4. Control flow

4.1 Logicals

We saw already the data type `logi` (logical).

4.1.1 Logical Operators

Logical operators are used to perform logical comparisons.

(a) And (`&`)

```
x <- c(TRUE, FALSE, TRUE, FALSE)
y <- c(TRUE, TRUE, FALSE, FALSE)
x & y  # returns TRUE FALSE FALSE FALSE
```

```
## [1]  TRUE FALSE FALSE FALSE
```

(b) Or (`|`)

```
x | y  # returns TRUE TRUE TRUE FALSE
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

(c) Negation (`!`)

```
! x # returns FALSE TRUE FALSE TRUE
```

```
## [1] FALSE  TRUE FALSE  TRUE
```

4.1.2 Logical Comparisons

Logical comparisons return `TRUE` or `FALSE` based on certain conditions.

(a) Equality (`==`)

```
2 == 2  # returns TRUE
```

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

```
3 == 2  # returns FALSE
```

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

(b) Inequality (`!=`)

```
3 != 2  # returns TRUE
```

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

(c) Greater than (>), Less than (<)

```
2 < 2 # returns FALSE
```

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

```
2 > 1 # returns TRUE
```

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

(d) Greater than or equal to (>=), Less than or equal to (<=)

```
2 <= 2 # returns TRUE
```

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

```r
2 >= 1 # returns TRUE
```

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

4.2 If-else conditions

Various actions depend often on certain conditions which are not known beforehand. Such situations may be handled with if-else statements.

4.2.1 If-else

The general syntax can be:

```
if (condition){
   statements to be executed if condition == TRUE
} else{
   statements to be executed if condition == FALSE
}
```

Example

```r
x <- 5

if (x > 3) {
  print("x is greater than 3")
} else {
  print("x is less than or equal to 3")
}
```

```
## [1] "x is greater than 3"
```

4.2.2 Using if without else

You can use a standalone if statement without an else clause if you only want to execute the code when the condition is TRUE. The notation then reduces to

```
if (condition){
   statements to be executed if condition == TRUE
}
```

Example

```r
x <- 5

if (x > 3) {
  print("x is greater than 3")
}
```

```
## [1] "x is greater than 3"
```

4.2.3 Chained Conditions

R also allows you to chain condition. This is useful when you have more than two possible branches. The notation is

```
if (condition1){
   statements to be executed if condition1 == TRUE
} else if (condition 2){
   statements to be executed if condition1 == FALSE
```

```
   and condition2 == TRUE
} else{
   statements to be executed if condition1 == FALSE
   and condition2 == FALSE
}
```

This can be repeated using additional conditions.

Example

```
x <- 10

if (x < 0) {
  print("x is negative")
} else if (x == 0) {
  print("x is zero")
} else {
  print("x is positive")
}
```

```
## [1] "x is positive"
```

Example 2

Expressing temperatures on the Fahrenheit, Celsius and Kelvin scale.

```
# Choose a scale (pick one):
Scale <- "Fahrenheit"
Scale <- "F"
Scale <- "Celsius"
Scale <- "C"
Scale <- "Kelvin"
Scale <- "K"

# Specify a temperature value:
Temp <- 30

# Show the temperature on all three scales:
if (substr(Scale,1,1)=="F" | substr(Scale,1,1) == "f"){
    TempC <- (Temp - 32)*5/9
    TempK <- TempC + 273.15
    c('Celsius'=TempC,'Fahrenheit'=Temp,'Kelvin'=TempK)
}else if (substr(Scale,1,1)=="C" | substr(Scale,1,1) == "c"){
    TempF <- 32 + 9*Temp/5
    TempK <- Temp + 273.15
    c('Celsius'=Temp,'Fahrenheit'=TempF,'Kelvin'=TempK)
}else{
    TempC <- Temp - 273.15
    TempF <- 32 + 9*TempC/5
    c('Celsius'=TempC,'Fahrenheit'=TempF,'Kelvin'=Temp)
}
```

```
##    Celsius Fahrenheit     Kelvin
##    -243.15    -405.67      30.00
```

Examples 2 will be turned into a user-friendly function later.

4.3 Loops

Loops are an essential part of programming because they allow you to automate repetitive tasks. Without loops, you would need to manually repeat code, which is inefficient, time-consuming, and prone to errors.

4.3.1 For-loops

Often, a certain task has to be executed repeatedly, probably with different given parameters stored, say, in a vector **zz**. This can be achieved with a for-loop. The general syntax can be:

```
for (i in 1:N){
    statements to be executed in round i
}
```

or

```
for (z in zz){
    statements to be executed with parameter
     z = zz[1] in the 1st round,
     z = zz[2] in the 2nd round,
     ...
     z = zz[length(zz)] in the last round.
}
```

Example
The Fibonacci sequence is defined by $1, 1, 2, 3, 5, 8, 13, 21, \ldots$, i.e. an infinite sequence with components $x_1 = 1, x_2 = 1$ and $x_n = x_{n-2} + x_{n-1}$ for n $= 3, 4, \ldots$.

```r
# Initialise x:
N <- 20  # should be > 2
x <- rep(1,N)

for (n in 3:N){
    x[n] <- x[n-2] + x[n-1]
}
x
```

```
##  [1]    1    1    2    3    5    8   13   21   34   55   89  144  233  377  610
## [16]  987 1597 2584 4181 6765
```

4.3.2 While-loops

Sometimes it is not known beforehand how often a specific task has to be executed. Instead, the task is repeated as long as a certain condition is satisfied. Here one can use a while-loop. The general syntax is

```
while (condition){
    statements to be executed until condition is no longer satisfied.
}
```

Example
Assume we want to add all natural numbers starting from 1 until the sum exceeds 100. This task can be performed using a while-loop.

```r
# Initialize variables
sum <- 0
i <- 1

# sum numbers until the total exceeds 100
while (sum <= 100) {
```

```
  sum <- sum + i
  i <- i + 1
}
print(paste("i =", i, "sum =", sum))
```

```
## [1] "i = 15 sum = 105"
```

4.3.3 Nested loops

One can use for- or while-loops within other for- or while-loops. For instance, if a matrix `M` with `r` rows and `c` columns has to be filled, one can do this via

```
for (i in 1:r){
    for (j in 1:c){
        M[i,j] <- ...
    }
}
```

Example

Let us calculate the sum of each row of a matrix.

```
M <- matrix(1:9, nrow = 3, byrow = TRUE)

# Outer loop iterates over rows
for (i in 1:nrow(M)) {
  row_sum <- 0
  # Inner loop iterates over all columns for that row
  for (j in 1:ncol(M)) {
    row_sum <- row_sum + M[i, j]
  }
  print(paste("Sum of row", i, "is", row_sum))
}
```

```
## [1] "Sum of row 1 is 6"
## [1] "Sum of row 2 is 15"
## [1] "Sum of row 3 is 24"
```