programming R: Functions and Methods

Adrian Waddell

University of Waterloo
Departement of Statistics and Actuarial Science

September 8, 2010

About these Slides

These slides were written on behalf of the Departement of Statistics and Actuarial Science at the University of Waterloo, Canada.

At the time of writing, the current software versions are

- GNU Emacs 23.1.1
- Eclipse SDK Version: 3.5.2
- R version 2.11.1
- ESS 5.11

There are more slides like these on our homepage.

Before we start with the Details

In the previous slides I described how to interact with R. Now, I will go into the detail of the R programming language. All my slides mainly focus on the conceptual part of the R language. That is, if you want to learn how to apply some cluster analysis or some linear regression with R, then these slides are likely not the right source of information for you.

I also advise you to read all my chapters on R in the suggested order, outlined on the homepage. That is, I'm not writing each chapter self contained. This has two advantages, I have to write less and you as the reader has to read less (if you plan to read all my slides).

Lastly, if you have never spent any time with programming you might find reading these slides frustrating –you might do that regardless–. If so, you will either have to struggle a bit with these slides and/or consult a friend or better read a good introduction book on C or on Java programming which likely will help you later on.

Writing your Code

While writing your code, you should follow some rules in order for the code to be read easier by another person or by yourself in the future.

There is the Google's R Style Guide which is a good starting point, however personally I like the Java guidelines.

In any case, comment your source code! Comments in R can be written after `#`, for example:

```
## Use two # if you want your comment at the
## beginning of a line

a <- c(1,2,3)          # and after an expression use one #
for(i in 1:3) {        # even in an open block
  cat(paste(i,'\n'))   # print i to the R output
}
```

That is, everything after an `#` on the same line won't be interpreted by the R interpreter.

Note that I did not write the `>` or `+` at the beginning of each code line. I will use the `>/+` notation when appropriate for example to indicate that it is an R session.

Writing your Code

You can organize your code by adding additional newline (hit return) characters at some parts of your code. However you need to be aware that if R parses one line into a complete working expression, it does not look at the next line for the command to continue, hence

```
1 + 2 + 3 +
 4 + 5 + 6
```

sums up one to 21, but

```
1 + 2 + 3
 + 4 + 5 + 6
```

returns 6 for the first line and an 15 for the next line. You can fix the code by using brackets

```
(1 + 2 + 3
  + 4 + 5 + 6)
```

A good example of using newlines to organize your code is when you make a function call with many arguments

```
mat <- matrix(data = c(1, 2, 3, 4,
                       2, 5, 6, 8),
              ncol = 4, byrow = TRUE)
```

Calling a Function

Presumably one of the most common things you will do in R is to call functions. Functions have a name and usually a set of arguments of whom some might be optional. For example

```
> seq(from = 1, to = 5)
[1] 1 2 3 4 5
```

calls the function `seq` with the arguments `from=1` and `to=5` which in turn returns the integer sequence from 1 to 5 in a `vector` object.

If you look at the documentation for `seq` (by entering `?seq`), you will see that there are different ways to use `seq`. For example

```
> seq(from = 1, to = 9, by = 2)
[1] 1 3 5 7 9
```

or

```
> seq(from = 1, to = 9, length.out = 6)
[1] 1.0 2.6 4.2 5.8 7.4 9.0
```

both work.

Writing your own Function

Now `seq` is an example of a pretty sophisticated function. Lets first, for now, define our own easy function, say `doNothing`, in its simplest form:

```
doNothing <- function() {}
```

The function gets invoked as follows

```
> doNothing()
NULL
```

`doNothing` returns a `NULL` object because nothing was done within the function execution block `{}`. In fact every function returns the last processed value from its execution block or the argument of the first executed `return` call. So

```
foo <- function() {
  5+3
}
```

yields

```
> foo()
[1] 8
```

More on Returned Objects

On the other hand

```
foo <- function() {
  a <- 3 + 2; b <- a + 8
  return(b + 7)
}
```

will make for the following function call

```
> foo()
[1] 20
```

Note that the semicolon `;` separates two expressions.

Read the documentation to `function`, i.e. `?'function'`.

Further, `foo` and `bar` have no R built in meaning. They are just good function names to demonstrate example code.

Define your own function names beginning with a lowercase letter.

Arguments

Function arguments must have unique names. Their ordering matters only if you do not specify the argument names in a function call.

```
foo <- function(a, b) {
  return(a*2 + b)
}
```

so

```
> foo(a = 2, b = 3)
[1] 7
> foo(2, 3)
[1] 7
> foo(b = 2, a = 3)
[1] 8
```

R will always parse the named arguments first and then associate the remaining unnamed arguments in the order of the function argument definition.

Optional Arguments

Optional arguments are either default parameters or assigned to `NULL`.

```
foo <- function(x, offset = 0) {
  scaled.x <- (x - mean(x))/sd(x)
  return(scaled.x + offset)
}
```

hence you have two possible function invocations

```
> foo(seq(1,4))
[1] -1.1618950 -0.3872983  0.3872983  1.1618950

> foo(seq(1,4),2)
[1] 0.838105 1.612702 2.387298 3.161895
```

or equivalently

```
> foo(x = seq(1,4))
[1] -1.1618950 -0.3872983  0.3872983  1.1618950

> foo(x = seq(1,4), offset = 2)
[1] 0.838105 1.612702 2.387298 3.161895
```

Assignments

You have probably noted that I use `<-` to assign objects to variables and `=` for function arguments. This is in fact how you should always use `<-` and `=`.

"The operator `<-` can be used anywhere, whereas the operator `=` is only allowed at the top level (e.g., in the complete expression typed at the command prompt) or as one of the subexpressions in a braced list of expressions." [see `?'<-'`]

Also

```
5 -> a
```

results like

```
a <- 5
```

in assigning the value 5 to the variable `a`.

Displaying the Code of a Function

Say you would like to see whether one of your own functions in your current R session is up to date. Or you would like to study someone else's function code.

You just have to enter the function name into the prompt without brackets and arguments and R will print the function definition, e.g.

```
> foo
function(x, offset = 0) {
  scaled.x <- (x - mean(x))/sd(x)
  return(scaled.x + offset)
}
```

Unfortunately, this wont work that easily with most functions from the R base system. Coming back to the `seq` function

```
> seq
function (...)
UseMethod("seq")
<environment: namespace:base>
```

probably does not mean much to you right now. For your information

```
> seq.default
```

will do the job. I will talk about methods later.

Even more on Functions

In fact all operators like `+`, `-`, `<-`, `%%` and `%in%` also get parsed into conventional function calls by R. That is, if you write

```
> 5 + 5
[1] 10
```

R really does (behind the scenes)

```
> '+'(5,5)
[1] 10
```

or for example

```
> a <- 5
```

is really

```
> '<-'(a, 5)
```

try it!

Hence, now you know how to find the R internal documentation to operators like `?'+'`, `?'-'`, `?'<-'`, `?'%%'` and `?'%in%'`

The Ellipsis

Say you write a wrapper function for `plot` for whatever reason. `plot` has many optional graphics parameter, see `?par`. It would be very tedious to include them all in your function definition. That is where the ellipsis comes into play

```
myPlot <- function(x, y, ...) {
  plot(x, y, ...)
}
```

Say you would call `myPlot` as follows

```
> z <- seq(from = 3, to = 10, length.out = 100)
> myPlot(x = z, y = z^2, col = "orange", pch = 19)
```

You would get the plot produced by

```
> plot(z, z^2, pch = 19, col = "orange")
```

As another example see the documentation `?dotchart`.

For normal functions (not methods), use the ellipsis only if it is really necessary. Always ask yourself whether some additional arguments would do the same job.

More about the Ellipsis

In the previous slide, the additional arguments are constrained to be a subset of the `plot` arguments. That is,

```
> myPlot(x = z, y = z^2, col = "orange", pch = 19, bob = TRUE)
```

would yield a warning message because `bob` is not a valid argument for the `plot` function.

You can access individual elements in the ellipsis using the following code

```
myPlot <- function(x, y, ...) {
  args <- list(...)
  if(!is.null(args[['bob']])) {
    plot(x, y, pch = 19, cex = 7)
  }else {
    plot(x, y, ...)
  }
}
```

assuming that bob is visually impaired and colour blind. (The passionate reader now enters `?'!'`, `?is.null`, `?'if'` and `?list`). However it would be much better to define the bob argument in the `myPlot` function!

Even more about the Ellipsis

The previous example would be better programmed as follows

```
myPlot <- function(x, y, bob = FALSE, ...) {
  if(bob) {
    plot(x, y, pch = 19, cex = 7)
  }else {
    plot(x, y, ...)
  }
}
```

Finally, elements of `...` can be accessed directly via `..1`, `..2`, etc... For –a bad– example

```
foo <- function(...) {
  return(..1 + ..2 + ..3)
}
```

would yield

```
> foo(1, 2, 3)
[1] 6
```

You should never –ever– program yourself into a situation where you would have to make use of `..1` (except for debugging)! Why would you not just add an additional argument?

do.call

Say you were to call two distinct functions within your own function with different optional arguments. The following function with the use of `do.call` will work

```
myPlot <- function(x, y, z, par2d = list(), par3d = list()) {
  require(lattice) || stop("library lattice is not installed!")
  do.call('plot', c(formula(y ~ x), par2d))
  dev.new() # open a new plotting device
  do.call('cloud', c(formula(z ~ x + y), par3d))
}
```

then

```
x <- runif(20); y <- runif(20); z = runif(20)
myPlot(x, y, z, par2d = list(pch = 19, col = 'red'))
```

The interested reader should read `?formula` and `?dev.new`.

Functions as Function Arguments

You can also pass by functions as a function argument as in the following example

```
foo <- function(FUN, arg) {
  FUN(arg)
}
```

which yields

```
> foo(FUN = function(x){x^2}, arg = 2)
[1] 4
```

Modifying Function Arguments and Function Bodies

For completeness, there are some other functions for working with functions.

- `args()` to show argument list and return value of the function passed by as its argument.
- `formals()` to access and change the default arguments,

```
> foo <- function(x, y = 0, z = 4) {}
> f <- formals(foo)
> f$y <- 3
> formals(foo) <- f
```

or

```
> formals(foo) <- alist(x=,y=2,z=3)
```

see `?alist`.

- `body()` to get and set the body of a function.

Read their documentation if you feel like these might be useful functions for you. I have not encountered situation where they were neccessary for me.

Methods

Coming back to the `seq` example. `seq` has an elementary difference between all the functions we have just discussed. That is, `seq` is a *method*.

The term *method* is used in the object oriented world. *Methods* differ to ordinary functions in that they are defined for a specific class of its passed by argument. Hence as an illustrative example

```
> plot(1,1)
```

and

```
> fit <- lm(rnorm(10)~runif(10))
> plot(fit)
```

behave completely different. That's why the plot function also has the definition

```
plot(x, y, ...)
```

because all the other arguments depend on what object get passed to the argument `x`. See `?plot`.

More about Methods

For this chapter, it is just important for you to understand that methods execute different code depending on the arguments that get passed by to the method. For example, we can check for which classes the `seq` method was defined

```
> apropos("^seq\\.")
[1] "seq.Date"    "seq.default" "seq.int"     "seq.POSIXt"
```

or

```
> apropos("^plot\\.")
 [1] "plot.default"        "plot.density"        "plot.design"
 [4] "plot.ecdf"           "plot.lm"             "plot.mlm"
 [7] "plot.new"            "plot.spec"           "plot.spec.
     coherency"
[10] "plot.spec.phase"     "plot.stepfun"        "plot.ts"
[13] "plot.TukeyHSD"       "plot.window"         "plot.xy"
```

Here I made some use of regular expressions. The carrot `^` means that the following characters have to be at the beginning of a line and the `\\.` stands for that it should search for a dot.

Everything after the dot refers to the class of the passed by argument.

A few words more about methods

There is more to the whole *method* story. In fact, there are two different class types, the S4 classes and the S3 classes. What I have shown you in the previous two slides applies to S3 classes. I will write more about methods in the "R and the object oriented model" chapter.

Finally, the double point `:` acts like a similar function as `seq`, see `?':'`

```
> 1:5
[1] 1 2 3 4 5
```