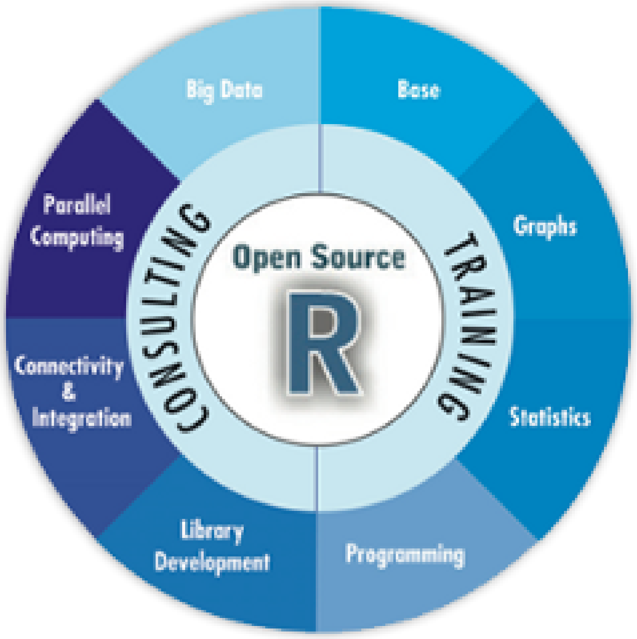
Introduction to R

Appendix 3. Basic Programming



April 2014

Version 1.2

Andrea Spanò
andrea.spano@quantide.com

Nicola Sturaro Sommacal
nicola.sturaro@quantide.com

Contents

Chapter 1

Functions in R

1.1 Introduction to functions structure

When working with R we all make constant use of functions and, when developing, we create new functions so that functions look like very familiar R objects. Nevertheless, understanding the theory and the rationals underlying R functions may help to create much more efficient and possibly elegant coding.

We can create and assign functions to a variable name as we do with any other object:

```
f <- function(x, y = 0) {
    z <- x + y
    z
}
```

Eventually, we can delete any function with the usual call to `rm()` or `remove()`

Functions are objects with three basic components:

- a formal arguments list
- a body
- an environment.

```
formals(f)
```

```
## $x
##
##
## $y
## [1] 0
```

```
body(f)
```

```
## {
##     z <- x + y
##     z
## }
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

1.1.1 Formals

Formals are the formal arguments of a function returned as an object of class `pairlist` where a `pairlist` can be thought as something similar to a list with an important difference:

```
is.null(pairlist())
```

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

```
is.null(list())
```

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

that is: a `pairlist` of length zero is `NULL` while a `list` is not.

When we call a function, formals arguments can be specified by position or by name and we can mix positional matching with matching by name so that the following are equivalent:

```
mean(x = 1:5, trim = 0.1)
```

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

```
mean(1:5, trim = 0.1)
```

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

```
mean(x = 1:5, 0.1)
```

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

```
mean(1:5, 0.1)
```

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

```
mean(trim = 0.1, x = 1:5)
```

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

Along with position and name, we can also specify formals by partial matching so that:

```r
mean(1:5, tr = 0.1)
```

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

```r
mean(tr = 0.1, x = 1:5)
```

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

would work anyway.

Functions formals may also have the construct `symbol = default`, that unless differently specified, forces any argument to be used with its default value.

Specifically, function `mean()` also have a third argument `na.rm` that defaults to `FALSE` and , as a result passing vectors with `NA` values to `mean()` returns `NA`

```r
mean(c(1, 2, NA))
```

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

While, by specifying `na.rm=TRUE` we get the mean of all non missing elements of vector `x`.

```r
mean(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] 1.5
```

The order `R` uses for matching formals against value is:

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Formals of a function are normally used within functions by the internal `R` evaluator but, we can use function `formals()` to expose formals explicitly.

```r
formals(f)
```

```
## $x
##
##
## $y
## [1] 0
```

`args()` is an other function that displays the formals in a more user friendly fashion. Actually, `args(fun)` returns a function with the same arguments as `fun` but with an empty body.

```
args(f)
```

```
## function (x, y = 0)
## NULL
```

Surely, for programming purposes, `formals()` is a better choice as it returns a simple `pairlist` that can be handled as a list:

```
is.list(formals(mean))
```

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

The "`...`" argument of a function is a special argument and can contain any number of `symbol=value` arguments . The "`...`" argument is transformed by `R` into a list that is simply added to the `formals` list:

```
h <- function(x, ...) {
    0
}
formals(h)
```

```
## $x
##
##
## $...
```

The "`...`" argument can be used if the number of arguments is unknown. Suppose we want to define a function that counts the number of rows of any given number of data frames we could write:

```
count_rows <- function(...) {
    list <- list(...)
    lapply(list, nrow)
}

count_rows(airquality, cars)
```

```
## [[1]]
## [1] 153
##
## [[2]]
## [1] 50
```

Similarly, the "`...`" arguments becomes very handy when the "`...`" arguments will be passed on to another function as it often happened when calling `plot()` from within another function. The following example shows a basic plot function used for depths plotting where additional graphics parameters are passed via "`...`":

```
time <- 1:13
depth <- c(0, 9, 18, 21, 21, 21, 21, 18, 9, 3, 3, 3, 0)

plot_depth <- function(time, depth, type = "l", ...) {
    plot(time, -depth, type = type, ylab = deparse(substitute(depth)), ...)
}
par(mfrow = c(1, 2))
plot_depth(time, depth, lty = 2)
plot_depth(time, depth, lwd = 4, col = "red")
```

1.1.2 Body of a function

The body of a function is a parsed R statement. In practice, this implies that the body of a function needs to be correct from a formal point of view but no evaluation of the body of a function occurred yet.

As a result, this function would return an error:

```
wrong <- function(x) {x =}
```

as its body is not a correct `R` statement.

While this function:

```
right <- function(x) {
    x + y
}
```

is accepted by `R` as is formally correct even thought, except under specific circumstances, will always return an error:

```
right(x = 2)
```

```
## Error: object 'y' not found
```

The body of a function, is usually a collection of statements in braces but it can be a single statement, a symbol or even a constant.

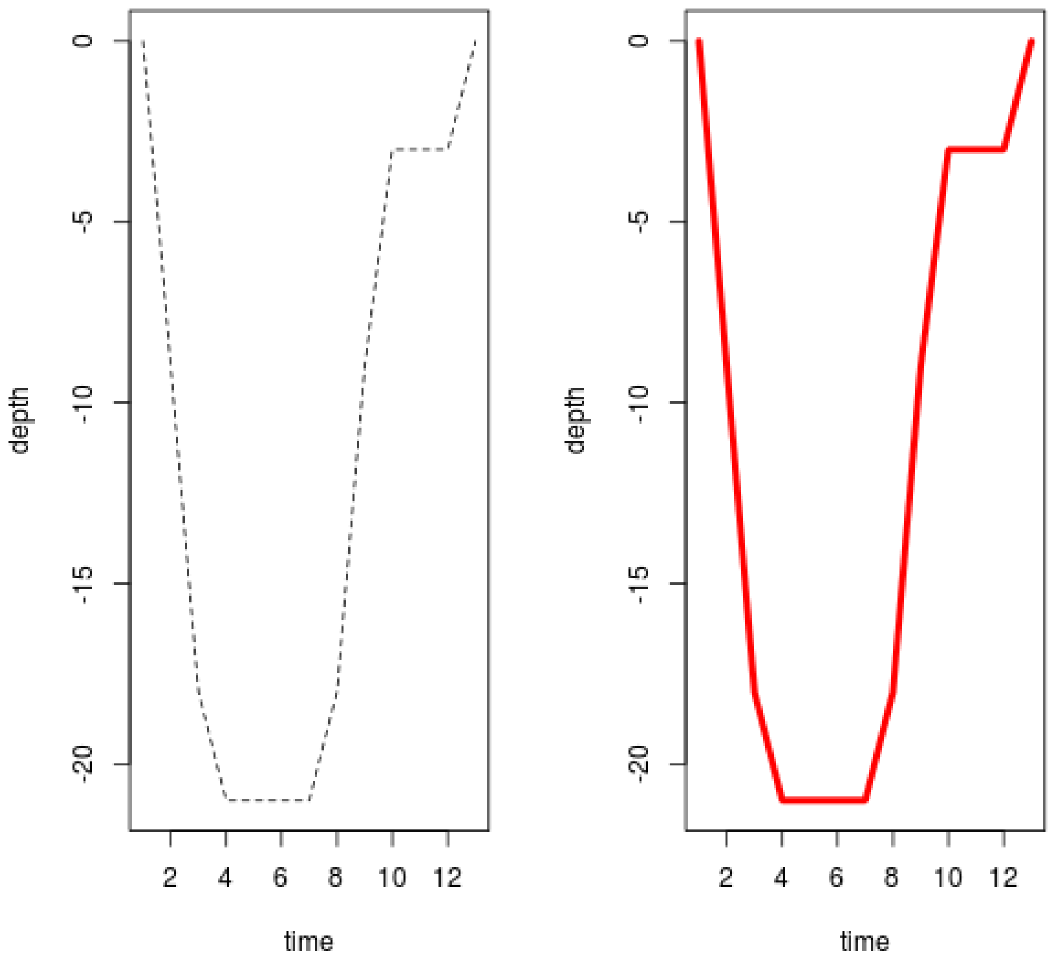


Figure 1.1: plot of chunk functions-dots3

1.1.3 Environment of a function

Environments are a fundamental concept in R and their knowledge is essential for advanced programming.

Environments are not threated here, but the following fundamental concepts:

- The environment of a function is the environment that was active at the time that the function was created. Generally, for user defined function, the Global environment:

  ```
  f <- function(x) {
      x + 1
  }
  environment(f)
  ```

  ```
  ## <environment: R_GlobalEnv>
  ```

  or, when a function is defined within a package, the environment associated to that package:

  ```
  environment(mean)
  ```

  ```
  ## <environment: namespace:base>
  ```

- Objects defined within a function, exists in the environment of the function itself.

  ```
  f <- function(x) {
      x + 1
  }
  x
  ```

  ```
  ## Error: object 'x' not found
  ```

1.2 Writing a function

A function in R can be defined with the `function()` function. Its use can be summed up as follows:

```
myFunction = function(a, b, c) {
    ...
}
```

`myFunction` is the name of the customised function; `a`, `b` and `c` are the arguments of `myFunction()` and the part in curly brackets contains the definition of the function.

As an example, write a function which calculates taxable income and taxes, given the total and the VAT percentage.

```
vat = function(amount, rate = 0.21) {
  taxable = amount / (1 + rate)
  tax = amount - taxable
  return(list(tax = tax, taxable = taxable))
}
vat(121)
```

```
## $tax
## [1] 21
##
## $taxable
## [1] 100
```

```
vat(104, rate = 0.04)
```

```
## $tax
## [1] 4
##
## $taxable
## [1] 100
```

The value assigned to the `rate` argument is the default value assumed by the argument itself.

It is good practice to end the function specifying its output with the `return()` or `invisible()` functions. The function output ought to be made of a single object. If there are multiple objects, they can be inserted in a list.

Chapter 2

Conditionals in R

2.1 The `if` statement

Conditionals are very useful when programming, as they allow to execute some code when a condition is satisfied.

The `if` statement is the most used conditionals. Its use is quite simple: `if(condition) code to be executed`

The condition ought be a lenght-one logical vector. Conditions of length greater than one are accepted with a warning, but only the first element is used. If condition is `TRUE` then the code is executed.

```
x <- 2
if (x > 0) print("x greater than zero")
```

```
## [1] "x greater than zero"
```

The code above print the text, because the condition is satisfied. When the condition is false, the code is ignored.

```
x <- -4
if (x > 0) print("x greater than zero")
```

When condition is longer than a one line of code, the code to be executed ought be enclosed in braces (`{...}`).

```
x <- 2
if (x > 0) {
    print("x greater than zero")
    return(x)
}
```

```
## [1] "x greater than zero"

## [1] 2
```

The `if` statement allows an `else` condition that will be executed when the condition is false. “‘ if(condition) code-when-true else code-when-false

As an example:

```
x = -4
if (x > 0) print("x greater than zero") else print("x is not greater than zero")
```

```
## [1] "x is not greater than zero"
```

In this case, it is strongly suggested to always use braces.

```
x <- -4
if (x > 0) {
    print("x greater than zero")
} else {
    print("x is not greater than zero")
}
```

```
## [1] "x is not greater than zero"
```

Of course, the `if` statement can be used to assign values to an object.

```
x <- -4
if (x > 0) {
    y <- 1
} else {
    y <- -1
}
y
```

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

2.2 The `ifelse()` function

As seen above, the `if` statement applies only to a length-one logical vector. If applied to vectors with length greater than one, only the first element is used and a warning message is returned.

```
if (x < 5) {
    "LESS"
} else {
    "GREATER"
}
```

```
## [1] "LESS"
```

The `ifelse()` function allows us to deal with vectors of any length. Its arguments are: a test condition, a return value when the test returns TRUE, a return value when the test returns FALSE.

```
x = 1:10
ifelse(x < 5, "LESS", "GREATER")
```

```
##  [1] "LESS"    "LESS"    "LESS"    "LESS"    "GREATER" "GREATER" "GREATER"
##  [8] "GREATER" "GREATER" "GREATER"
```

Moreover, the `ifelse()` function return a vector, so it can be assigned to an object.

```
x <- -4
y <- ifelse(x > 0, 1, -1)
y
```

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

2.3 Select one of a list of alternatives: `switch`

The `switch()` function evaluates an expression and accordingly chooses one of the further arguments. It is usually used inside a function to select from a list of alternatives.

The following code, provide a self-written `centre()` function, that accepts two input arguments: a numeric vector and a centroid function (mean, median, trimmed mean) to be applied to the numeric vector.

The `switch` read the string with the centroid function (`type`) and applies the accordingly function.

```
centre <- function(x, type) {
  switch(type,
    mean = mean(x),
    median = median(x),
    trimmed = mean(x, trim = .1)
  )
}
```

```
x <- rcauchy(10)
centre(x, "mean")
```

```
## [1] 0.3433
```

```
centre(x, "median")
```

```
## [1] 0.2169
```

```r
centre(x, "trimmed")
```

```
## [1] 0.3204
```

Chapter 3

Loops in R

3.1 for loops

`for` loops, as any other iterators, are common components of almost any programming language. The common structure of a `for` loop in `R` is:

```
languages <- c("C", "C++", "R", "Java", "Python")
for (lang in languages) {
    cat("I love ", lang, "\n")

}
```

```
## I love  C
## I love  C++
## I love  R
## I love  Java
## I love  Python
```

or alternatively:

```
languages <- c("C", "C++", "R", "Java", "Python")
n <- length(languages)
for (i in 1:n) {
    cat("I love ", languages[i], "\n")

}
```

```
## I love  C
## I love  C++
## I love  R
## I love  Java
## I love  Python
```

3.2 Good programming practices with `for` loops

3.2.1 Pre-allocating memory

`for` loops, as any equivalent iterator structure may require quite a long time to complete. Long computing time is quite natural as computational effort linearly increases with the number of iterations in the loop.

Nevertheless, few little cautions may help to save a sensible amount of time.

As an example, we may calculate the 95% quantile of the distribution of the correlation coefficient between two `N(0,1)` vectors of given sizes `n=10` by using a simple for loop over `k = 100,000` iterations.

```
k = 1e+05
n = 10
z = NULL
system.time({
    for (i in 1:k) {
        x = rnorm(n, 0, 1)
        y = rnorm(n, 0, 1)
        z = c(z, cor(x, y))
    }
    cat("95th quantile = ", quantile(z, 0.95), "\n")
})[3]
```

```
## 95th quantile =  0.5509
```

```
## elapsed
##   34.52
```

We can gain a first improvement by *'pre-allocating'*, the vector of results, We tell `R` what the size of the vector is before we begin filling it up. The wrong way to fill in a vector is to allow it to grow dynamically as it happens in the above loop.

In the above case, R has to store a vector of one element than a second vector of two elements and so on up to a vector of `100,000` elements. As each new vector can't fit inside the RAM allocated to the previous vector, R has to use a ''new" bit of contiguous RAM for new vector.

Thus, instead of using just one chunk of RAM that it takes to make a vector of `k=100,000` and filling it up with one more element at each iteration, R is forced to write the new enlarged vector in Ram at each iteration. Clearly, all of this memory writing takes up time.

Fortunately, R has a quite efficient mechanism for garbage collection and, as a result, memory usage is kept under control. Otherwise, looping without *pre-allocating* may also cause serious memory usage problems.

Running the previous example, by simply pre-allocating vector `z` before looping, reduces computing time to about half of the previous time.

```
k = 1e+05
n = 10
z = numeric(k)
system.time({
    for (i in 1:k) {
        x = rnorm(n, 0, 1)
        y = rnorm(n, 0, 1)
        z[i] = cor(x, y)
    }
    cat("95th quantile = ", quantile(z, 0.95), "\n")
})[3]
```

```
## 95th quantile =  0.5494
```

```
## elapsed
##   6.197
```

Clearly, pre-allocation requires to know in advance the size of the output vector. Whenever this is not possible, as it may happen in while loops, computing time may noticeably increase.

3.2.2 Vectorized `for` loops

Whenever possible, loops should be replaced with vectorized calculation. This approach improve both performance and clarity.

As an example, we examine a double for loop:

```
slow = function(x, y) {
    nx = length(x)
    ny = length(y)
    xy = numeric(nx + ny - 1)

    for (i in 1:nx) {
        for (j in 1:ny) {
            ij = i + j - 1
            xy[ij] = xy[ij] + x[i] * y[j]
        }
    }
    xy
}
```

```
system.time(slow(runif(1000), runif(1000)))[3]
```

```
## elapsed
##   1.917
```

An attempt to vectorize the previous code leads to:

```
fast = function(x, y) {
    nx = length(x)
    ny = length(y)
    xy = numeric(nx + ny - 1)
    j = 1:ny
    for (i in 1:nx) {
        ij = i + j - 1
        xy[ij] = xy[ij] + x[i] * y
    }
    xy
}
system.time(fast(runif(1000), runif(1000)))[3]
```

```
## elapsed
##    0.06
```

Simply working with vectorized dimensions drastically reduce computing time. Note that `j` index is computed within the `i` loop and that `y[j]` has been replaced with `y`, as a result, a single for loop is required.

3.2.3 Functionals: `lapply`

`for` loops are quite often nested and mixed up with `if()` statements. As a result, the whole coding structure may result in a complicated muddle to understand and, as a consequence, the objective of our coding may get lost within the code itself.

`for()` loops do have a bad reputation of being slow especially when compared with equivalent functionals structures as `lapply()`. Nowadays, speed is no more a crucial point when dealing with `for()` loops

In other words, `for()` loops, whenever possible, should be avoided not mainly because of a performance issue but to to achieve modularity and clarity in our codes.

Functionals are functions that take a function as input and return a data object as output. Functionals are very often excellent substitutes to `for` loops as they allow to communicate the objective of our code in a more clear and concise manner as the code will be cleaner and it will more closely adhere to `R`'s idioms.

`lapply()` is, possibly, the most used functional. `lapply()` takes a function and applies it to each element of a list, saving the results back into a result list. `lapply()` is the building block for many other functionals. In principle, `lapply()` is a wrapper around a standard for loop.

`lapply()` takes three arguments:

- a list `X`, or anything that can be coerced to a list by `as.list()`
- a function `FUN` that takes, as first argument, each element of `X`
- the `''...''` argument that can be any argument to be passed to `FUN`

Suppose we want to gain the maximum of each column for the `airquality` data frame. By using a `for` loop we could write:

```
n <- ncol(airquality)
out <- numeric(n)
for (i in 1:n) {
    out[i] <- max(airquality[, i], na.rm = TRUE)
}
out
```

```
## [1] 168.0 334.0  20.7  97.0   9.0  31.0
```

alternatively, as a data frame is a list:

```
lapply(X = airquality, FUN = max, na.rm = TRUE)
```

```
## $Ozone
## [1] 168
##
## $Solar.R
## [1] 334
##
## $Wind
## [1] 20.7
##
## $Temp
## [1] 97
##
## $Month
## [1] 9
##
## $Day
## [1] 31
```

The second chunk of code is by far more clear and concise than the first one even though a vector would be preferable than a list as output.

Moreover, `lapply()` as opposite to `for` loops does not produce any intermediate result when running. In the above `for` loop, the value of the result of the loop, vector `out`, changes at each iteration. The result of `lapply()`, instead, can be assigned to a variable but does not produce any intermediate result.

By default `lapply()` takes each element of list `X` as the first argument of function `FUN`. This works perfectly, as long as each element of `X` is the first of `FUN`. This is true in case we want to compute the mean of each column of a data frame as each column is passed as first argument to function `mean()`.

But, suppose we want to compute various trimmed means of the same vector, `trim` is the second parameter of `mean()`, so we want to vary `trim`, keeping the first argument `x` fixed.

This can be easily achieved by observing that the following two calls are equivalent:

```
mean(1:100, trim = 0.1)
```

```
## [1] 50.5
```

```
mean(0.1, x = 1:100)
```

```
## [1] 50.5
```

So, in order to use `lapply()` with the second argument of unction `FUN`, we just need to name the first argument of `FUN` and pass it to `lapply()` as part as the `''...''` argument:

```
x <- rnorm(100)
lapply(X = c(0.1, 0.2, 0.5), mean, x = x)
```

```
## [[1]]
## [1] 0.1133
##
## [[2]]
## [1] 0.1444
##
## [[3]]
## [1] 0.1292
```

3.3 while loops

`for` loops apply when the number of loops is known. Sometimes, one wants to loop until a condition is satisfied. In this case, the `while` statement should be used. Its use is quite simple: `while(condition) code-to-be-execute`

If the code is more than one row long, than it must be placed between braces (`{...}`).

The following code, generates random numbers from a Normal Distribution and stops when a value greater than 2 is obtained. The number is assigned to `x` and a message with the number of loops is printed.

```
n <- 0
x <- 0
while (x <= 3) {
    x <- rnorm(1)
    n <- n + 1
}
cat(n, "loops were executed")
```

```
## 30 loops were executed
```