Notes for Week 2

Rob Hayward

October 3, 2013

1 Control structures

The are a number of structures that can be used.

```
if(<condition>) {
  ## do something
} else {
  ## do something else
}
```

The loop structure.

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

While

```
count <- 0
while (count < 10) {
    print(count)
    count <- count + 1
}
```

Example using the logical operator.

```
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
    }
}
```

Repeat can be used with break.

```
x0 <- 1
tol <- 1e-08

repeat {
    x1 <- computeEstimate()
    if (abs(x1 - x0) <= tol) {
        break
    } else {
        x0 <- x1
    }
}
```

This is often used in an optimisation algorithm. However, it is Usually better to use a for loop with a set length so that there is a break and you avoid an infinite loop.

```
for (i in 1:10) {
    if (i <= 20) {
        ## skip the first 20 iterations
        next
    }
    ## do something here
}
```

Does not use the first 20 attempts.

2 Functions

The basic style is

```
x <- function(<arguments>){
  ## do something interesting
}
```

Functions are *first class objects*. They can be passed on and nested inside another function. The last expression is returned. Formals will return a list of all the formal arguments in a function. Arguments can be matched by position or name. Lazy evaluation means that arguments are only evaluated when they are needed. This prevent errors happening easiy. It is posible to use ... to pass

a function that you do not want to repeat. For example, if you were to create a version of plot, it is possible to do the following

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

The ... can also be used when dealing with *generic functions*. These are functions that dispatch methods to different types of data. More on this later. The ... can also be used when the arguments are not known in advance. For example, the *paste* function uses this because it is possible to push as many strings as possible. Anything after ... must be named explicitly.

One issue that emerges is how does R know what to do if I assign a symbol that is already attatched to a function to a new function. For example, what happens if I assign something to lm? R needs to bind a value to the symbol. In this case it is a function that is bound to the lm symbol. R looks through the various *environment*. It first looks in the *Global Environment*. Therefore, if you assign something it is in the Global Environment. It will look through the list of Environments until it finds a match. The base package is always the last. The order of the packages matters. If library() is used to load package, it is placed behind the Global Environment (second in the list). It is possible to have different objects with the same symbol. For example, there could be a function called *c* and a vector called *c*.

3 Scoping Rules

The scoping rules determine how a value is associated with a free variable in a function. R uses lexical scoping or static scoping rather than dynamic scoping. Free variables are those that are not formal arguments and not local variables (those that are assigned inside the function body). The values of free variables are searched for in the environment in which the function was defined. An environment is a collection of symbol-value pairs (x is the symbol and 2.3 may be its value). Every environment (collection of symbol-value paris) has a partent environment and an environment may have a number of children. Once a function is associated with an environment, there is *closure* or *environment closure*.

R will look in the environment that the function was defined if it cannot find a value in the function. If it cannot find it there, it will look in the parent environment. It will continue looking down the list of environments until it comes to the Globa Environment. It wll continue until it gets to the empty environment and then throw an error.

Typlically, the function is defined in the global environment. This is what is usual and the response is expected. However, sometimes it is possible to return a function from a function. In this case, the function has been defined in the

fucntion not the global environment. These are often of a sort of *constructor functions*. The function is constructing another function. For example,

```
make.power <- function(n) {
    pow <- function(x) {
        x^n
    }
    pow
}
```

The function pow is returned. n is a free variable. It is defined in the pow function.

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

## [1] 27

square(3)

## [1] 9
```

it is possible to look into the environment

```
ls(environment(cube))

## [1] "n"   "pow"

get("n", environment(cube))

## [1] 3
```

R looks for the value of free variable in the *defining* environment. In dynamic scoping, the programe looks for the free variable in the *calling* environment. This means that all objects have to be stored in memory. All functions must have a pointer to its defining environment. This can be complex becaus there can be functions within functions.

This is important because it will facilitate optimisation. There are a number of optimisation functions in R such as *optim*, *nlm* and *optimize*. They all require functions be passed to the function where the arguments are the vector parameters. The aim is to create a constructor function which *construts* the objective function. All the data and all the things that the objective function depends on will be included in the defining environment. For example,

```
make.NetLogLik <- 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)
    }
}
```

This is a log likelihood function that depends on the data and a logical vector that will allow some of the parameters to be fixed. Inside the constructor function, another function is defined that will optimise over the mean and the standard deviation. The constructor function returns the function as the return value.

```
set.seed(1)
normals <- rnorm(100, 1, 2)
nLL <- make.NetLogLik(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)
##     }
## <environment: 0x000000000707cd28>

ls(environment(nLL))

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

The address of the environment that the function is in is returned. "Data" is a free variable (not a formal or informal argument). The data has to be looked up in the environment where the data was defined. If you look at this environment by calling ls(), you will see the free variable that are defined in the defining environment.

Now optim() can be called on the fuction.

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

##    mu sigma
## 1.218 1.787
```

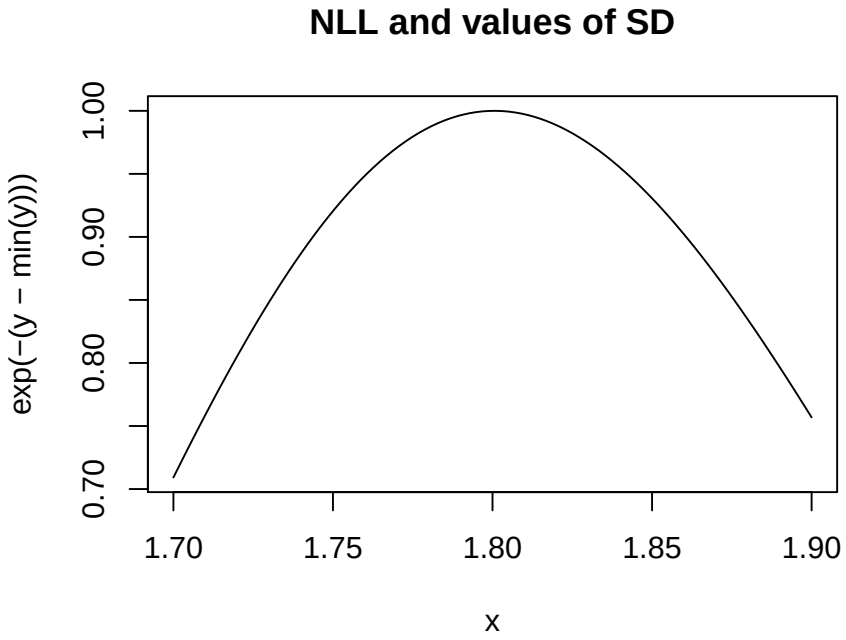
The estimates of the mean and sigma are pretty accurate. Alternatively, it is possible to fix sigma at its actual value and estimate the mean. First, re-set the negative log likelihood function.

```
nLL <- make.NetLogLik(normals, c(FALSE, 2))
optimize(nLL, c(1e-06, 10))$minimum

## [1] 1.218
```

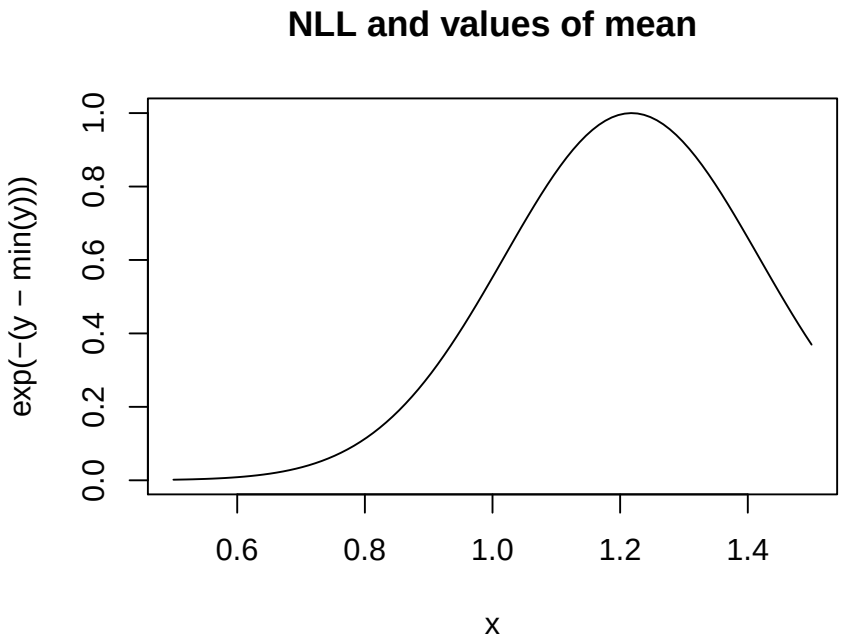
The log likeihood can also be plotted. Create a sequence of x values and apply those to the nLL function (with the mean fixed at 1) and plot the result for the standard deviation.

```
nLL <- make.NetLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, length = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l", main = "NLL and values of SD")
```



Similarly for the mean.

```
nLL <- make.NetLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, length = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l", main = "NLL and values of mean")
```



4 Loop functions

The main loop functions are lapply, sapply apply, tapply, mapply. There is also the function split that will split things into small items. lapply will also return a list. The function will apply a function to each element of the list. the . . . can be used to send arguments to the functiont that is being applied.

There is a lot of use of *anonymous functions*. An an anonymous function is one that is made on the fly. For example, to extract the first column,

```
x = list(a = matrix(1:4, 2, 2), b <- matrix(1:6, 3, 2))
x

## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

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

lapply(x, function(elt) elt[, 1])

## $a
## [1] 1 2
##
## [[2]]
## [1] 1 2 3
```

The function to extract the column has to be made up on the spot.

sapply will try to simplify the result of lapply. It can change a list into a vector or matrix if it is more appropriate (where each element has a length of onme or there are seveal elements of the same length). For example,

```
x <- list(rnorm(100), runif(100), rpois(100, 1))
sapply(x, quantile, probs = c(0.25, 0.75))

##        [,1]   [,2] [,3]
## 25% -0.6510 0.2557    0
## 75%  0.5009 0.7655    2
```

Apply is the same sort of function that works on arrays. A matrix is the simplest form of arrays. The *margin* is the dimension over which the function is to be appled. Dimension 1 applies to rows, dimension 2 is the columns.

```
x <- matrix(rnorm(200), 20, 20)
apply(x, 2, mean)

##  [1]  0.21051  0.09443  0.02218 -0.15933  0.09021  0.14723 -0.22431
##  [8] -0.49658  0.30095  0.07704  0.21051  0.09443  0.02218 -0.15933
## [15]  0.09021  0.14723 -0.22431 -0.49658  0.30095  0.07704
```

There are other functions that are optimised to work more swiftly.

- rowSums = apply(x, 1, sum)
- rowMeans = apply(x, 1, mean)
- colSums = apply(x, 2, sum)
- colMeans = apply(x, 2, mean)

If you want to apply a function across an array with more than two dimensions, identify the dimensions that are to be preserved. For example, if there is an array of 2 by 2 matrices, to take the mean of the row and columns of all of them.

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a, c(1, 2), mean)

##          [,1]    [,2]
## [1,] 0.11176 -0.5831
## [2,] 0.01838  0.1728
```

This will return the mean of the rows and the mean of the columns.

tapply will apply some function to a subset of the vector. the Index argument will determine which parts of the vector or matrix will be assessed. It should be a list of factors or something that can be coerced into a factor. The result can be simplified (default is TRUE). For example, if 30 variables are created and factors are allocated, the mean by factor can be calculated.

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
tapply(x, f, mean)

##        1        2        3
## -0.02514  0.43427  0.49939
```

If you do not simplify the result, you will get a list back.

tapply is useful because it splits the matrix and then puts it back together. The split() function takes a vector and a factor variable and splits the vector into the number of groups implied by the factors. Split will always return a list. This is more or less what tapply does. However, the split function can be applied to much more complicated objects. For example, to calculate mean temerature for each month in the following dataframe.

```
library(datasets)
head(airquality)

##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6

s <- split(airquality, airquality$Month)
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
##              5      6       7       8      9
## Ozone    23.62  29.44  59.115  59.962  31.45
## Solar.R 181.30 190.17 216.484 171.857 167.43
## Wind     11.62  10.27   8.942   8.794  10.18
```

Using the sapply function will ensure that a matrix is returned rather than a lsit. The na.rm = TRUE allows the mean() function to be evaluated.

It is also possible to split over more than one level. If there is more than one factor. For example, here two factors are created, one with two levels (f1) and one with five levels (f2). This means that there are 10 possible combinations.

```
x <- rnorm(10)
f1 <- gl(2, 5)
f2 <- gl(5, 2)
interaction(f1, f2)

##  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5

str(split(x, list(f1, f2), drop = TRUE))

## List of 6
##  $ 1.1: num [1:2] 1.097 -0.248
##  $ 1.2: num [1:2] -0.16 -0.626
##  $ 1.3: num 0.9
##  $ 2.3: num -0.994
##  $ 2.4: num [1:2] 0.849 0.806
##  $ 2.5: num [1:2] -0.468 0.848
```

Drop = TRUE will get rid of the new combined factors (10) that do not have any members.