Part IV

Control Statements and Functions

4.1.1 Basic functions

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

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

4.1.1 Basic functions

The key elements in the function call are:

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

4.1.1 Basic functions

The key elements in the function call are:

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

4.1.1 Basic functions

The key elements in the function call are:

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

4.1.1 Basic functions

The key elements in the function call are:

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

4.1.1 Basic functions

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
```

4.1.1 Basic functions

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
```

4.1.1 Basic functions

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
```

4.1.1 Basic functions

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
```

4.1.1 Basic functions

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
```

Question 1

What is the value of y?

```
> fun1 = function(x) {
+         return(2*x)
+ }
> y = fun1(2)
```

- 5
- 10
- 2
- 4

Question 1

What is the value of y?

```
> x = 5
> fun1 = function(x) {
+         return(2*x)
+ }
> (y= fun1(2))
[1] 4
```

- 5
- 10
- 2
- 4

Question 2

What is the value of y?

```
> fun1 = function(x) {
+   z = 3
+   return(x*z)
+ }
> y = fun1(2)
```

- 2
- 3
- 4
- 6

Question 2

What is the value of y?

```
> fun1 = function(x) {
+   z = 3
+   return(x*z)
+ }
> (y = fun1(2))
[1] 6
```

- 2
- 3
- 4
- 6

Question 3

What is the value of y?

```
> fun1 = function(x) {
+   z = 3
+   return(x*z)
+ }
> k = 5
> y = fun1(k)
```

Question 3

What is the value of y?

```
> fun1 = function(x) {
+   z = 3
+   return(x*z)
+ }
> k = 5
> (y = fun1(k))
[1] 15
```

4.1.1 Basic functions

Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
```

```
> Fun2()
[1] 1
```

```
> Fun2(4)
[1] 16
```

4.1.1 Basic functions

Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
```

```
> Fun2()
[1] 1
```

```
> Fun2(4)
[1] 16
```

4.1.1 Basic functions

Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
```

```
> Fun2()
[1] 1
```

```
> Fun2(4)
[1] 16
```

4.1.1 Basic functions

```
> Fun3 = function(x, y) {
+   return (x*y)
+ }
```

```
> Fun3(3, 4)
[1] 12
```

4.1.1 Basic functions

```
> Fun3 = function(x, y) {
+   return (x*y)
+ }
```

```
> Fun3(3, 4)
[1] 12
```

Question 4

What is the value of y?

```
> fun1 = function(x=3) {
+         return(2*x)
+ }
> y = fun1(2)
```

- 3
- 4
- 6

Question 4

What is the value of y?

```
> fun1 = function(x=3) {
+         return(2*x)
+ }
> (y = fun1(2))
[1] 4
```

- 3
- 4
- 6

Question 5

What is the value of y?

```
> fun1 = function(x=3) {
+   return(2*x)
+ }
> y = fun1()
```

- 3
- 4
- 6

Question 5

What is the value of y?

```
> fun1 = function(x=3) {
+   return(2*x)
+ }
> (y = fun1())
[1] 6
```

- 3
- 4
- 6

Question 6

What is the value of y?

```
> fun1 = function(x=3) {
+   x = 2
+   return(2*x)
+ }
> y = fun1()
```

- 3
- 4
- 6

Question 6

What is the value of y?

```
> fun1 = function(x=3) {
+   x = 2
+   return(2*x)
+ }
> (y = fun1())
[1] 4
```

- 3
- 4
- 6

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)
+ }
```

4.1.2 A more useful function

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

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

```
> Investigate(movies$Length)
[1] 100.87807  17.34152
```

```
> Investigate(movies$Budget)
[1] 10286893 23039711
```

4.1.2 A more useful function

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

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

```
> Investigate(movies$Length)
[1] 100.87807  17.34152
```

```
> Investigate(movies$Budget)
[1] 10286893 23039711
```

4.1.2 A more useful function

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

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

```
> Investigate(movies$Length)
[1] 100.87807  17.34152
```

```
> Investigate(movies$Budget)
[1] 10286893 23039711
```

4.1.3 Variable scope

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
```

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
```

4.1.3 Variable scope

But Fun5 uses a local variable:

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

```
> Fun5()
[1] 6
```

```
> blob
[1] 5
```

4.1.3 Variable scope

But Fun5 uses a local variable:

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

```
> Fun5()
[1] 6
```

```
> blob
[1] 5
```

4.1.3 Variable scope

But Fun5 uses a local variable:

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

```
> Fun5()
[1] 6
```

```
> blob
[1] 5
```

Question 7

What is the value of y?

```
> x = 1
> fun1 = function(x) {
+   x = 2
+   return(x)
+ }
> y= fun1(3)
```

- 1
- 2
- 3

Question 7

What is the value of y?

```
> x = 1
> fun1 = function(x) {
+   x = 2
+   return(x)
+ }
> (y= fun1(3))
[1] 2
```

- 1
- 2
- 3

Question 8

What is the value of y?

```
> x = 1
> fun1 = function(x=3) {
+   x = 2
+   return(x)
+ }
> y= fun1()
```

- 1
- 2
- 3

Question 8

What is the value of y?

```
> x = 1
> fun1 = function(x=3) {
+   x = 2
+   return(x)
+ }
> (y= fun1())
[1] 2
```

- 1
- 2
- 3

Question 9

What is the value of y?

```
> x = 1
> fun1 = function(x=3) {
+   return(2*x)
+ }
> x = fun1(x)
> y = fun1(x)
```

Question 9

What is the value of y?

```
> x = 1
> fun1 = function(x=3) {
+   return(2*x)
+ }
> x = fun1(x)
> (y = fun1(x))
[1] 4
```

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`.

4.2.1 `If` statements

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.

4.2.1 `If` statements

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.

4.2.1 If statements

We test for greater than in a similar manner:

```
> 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 $=>$.

4.3.2 If else statements

We can link together a number of `if` statements:

```
> 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.

4.2.2 If else statements

```
> IsNegative = function(value) {
+   is_neg = FALSE
+   if(value < 0) {
+     is_neg = TRUE
+   }
+   return(is_neg)
+ }
```

```
> IsNegative(1)
[1] FALSE
```

```
> IsNegative(-4.6)
[1] TRUE
```

4.2.2 If else statements

```
> IsNegative = function(value) {
+   is_neg = FALSE
+   if(value < 0) {
+     is_neg = TRUE
+   }
+   return(is_neg)
+ }
```

```
> IsNegative(1)
[1] FALSE
```

```
> IsNegative(-4.6)
[1] TRUE
```

4.2.2 If else statements

```
> IsNegative = function(value) {
+   is_neg = FALSE
+   if(value < 0) {
+     is_neg = TRUE
+   }
+   return(is_neg)
+ }
```

```
> IsNegative(1)
[1] FALSE
```

```
> IsNegative(-4.6)
[1] TRUE
```

4.2.2 If else statements

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)
+ }
```

4.2.2 If else statements

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

```
> IsGreaterThan(10, 10)
[1] FALSE
```

4.2.2 `If else` statements

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

```
> IsGreaterThan(10, 10)
[1] FALSE
```

Question 1

What is the value of y?

```
> x = 5
> if(x==6)  {
+   y = FALSE
+ } else {
+   y = TRUE
+ }
```

- TRUE
- FALSE

Question 1

What is the value of y?

```
> x = 5
> if(x==6)  {
+   y = FALSE
+ } else {
+   y = TRUE
+ }
> y
[1] TRUE
```

- TRUE
- FALSE

Question 2

What is the value of y?

```
> x = 5
> if(x > 6){
+   y = 0
+ } else if(x >= 5) {
+   y = 1
+ } else {
+ y = 2
+ }
```

- 0
- 1
- 2

Question 2

What is the value of y?

```
> x = 5
> if(x > 6){
+   y = 0
+ }else if(x >= 5) {
+   y = 1
+ } else {
+  y = 2
+ }
> y
[1] 1
```

- 0
- 1
- 2

Question 3

What is the value of z?

```
> x = 5
> y = "male"
> if(x > 6 & y == "Female"){
+   z = 0
+ }else if(x < 5 & y == "Male") {
+   z = 1
+ } else {
+   z = 2
+ }
```

- 0
- 1
- 2

Question 3

What is the value of z?

```
> x = 5
> y = "male"
> if(x > 6 & y == "Female"){
+   z = 0
+ }else if(x < 5 & y == "Male") {
+   z = 1
+ } else {
+   z = 2
+ }
> z
[1] 2
```

- 0
- 1
- 2

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
```

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
```

4.3 Control statements

$\sum_{j=-5}^{-1} e^j / j^2$,

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

```
> total
[1] 0.4086594
```

4.3 Control statements

$\sum_{j=-5}^{-1} e^j / j^2$,

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

```
> total
[1] 0.4086594
```

4.3 Control statements

A more tricky example is: 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
```

Question 4

What is the value of x?

```
> total = 0
> for(blob in 1:4) {
+   total = total + 1
+ }
> x = total
```

Question 4

What is the value of x?

```
> total = 0
> for(blob in 1:4) {
+   total = total + 1
+ }
> (x = total)
[1] 4
```

Question 5

What is the value of x?

```
> total = 0
> for(blob in 1:10) {
+   if(blob > 9) {
+     total = total + 1
+   }
+ }
> x = total
```

Question 5

What is the value of x?

```
> total = 0
> for(blob in 1:10) {
+   if(blob > 9) {
+     total = total + 1
+   }
+ }
> (x = total)
[1] 1
```

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:

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

4.4 Putting it all together

So in R we have:

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

```
> Summation1(-5, -1, 1)
[1] 0.4086594
```

```
> Summation1(3, 21, 3)
[1] 3208925
```

4.4 Putting it all together

So in R we have:

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

```
> Summation1(-5, -1, 1)
[1] 0.4086594
```

```
> Summation1(3, 21, 3)
[1] 3208925
```

4.4 Putting it all together

So in R we have:

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

```
> Summation1(-5, -1, 1)
[1] 0.4086594
```

```
> Summation1(3, 21, 3)
[1] 3208925
```

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
```

4.5.1 The apply function

- 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 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.

Standard deviation of every column

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

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
```

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
```

Question 6

What type of variable is z?

```
> apply(z, 1, mean)
```

- Data frame
- Double
- String
- logical

Question 6

What type of variable is z?

```
> apply(z, 1, mean)
```

- Data frame
- Double
- String
- logical

Question 7

Which of these statements is correct?

```
> apply(z, 2, median)
```

- We calculate the median of z
- We calculate the median for each row of z
- We calculate the median for each column of z
- We calculate the mean of z

Question 7

Which of these statements is correct?

```
> apply(z, 2, median)
```

- We calculate the median of z
- We calculate the median for each row of z
- We calculate the median for each column of z
- We calculate the mean of z

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
```

4.5.2 The `tapply` function

- 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.

4.5.2 The `tapply` function

- For example, in the next piece of code, we plot the average movie rating conditional on its length:

```
> 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)
```

4.5.2 The `tapply` function

- For example, in the next piece of code, we plot the average movie rating conditional on its length:

```
> 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)
```

4.5.2 The `tapply` function

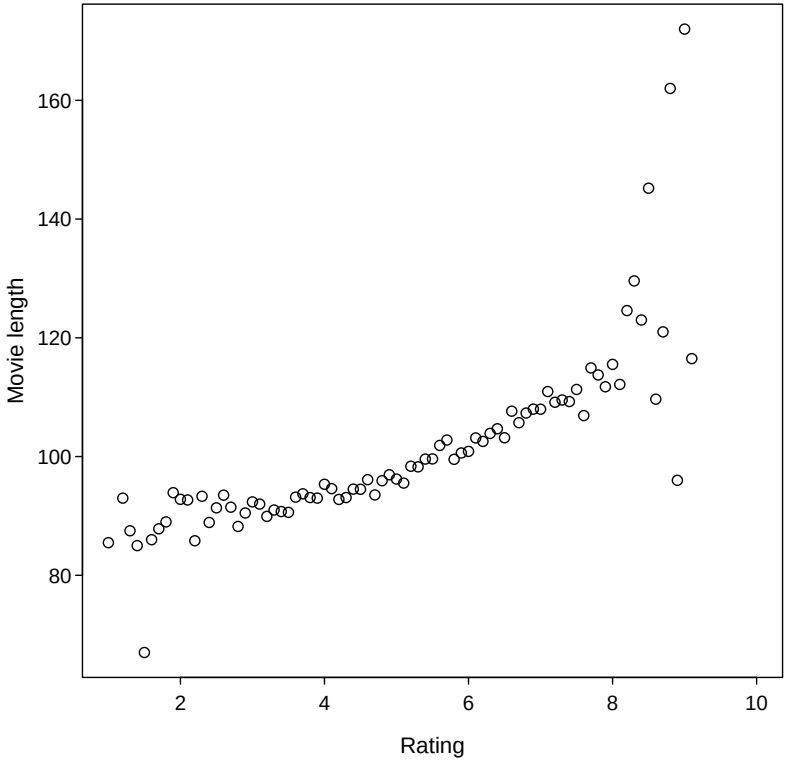

Figure : Plot of movie length conditional on it's rating.

4.6 Help

- R has a very good help system.
- If you need information about a particular function, say `plot`.
- Then typing `?plot` in a 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.