Chapter 7 - Programming Tools

1 Aims of the Chapter

By the end of this chapter you should know

- how to create `for` loops
- how to control `for` loops
- how to use conditional statements (if, else etc)
- how to use apply, lapply, sapply, mapply

This chapter introduces some of the facilities in R for controlling the flow of execution of commands, and the facilities for looping which make it a powerful and flexible tool for general statistical programming.

Now that you have seen R Markdown syntax, in future notes, I am going to provide you with the R Markdown files rather than the HTML files. This way you can modify and run the code interactively in the code chunks. Press the right hand button (green triangle) within the code chunk to run it

2 Looping versus exploiting vectorisation

As you have started to see, R has thousands of functions that perform a wide variety of statistical and computational tasks. You should **always** use existing functions wherever possible.

You are about to see how to use loops. An important point to remember is that R operates very efficiently on vectors. Many calculations that appear to need loops can be done more efficiently using operations on whole vectors instead. Look for opportunities to do this whenever possible.

3 branching

3.1 `if`

The `if` command allows a statement to be executed only if some condition is true.

For example

```
x <- 10
if (x > 0) y <- log(x)
y
```

```
## [1] 2.302585
```

ensures that $\log(x)$ is calculated only if $x > 0$. If there are multiple lines of code that depend on the result of the `if` condition then they should go in `{}`. For example

```
x <- -2
if (x < 0){
  y <- x ^ 2
  z <- x ^ 3
}
y
```

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

```
z
```

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

3.2 `&&` and `||` versus `&` and `|`

The condition that is checked can have several conditions; all that is needed is that it produces a single logical (`TRUE` or `FALSE`) value.

For example

```
x <- 3
if (x > 0 &&  x < 5) y <- x / (x + 1)
y
```

```
## [1] 0.75
```

Note the `&&` operator here. We have already worked with the the `and` and `or` operators `&` and `|`. These operate component wise on vectors, use recycling and can return logical vectors of length more than 1. This makes them potentially inappropriate to use in `if` statements since we require a **single** `TRUE` or `FALSE` value.

Instead it is better to use `&&` for the *and* operator, `||` for the *or* operator. These operate only on scalar logical expressions. Using `&` and `&&` inappropriately is a common mistake. Consider the following simple example

```
x <- 1:3
y <- 4:6
x > 1 & y > 5
```

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

```
x > 1 && y > 5
```

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

Here `&` tests each of the three components of the vector whereas `&&` only tests the first. Because of **recycling** we have already seen that this can provide incorrect answers without any warning or error messages if we are not careful. Suppose we want to extract the elements of a vector `x` that have a value more than 0 but less than 5. Why does the following code not return the values 1,2,3,4 and why does it not return an error?

```
x <- 1:10
x[x < 5 && x > 0]
```

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

Try this

What would make the code correctly return the values 1,2,3,4?

3.3 `if ... else`

This allows alternative commands to be executed if the condition is false. For example

```
  if (x > 0) y <- log(x) else y <- NA
```

```
## Warning in if (x > 0) y <- log(x) else y <- NA: the condition has length >
## 1 and only the first element will be used
```

or

```
x <- 5
y <- -8
if(x > 0 && y > 0){
  a <- 10
  b <- 50
} else{
    a <- 100
    b <- 200
  }
a
```

```
## [1] 100
```

```
b
```

```
## [1] 200
```

Sometimes a sequence of conditions might be used

```
print_sign_of_x <- function(x){
  if  (is.na(x)) print("x is NA") else
  if (x < 0) print("x is negative") else
  print( "x is non-negative")
}
print_sign_of_x(x = NA)
```

```
## [1] "x is NA"
```

```
print_sign_of_x(x = 4)
```

```
## [1] "x is non-negative"
```

```
print_sign_of_x(x = -7)
```

```
## [1] "x is negative"
```

3.4 `ifelse`

The structure of the R language ensures that most calculations can be done much more quickly and conveniently on vectors or arrays than by explicitly looping through individual elements. R has several functions to aid vectorisation. One is a vectorised form of the `if` function, `ifelse`, which operates component wise on vectors. So

```
x <- 1:10
(y <- ifelse(test = x < 4.3 , yes = 1, no = 0))
```

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

Here `ifelse` tests each element of `x` to see if it is less than 4.3, if the test is `TRUE` then `y` is assigned the value 1, if the test is `FALSE` `y` is assigned the value 0. Here `test` is a vector with 10 `TRUE` or `FALSE` values.

You can also supply vectors for the `yes` and `no` values. For example

```
x <- c(-3:3)
ifelse(x > 2, yes = 1:7, no = 11:17)
```

```
## [1] 11 12 13 14 15 16  7
```

produces a vector whose values are the elements of

- `1:7` when the elements of `x` are greater than 2
- `11:17` when the elements of `x` are **not** greater than 2

Try this

What do you think the following would produce?

```
x <- c(-3:3)
ifelse(x > 1, yes = 1:4, no = 11:13)
```

3.5 `switch`

This allows choice between multiple options by name or number. For example

```
generate_realisations <- function(distribution, ..., n = 1){
  switch(distribution,
         "uniform" = runif(n = n, ...),
         "gaussian" = rnorm(n = n, ...),
         "binomial" = rbinom(n = n, ...))
}
generate_realisations("uniform", min = 2, max = 100)
```

```
## [1] 48.97521
```

```
generate_realisations("uniform", n = 5)
```

```
## [1] 0.2425230 0.4386282 0.8584985 0.7394161 0.3870508
```

```
generate_realisations("gaussian", mean = 2, sd = 0.5)
```

```
## [1] 1.632214
```

```
generate_realisations("binomial", n=3, size = 10, prob = 0.3)
```

```
## [1] 1 4 3
```

```
generate_realisations(3, n=8, size = 1, prob = 0.3)
```

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

The `switch` function matches the character string argument supplied and executes that line of code. If a number is supplied it runs the line of code corresponding to that number.

Note the use of `...` to pass arguments which will have different names for different distributions (e.g. `size` for rbinom, `sd` for `rnorm`, `min` for `runif`). Using `...` means we don't need to say exactly what the arguments have to be, which is important since we don't know which distribution the user is going to sample from

Try this

Looking at the help files for `runif` and `rbinom`, which of the following lines of code do you think will run?

```
generate_realisations("uniform")
generate_realisations("binomial")
```

4 looping

4.1 `for` loops

`for` can be used for simple repetition. For example the iterative relationship

$$x_{t+1} = ax_t(1 - x_t), \quad t = 0, 1, \dots,$$

where $0 < a < 4$, has been used as a simple model of behaviour over time of the (scaled) size of a population which is affected by overcrowding ($0 \le x_t \le 1$). The model can be implemented as follows.

```
n <- 100
a <- 2.5 # or any value between 0 and 4
x <- rep(NA, n)
x[1] <- runif(1)
for (j in 2:n) x[j] <- a * x[j-1] * (1-x[j-1])
```

Note that if we want to specify the value of elements of a vector within a `for` loop we need to **initialise** the vector first. There are lots of ways to do this. If you know the length of the vector in advance then you can use the `rep` function to create it, for example `x <- rep(NA, n)`. I tend to create vectors of `NA` values so that I know if I don't overwrite all the elements in the vector.

In the above code `j` is the variable that is used to refer to the index of the `x` vector. `for` loops increment by one, meaning that in the code above

- `j` starts at 2, `x[2]` is then calculated using value of `x[1]`
- `j` increments and takes the value 3, `x[3]` is then calculated using `x[2]`
- `j` continues to increase by 1 until it reaches `n`, then `x[n]` is calcualted using `x[n-1]`

It is sometimes useful to display the calculations at each iteration on the screen to check the output as it progresses. This can be very useful for slow loops where the output doesn't print to the screen too fast or where the calculations at each iteration are not assigned or are overwritten (making it impossible to look at them when the loop terminates). In the above it could be achieved using the `cat` function.

```
   n <- 10
   a <- 2.5 # or any value between 0 and 4
   x <- rep(NA, n)
   x[1] <- runif(1)
   for (j in 2:n) {
     x[j] <- a * x[j-1] * (1-x[j-1])
     cat("iteration=", j, "\t", "value=", x[j], "\n")
   }
```

```
## iteration= 2      value= 0.6247299
## iteration= 3      value= 0.5861062
## iteration= 4      value= 0.6064643
## iteration= 5      value= 0.5966634
## iteration= 6      value= 0.6016405
## iteration= 7      value= 0.599173
## iteration= 8      value= 0.6004118
## iteration= 9      value= 0.5997937
## iteration= 10     value= 0.600103
```

4.2 `while`

This loop continues to run, as long as some condition is `TRUE`. Something inside the loop has to eventually make the condition `FALSE` (otherwise it will never stop running!) For example, here is a way of checking convergence of an iteration where `find_root` is the function that is evaluated in each iteration

```
set.seed(6578)
find_root <- function(x) sqrt(4 * x - 1)
x <- 0.3
delta <- 100
while (delta > 0.01) {
      newx <- find_root(x)
      delta <- abs(x - newx)
      x <- newx
      cat("x=", x, "\n")
}
```

```
## x= 0.4472136
## x= 0.8881747
## x= 1.597717
## x= 2.321824
## x= 2.878766
## x= 3.242694
## x= 3.459881
## x= 3.583228
## x= 3.651426
## x= 3.688591
## x= 3.708688
## x= 3.719509
## x= 3.725324
```

4.3 `next`

Within any loop, the command `next` produces a skip to the next iteration. Suppose you wanted to produce m realisations of a random variable with a truncated standard normal distribution (so that we only have positive values). I.e. we are using $a = 0,\ b = -\infty$ in the Wiki page here (https://en.wikipedia.org/wiki/Truncated_normal_distribution)

```
m <- 8
counter <- 0
pos_normals <- NULL
while(counter < m){
  value_to_check <- rnorm(1)
  if(value_to_check < 0) next else {
    pos_normals <- c(pos_normals, value_to_check)
    counter <- counter +1
  }
}
pos_normals
```

```
## [1] 0.2925345 0.4938734 1.5390101 0.2024994 1.8408790 0.0566255 1.8940435
## [8] 3.3022327
```

Notice in the above code how we set up a counter that increases by 1 when we grow our `pos_normals` vector.

4.4 `break`

Within a loop, the command `break` exits the loop. This is useful when you need to stop the loop because

- the code would not run if some condition is `TRUE`
- you just want the loop to finish if some condition is `TRUE`
- you are checking your code in a loop and want it to exit if some condition is `TRUE`

Here is an example where you just want to stop the loop if some condition is `TRUE`. We have a random process where the ith value of the random variable (x_i) is given by $x_i = x_{i-1}^2 + \epsilon_i$ for $i \geq 2$ where x_1 is known and $\epsilon_i \sim N(0, 0.1^2)$, i.e. we square the previous value and add a realisation from a standard normal to it. We want to stop the process if any value of x_i goes above 10

```
set.seed(3644)
x <- rep(NA, 100)
x[1] <- 1.005
for(i in 2:100){
  x[i] <- x[i-1] ^ 2 + rnorm(1, sd = 0.1)
  if(x[i] > 10){
    cat("Value has gone above 10\n\n")
    print(x)
    break
  }
}
```

```
## Value has gone above 10
##
##   [1]  1.005000  1.021143  1.053471  1.088104  1.135454  1.107296  1.162447
##   [8]  1.432274  2.051039  4.362992 18.932834        NA        NA        NA
##  [15]        NA        NA        NA        NA        NA        NA        NA
##  [22]        NA        NA        NA        NA        NA        NA        NA
##  [29]        NA        NA        NA        NA        NA        NA        NA
##  [36]        NA        NA        NA        NA        NA        NA        NA
##  [43]        NA        NA        NA        NA        NA        NA        NA
##  [50]        NA        NA        NA        NA        NA        NA        NA
##  [57]        NA        NA        NA        NA        NA        NA        NA
##  [64]        NA        NA        NA        NA        NA        NA        NA
##  [71]        NA        NA        NA        NA        NA        NA        NA
##  [78]        NA        NA        NA        NA        NA        NA        NA
##  [85]        NA        NA        NA        NA        NA        NA        NA
##  [92]        NA        NA        NA        NA        NA        NA        NA
##  [99]        NA        NA
```

4.5 Using subsetting to avoid looping

A general way to avoid `for` loops and `if` statements is to use logical subsetting. Suppose, for example, we want a function to set to zero elements of a vector that fall below a certain user-specified threshold. A function for this using looping is

```
(x <- rnorm(10))
```

```
##  [1]  1.41454761 -0.22066387 -1.15474992  1.43167287  1.31013970
##  [6] -0.05459718  0.16043011  1.48723001 -0.86739327 -0.08146871
```

```
cutoff1 <- function(x, threshold)   {
  for (i in 1:length(x))  {
    if (x[i] < threshold)  x[i] <- 0
  }
  return(x)
}
cutoff1(x, threshold = 0)
```

```
##  [1] 1.4145476 0.0000000 0.0000000 1.4316729 1.3101397 0.0000000 0.1604301
##  [8] 1.4872300 0.0000000 0.0000000
```

The following alternative uses a vectorised method

```
cutoff2 <- function(x, threshold){
  x <- ifelse( x < threshold, 0, x)
  return(x)
}
cutoff2(x, threshold = 0)
```

```
##  [1] 1.4145476 0.0000000 0.0000000 1.4316729 1.3101397 0.0000000 0.1604301
##  [8] 1.4872300 0.0000000 0.0000000
```

but a faster, more efficient way is to use a logical subscript

```
cutoff3 <- function(x, threshold){
  x[x < threshold] <- 0
  return(x)
}
cutoff3(x, threshold = 0)
```

```
##  [1] 1.4145476 0.0000000 0.0000000 1.4316729 1.3101397 0.0000000 0.1604301
##  [8] 1.4872300 0.0000000 0.0000000
```

or even more simply

```
cutoff4 <- function(x, threshold)  {
  x <- x*(x >= threshold)
  return(x)
}
cutoff4(x, threshold = 0)
```

```
##  [1] 1.4145476 0.0000000 0.0000000 1.4316729 1.3101397 0.0000000 0.1604301
##  [8] 1.4872300 0.0000000 0.0000000
```

which uses the fact that logical values, when used in an arithmetic expression, are converted into the numerical values 0 for `FALSE` and 1 for `TRUE` .

5 Exercises A

1. For a general vector e of length n, write functions to calculate $= \sum_{i=1}^{n} e_i^2$ in the following ways:

- using a `for` loop
- by using the matrix multiplication symbol `%*%`
- exploiting vectorisation

Generate a vector of length n to try your code on.

2. Write a function that calculates a moving average of span 2 of the values in a given vector. That is, given a vector $\boldsymbol{x}$ your function should return a vector with components $(x_i + x_{i-1})/2$ for $i = 2, \ldots, n$. Do this

- using a `for` loop
- without using a `for` loop

3. Write a function to calculate and plot the probability density

$$f(x) = \begin{cases} x/3 & \text{if } 0 \le x < 2 \\ (2/3)\exp(4 - 2x) & \text{if } 2 \le x \\ 0 & \text{otherwise.} \end{cases}$$

4.

- Write a function to carry out the iteration

$$x_{t+1} = a x_t (1 - x_t), \quad t = 0, 1, \ldots$$

  and plot the results.

- Try it out for various values of a. By choosing appropriate values, you should be able to observe three distinct types of behaviour.
- Extend the function to check whether the iteration has converged (within some tolerance ϵ), and to stop when it has. Check it works.

5. (Challenging) Find out what the functions `%/%` , `floor` , `ceiling` , `round` and `%%` do

Given a positive integer (a vector of length 1), write a function to return a vector of length equal to the number of digits consisting of its individual digits. For example if the integer 349123 is entered into the function it should return the length-6 vector 3 4 9 1 2 3. Do this:

- using a `for` loop
- without using a `for` loop

6 The apply Family of Functions

This family of functions allows you to avoid using loops in many programming tasks. They are powerful programming tools but many people find them difficult to use initially. There is a good summary of this family here (https://www.datacamp.com/community/tutorials/r-tutorial-apply-family). In summary

- `apply` acts on arrays (matrices and higher-order objects). It is useful when you want to apply the same function to the rows or columns of a matrix.
- `lapply` acts on lists, data frames, vectors. It applies a function to every element in the list. vector etc. It returns a list containing the results. The l in `lapply` stands for list.
- `sapply` is similar to `lapply` except that it **attempts** to return a simpler object (vector etc), rather than a list. The s in `sapply` stands for simplify.
- `mapply` allows functions that only accept scalar arguments to work on multiple vectors. The m in `mapply` stands for multivariate.

We now look at these in more detail.

6.1 apply (using existing functions)

Suppose we want to calculate the sums of each row of a matrix. We could loop over the rows but we can easily do this using `apply` .

```
(a_mat <- matrix(1:15, nrow=3))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
```

```
apply(a_mat, MARGIN = 1, FUN = sum)
```

```
## [1] 35 40 45
```

In the `apply` function:

- argument 1 is the name of the array
- argument 2 is the `MARGIN` (or dimension) of the array you want to apply the function to. `MARGIN = 1` applies the function to the **rows** of a matrix, `MARGIN = 2` applies the function to the **columns** of a matrix. If you have a higher dimensional array then you can apply the function to any dimension you like by specifying the correct value of `MARGIN`.
- argument 3 is the function to apply to the specified dimension

This is fine is the function we want to apply already exists but what is we want to apply a function we have created ourselves.

6.2 apply (using functions you create)

You can also create your own function to use with apply. For example if, for each row of `a_mat` you want to find the sum of the elements greater than or equal to 8, you could do so as follows

```
apply(a_mat, 1, function(x) sum(x[x >= 8]))
```

```
## [1] 23 33 36
```

It is important to understand what R is doing here and what the function argument `x` is. We are applying the function to the rows of the matrix, so `x` here is a row of the matrix. It starts with the first row where `x` is the vector `x <- c(1, 4, 7, 10, 13)` and calculates `sum(x[x >= 8])` to give 23.

It then moves on to the second row where `x <- c(2, 5, 8, 11,14 )` and then applies `sum(x[x >= 8])` to give 33.

Here we wrote our function within the `apply` function itself, but if it is a longer function then it is clearer to define the function first and then call it within `apply`. For example, if we wanted to subtract the column mean from those elements of each column that exceeded the mean we could do this as follows

```
selectively_subtract_mean_fun<-function(data)
{
  bigger_than_mean <- data > mean(data)
  return(data - bigger_than_mean * mean(data))
}
apply(a_mat, 2, selectively_subtract_mean_fun)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    1    1    1    1    1
```

Finally we can also allow other arguments to be included in our function to be called in apply. So if we wanted to remove the mean from those elements of each column that exceeded some value y we could do so as follows (where we choose `y` to be 7)

```
subtract_mean_bigger_than_y_fun<-function(data, y)
 {
  bigger_than_y <- data > y
  return(data - bigger_than_y * mean(data))
 }
apply(a_mat, 2, subtract_mean_bigger_than_y_fun, y = 7)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   -1   -1
## [2,]    2    5    0    0    0
## [3,]    3    6    1    1    1
```

6.3 sapply and lapply

If you have a list (or vector) and you want to apply the same function to each element of it, then use the `lapply` or `sapply` functions. The difference is in what is returned.

Consider again the list we looked at before

```
p_lst <- list(patient = "Barack", spouse = "Michelle", children = 2, child_ages = c(8,
11))
```

The arguments of `lapply` and `sapply`:

- argument 1 is the name of object to apply the function to
- argument 2 is the function to apply

So if we wanted to determine the length of the each element of `p_lst` we could use

```
(length_p_lst <- sapply(p_lst, length))
```

```
##    patient     spouse   children child_ages
##          1          1          1          2
```

```
is.list(length_p_lst)
```

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

If we wanted to return the vectors lengths as a list we could use `lapply`

```
(length_p_lst <- lapply(p_lst, length))
```

```
## $patient
## [1] 1
##
## $spouse
## [1] 1
##
## $children
## [1] 1
##
## $child_ages
## [1] 2
```

```
is.list(length_p_lst)
```

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

Consider `a`, `b` and `c` defined as

```
a <- 1
b <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
c <- array(data = 1:12, dim = c(2, 2, 3))
```

If we wanted to return a list containing the dimensions of each list element we could use

```
array_lst <- list(a, b, c)
lapply(array_lst, dim)
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] 2 2
##
## [[3]]
## [1] 2 2 3
```

> Try this
>
> What would the following return?

```
sapply(array_lst, dim)
```

So using `sapply` doesn't guarantee that the result will be a vector. R will try to simplify the output structure but sometimes it will have to keep the output in a list (as it does here)

6.4 mapply

Suppose we have a function that has two arguments, `x` and `y`. The function either calculates `log(x * y)` if both `x` and `y` are positive. Otherwise it returns 0. It is written in such a way that the arguments are expected to be scalars.

```
return_log_or_zero <- function(x, y){
  if (x > 0 & y > 0) value <- log(x * y) else value <- 0
  return(value)
}
return_log_or_zero(x = 3, y = 7)
```

```
## [1] 3.044522
```

```
return_log_or_zero(x = 3, y = -7)
```

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

```
x <- -3:2
y <- 2:7
```

What if we want to apply this function to vectors of `x` and `y`. The problem is that the function `return_log_or_zero` expects `x` and `y` to be scalars since the function uses an `if` statement which expects a scalar value. So

```
return_log_or_zero(x = -3:2, y = 2:7)
```

```
## Warning in if (x > 0 & y > 0) value <- log(x * y) else value <- 0: the
## condition has length > 1 and only the first element will be used
```

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

```
x <- -3:2
y <- 2:7
```

produces a warning and returns only a single value. How can we apply the function `return_log_or_zero` to all the elements of `x` and `y`? We use `mapply` to do this.

```
mapply(return_log_or_zero, x = -3:2, y = 2:7)
```

```
## [1] 0.000000 0.000000 0.000000 0.000000 1.791759 2.639057
```

7 Exercises B

1. Generate a 3 by 3 matrix containing any integers in `1:10`. Use the `apply` function to calculate, for each column, the product of numbers in that column.
2. Generate a 3 by 3 matrix containing any integers in `1:10`. Use the `apply` function to calculate, for each row, the number of even numbers (you may find the operator `%%` useful here; `a %% b` gives the remainder when a is divided by `b`).
3. Use the `sapply` function to return

- the `log` of each positive element
- 0 for each negative element

of the vector `c(3, 6, 7, -4, -8, 1)`. How could you do it **without** using sapply or a `for` loop?

4. A string is just a collection of characters inside quotation marks (e.g. “gh$y7” or “9@%B6 (mailto:9@%B6)”). Write a function to return a **vector** containing all the strings in a list. For example, if we enter the list `list("cat", "d0kg", 4, TRUE, NA, "m89^e")` your function should return the vector `c("cat", "d0kg", "m89^e")`. You may find the function `is.character` and `unlist` useful.

5. Write a function that acts on a character string, and returns:

- the whole string if the number of characters is three or less
- the first three characters if the number of characters is four or more

Use one of the `apply` family functions to apply it to `list("cat", "d0kg", "m89^e")`

Hint: You may find the functions `nchar` and `substr` useful for this.

6. Consider two vectors x and y. Let x_i and y_i be the ith element of x and y respectively.

- Write a function that uses `mapply` to return a vector whose ith element is the maximum of x_i and y_i. Try it on some randomly generated integer vectors of length 10
- Write a function that uses `mapply` to return a vector whose ith element is the maximum of $|x_i|$ and $|y_i|$. Try it on some randomly generated integer vectors of length 10. Make sure you have some negative integers in your vectors

8 References