}
x
```

```
## [1] 256
```

An additional construct `next` can be used in the same way as `break`, but instead than terminating the loop, it only stops the evaluation of the expression(s) in the current iteration and starts from the next one. However, `while()` and `repeat` are less commonly used to produce loops: the main reason is that the number of iterations is not fixed, and in the case of errors this may produce endless loops (for instance by setting `x <- 1` as the starting value in the example above).

In R, control-flow constructs are replaced when possible by vectorized computations that provide advantages in terms of speed and clarity of code. Loops, in particular, are not applied in R as much as in other programming languages, as frequently computations along elements of vectors or lists can be more easily performed using vectorized functions or through functions within the apply family. A vectorized counterpart exists also for the `if()` construct: the function `ifelse()`. Let's see an example:

```
ifelse(test=c(1,4,-2,4,-1) > 0, "pos", "non-pos")
```

```
## [1] "pos"     "pos"     "non-pos" "pos"     "non-pos"
```

The function loops through the elements of the logical vector defined in the argument `test`, and returns the value defined in the second argument if `TRUE` or the third argument if `FALSE`, respectively. The last two arguments can be vectors as well, and in this case recycling rules apply.

See `help(Control)` for more info on control-flow constructs.

Creating functions in R

A key aspect of programming in R is the possibility of creating additional functions. Although R offers a huge list of pre-defined functions in standard or recommended packages, the user may find it useful to create others with the aim to simplify standard computations or to perform non-standard ones.

In R, functions can be created with the construct `function()`, using the following syntax:

```
somefun <- function(arg1, arg2, arg3) {
  expr1
  expr2
  expr3
  ...
  return(value)
}
```

This syntax, resembling the output returned when printing an existing function, defines the *header* and *body* of the function. The header identifies the arguments of the function, optionally providing default values. The body contains a single expression, or multiple expressions within curly brackets, that generate the computations. The last line of the body includes the constructor `return()` that provides the final result `value`, as an object or an expression to be evaluated.

The expressions in the body are defined in terms of the arguments: different sets of arguments generate different return values. The last return line is optional: if absent, the function returns the result of the last expression. The function object defined above is assigned to the name `somefun`, and it can now be used in function calls.

As an example, let's pretend there is no `mean()` function in R, so we need to create one called `mymean()`. Everything we need is this simple expression:

```
mymean <- function(x) sum(x) / length(x)
```

The function is based on a single argument `x`, presumably a numeric vector. We can now apply the newly defined `mymean()` directly in another expression:

```
numvec1 <- c(2,-6, 4, 8, -1)
mymean(numvec1)
```

```
## [1] 1.4
```

and it already works. The hard part of programming a new function, however, is to make it general and applicable in different scenarios. The inclusion of additional arguments may help for this purpose. For example, we soon realise that our function `mymean()` cannot deal with missing elements in the vector. Using another vector that includes missing values:

«chap13,echo=F,eval=F»= # additional arguments @

```
numvec2 <- c(3,-5, NA, -2, -9, 6, NA)
mymean(numvec2)
```

```
## [1] NA
```

Similarly to the original function `mean()`, we can therefore add another argument `na.rm` to exclude the missing values when required. Our function can be modified by including additional expressions between curly brackets and (optionally) an explicit return expression:

```
mymean2 <- function(x, na.rm=FALSE) {
  if(na.rm) x <- x[!is.na(x)]
  mean <- sum(x) / length(x)
  return(mean)
}
```

The new argument `na.rm`, consistently with other R functions, is given the default value of `FALSE`. We can now use `mymean2()` with missing values, allowing or not their exclusion from the computation:

```
mymean2(numvec2)
```

```
## [1] NA
```

```
mymean2(numvec2, na.rm=TRUE)
```

```
## [1] -1.4
```

New functions can be written with the purpose of addressing specific issues in a script, or for generating a general tool to perform a computational task. In the latter case, the code of the function should work more generally under different scenarios, and care should be taken to define its scope, in particular regarding the arguments to be accepted and the results to be returned.

Functional aspects and scoping rules

The functional style of R implies some aspects that should be remembered when programming new functions. An important recommendation is that each function should be *free from side effects*, meaning that a call to the function should not change the surrounding environment. A simple example will illustrate this feature:

```
x <- 1
f1 <- function(y) {
  x <- 2
  return(y+x)
}
f1(1)
```

```
## [1] 3
```

```
x
```

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

From the results above is clear that the assignment `x <- 2` is only valid within the body of the function `f1()`, but it does not change the value of the existing object `x` in the global environment. This feature avoids unexpected results due to changes that would otherwise occur when calling the function in other environments. This behaviour can be overcome with the *super-assigment operator* '`<<-` or with the function `assign()` (see `?'<-'`), although these are neither often applied nor recommended in standard computing tasks.

Another general recommendation when writing new functions is to make them *free from external influences*. That is, the function must be self-contained, and the value it returns must usually be determined only by its arguments, independently from the environment in which the function is called. A simple example will clarify the issue:

```
f2 <- function(y) x + y
f2(1)
```

```
## [1] 2
```

```
x <- 2
f2(1)
```

```
## [1] 3
```

Here the expressions in the body of the function `f2()` include an object `x` that is not specified as an argument. This causes two problems: first, the evaluator needs to find `x` when the function is called, or otherwise an error will be generated; second, if the value of `x` is changed, this will modify the result of a call to `f1()`, whatever its arguments. When writing new functions, this behaviour should be avoided. Nonetheless, exceptions to this rule are allowed. For instance, in functions for plotting or printing objects, graphical parameters and number of digits, among other features, are set to default values outside the function environment and can

be permanently changed through the functions `par()` and `options()`. However, functions in R are usually self-contained.

In a programming language, the issues above are resolved by the *scoping rules*, a set of laws that dictate how the evaluator finds a value for a variable. In R, the scoping rules are determined by a pre-defined hierarchy between environments. Specifically, when a new function is created, it is also assigned a reference to a new environment. When the function is called, this environment is populated by objects corresponding to the arguments of the function, and by new objects created as soon as the expressions are evaluated. These *local* objects exist only during the function call, without modifying *global* objects already present in other environments. When an expression in the body is evaluated, objects are searched in various environments in the following order: first within the local environment of the function, then in the environment where the function was originally created, and finally the object is searched using the standard search path.

This behaviour prevents side effects such as the modification of global objects, and it avoids conflicts between local and global objects with the same name, removing external influences. This is particularly useful when multiple packages are loaded into the session: objects and functions within the same package take precedence in functions calls, avoiding problems with conflicting names.