Flow control in R

Up to now we have worked with so-called linear code, in which every step from top to bottom is executed in sequence. Oftentimes, this might not be the best approach for a given problem. This is where flow control plays a role. We will be examining how we can use the **i**f and **f**or statements to make meaningful changes to how our code is executed

If statement By far the most common flow control tool is the conditional or if statement. The syntax of an if statement consists of three parts:
a. ”if”
b. a Boolean (TRUE/FALSE) test, enclosed in brackets
c. block of code to be executed if the test returned TRUE
Let us try it:

```
x <- 6
if (x == 4) {x <- x-2}
if (x == 6) {x <- x+2}
x
```

```
[1]8
```

Here, x was not equal to four, so the first if statement evaluated FALSE and was not executed. The second if statement evaluated TRUE, and x was increased by 2.

Now imagine a situation where you want to execute one block of code when some condition is true, but a different block of code the rest of the time. We could use two if statements for the two cases, but a faster and more clean solution is to use the else keyword. When directly following an if statement, the else statement allows us to define a block of code that is executed only if the if statement was evaluated as false. For example:

```
x <- 5
if ((x %% 2) == 0) {x <- X/2}
else {x <- (x+1)/2}
x
```

```
[1]3
```

Clearly, if and if/else statements are powerful tools to control what happens with your data.

For loops The second type of flow control is a loop called the for loop. for loops allow us to repeatedly execute a block of code a certain amount of times. The syntax of a for loop consists of three parts:
a. "for"
b. an iterator definition between brackets, consisting of:

- a name you choose for your iterator variable
- "in"
- a vector that contains all values for your iterator

c. a block of code to be executed
An example of a simple for loop would be:

```
for (i in 1:3) {print(2)}
```

```
[1] 2
[1] 2
[1] 2
```

In this example, the block code is executed exactly three times in a loop. The only difference between these loops is the value of the iterator "i" during these executions. If we print the value of the iterator during each loop, we can see this more clearly:

```
for (i in 1:3) {print(i)}
```

```
[1] 1
[1] 2
[1] 3
```

Having learned how both if statements and for loops work, gives us many possibilities for flow control. One exceptionally powerful tool is the ability to ”nest” flow control inside the code block of another flow control tool:

```
for (i in 1:3) {
if (i == 2) {print (15}
else {print (i)}
}
```

```
[1] 1
[1] 15
[1] 3
```

Here, the if statement was executed inside each loop of the for loop.

Break and next

For loops can get rather big and bulky when iterating over large data sets, especially if the code block contains computationally expensive functions. In this case, we want to run as little code as possible. For this we have two additional keywords for flow control: break and next.

The next keyword stops the current for loop and goes directly to the next iteration. For example:

```
for (i in -3:3){
if (i <= 0) {next}
else {print(5/i)}
}
```

```
[1] 5
[2] 2.5
[3] 1.666667
```

In this case, the first four loops were quickly skipped with the next keyword.

The break keyword goes one step further: using it during any loop exits the entire for statement, skipping all loops that have not yet been executed. This is extremely useful if you are using your for statement to look for just one value or solution to a problem. For example:

```
mydata <- c(1:1000)
```

```
for (i in mydata){
  if ((i %% 13) == 0) {
  print(i)
  break
  }
}
```

```
[1] 13
```

This code looks for the first value in mydata that is divisible by 13, prints that value, and then breaks out of the entire for loop.

Examples As an extra example, let us write a function that finds the minimum number in a vector. This exercise uses both if and for for the flow control

```
minimum <- function (input_vector) {
 lowest <- input_vector[1]
 for (i in input_vector) {
 if (i < lowest) {
 lowest <- i }
 }
 return (lowest)
}
myvec<- c(9, 21, 3, 45)
minimum(myvec)
```

```
[1] 3
```

When we call the function *m*inimum, *lowest* variable is set to 9 (in our example). i takes the values $9, 21, 3$, and 45. When i is 9, the variable *lowest* does not change its value. Also when i is 21, *lowest* remains to be 9. But when i is 3, *lowest* becomes 3 because the expression $i < lowest$ becomes TRUE. When i is 45 the value of *lowest* does not change. Our function *minimum* returns 3, the value of the local variable *lowest*.

It is also possible to make nested if loops. For example:

```
x <- 0
if (x < 0) {
print("Negative number")
} else if (x > 0) {
print("Positive number")
} else
print("Zero")
```

```
[1] "Zero"
```

The for loop counters do not always need to be numbers; it is also possible to use character vectors. For example:

```
teams <- c("team_A","team_B")
for (value in teams){
   print(value)}
```

```
[1] "team_A"
[1] "team_B"
```

Imagine we want to get the total goals scored in a game and store them in the vector. We can this using a for loop. We need to start with an empty vector and add a new value to this vector at each for loop:

```
matches <- list(c(2,1),c(5,2),c(6,3))
total_goals <- c()
for (match in matches){
    total_goals <- c(total_goals, sum(match))
}
total_goals
```

```
[1] 3 7 9
```

Our program can also print whether the first team won or lost the game. Assuming the first team's goals is the first index of each pair of values and the opponents is the second index:

```
matches <- list(c(2,1),c(5,2),c(6,3))
for (match in matches)
{
    if (match[1] > match[2]){
        print("Win")
    }
else {
         print ("Lose")
        }
}
```

```
[1] "Win"
[1] "Win"
[1] "Win"
```