7 Programming in R

7.1 Functions

The R language provides a way to define your own functions just like most programming languages. As an example, suppose we wish to write a function that takes as input the body mass vector and calculates the average body mass for all animals with body mass larger than 500 kg. We can do this as follows. Each line must be entered sequentially. After the first line the prompt changes to a + sign. This allows commands that span multiple lines.

```
> calc_mass <- function( body_mass ){
+ subset <- body_mass[ which(body_mass > 500) ]
+ av <- mean( subset )
+ return(av)
+ }
```

Note here the prompt turns into a + sign - R changes the command line when it is expecting further input. This is initiated by the open curly bracket, {, and then completed by the close curly bracket at the end, }.

In the first line we create a new variable called `calc_mass` and assign it a function type that contains one argument; this will be the body mass vector. The function definition is then contained within the curly brackets. The first line of the function definition uses subsetting to create a smaller vector containing only those entries that have a value greater than 500. The next line calculates the arithmetic average using another inbuilt R function called `mean` and assigns it to a variable called `av`. Finally the function **returns** the calculated average value using the `return` statement. This last part is important since it is this command that makes our result available from outside the function. We can use our new function as follows.

Access the documentation for `mean` using `help(mean)`.

```
> av <- calc_mass( massFrame$Body_mass )
> av
 [1] 16907.29
```

7.2 Repeated execution : for-loops

Often in programs we want to perform a certain operation repeatedly. A very common programming concept is that of a **for-loop** and it is used to perform exactly this task. Let's start with a basic example.

```
> for(i in c(1,2,3) ){
+  cat("hello world", i, "\n")
+ }
 hello world 1
 hello world 2
 hello world 3
```

On line 1 we start the for-loop by giving the keyword `for` and then defining the number of **iterations** that we wish to perform. In this

particular case we want the variable `i` to take all the values in the vector defined using the combine operator, `c(1,2,3)`. What follows the loop definition on line 1 is the the **loop body** which is the part we wish to be repeated and is enclosed in **braces** { }. In this case it is a single call to the `cat` function which concatenates its arguments (it is similar to combine but more flexible). The first argument is `"hello world"`, the second is the value of the variable, i, in the loop and the third is the new line character which means "go to a new line"..

Recall that `c(1,2,3)` defines a vector containing the elements $1, 2, 3$.

We can use for-loops to fill vectors. In the following we will generate the first ten Fibonacci numbers. These are an interesting sequence of numbers where the i^{th} number in the series is given by the sum of the previous two numbers (the first two numbers are both 1). They appear in natural phenomena such as the arrangements of flower petals.

You should immediately type `help(cat)` to look at the R documentation on this function.

Wikipedia: Fibonacci number

Wikipedia: Fibonacci in nature

Table 1 gives the first 10 Fibonacci numbers. The letter i denotes the i^{th} number of the sequence and fibonacci(i) denotes the i^{th} Fibonacci number.

i	1	2	3	4	5	6	7	8	9	10
fibonacci(i)	1	1	2	3	5	8	13	21	34	55

Table 1: The first 10 Fibonacci numbers

```
> result = rep(0,10)
> result[1] = 1
> result[2] = 1
> for(i in 3:10){
+       result[i] = result[i-1] + result[i-2]
+ }
> result
 [1]  1  1  2  3  5  8 13 21 34 55
```

This is slightly more complex, demonstrating a number of principles. On line 1 we use another technique for generating an vector; the `rep` function. This creates an vector by repeating the first argument by the number of times in the second argument.

On lines 2 and 3 we **initialise** our vector by setting the first and the second elements to 1 which are the first two numbers in the Fibonacci sequence. Lines 4-6 define our for-loop. Notice that we iterate from $i = 3$ to $i = 10$ because we have already filled the first two elements of the sequence. Finally we print the results to the screen by simply giving the variable name `result`.

Exercise 7.1

1. What is the 17^{th} value of the Fibonacci sequence?
   Hint: You will need to edit the code in only two places to do this.
   Solution:
   fibonacci(17) = 1597

   The previous code generates the first ten numbers of the Fibonacci sequence. To find the 17^{th} Fibonacci number with this code, we need to do only two things: enlarge the array, `result`, to allow the storage of 17 numbers rather than 10, and increase the number of iterations performed in the for loop.

```
result = rep(0,17)
> result[1] = 1
> result[2] = 1
> for(i in 3:17){
+       result[i] = result[i-1] + result[i-2]
+ }
> result[17]
 [1] 1597
```

7.3 Conditional execution : if statements

Conditional execution means that a program may do different tasks depending on the values of some variables. Here is a function that generates a random number between 0 and 1 using the `runif` function. The program tests whether this value is greater than 0.5 using an **if statement**. If the argument in the if statement (enclosed in round brackets `()`) evaluates to `TRUE` then the program performs the code contained within the braces {}. In this simple case it returns the string "heads".

```
> cointoss <- function(){
+        u <- runif(1,0,1)
+
+        if(u > 0.5){
+             return("heads")
+        }
+ }
```

Try running the function using the command `cointoss()`. You should sometimes see (actually it is 50% of the time) that there is no output and sometimes see that the string "heads" is output.

Let's go ahead and complete the `cointoss` function by making it return the string "tails" when $u \leq 0.5$. We can do this with an **if-else** statement.

```
> cointoss <- function(){
+        u <- runif(1,0,1)
+
+        if(u > 0.5){
```

```
+             return("heads")
+         }else{
+             return("tails")
+         }
+ }
```

It should be straightforward to see what is now happening when you issue the function call `cointoss()`. If the logical test $u > 0.5$ evaluates to `FALSE` then the code in the braces, {}, immediately after the `else` statement is executed.

This was a very simple example of conditional execution. However, it should be clear that conditional execution and repeated operations form the core of more complex programs.

7.4 The `apply` family of functions

Another important way to perform repeated operations over an object is using the `apply` family of functions. Here we will use `apply` to calculate the row and column sums of a matrix.

```
> m <- matrix( nrow=3, ncol=3, data=c(1,2,3,4,5,6,7,8,9), byrow=T )
> m
      [,1] [,2] [,3]
 [1,]    1    2    3
 [2,]    4    5    6
 [3,]    7    8    9
> apply(m,MARGIN=1,sum)
 [1]  6 15 24
> apply(m,MARGIN=2,sum)
 [1] 12 15 18
```

The first argument of `apply` takes in a matrix and the second argument is a flag for whether we are interested in the rows (`MARGIN=1`) or the columns (`MARGIN=2`). The third argument to `apply` is always a function and here we have supplied the `sum` function. In this case what `apply` returns is a vector of length 3 with the row sums in the first example (`MARGIN=1`) and the column sums in the second example (`MARGIN=2`). The apply family of functions are incredibly useful and much more efficient than for-loops so use them if you can!

7.5 Writing scripts

Scripts are just collections of R commands that you wish to run together. If you find yourself repeatedly running commands to perform some task then you should use a script. Scripts provide more flexibility and provide a better way to handle your code than purely using the command line. Also, since R can save all your commands in the current work space,

having all your commands in a script can make sure you are running the correct code. The R console has its own editor and it is **strongly** recommended you to get into the habit of using it. Doing so will save you a huge amount of time.

You can use a different text editor available on your computer if you wish (one favourite is Emacs although you could use Notepad on Windows or TextWrangler on Mac.

We can define and run the above functions in a script by opening the editor and placing the following code into a script file called `func_example.R` and saving it

```
# read in the data
massFrame <- read.table("body_brain_mass.txt",header=T)

# define function to calculate average mass above 500
calc_mass <- function( body_mass ){
  subset <- body_mass[ which(body_mass > 500) ]
  av <- mean( subset )
  return(av)
}

av <- calc_mass( massFrame$Body_mass )
print(av)
```

Note in the above code that comments are preceded by the hash symbol, `#`. Note also the `print` function to print our result to the screen. We can run the script by using the `source` function within R.

Do `help(print)`.

```
> source("func_example.R")
 [1] 16907.29
```

Note that the script must be in the current directory where R is running. Alternatively, in the R console, you can use the `File->Source File` option from the menu at the top.

8 Using external packages

Finally it is worth mentioning that one of the greatest features of R is that there is a huge community of R developers and people who have contributed code to R in the form of **packages**. If the core R program does not contain a particular function or functionality you can look for libraries that do. You can browse some of the packages available in the CRAN package repository http://cran.r-project.org/.

CRAN stands for Comprehensive R Archive Network.

One package that is useful for doing two dimensional smoothing is `sm`. To install the package use the following R code.

```
> install.packages("sm",dependencies=T)
```

Note that you will have to choose a mirror from which to download the package. The `dependencies=T` flag tells R to download all the packages that `sm` depends on. To use the library and make a smoothed contour plot we do do the following.

```
library(sm)
m <- data.matrix( massFrame[,c(4,5)] )
dens <- sm.density( m, display="none", nbins=0 )
contour(dens$eval.points[,1],dens$eval.points[,2], dens$estimate,  col
    =2 )
```

The first line loads the downloaded package. The second line creates a matrix from the 4th and 5th rows of the data frame which we then input into the `sm.density` function on the third line. We then use the `contour` function (another plotting function) to make a contour plot which is shown below.

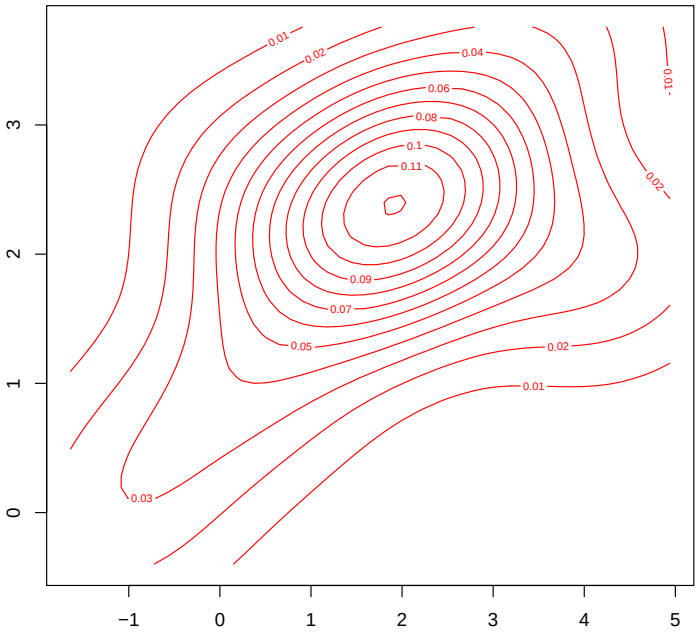

Figure 8.1: Contour plot of the body and brain mass data using two dimensional smoothing in the `sm` package.