Data Programming with R

Isabella Gollini

Lecture 6 - R programming structures

R programming structures

- Programming in R
  - loops - if-else
  - Boolean operators
- Writing functions
  - default values
  - `return()` and `invisible()`
  - functions are objects
  - viewing and editing flexible arguments
- Environments and scope
  - `ls` and `ls.str`
  - global assignment
- Clever coding
  - recursion
  - replacement functions
  - writing operators
  - anonymous functions

Programming in R

- Without mentioning it, we've learned quite a bit about the way R works:
  - In R everything is an object (even a function).
  - Functions have arguments given in brackets.
  - New statements are given a new line but can alternatively be separated by a semi-colon.
  - There's no need to declare (most types of) variables in R.
- In this lecture we will go into the structure of language in a bit more detail and cover a few more important concepts.

Loops

Looping in R is slow but sometimes necessary. The simplest loop is a `for` loop:

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

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

This says: set `i` to be `1` and then carry out instructions, then set `i` to be `2` and carry out instructions, etc

We can make this richer by expressing the looping vector to be an object:

```
x <- c(2, 4, 6)
for(i in x) { # for loop over the elements of x
  print(i) # print the current value of i
  print(i^2) # print the current value of i^2
}
```

```
## [1] 2
## [1] 4
## [1] 4
## [1] 16
## [1] 6
## [1] 36
```

`i` is set to be `x[1]`, `x[2]`, etc and the curly braces encapsulate a block of commands to be carried out.

Note that it is common to indent text inside curly brackets to make it easier to read.

`while` and `repeat` loops

`while` loops keep repeating until a certain condition is met.

`repeat` loops keep repeating until a `break` command is found.

The `break` command can also be used in `for` and `while` loops.

```r
i <- 2 # initialise i = 2
while(i < 10){ # stop if i >= 10
  print(i)
  print(i^2)
  i <- i + 2
}
```

```
## [1] 2
## [1] 4
## [1] 4
## [1] 16
## [1] 6
## [1] 36
## [1] 8
## [1] 64
```

```r
i <- 2 # initialise i = 2
repeat{ # repeat the code in curly brackets
  print(i)
  print(i^2)
  i <- i + 2
  if(i > 8) break # exit the loop if i > 8
}
```

```
## [1] 2
## [1] 4
## [1] 4
## [1] 16
## [1] 6
## [1] 36
## [1] 8
## [1] 64
```

Another useful command is `next` which tells a loop to skip to the next iteration without conducting any of the later commands.

Looping over non-vectors

Almost all loops take place over a vector. If you want to loop over something else (e.g. a list of objects or a matrix) you need to do something smarter.

Example: looping over matrix objects

```
u <- matrix(1:6, 3, 2)
v <- matrix(7:12, 3, 2)
for(i in c('u', 'v')) {
  print(i) # print the value of i
  z <- get(i) # get the object with the name given by the current value of i
  print(colMeans(z)) # print the values of the column means
}
```

```
## [1] "u"
## [1] 2 5
## [1] "v"
## [1]  8 11
```

The (very powerful) `get` function here turns the character vector `(u, v)` into objects from their names.

if-else statements

We have already seen simple if statements used in commands. A useful extension is if-else:

```
r <- 3
if(r == 4) {
  x <- 2
  } else { # if( r != 4)
    x <- 3
    y <- 4
  }
```

- If you have multiple commands to carry out inside an if statement you should always use curly brackets.
- Remember that the value inside the `if` statement is a logical condition (i.e. must evaluate to `TRUE/FALSE`).

Arithmetic and Boolean operators

We are already used to writing e.g.`x + y`, `x - y`, `x * y`, `x / y`; and `x^y` and `x %% y`

Another useful command is `x %/% y` which is for integer division, e.g. `12 %/% 7 = 1` because 7 goes into 12 once.

Boolean operations include `x == y` (equality), `x > y`, `x < y`, `x >= y`, `x <= y` (inequalities), `x != y` (negation).

There are also **AND** and **OR** Boolean operations for Boolean variables: `x && y`, `x || y` which produce scalar output and `x & y`, `x | y` which produce vector output.

```
a <- c(3, 2, 1)
b <- c(1, 2, 3)
# check if: (a[1] > 1 AND b[1] < 2); (a[2] > 1 AND b[2] < 2); (a[3] > 1 AND b[3] < 2)
a > 1 & b < 2
```

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

```
# check only if: a[1] > 1 AND b[1] < 2
a > 1 && b < 2
```

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

```
# check if: (a[1] > 1 OR b[1] < 2); (a[2] > 1 OR b[2] < 2); (a[3] > 1 OR b[3] < 2)
a > 1 | b < 2
```

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

```r
# check only if: a[1] > 1 OR b[1] < 2
a > 1 || b < 2
```

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

The difference is that `x && y` and `x || y` look at only the first elements of `x` and `y`. It's usually safer to use `x & y` or `x | y`.

More on Boolean operations

`TRUE` and `FALSE` can be shortened to `T` and `F` provided you have not used the variables `T` and `F` elsewhere in your code (try not to!).

`TRUE` maps to (i.e. is stored by R as) the value 1 and `FALSE` as 0.

```r
TRUE + 2
```

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

```r
TRUE * 5
```

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

```r
(1 < 2) == 1 # same as: TRUE == 1
```

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

```r
FALSE * 5
```

```
## [1] 0
```

R has a specific mode for Boolean variables: `logical`. There are associated `as.logical` and `is.logical` functions available.

Writing functions

- default values
- `return()` and `invisible()`
- functions are objects
- viewing and editing flexible arguments

Writing functions

Here is a simple function with 3 arguments:

```r
fun1 <- function(x, printx = TRUE, printy = TRUE) {
  y <- x^2
  if(printx) print(x) # if(printx) is the same as if(printx == TRUE)
  if(printy) print(y) # if(printy) is the same as if(printy == TRUE)
  return(y)
}
```

We can run the function with, e.g.:

```r
answer <- fun1(5) # by default it prints x and y
```

```
## [1] 5
## [1] 25
```

```r
answer # show the value returned by fun1
```

```
## [1] 25
```

Default values for arguments

The first line of the function specifies that `printx` and `printy` have default arguments (both set to `TRUE`). The argument `x` has no default and must be supplied.

If we want to turn the `printx` and `printy` arguments off we can specify them:

```r
answer <- fun1(5, printx = FALSE, printy = FALSE) # does not print x and y
answer # show the value returned by fun1
```

```
## [1] 25
```

We can give the arguments in any order provided we specify the argument names:

```r
answer <- fun1(printy = FALSE, x = 7) # does not print y, and the value of x is 7
```

```
## [1] 7
```

```r
answer
```

```
## [1] 49
```

However if we don't give the argument names we must specify them in order:

```r
answer <- fun1(5, FALSE) # this is the same as fun1(x = 5, printx = FALSE, printy = TRUE)
```

```
## [1] 25
```

```r
answer
```

```
## [1] 25
```

`return` or not?

At the end of the function I have used `return()` to explicitly tell R that this is the value we want to give back. Thus if we write:

```r
answer <- fun1(5)
```

```
## [1] 5
## [1] 25
```

then `answer` will store the value(s) specified by `return`.

As we are only returning a single value, we can ignore `return` and just have `y` as the last line.

If we want to return multiple values we can return them as a `list`:

```r
fun2 <- function(x, printx = TRUE, printy = TRUE) {
  y <- x^2
  if(printx) print(x)
  if(printy) print(y)
  return(list(input = x, output = y))
}
```

Although not compulsory it's usually a good idea to put `return` at the bottom of the function.

`invisible`

By default the function will print out the answer every time:

```r
fun1(5, FALSE, FALSE)
```

```
## [1] 25
```

If we use `invisible` instead of `return` we can remove this behaviour:

```r
fun3 <- function(x, printy = TRUE) {
  y <- x^2
  if(printy) print(y)
  invisible(y) # do not print the value returned by the function if that's
               # not assigned to an object
}
fun3(5, FALSE)
answer <- fun3(5, FALSE)
answer # the value is returned since is saved in 'answer'
```

```
## [1] 25
```

The function `function`

The R function `function` has essentially two arguments, the argument list and the body

These can be accessed via the functions `formals` and `body`

Consider:

```r
fun4 <- function(x = 7){
  return(x^2)
}
formals(fun4) # get the arguments of the function fun4
```

```
## $x
## [1] 7
```

```r
body(fun4) # get the body of the function fun4
```

```
## {
##     return(x^2)
## }
```

Viewing and editing R functions

- When you type in the name of a function without brackets R will print out the arguments and body of the function.
- This works well for functions that you have written and for some R functions (see, e.g. `abline`, `read.table`) but not for others which are programmed in other languages (e.g. `mean`, `sum`).
- If a function is very long you can use `page(function)` to view it more neatly, or `edit(function)` to edit it.

If you're going to edit an existing R function it is often best to rename it first:

```r
abline2 <- abline
page(abline2) # show the function in a "R page"
edit(abline2) # get access to the text editor to edit the function 'abline'
```

- Notice that function `abline` uses `invisible` instead of `return`

Flexible arguments in functions

- Many of the default R functions (e.g. `plot`) contain an ellipsis `...` at the end of the argument list

This allows you to pass in other extra named arguments to functions that may be called as part of the main function.

```
fun5 <- function(x,...) {
  return(mean(x,...))
}
```

Here mean has the extra arguments `na.rm` and `trim`, so:

```
fun5(c(1:10))
```

```
## [1] 5.5
```

```
fun5(c(1:10, NA))
```

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

```
fun5(c(1:10, NA), na.rm = TRUE)
```

```
## [1] 5.5
```

So I can give `fun5` the argument `na.rm = TRUE` despite it not being detailed in the original function arguments.

Environments and scope

SC 1

- `ls` and `ls.str`
- global assignment
- Those of you without a programming background might prefer to watch/read this section twice. See the screencast for an extra walkthrough.

Consider the following example:

```
w <- 12
f <- function(y) { # create the function f
  d <- 8
  h <- function(){ # create the function h inside the function f
    return(d * (w + y)) # d is passed by the environment of f
                        # w is passed by the global environment
                        # y is passed as an argument of f
  }
  return(h()) # function f returns the value of h()
}
f(2)
```

- Without running this code, what do you think the value of `f(2)` will be?
- When you are at the standard R prompt you are in the 'Global Environment'. Any objects you create here are available to every function you might create.
- Once you are inside a function, however, any objects you create are only available to that function. They are created in a separate environment.
- Functions can be created inside other functions creating nested environments where only certain objects are available.
- The `environment` function will tell you which environment you are in when you run it.
- The `ls` (and `ls.str`) functions will list all the objects in the current environment

`environment`, `ls` and `ls.str`

- Returning to our example function:

```r
environment() # show the current environment
```

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

```r
ls() # give the list the objects in the current environment
```

```
## [1] "f" "w"
```

```r
ls.str() # give the list the objects in the current environment and their structure
```

```
## f : function (y)
## w :  num 12
```

- The global environment (i.e. available to everything) is known as `.GlobalEnv`, though confusingly the environment function calls it `R_GlobalEnv`.
- Note that `h` and `d` don't exist as they are only created inside the function so are not in the global environment.

Environments inside functions

- Now suppose we change our function slightly to print the environment whilst we're in the function

```r
f2 <- function(y) {
  d <- 8
  h <- function(){
    return(d * (w + y))
    }
  print(environment())
  print(ls.str())
  return(h())
}

f2(2)
```

```
## <environment: 0x7fc6c6cf2920>
## d :  num 8
## h : function ()
## y :  num 2
```

```
## [1] 112
```

Getting it wrong

Suppose instead we gave the functions separately:

```r
f <- function(y) {
  d <- 8
  return(h())
}

h <- function() {
  return(d * (w + y))
}

f(2)
```

```
## Error in h(): object 'd' not found
```

```r
f <- function(y) {
  d <- 8
  return(h())
}
# change the order in the multiplication in h
h <- function() {
  return((w + y) * d)
}

f(2)
```

```
## Error in h(): object 'y' not found
```

- These errors are returned because the objects `d` and `y` do not exist in the environment for `h`.
- The fix is to give `h` some named arguments - see the screencast (mentioned in slide 20) for details.
- Notice that R stops the function when the first error it founds.

More on `ls`.

- So far we have called `ls` without any arguments to give the list of objects in the current environment.

By adding the `envir` argument, we can list all the objects in environments above that of where `envir` is called.

```r
f <- function(y) {
  d <- 8
  return(h(d, y))
}
h <- function(dd, yy) {
  print(ls()) # list the arguments in the current environment
  print(ls(envir = parent.frame(n = 1))) # list the arguments in the environment above
  return(dd * (w + yy))
}

# When h is called from f ls(envir = parent.frame(n = 1)) show the arguments in f
f(2)
```

```
## [1] "dd" "yy"
## [1] "d" "y"

## [1] 112
```

```r
# When h is used directly ls(envir = parent.frame(n = 1)) show the arguments
# in the global environment
h(8, 2)
```

```
## [1] "dd" "yy"
## [1] "f"  "f2" "h"  "w"

## [1] 112
```

Objects can exist only in functions

If you change an object inside a function the object will remain unchanged once the function has finished its execution

```r
d <- 5
f <- function() {
  d <- 8
  return(d)
}
f()
```

```
## [1] 8
```

```
d
```

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

- This behaviour occurs because `d` lives only inside the function `f`
- However, there is an exception to this rule…

Global assignment

The `<<-` and `assign` functions allow you to change an object inside a function and have it persist in different environments:

```
d <- 5
f <- function() {
  d <<- 8
  return(d)
}
f()
```

```
## [1] 8
```

```
d
```

```
## [1] 8
```

```
d <- 5
f <- function() {
  d <- 8
  assign('d', d, pos = .GlobalEnv)
  return(d)
}
f()
```

```
## [1] 8
```

```
d
```

```
## [1] 8
```

- The extra advantage of using `assign` is that the first argument is a character string so you can assign a vector of names to different values and then call them all back in with `get`

Clever coding

- recursion
- replacement functions
- writing operators
- anonymous functions

Recursion

- A recursive function is one that calls itself.
- This can be surprisingly useful and efficient though is often confusing for those who are not used to writing functions.
- Consider finding

$$\begin{aligned} x! &= x \times (x-1) \times (x-2)\ \times \ldots \times\ 1 \\ &= x \times (x-1)! \end{aligned}$$

A simple function is:

```
fact <- function(x) {
  if(x == 1) return(x)
  return(x * fact(x - 1))
}
fact(5)
```

```
## [1] 120
```

- Note that there is a break as soon as `x` becomes `1`, otherwise the function would run on forever.
- The `return` function breaks the problem down into a smaller problem and re-runs the function.

Quick sort

- Another nice recursion example is the quick sort algorithm.

Given a vector, e.g. `(5, 4, 12, 13, 3, 8, 88)` this algorithm compares the first element `5` with the others and produces two sub-vectors; the first with all the elements smaller than it `4, 3`, the second with all the elements bigger than it `12, 13, 8, 88` and re-runs `quicksort` on these two sub-vectors.

```
quicksort <- function(x) {
  if(length(x) <= 1) return(x)
  pivot <- x[1] # take the first value as pivot
  therest <- x[-1] # create a new vector with all values but the pivot
  sv1 <- therest[therest < pivot] # subset of values smaller than the pivot
  sv2 <- therest[therest >= pivot] # subset of values greater than or = the pivot
  sv1 <- quicksort(sv1) # quicksort the values smaller than the pivot
  sv2 <- quicksort(sv2) # quicksort the values greater than or equal the pivot
  return(c(sv1, pivot, sv2))
}
quicksort(c(5, 4, 12, 13, 3, 8, 88))
```

```
## [1]  3  4  5  8 12 13 88
```

Replacement functions

R allows you to use replacement functions in certain circumstances:

```
names(x)
```

```
## NULL
```

```
names(x) <- c('a', 'b', 'ab')
names(x)
```

```
## [1] "a"  "b"  "ab"
```

- The `names(x) <- ...` line looks like it violates some of our rules - we're forcing the outcome of a function to be a set of values.
- Bizarrely `names<-` is actually a function! (Side note: this is one of those situations where `<-` works and `=` does not.)

What we're really calling is:

```
x <- 'names<-'(x, c('a', 'b', 'ab'))
```

- Whenever you see a function on the left hand side this is really a replacement function of the form above.

Writing your own operator

If you want to write your own operator, you simply need to start and end your function with `%` and enclose it in quotes:

```
"%a2b%" <- function(a, b) return(a + 2 * b)
3 %a2b% 5
```

```
## [1] 13
```

- Note that your function must take two arguments and return a single value.

Anonymous functions

Sometimes it's not even worth giving your function a name. Eg. if you're just using it and disposing of it instantly:

```
M <- matrix(1:6, nrow = 3, ncol = 2)
f <- function(x) x / max(x)
apply(M, 1, f)
```

```
##      [,1] [,2] [,3]
## [1,] 0.25  0.4  0.5
## [2,] 1.00  1.0  1.0
```

```
apply(M, 1, function(x) x / max(x))
```

```
##      [,1] [,2] [,3]
## [1,] 0.25  0.4  0.5
## [2,] 1.00  1.0  1.0
```

- In the second example we haven't had to define the function at all!

Lessons from this week

- Lots of important things this week!
- Loop through with `for`, `while`, `repeat`, but avoid them whenever you can.
- Be careful when writing functions to specify your arguments correctly and return your output in the way that you want it.
- Pay close attention to the environment in which your function is called.