# notes_week2

Derek Hall

19/03/2020

## Week 2

### Control structures in R:

- if, else: testing a condition
- for: loop a fixed number of times
- whie: loop while a condition is true
- repeat: infinite loop
- break: break the loop
- next: skip an interaction of a loop
- return: exit a function

#### If/Else

The *else* is optional. There can be infinite *else if* statements.

*if () {f(x)}* is valid. Essentially, you test to see if something is true and if it is you do something and if it is not you skip the code.

classical program method:

```
x <- sample(1:6, 1)
if(x > 3) {
  y <- 10
} else {
  y <- 0
}
```

alternatively (in R):

```
y <- if(x > 3) {
  10
} else {
  0
}
```

#### For

*for* allows you to specify a function to occur based on an index of a vector.

```r
for (i in 1:10) {print(i)}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```r
x <- c("a", "b", "C", "d")

for (i in 1:4) {print(x[i])}
```

```
## [1] "a"
## [1] "b"
## [1] "C"
## [1] "d"
```

```r
for(i in seq_along(x)) {print(x[i])}
```

```
## [1] "a"
## [1] "b"
## [1] "C"
## [1] "d"
```

```r
for (letter in x) {print(letter)}
```

```
## [1] "a"
## [1] "b"
## [1] "C"
## [1] "d"
```

*for* loops can be nested. be careful, >2-3 levels becomes difficult to read and understand

```r
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
              print(x[i, j])
        }
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 2
## [1] 4
## [1] 6
```

### While

*while* executes a loop based on a logical expression. Careful! Can result in infinite loops if not written correctly.

```
count <- 0

while(count < 10) {
  print(count)
  count <- count + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

*while* loops can have more than one condition

```
z <- 5

while(z >= 3 && z <= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)

  if(coin == 1) { ## random walk
    z <- z + 1
    } else {
      z <- z - 1

    }
}
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] 4
## [1] 5
## [1] 4
## [1] 5
## [1] 6
## [1] 5
## [1] 6
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 9
## [1] 10
```

### Repeat

x0 <- 1 tol <- 1e-8

repeat { xl <- computeEstimate()

if(abs(x1 - x0) < tol) { break } else { x0 <- x1 } }

*repeat* is useful for optimization or finding the best version of something. You run the code over and over again until you find a solution that meets your criteria.

### Next, Return

*next* is used to skip an iteration of a loop

```
for (i in 1:100) {
  if(i <= 20) {
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}
```

*return* signals that a function should exit and return a given value

### Control Structure summary

- Control structures *like*, *if*, *while*, and *for* allow you to control the flow of a program.
- Infinite loops should generally be avoided, even if they are theoretically correct.
- Control structures mentioned here are primarily useful for writing programs; for command line interactive work, the *apply* functions are more useful.

## Functions

Functions are created using the *function()* directive and are stored as R objects just like anything else. Their R class is "function".

This means functions can be passed to other functions and can be nested.

Functions do not need to use all formal arguments. Some can be mssing or might have default vaules

*formals* fuction returns a list of all formal arguments in a function

### Argument Matching

Arguments can be matched done positionally or by name. It is not recommended to messing with argument order as it can cause confusion.

All of these sd calls are equivalent

```
mydata <- rnorm(100)

sd(mydata)
```

```
## [1] 1.05707
```

```
sd(x = mydata)
```

```
## [1] 1.05707
```

```
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 1.05707
```

```r
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 1.05707
```

```r
sd(na.rm = FALSE, mydata)
```

```
## [1] 1.05707
```

*Note:* when an argument is called by name, it is removed from the arguement order list.

### Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed. In the second example below, note that the function is fine until it encountered *print(b)*. Because *b* did not need to be evaluated until that point, it proceeded *lazily* until then.

```r
f <- function(a, b) {
  a^2
}
f(2)
```

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

```r
## b was not inputed and so it is lazily ignored
```

f <- function(a, b) { print(a) print(b) }

f(45)

[1] 45 Error in print(b) : argument "b" is missing, with no default

### The "..." Argument**

... is used to pass arguments from one function to another. Used for when you want to extend another function

```r
myplot <- function(x, y, type = "1", ...) {
  plot(x, y, type = type, ...)
}

## Changes the default type of the base plot function to 1 and keeps all other arguements without haven
```

The ... argument is also necessary when the number of arguments to be passed to a function can not be known in advance. *e.g.* the *paste* function pastes a series of character variables and merges them, but it is unknown ahead how many variables will be placed into the function ahead of time.

```r
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

One catch of ... is that arguments after ... must be named explicitly and cannot be partially matched.

```r
paste("a", "b", sep = ":") ## Changes seperator to :
```

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

```r
paste("a", "b", se = ":") ## Partial matching is ignored allowed
```

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

```r
paste("a", "b", ":") ## Arguements must be called explicitly
```

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

### Scoping and Symbol Binding

When R tries to bind a value to a symbol, it performs an ordered search through a series of *environments*. The order is roughly

1. Search the global environment
2. Search the namespaces of each package on the search list

The search list can be found using the *search* function

```r
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

### Binding Values

- The *global environment* or workspace is always searched first and the *base* package is always last.
- The order of the search list matters and is unique to each User. You cannot assume a set list will be available.
- When a user loads a package with *library()*, it is placed in the second spot and the rest of the list shifts down.
- R has separate namespaces for functions and non-functions, so it is possible to have an object named *c* and a function named *c*.

### Scoping Rules

The scoping rules for R are the main feature distinguishing it from the OG S language.

R uses *lexical* or *static* scoping in contrast to *dynamic* scoping.

R uses the search list and the *lexical* scoping is particularly useful for simplifying statistical computations.

*e.g.*

```r
f <- function (x, y) {
  x^2 + y / z

}
```

x and y are formal arguements, but z is a *free variable*

Lexical scoping means that the values of free variables are searched in the environment in which the function was defined.

What is an environment?

- A collection of (symbol, value) pairs.
- Every environment has a parent environment and can have multiple children.
- Only the empty environment does not have a parent.
- A function + an environment = a closure or function closure.

If the value of a symbol is not defined and is not found in the environment in which the function was defined, the search will continue in the *parent directory*. The *parent directory* will be the environment in which the function was called.

The search continues until it hits the top-level environement, usually the global environment or the package namespace.

After the top-level environment, the search continues down the search list until reaching the empty environment, at which point an error is thrown.

Scoping is important because, unlike other languages, R allows a function to return another function.

```
## Creates a function that returns the pow function with the n defined.
make.power <- function(n) {
  pow <- function(x) {
    x^n
  }
}

## The make.power function can then be stored as a new function, which in essence is the pow function w

cube <- make.power(3)
square <- make.power(2)

cube(3)
```

```
## [1] 27
```

```
square(3)
```

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

### Exploring a Function Closure

A function's environment can be explored with the *ls()* and *get()* functions

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube))
```

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

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(square))
```

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

An example of lexical vs. dynamic:

```
y <- 10

f <- function(x) {
  y <- 2
  y^2 + g(x)
}
```

```
g <- function(x) {
  x*y
}
```

What is the value of *f(3)* in lexical vs. dynamic scoping?

*Lexical:* In g, the y is called from the environment where g was defined, which is the global environment. Therefore g will be 3 * 10 = 30. In f, the y will be called from where it was defined, and will be 2, so 2^2 + 30 = 34. This is the logic R uses:

```
f(3)
```

```
## [1] 34
```

*Dynamic:* In Dynamic, y will be called from the environment in which it is *called*. Therefore, it will be 2 in both cases. 2^2 + 3*2 = 10.

### Consequences of Lexical Scoping

1. In R, all objects must be stored in local memory. This can be a problem with large datasets.
2. In R, all functions must carry a pointer to their defining environments, which could be anywhere.

### Aplication of lexical scoping rules - Optimization

Optimization routines in R pass a function whose argument is a vector of parameters, with the goal to minimize or maximize an output based on a combination of those parameters.

However, the object function might depend on a host of other things besides parameters, such as data.

Therefore, it might be important to hold certain parameters fixed during the optimization.

Write a "constructor" function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma*2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
  }
}

set.seed(1); normals <- rnorm(100,1,2)

nLL <- make.NegLogLik(normals)
nLL
```

```
## function(p) {
##     params[!fixed] <- p
##     mu <- params[1]
##     sigma <- params[2]
##     a <- -0.5*length(data)*log(2*pi*sigma*2)
##     b <- -0.5*sum((data-mu)^2) / (sigma^2)
##     -(a + b)
##   }
```

```
## <bytecode: 0x0000000017e0f870>
## <environment: 0x0000000012269e80>
```

```
ls(environment(nLL))
```

```
## [1] "data"   "fixed"  "params"
```

```
optim(c(mu = 0, sigma = 1), nLL)$par
```

```
##       mu    sigma
## 1.217666 2.527922
```

```
nLL <- make.NegLogLik(normals, c(FALSE, 2))

optimize(nLL, c(-1, 3))$minimum
```

```
## [1] 1.217775
```

### Coding Standards

1. Write code in a .txt editor.
2. Using indenting
3. Limit the width of your code (<80 columns)
4. Limit the lengths of your functions. One task per function ideal.

### My first R functions

```
add2 <- function(x, y) {  ## Creates the function
  x + y
}
## Example
add2(3, 5)
```

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

This code creates a function called *add2* that adds two numbers together

```
above10 <- function(x) {
  use <- x > 10
  x[use]
}
```

This function returns a subset of the vector x that are greater than 10

```
above <- function(x, n = 10) { ## Specifies a default value for n
  use <- x > n
  x[use]
}

## e.g.
x <- 1:20

above(x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
above(x, 12)
```

```
## [1] 13 14 15 16 17 18 19 20
```

This function returns the values of a vector $x$ above a specified number $n$. If no value is provided for $n$, the default is 10.

```
columnmean <- function(y) {
  nc <- ncol(y)
  means <- numeric(nc)
  for(i in 1:nc) {
    means[i] <- mean(y[, i])
  }
  means
}
```

*columnmean* calculates the mean values of the columns in a dataset. However, it will return an error for datasets containing NAs or none numeric columns.

```
columnmean <- function(y, removeNA = TRUE) {
  nc <- ncol(y)
  means <- numeric(nc)
  for(i in 1:nc) {
    means[i] <- mean(y[, i], na.rm = removeNA)
  }
  means
}
```

This version of the function allows the user to specify if Na values should be remove, with the default behavior being to remove them.

## Dates and Times

R has developed a special representation of dates and times.

Dates are represented by the *Date* class. Times are represented by the *POSIXct* or the *POSIXlt* classes.

*POSIXct* is a very large interger. *POSIXlt* is a list and stores other info such as day of the week, year, month, day of the month.

Dates are stored internally as the number of days since 1970-01-01 Times are stored internally as the number of seconds since 1970-01-01

Character strings can be coerced into these formats with *as.Date, as.POSIXlt, as.POSIXct.*

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

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

A number of generic functions exist.

weekdays: gives the day of the week months: give the month name quarters: give the quarter number

```
weekdays(as.Date("2020-07-30"))
```

```
## [1] "Thursday"
```

```
months(as.Date("2020-07-30"))
```

```
## [1] "July"
```

```
quarters(as.Date("2020-07-30"))
```

```
## [1] "Q3"
```

*strptime* function can be used if date and times are written in a different format.

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-10 10:40:00 EST" "2011-12-09 09:10:00 EST"
```

When manipulating data, make sure the format is the same between different variables.

Date time classes auto track tricky things like daylight saving and leap years.

# Swirl Notes

## R boolean operators

- < less
- greater
- == equal
- <= less or equal
- = greater or equal
- ! negates logical expression (e.g. !TRUE = FALSE)
- & and && tests multiple operands with AND logic. & evaluates across a vector and && evaluates the first member of a vector
- and || tests multiple operands with OR logic. | evaluates across a vector and || evaluates the first member of a vector

AND operators are always evaluated before OR operators.

- *isTRUE()* evaluates an argument and returns TRUE if it is TRUE or FALSE if it is FALSE.
- *identical()* evaluates if two R objects are identical. If so, it returns TRUE, otherwise it returns FALSE.
- *xor()* or exclusive OR function will evaluate two arguments and return TRUE as long as one of them is TRUE. Otherwise it returns false.
- *which()* returns an index of which values of a object are true for a given arguement.
- *any()* returns TRUE if one or more elements of a logical vector are TRUE.
- *all()* returns TRUE if every element of a logical vector are TRUE.

## Functions

Everything that exists in a function is an *object*. Everything that happens because of a function is a *call*.