STRUCTURE PROGRAMMING IN R

2 Loops

Loops can come in handy on numerous occasions. While loops are like repeated if statements; the for loop is designed to iterate over all elements in a sequence. Learn all about them in this chapter.

While loop

In this chapter I'll be talking about while loops. The while loop is somewhat similar to the if statement: it executes the code inside if the condition is true. However, as opposed to the if statement, the while loop will continue to execute this code over and over again as long as the condition is true.

The syntax of a while loop is very similar to the if statement, as you can see here. Let's have a look at very simple example: we'll simply make R increment a counter until it reaches value 7, my lucky number. We start by defining the variable ctr, short for counter, and setting it to 1. Let's first set the "condition" of the while loop, without worrying about the expressions inside it. We want the while loop to execute as long as the ctr variable is less than or equal to 7. For the initial value of ctr equal to 1, the condition will evaluate to TRUE, but also for other values, such as 3, -5 and 7, this condition will be TRUE. Next up is the expression. What do we want to while loop to do on every run? We want some information on how the while loop is progressing, so we'll throw in a print statement, together with the paste function. If ctr equals 2, for example, this expression will print out "ctr is set to 2". We're not done yet!

```
while loop
while(condition)
{
  expr
}
```

```
while loop
while(ctr <= 7) {
  print(paste("ctr is set to", ctr))
  ctr <- ctr + 1
  }
```

We still have to add another line of code to inform R that we want to increment the ctr variable on every run. We add ctr assign operator ctr +1 to the loop code.

Let's first try to guess how R will handle this while loop. Before R arrives at the while loop, ctr will be 1. The condition evaluates to TRUE, so the code inside the while loop gets executed. R will print "ctr is set to 1", and then set ctr to ctr + 1; ctr now equals 2.

Now, as opposed to the if statement, R takes another look at the condition ctr less than or equal to 7. The current value of ctr is 2 so condition is TRUE. R executes the code inside the while loop again, prints out "ctr is set to 2" and increments the ctr variable. This will go on for ctr equal to 3, 4, 5 and 6. What happens after ctr is set to 7 on the 6th run? R checks the while loop's condition: it's still TRUE, because 7 is less than or equal to 7. R prints ctr is set to 7 and then increments the ctr. Now, the condition will be checked once more. But this time ctr will be equal to 8, which is greater than 7, so the condition evaluates to FALSE, forcing R to abandon the while loop. Curious if our abstract thinking was correct? We'll simply execute the R code and can find out. Indeed, the "ctr is set to" sentence are printed out for numbers 1 to 7. If we now check the value of ctr we see that indeed, ctr is equal to 8. 8 is the first value for ctr for which the condition fails, so R does not increment ctr further.

The line of code to increment ctr is crucial. Suppose we remove this line. If we run this code in R, the line "ctr is set to 1" would be printed indefinitely, until we stop the session manually with Control C. Why? Because ctr does not get updated; this would mean that the condition is always true, and R keeps on re-executing the code in the while loop. You'll have to hit the stop

```
indefinite while loop
while(ctr <= 7) {
  print(paste("ctr is set to", ctr))
  }
```

```
"ctr is set to 1"
"ctr is set to 1"
"ctr is set to 1"
"ctr is set to 1"
"ctr is set to 1"
...
```

sign in your R console to stop this. What I truly want to say here: always make sure your while loop will end at some point! There's one more thing I want to discuss before you get started with the exercises.

The break statement. The break statement simply breaks out of the while loop: when R finds it, it abandons the currently active while loop.

```
break statement
ctr <-1
while(ctr <= 7) {
  if(ctr %% 5 == 0) {
    break
  }
  print(paste("ctr is set to", ctr))
  ctr <- ctr + 1
}
```

```
cities <- list("New York", "Paris",
            "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
for(city in cities) {
    print(city)
  }
```

```
"ctr is set to 1"
"ctr is set to 2"
"ctr is set to 3"
"ctr is set to 4"
```

Suppose we want R to stop our while loop from before as soon as the value of ctr is divisible by 5. We can do this with a break statement. If we now run this pieces of code, the sentence is only printed out 4 times, for the ctr values 1 to 4. If we check out the ctr variable, it is equal to 5, because for ctr equal to 5, the condition that checked if ctr was divisible by 5 became TRUE, and the while loop was abandoned.

For loop

The for loop is somewhat different from the while loop. Have a look at this 'recipe'. This can be read as: for each var, a variable, in seq, a sequence, execute expressions. Make sense? Let's see how this actually works with an example. Suppose you have a vector, cities containing the names of a number of cities. We can simply print the cities vector to the console. But suppose we want to have a different printout for every element in the vector. We can accomplish this using a for loop. Let's start from the recipe and convert it to a functional for loop step by step. Inside the parentheses, we write 'city in cities', meaning that we want to execute the code in the expression block for every city in the cities vector. We'll simply replace the expression by a simple print statement for starters.

```
for loop
for(var in seq) {
  expr
}
```

```
cities <- c("New York", "Paris",
            "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
cities
```

```
"New York" "Paris" ... "Cape Town"
```

```
cities <- c("New York", "Paris",
            "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
for(city in cities) {
    print(city)
  }
```

How does R handle this code? At the start of the loop, R evaluates the seq element, being cities in our case. It realizes that it is a vector containing 6 elements. Next, R stores the first element of this sequence in the variable city, so city equals "New York". Then, the expression, print(city), is executed, printing out "New York" to the console. After the execution, R stores the second element of the cities vector, "Paris", in city and re-runs the code. This process repeats itself until all cities in the cities vector are iterated over. The final result looks like this: for each city, a separate printout was done.

```
"New York"
"Paris"
"London"
"Tokyo"
"Rio de Janeiro"
"Cape Town"
```

```
cities <- list("New York", "Paris",
            "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
for(city in cities) {
    print(city)
  }
```

The for loop does not only work on vectors: it also works with lists for example. Suppose that the cities vector is a list instead of a vector: The exact same for loop as we've been using before can be used for lists, and the result is exactly the same. So, there's no need to worry about the difference between subsetting vectors and lists, the for loop does this for us. I would encourage you to try the for loop with different data structures as well, such as matrices and data frames. I won't go into detail on these in this chapter. Instead, I want to talk about two control statements for loops.

```
"New York"
"Paris"
"London"
"Tokyo"
"Rio de Janeiro"
"Cape Town"
```

```
break statement
for(city in cities) {
  if(nchar(city) == 6) {
    break
  }
  print(city)
}
```

The first one is break, the second one is next. The break statement is a statement that you already know: just like in the while loop, break in a for loop simply stops the execution of the code and abandons the for loop altogether. Suppose we want to leave the for loop as soon as we encounter a city that consists of 6 characters.

We can use the nchar function, which stands for number of characters, inside an if statement for this: How will R deal with this code? Well, for the city in the cities vector, "New York", the nchar conditional is false, so the "New York: still gets printed to the console. The same happens for "Paris". But in the third iteration, when city is equal to "London", the nchar condition is TRUE, causing the for loop to break. Since the break construct comes before the print command, the character string "London" is not printed to the console anymore. If we run the code, we see that indeed, only "New York" and "Paris" get printed to the console, after which the for loop is abandoned.

```
"New York"
"Paris"
```

The next statement also alters the flow of your for loop, but does so in a slightly different way. Let's see what happens if we change the break statement by the next statement and execute the entire for loop again. All city names except for "London" get printed to the console. How could this happen? Because the next statement skips the remainder of the code inside the for loop and proceeds to the next iteration. So as soon as next is encountered, the print(city) part is not processed and the for loop is continued. Of course, it is perfectly possible to use both break and next in the for loop.

```
next statement
for(city in cities) {
  if(nchar(city) == 6) {
    next
  }
  print(city)
}
```

```
"New York"
"Paris"
"Tokyo"
"Rio de Janeiro"
"Cape Town"
```

Before you can nave some more looping fun in the exercises, I want to talk about another way we can loop over different data structures. Let's retake the basic for loop that prints the city names that are stored in a vector. Suppose that instead of simply printing out the city's name, we also want to give information on the city's position in the vector. We can't use this construct, given that we don't have access to the so-called looping index. This index is a counter that R uses behind the scenes to know which element to select on every iteration. In the first iteration, the looping index is 1, and the first element of the cities vector is selected. But what if we want to use this looping vector ourselves? There's no way for us to access it.

```
for loop: v2
cities <- c("New York", "Paris",
            "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
for(i in 1:length(cities)) {
    print(city)
}
```

Fortunately, we can easily solve this. Instead of iterating over the cities, we can manually create a looping index ourselves. Let's start with changing the looping details.

Now, we let I progress from 1 to length of the cities vector, which is 6, by steps of 1. Remember that 1 colon 6 is a compact way of coding a vector containing the element 1, 2, 3, 4, 5 and 6. By using a manual looping index, we lose our city variable, so we have to change the contents of the for loop as well. We now do the subsetting of the vector explicitly, using square brackets. The result is exactly the same as before. This might seem a bit more work, but we now gain access to the index as well. Adding some more information is easier now:

```
"New York"
"Paris"
"London"
"Tokyo"
"Rio de Janeiro"
"Cape Town"
```

```
for loop: v2
cities <- c("New York", "Paris",
            "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
for(i in 1:length(cities)) {
    print(paste(cities[i], "is one position",
                i, "in the cities vector."))
}
```

I can imagine that you're wondering, "Which one of the two is best?" It depends. The first one, the city in cities version, is typically more concise and easier to read, but does not give access to all looping information. The version with the explicit looping index takes more thought to write, but gives you all the information you need.

```
"New York is on position 1 in the cities vectors."
"Paris is on position 2 in the cities vectors."
"London is on position 3 in the cities vectors."
"Tokyo is on position 4 in the cities vectors."
"Rio de Janeiro is on position 5 in the cities vectors."
"Cape Town is on position 6 in the cities vectors."
```