Conditions and loops

Marcus Wurzer

Conditions

Conditions makes the distinction of cases possible, i.e., which operations are exectued on different parts of the data. In R, `if()` and `ifelse` are the two most commonly used statements.

The `if()` statement

The `if` function works the following way:

- `if (condition) {commands when TRUE}`
- `if (condition) {commands when TRUE} else {commands when FALSE}`

The `else` part is optional. Since the following condition evaluates to FALSE, `y` is not created:

```
x <- 3
if (x == 2) y <- 2 * x
y
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

The following condition evaluates to `TRUE`:

```
if (x > 2) y <- 3 * x
y
```

```
## [1] 9
```

Using an `else` part:

```
if (x == 2) y <- 3 * x else y <- 3 / x
y
```

```
## [1] 1
```

The following evaluated to FALSE, and `y` was created according to the `else` part. We didn't use curly braces for these very simple commands, but we need them when we have blocks of commands that may comprise multiple lines:

```
x <- 3
if (x == 2) {
  y <- 3 * x
} else {
  x <- 2 * x
  y <- 3 / x
}
x
```

```
## [1] 6
```

```
y
```

```
## [1] 0.5
```

x was multiplied with 2 and y was than created using the altered x. Be careful: The following code will trigger an error when executed because R executes the first line before the second is entered:

```
if (x == 2) {y <- 3 * x}
else {y <- 3 / x}
```

```
## Error: <text>:2:1: unexpected 'else'
## 1: if (x == 2) {y <- 3 * x}
## 2: else
##    ^
```

The `ifelse()` statement

When using `if () {} else() {}`, the condition must be a logical vector of length 1 that is not `NA` (i.e., only `TRUE` or `FALSE`). If it has a length greater than one (i.e., a sequence of `TRUE`/`FALSE` values), an error is thrown:

```
x <- 3
if (x == c(5, 3, 1)) y <- 3 * x else y <- 3 / x
```

```
## Error in if (x == c(5, 3, 1)) y <- 3 * x else y <- 3/x: the condition has length > 1
```

```
y
```

```
## [1] 0.5
```

In this case, we can use `ifelse` to evaluate the vector-valued condition and get a component-wise evaluation:

```
ifelse (x == c(5, 3, 1), y <- 3 * x, y <- 3 / x)
```

```
## [1] 1 9 1
```

Loops

Loops are needed when we have a large number of commands that are evaluated repeatedly, e.g., when performing operations with different start values and/or parameters or when using iterative algorithms, where the inputs of later iterations are dependent on the preceding ones. There are three variants of loops that are used when working with R:

- `repeat {while}`
- `while {condition} {command when TRUE}`
- `for (name in vector) {command}`

The `repeat()` statement

`repeat()` is used if we want to repeat a command without having a fixed number of replications (press *Escape* to interrupt R).

```
i <- 0
repeat {
  i <- i + 1
}
```

To avoid infinite loops, we need a `break` statement:

```r
i <- 0
repeat {
  i <- i + 1
  break
}
i
```

```
## [1] 1
```

We need to set some break condition, otherwise the loop breaks immediately:

```r
i <- 0
repeat {
  i <- i + 1
  if (i == 3) break
}
i
```

```
## [1] 3
```

We may print the results after each iteration:

```r
i <- 0
repeat {
  i <- i + 1
  print(i)
  if (i == 3) break
}
```

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

Using the `next` statement, the loop directly jumps back to the beginning - only after the last iteration, the final result is printed:

```r
i <- 0
repeat {
  i <- i + 1
  if (i < 3) next
  print(i)
  if (i == 3) break
}
```

```
## [1] 3
```

The `while()` statement

If we want to repeat statements, but do not know about the pattern of repetition beforehand, we make use of the `while()` statement. As long as the condition holds, the commands are evaluated:

```r
i <- 0
while (i < 3)
  i <- i + 1
i
```

```
## [1] 3
```

We can see that this is a simpler way to get the same result as for the `repeat()` loop we used above.

The `for()` statement

`for ()` statements are used if we want to repeat a certain operation a fixed number of times:

```
x <- c(3, 6, 4, 8, 0)
for (i in x) print (i ^ 2)
```

```
## [1] 9
## [1] 36
## [1] 16
## [1] 64
## [1] 0
```

Using a continuous index in order to index objects is a common task when using loops. Modifying the above example, we get:

```
for (i in seq(along = x)) print(x[i] ^ 2)
```

```
## [1] 9
## [1] 36
## [1] 16
## [1] 64
## [1] 0
```

Vectorization

R works vectorized (remember: even scalars are just represented by vectors of length 1), which often leads to more efficient computation (and simpler code) when making use of it. This is true for most of the functions as well, and we rarely need loops when applying an operation to all elements of a vector. For the example above, we could simply write:

```
x ^ 2
```

```
## [1]  9 36 16 64  0
```

This is also valid for self-defined functions, provided that they call vectorized functions themselves:

```
square <- function(x) x * x
square(x)
```

```
## [1]  9 36 16 64  0
```

More information on functions can be found in the corresponding notebook.

Why does vectorization lead to more efficiency (i.e., speedup)? Because R is an interpreted language, i.e., instructions are executed directly/in real time. (Note that this is true for your own R code, but many commands are written in C and thus, run in fast, pre-compiled machine code.) When working with loops, every line has to be interpreted again for every iteration, whereas this has to be done only one time when using vector-valued operations. Loops have a bad reputation in R for being slow. This is not always true, but in principle, vectorization can lead to dramatic speedups. The `apply()` family of functions that is of special importance for vector-valued programming is discussed in the next chapter.