4
Control Statements and Functions

4.1 Functions

A very powerful aspect of R is that it is relatively easy to write your own functions. Functions can take inputs (or *arguments*) and return a *single* value. Let's look at some simple functions.

4.1.1 Basic functions

This function takes in a single argument x and returns x^2:

```
> Fun1 = function(x) {
+   return (x*x)
+ }
```

The key elements in the function call are:

- The word `function`;
- The brackets () which enclose the **argument** list.
- A sequence of statements in curly braces { }.
- A `return` statement.

We *call* `Fun1` in the following manner:

```
> Fun1(5)
[1] 25
> y = Fun1(10)
> y
[1] 100
> z = c(1, 2, 3, 4)
> Fun1(z)
[1]  1  4  9 16
```

Of course, the old saying 'garbage in, garbage out' is true:

```
> Fun1()
Error in Fun1() :  argument "x" is missing, no default
```

```
> Fun1("5")
Error in x * x :  non-numeric argument to binary operator
```

The error messages give you an idea of what went wrong.

Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
> Fun2()
[1] 1
> Fun2(4)
[1] 16
> Fun3 = function(x, y) {
+   return (x*y)
+ }
> Fun3(3, 4)
[1] 12
```

4.1.2 A more useful function

Here the function below takes in a vector, plots a histogram and returns a vector containing the mean and standard deviation:

```
> Investigate = function(values) {
+   hist(values)
+   m_std = c(mean(values), sd(values))
+   return(m_std)
+ }
```

Once we have created our function, we can put it to good use[1]:

```
> Investigate(movies$Rating)
[1] 5.522715 1.451864
```

[1] Obviously, a histogram would also be created – it's just not shown here. See Section 3.3.3 for examples of histograms.

4.1.3 Variable scope

When we call a function, R first looks for *local* variables, then *global* variables. For example, `Fun4` uses a global variable:

```
> blob = 5
> Fun4 = function() {
+   return(blob)
+ }
> Fun4()
[1] 5
```

R scoping rules are actually a bit more complicated than described below. R uses something called *lexical* scope, but this doesn't affect us.

However, in `Fun5`, we use a local variable:

```
> Fun5 = function() {
+   blob = 6
+   return(blob)
+ }
> Fun5()
[1] 6
> blob
[1] 5
```

As a general rule, functions should only use **local** variables[2].

[2] This makes your code more portable and less likely to have bugs.

4.2 Conditionals

Conditional statements are features of a programming language which perform different computations or actions depending on whether a condition evaluates to `TRUE` or `FALSE`. They are used in almost all computer programs.

4.2.1 *If* statements

The basic structure of an `if` statement is:

```
> if(expr) {
+   ##do something
+ }
```

where `expr` is evaluated to be either `TRUE` or `FALSE`. The following example illustrates `if` statements in R:

```
> x = 5
> y = 5
> if(x<5) {
+   y = 0
+ }
> y
[1] 5
```

In this code chunk, `x < 5` evaluates to be `FALSE` so the following brackets are not evaluated. Similarly, we can test for greater than:

```
> x = 5
> y = 5
> if(x > 0) {
+   y = 0
+ }
> y
[1] 0
```

Here `x > 0` evaluates to be `TRUE` so, `y` is set equal to 0. If we wanted to test for equality with zero, then we would use `==`.

4.2.2 *If else* statements

We can link together a number of `if` statements[3]:

```
> x = 0
> if(x > 0) {
+   cat("x is greater than zero")
+ } else if(x < 0) {
+   cat("x is less than zero")
+ } else {
+   cat("x must be zero!")
+   cat("\n")
+ }
x must be zero!
```

The final `else` is optional.

[3] What do you think the `cat` command does?

We can also use `if` statements in functions, for example to check that our data is negative we can construct the following function:

```
> IsNegative = function(value) {
+   I = FALSE
+   if(value < 0) {
+     I = TRUE
+   }
+   return(I)
+ }
> IsNegative(1)
[1] FALSE
> IsNegative(-5.6)
[1] TRUE
```

A more sophisticated function could be:

```
> IsGreaterThan = function(value1, value2) {
+   is_greater_than = FALSE
+   if(value1 > value2) {
+     is_greater_than = TRUE
+   }
+   return(is_greater_than)
+ }
```

Which we can then call:

```
> IsGreaterThan(-5, -6)
[1] TRUE
> IsGreaterThan(10, 10)
[1] FALSE
```

4.3 Control statements

At times we would like to perform some operation on a vector or a data frame. Often R has built-in functions that will do this for you, e.g. `mean`, `sd`,... Other times we have to write our own functions. For example, suppose we want to calculate $\sum_{i=1}^{10} i^2$.
In R we can use a `for` loop:

```
> x = 0
> for(i in 1:10) {
+   x = x + i^2
+ }
> x
[1] 385
```

Or perhaps $\sum_{j=-5}^{-1} e^j/j^2$, then:

```
> total = 0
> for(j in -5:-1) {
+   total = total + exp(j)/j^2
+ }
> total
[1] 0.4086594
```

A more tricky example: Calculate $\sum e^k/k^2$, for $k = 3, 6, 9, \ldots, 21$:

```
> total = 0
> for(i in 1:7) {
+   k = i*3
+   total = total + exp(k)/k^2
+ }
> total
[1] 3208939
```

Exercise: Using the inbuilt R function `sum`, calculate the above summations without using `for` loops.

4.4 *Putting it all together*

Rather than have to constantly write R code to solve the summations in §4.3 we can create a function to solve the general form:

$$\sum_{i=i_s}^{i_e} \frac{e^i}{i^2} \quad \text{for } i = i_s,\ i_s + j,\ i_s + 2j,\ \ldots, i_e\ .$$

Can you see what's wrong with the function `Summation1`? There's a prize for anyone who can spot this – and write a function which works all the time!

So in R we have:

```
> Summation1 = function(i_s, i_e, j) {
+   total = 0
+   for(i in 1:(i_e/j)) {
+     k = i*j
+     total = total + exp(k)/k^2
+   }
+   return(total)
+ }
> Summation1(3, 21, 3)
[1] 3208939
```

4.5 *The apply family*

R has been designed with manipulating data in mind. Due to this, there are two important functions that are unique to R.

Probably not unique, but not common in other programming languages.

4.5.1 *The `apply` function*

We use the `apply` function when we want to *apply* the same function to every row or column of a data frame. For example, suppose we have a data frame with three columns:

```
> (df4 = data.frame(c1 = 1:4, c2 = 4:7, c3 = 2:5))
  c1 c2 c3
1  1  4  2
2  2  5  3
3  3  6  4
4  4  7  5
```

The `apply` function takes (at least) three arguments. The first argument is the data frame, the second the number `1` or `2` indicating row or column

(respectively) and the third a function to apply to each row or column. So:

```
> apply(df4, 1, mean)
[1] 2.333333 3.333333 4.333333 5.333333
```

calculates the mean value of every row, while:

```
> apply(df4, 2, sd)
      c1       c2       c3
1.290994 1.290994 1.290994
```

calculates the standard deviation of every column.

Suppose one of the columns was non-numeric:

```
> (df5 = data.frame(c1 = 1:3, c2 = 4:6,
+    c3 = LETTERS[1:3]))
  c1 c2 c3
1  1  4  A
2  2  5  B
3  3  6  C
```

then taking the mean doesn't really make sense:

```
> apply(df5, 1, mean)
[1] NA NA NA
```

Instead, we remove the column, then calculate the mean:

```
> apply(df5[ ,1:2], 1, mean)
[1] 2.5 3.5 4.5
```

4.5.2 The `tapply` function

The function `tapply` is very useful, but at first glance can be tricky to understand. It's best described using an example:

```
> tapply(movies$Length, movies$mpaa, mean)
    NC-17        PG     PG-13         R
110.18750  97.23384 104.97877 100.18818
```

In the above code, we have calculated the average movie length conditional on its MPAA rating. So the average length of a PG movie is 97 minutes and the average NC-17 movie length is 110mins. With `tapply` we can do we very interesting things. For example, in the next piece of code, we plot the average movie length conditional on it's rating:

```
> tapply(movies$Length, movies$Rating, mean)[1:6]
   1  1.2  1.3  1.4  1.5  1.6
85.5 93.0 87.5 85.0 67.0 86.0
> rating_by_len = tapply(movies$Length, movies$Rating,
+   mean)
> plot(names(rating_by_len), rating_by_len)
```

Imagine trying to produce Figure 4.1 in Excel!

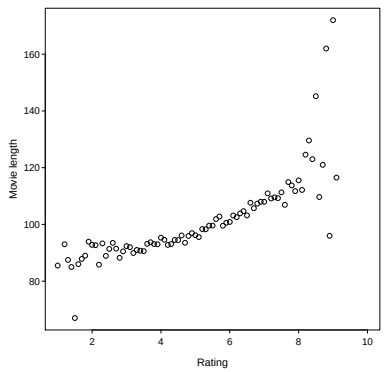

Figure 4.1: Plot of mean movie length conditional on it's rating.

4.6 Help

R has a very good help system. If you need information about a particular function – say `plot` – then typing `?plot` in an R terminal will bring up the associated help page.

The internet is another very good source of R help. Unfortunately, using Google isn't particularly useful since the letter "R" appears on most web pages! However, you can use

http://www.rseek.org/

Using this search engine limits searches to R web-pages.

Summary of R commands

Command	Comment
`for`	A for loop. See §4.3
`if` or `else`	A conditional statement. See §4.2.
`function`	An R function constructor. See §4.1
`cat`	Print command.
`apply` or `tapply`	See §4.5

Table 4.1: Summary of R commands in this chapter.