13 structures de contrôle

Regarder une vidéo de cette section

Les structures de contrôle dans R permettent de contrôler le flux d’exécution d’une série d’expressions R. Fondamentalement, les structures de contrôle vous permettent d’intégrer une certaine «logique» dans votre code R, plutôt que d’exécuter toujours le même code R à chaque fois. Les structures de contrôle vous permettent de répondre aux entrées ou aux caractéristiques des données et d'exécuter différentes expressions R en conséquence.

Les structures de contrôle couramment utilisées sont

- `if` et `else` : tester une condition et agir dessus
- `for` : exécuter une boucle un nombre fixe de fois
- `while` : exécuter une boucle *tant* qu'une condition est vraie
- `repeat` : exécuter une boucle infinie (il faut en `break` sortir pour s'arrêter)
- `break` : interrompre l'exécution d'une boucle
- `next` : ignorer l'interaction d'une boucle

La plupart des structures de contrôle ne sont pas utilisées dans les sessions interactives, mais plutôt lors de l'écriture de fonctions ou d'expressions plus longues. Cependant, ces constructions ne doivent pas nécessairement être utilisées dans les fonctions et il est judicieux de se familiariser avec elles avant de se plonger dans les fonctions.

13.1 `if` - `else`

Regarder une vidéo de cette section

La combinaison `if` - `else` est probablement la structure de contrôle la plus couramment utilisée dans R (ou peut-être n'importe quelle langue). Cette structure vous permet de tester une condition et d’agir en conséquence, que celle-ci soit vraie ou fausse.

Pour commencer, vous pouvez simplement utiliser la `if` déclaration.

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```

The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an `else` clause.

```
if(<condition>) {
        ## do something
}
else {
        ## do something else
}
```

You can have a series of tests by following the initial `if` with any number of `else if` s.

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

Here is an example of a valid if/else structure.

```
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

The value of `y` is set depending on whether `x > 3` or not. This expression can also be written a different, but equivalent, way in R.

```
y <- if(x > 3) {
        10
} else {
        0
}
```

Neither way of writing this expression is more correct than the other. Which one you use will depend on your preference and perhaps those of the team you may be working with.

Of course, the `else` clause is not necessary. You could have a series of if clauses that always get executed if their respective conditions are true.

```
if(<condition1>) {

}

if(<condition2>) {

}
```

13.2 `for` Loops

Watch a video of this section

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I've found very few situations where a for loop wasn't sufficient.

In R, for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10) {
+         print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, …, 10, executes the code within the curly braces, and then the loop exits.

The following three loops all have the same behavior.

```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+         ## Print out each element of 'x'
+         print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

The `seq_along()` function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object `x`).

```
> ## Generate a sequence based on length of 'x'
> for(i in seq_along(x)) {
+         print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

It is not necessary to use an index-type variable.

```
> for(letter in x) {
+         print(letter)
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

For one line loops, the curly braces are not strictly necessary.

```
> for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

However, I like to use curly braces even for one-line loops, because that way if you decide to expand the loop to multiple lines, you won't be burned because you forgot to add curly braces (and you *will* be burned by this).

13.3 Nested `for` loops

`for` loops can be nested inside of each other.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

13.4 `while` Loops

Watch a video of this section

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
> while(count < 10) {
+         print(count)
+         count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

While loops can potentially result in infinite loops if not written properly. Use with care!

Sometimes there will be more than one condition in the test.

```
> z <- 5
> set.seed(1)
>
> while(z >= 3 && z <= 10) {
+         coin <- rbinom(1, 1, 0.5)
+
+         if(coin == 1) {  ## random walk
+                 z <- z + 1
+         } else {
+                 z <- z - 1
+         }
+ }
> print(z)
[1] 2
```

Conditions are always evaluated from left to right. For example, in the above code, if `z` were less than 3, the second test would not have been evaluated.

13.5 `repeat` Loops

Watch a video of this section

`repeat` initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a `repeat` loop is to call `break` .

One possible paradigm might be in an iterative algorith where you may be searching for a solution and you don't want to stop until you're close enough to the solution. In this kind of situation, you often don't know in advance how many iterations it's going to take to get "close enough" to the solution.

```r
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()

        if(abs(x1 - x0) < tol) {  ## Close enough?
                break
        } else {
                x0 <- x1
        }
}
```

Note that the above code will not run if the `computeEstimate()` function is not defined (I just made it up for the purposes of this demonstration).

The loop above is a bit dangerous because there's no guarantee it will stop. You could get in a situation where the values of `x0` and `x1` oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a `for` loop and then report whether convergence was achieved or not.

13.6 `next` , `break`

`next` is used to skip an iteration of a loop.

```r
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

`break` is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
      print(i)

      if(i > 20) {
              ## Stop loop after 20 iterations
              break
      }
}
```

13.7 Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Les boucles infinies devraient généralement être évitées, même si (vous pensez), elles sont théoriquement correctes.
- Les structures de contrôle mentionnées ici sont principalement utiles pour écrire des programmes; pour un travail interactif en ligne de commande, les fonctions «appliquer» sont plus utiles.