Manuel J. A. Eugster

Programming R

Chapter 5: Control structures

Last modification on May 21, 2012

Draft of the R programming book I always wanted to read
http://mjaeugster.github.com/progr

5 Control structures

Control the flow.

The main literature for this section is:

- *An Introduction to R* by R Core Team (2012a)
- *R Language Definition* by R Core Team (2012b)

5.1 Control flow

R provides basic control-flow constructs; they allow to define the order in which the individual statements of a script or function are evaluated.

The main control flow statements are (cf. Wikipedia, 2012):

Choices: execute a set of statements only if some condition is met;

Loops: execute a set of statements zero or more times, until some condition is met;

Functions: execute a set of distant statements, after which the flow of control usually returns;

Ending: stop the program, preventing any further execution;

Exceptions: handle the occurrence of exceptions;

See Chapter 4 for a detailed introduction into functions.

5.2 Choices

The **if-else** statement conditionally evaluates two statements. The formal syntax is

```
> if ( condition ) {
+   statement1
+ } else {
+   statement2
+ }
```

First, `condition` is evaluated. If it is `TRUE` then `statement1` is evaluated; if it is `FALSE` then `statement2` is evaluated.

A simple example is the sign function which returns -1 if the number x is smaller than 0 and $+1$ if the number x is greater than zero:

```
> sign <- function(x) {
+   if ( x > 0 ) {  # condition1
+     y <- +1       # statement1
+   } else {        # !condition1
+     y <- -1       # statement2
+   }
+   y
+ }
>
> c(sign(-10), sign(pi), sign(10))

[1] -1  1  1
```

The function, however, is not correctly defined for $x = 0$:

```
> sign(0)

[1] -1
```

According to the mathematical definition of the sign function the return value should be 0. In order to achieve this, we need nested if/else statements:

```
> sign <- function(x) {
+   if ( x > 0 ) {         # condition1
+     y <- +1              # statement1
+   } else if ( x < 0 ) {  # condition2
+     y <- -1              # statement2
+   } else {               # !(condition1 | condition2)
+     y <- 0               # statement3
+   }
+   y
+ }
>
> c(sign(-10), sign(0), sign(10))

[1] -1  0  1
```

Note that if the conditions in if-else statements are vectors, only the first element is used; therefore vectorized computation is not possible with our implementation of the sign function:

```
> sign(c(-1.2, exp(1), 13.873))

Warning message: the condition has length > 1 and only the first element will be used

Warning message: the condition has length > 1 and only the first element will be used

[1] -1
```

See `ifelse()` for a vectorized conditional statement.

Note that in some cases the function `switch()` allows to write nested if-else statements more elegantly (multiway branches).

5.3 Loops

A loop is a sequence of statements which is specified once but which may be carried out several times in succession. R has three statements that provide explicit looping (see `?Control`); here we discuss the **for** and **while** loops.

The most common one (in my feeling) is the count- and collection-controlled **for** loop; its formal syntax is

```
> for ( var in seq ) {
+   statement1
+ }
```

where `var` is the loop variable, `seq` is a vector expression (often a sequence like `1:20`) and `statement1` is evaluated as often as there are elements in `seq`.

A simple example is a function which sums the elements of a vector:

```
> adder <- function(x) {
+   y <- 0
+   for ( e in x ) {  # var in seq
+     y <- y + e      # statement1
+   }
+   y
+ }
```

```
>
> adder(1:10)


[1] 55
```

The loop body is executed for each element of x; in the first iteration `e` is `x[1]`, in the second iteration `e` is `x[2]`, and so on.

In case of the for loop, the number of iterations is defined (i.e., the number of elements in `seq`). This is not always true, sometimes we want to iterate until a specific condition is true—the condition-controlled **while** (and **repeat**) loop allow this. Its formal syntax is

```
> while ( condition ) {
+   statement1
+ }
```

While `condition` is `TRUE`, `statement1` is executed.

A simple example is a function which adds up random numbers between 1 and 100 until the sum is greater than a number x:

```
> radder <- function(x) {
+   y <- 0
+   i <- 0
+   while ( y <= x ) {          # condition
+     y <- y + sample(100, 1)   # state...
+     i <- i + 1                #       ...ment1
+   }
+   c(x = x, i = i, y = y)
+ }
>
> set.seed(1234)
> radder(20)


 x  i  y
20  2 75


> radder(831)


  x   i   y
831  16 837
```

Beware of creating infinite loops, i.e., loops where `condition` is always `TRUE`; e.g.,

```
> x <- 1
> while ( x > 0 ) {
+   x <- x + 1
+ }
```

The escape key `ESC` interupts the execution in such a case.

5.4 Ending

The **return** statement exits from the current function, and control flow returns to where the function was evaluated. Its formal syntax is

```
> return(value)
```

where `value` is the the R object which is returned to the caller (either an other function or the user).

This now allows us to capture an "unfavorable" input, namely

```
> radder(Inf)
```

which leads to an infinite loop.

```
> radder <- function(x) {
+   if ( x == Inf ) {
+     return(c(x = x, i = Inf, y = Inf))
+   }
+
+   y <- 0
+   i <- 0
+   while ( y <= x ) {
+     y <- y + sample(100, 1)
+     i <- i + 1
+   }
+   c(x = x, i = i, y = y)
+ }
>
> radder(Inf)


  x   i   y
Inf Inf Inf
```

Note that `stopifnot()` and `stop()` also stop the evaluation of the current function and exit it with an error message.

5.5 Exceptions

Exceptions are anomalous or exceptional situations requiring special processing during computation (e.g., an error occurs in a called function). The process of responding to such exceptions is called **exception handling**.

R provides `tryCatch()` for handling conditions. Its formal syntax is

```
> tryCatch(expr, ...)


Error: '...' used in an incorrect context
```

where `expr` is evaluated and might fail. The programmer then has the change to define handlers for the possible problems.

For an example let us implement a function which draws n random numbers between -0.5 and 1, draws an error if the sum of these numbers is smaller than 0, and otherwise returns the sum.

```
> f <- function(n) {
+   x <- runif(n, min = -0.5, max = 1)
+   if ( sum(x) < 0 ) {
+     stop("Sorry, sum(x) < 0")
+   }
+   sum(x)
+ }
>
> set.seed(1234)
> f(10)


[1] 2.338


> f(10)


[1] 1.82
```

Now, we use this function in another function which implements a simulation with m replications. However, we do not want that the function stops when `f` throws an error—we just want to ignore the error and continue with the remaining iterations.

```
> g <- function(n, m) {
+   y <- numeric(length = m)
+
+   for ( i in seq(length = m) ) {
+     y[i] <- tryCatch(f(n),
+                      error = function(e) {
+                        warning("Error in f(); using NA instead.")
+                        NA_real_
+                      })
+   }
+   y
+ }
>
> set.seed(1222)
> g(10, 10)

 [1] 1.344 1.628 3.297 3.214 1.588 2.835 3.933 4.026 2.210 3.473


> g(10, 10)

Warning message: Error in f(); using NA instead.


Warning message: Error in f(); using NA instead.


 [1] 2.1004 3.0961 2.4744 3.3155 1.3407 4.8220     NA 2.5773 0.9176     NA
```

This is especially useful in longer simulations with a huge number of iterations, where the error is catched (and saved) but the execution is continued.

Bibliography

John Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008. ISBN 9780387759357.

R Core Team. *An Introduction to R*, 2012a. URL http://cran.r-project.org/doc/manuals/R-intro.html.

R Core Team. *The R language definition*, 2012b. URL http://cran.r-project.org/doc/manuals/R-lang.html.

Wikipedia. Control flow. Wikipedia entry, accessed on 2012-05-21, 2012. URL http://en.wikipedia.org/wiki/Control_flow.