# Data Handling: Import, Cleaning and Visualisation

Lecture 5: Programming with Data

*Prof. Dr. Ulrich Matter*
*(University of St.Gallen)*

*19/10/2019*

---



---

## 1 First steps with R[1]

Once we understand the basics of the R language and how to write simple programs, understanding and applying already implemented programs is much easier.[2]

### 1.1 Variables and vectors

```
# a simle integer vector
a <- c(10,22,33, 22, 40)

# give names to vector elements
names(a) <- c("Andy", "Betty", "Claire", "Daniel", "Eva")
a
```

```
##   Andy  Betty Claire Daniel    Eva
##     10     22     33     22     40
```

```
# indexing either via number of vector element (start count with 1)
# or by element name
a[3]
```

```
## Claire
##     33
```

```
a["Claire"]
```

```
## Claire
##     33
```

```
# inspect the object you are working with
class(a) # returns the class(es) of the object
```

```
## [1] "numeric"
```

[1] The following sections and examples are based on Matter (2018).

[2] In fact, since R is an open source environment, you can directly look at already implemented programs in order to learn how they work.

```
str(a) # returns the structure of the object ("what is in variable a?")
```

```
##  Named num [1:5] 10 22 33 22 40
##  - attr(*, "names")= chr [1:5] "Andy" "Betty" "Claire" "Daniel" ...
```

## 1.2 Math operators

R knows all basic math operators and has a variety of functions to handle more advanced mathematical problems. One basic practical application of R in academic life is to use it as a sophisticated (and programmable) calculator.

```
# basic arithmetic
2+2
```

```
## [1] 4
```

```
sum_result <- 2+2
sum_result
```

```
## [1] 4
```

```
sum_result -2
```

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

```
4*5
```

```
## [1] 20
```

```
20/5
```

```
## [1] 4
```

```
# order of operations
2+2*3
```

```
## [1] 8
```

```
(2+2)*3
```

```
## [1] 12
```

```
(5+5)/(2+3)
```

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

```
# work with variables
a <- 20
b <- 10
a/b
```

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

```
# arithmetics with vectors
a <- c(1,4,6)
a * 2
```

```
## [1]  2  8 12
```

```
b <- c(10,40,80)
a * b
```

```
## [1]  10 160 480
```

```
a + b
```

```
## [1] 11 44 86
```

```
# other common math operators and functions
4^2
```

```
## [1] 16
```

```
sqrt(4^2)
```

```
## [1] 4
```

```
log(2)
```

```
## [1] 0.6931472
```

```
exp(10)
```

```
## [1] 22026.47
```

```
log(exp(10))
```

```
## [1] 10
```

```
# special numbers
# Euler's number
exp(1)
```

```
## [1] 2.718282
```

```
# Pi
pi
```

```
## [1] 3.141593
```

# 2 Basic programming concepts in R

## 2.1 Loops

A loop is typically a sequence of statements that is executed a specific number of times. How often the code 'inside' the loop is executed depends on a clearly defined control statement. If we know in advance how often the code inside of the loop has to be executed, we typically write a so-called 'for-loop'. If the number of iterations is not clearly known before executing the code, we typically write a so-called 'while-loop'. The following subsections illustrate both of these concepts in R.

### 2.1.1 For-loops

In simple terms, a for-loop tells the computer to execute a sequence of commands 'for each case in a set of n cases'. The flow-chart in Figure **??** illustrates the concept.

For example, a for-loop could be used to sum up each of the elements in a numeric vector of fix length (thus the number of iterations is clearly defined). In plain English, the for-loop would state something like: "Start with 0 as the current total value, for each of the elements in the vector add the value of this element to the current total value." Note how this logically implies that the loop will 'stop' once the value of the last element in the vector is added to the total. Let's illustrate this in R. Take the numeric vector `c(1,2,3,4,5)`. A for loop to sum up all elements can be implemented as follows:

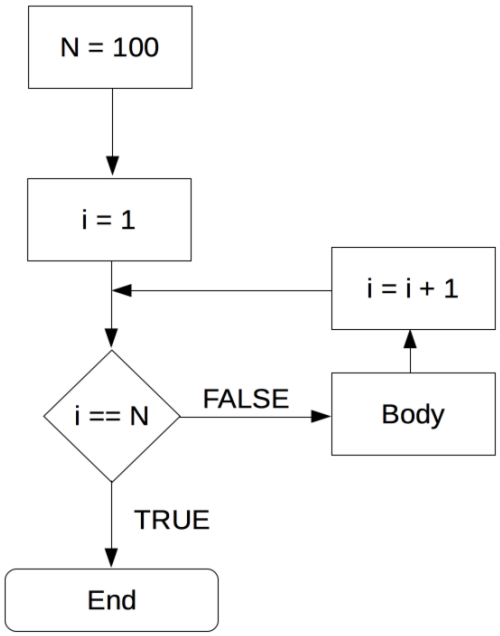


Figure 1: For-loop illustration.

```
# vector to be summed up
numbers <- c(1,2.1,3.5,4.8,5)
# initiate total
total_sum <- 0
# number of iterations
n <- length(numbers)
# start loop
for (i in 1:n) {
     total_sum <- total_sum + numbers[i]
}

# check result
total_sum
```

```
## [1] 16.4
```

```
# compare with result of sum() function
sum(numbers)
```

```
## [1] 16.4
```

### 2.1.2 Nested for-loops

In some situations a simple for-loop might not be sufficient. Within one sequence of commands there might be another sequence of commands that also has to be executed for a number of times each time the first sequence of commands is executed. In such a case we speak of a 'nested for-loop'. We can illustrate this easily by extending the example of the numeric vector above to a matrix for which we want to sum up the values in each column. Building on the loop implemented above, we would say 'for each column `j` of a given numeric matrix, execute the for-loop defined above'.

```
# matrix to be summed up
numbers_matrix <- matrix(1:20, ncol = 4)
numbers_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
# number of iterations for outer loop
m <- ncol(numbers_matrix)
# number of iterations for inner loop
n <- nrow(numbers_matrix)
# start outer loop (loop over columns of matrix)
for (j in 1:m) {
     # start inner loop
     # initiate total
     total_sum <- 0
     for (i in 1:n) {
          total_sum <- total_sum + numbers_matrix[i, j]
          }
     print(total_sum)
     }
```

```
## [1] 15
## [1] 40
## [1] 65
## [1] 90
```

### 2.1.3 While-loop

In a situation where a program has to repeatedly run a sequence of commands but we don't know in advance how many iterations we need in order to reach the intended goal, a while-loop can help. In simple terms, a while loop keeps executing a sequence of commands as long as a certain logical statement is true. The flow chart in Figure 2 illustrates this point.

For example, a while-loop in plain English could state something like "start with 0 as the total, add 1.12 to the total until the total is larger than 20." We can implement this in R as follows.

```
# initiate starting value
total <- 0
# start loop
while (total <= 20) {
     total <- total + 1.12
}

# check the result
total
```

```
## [1] 20.16
```

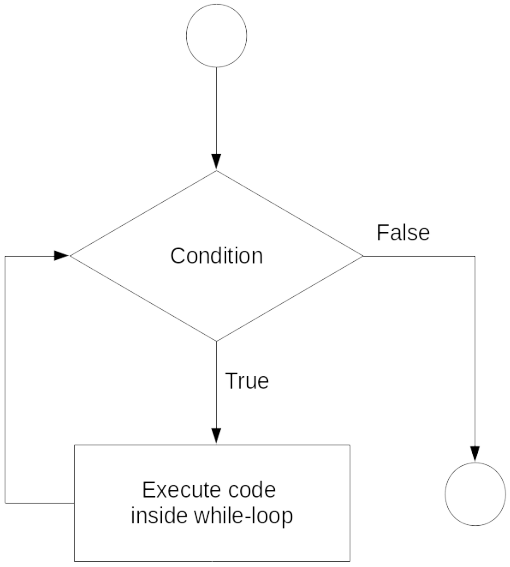


Figure 2: While-loop illustration.

## 2.2 Booleans and logical statements

```
2+2 == 4
```

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

```
3+3 == 7
```

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

```
condition <- TRUE
if (condition) {
     print("This is true!")
} else {
     print("This is false!")
}
```

```
## [1] "This is true!"
```

```
condition <- FALSE
if (condition) {
     print("This is true!")
} else {
     print("This is false!")
}
```

```
## [1] "This is false!"
```

## 2.3 R functions

R programs heavily rely on functions. Conceptually, 'functions' in R are very similar to what we know as 'functions' in math (i.e., $f : X \rightarrow Y$). A function can thus, e.g., take a variable $X$ as input and provide value $Y$ as output. The actual calculation of $Y$ based on $X$ can be something as simple as $2 \times X = Y$. But it could

also be a very complex algorithm or an operation that has not directly anything to do with numbers and arithmetic.[3]

In R—and many other programming languages—functions take 'parameter values' as input, process those values according to a predefined program, and 'return' the result. For example, a function could take a numeric vector as input and return the sum of all the individual numeric values in the input vector.

When we open RStudio, all basic functions are already loaded automatically. This means we can directly call them from the R-Console or by executing an R-Script. As R is made for data analysis and statistics, the basic functions loaded with R cover many aspects of tasks related to working with and analyzing data. Besides these basic functions, thousands of additional functions covering all kind of topics related to data analysis can be loaded additionally by installing the respective R-packages (`install.packages("PACKAGE-NAME")`), and then loading the packages with `library(PACKAGE-NAME)`. In addition, it is straightforward to define our own functions.

### 2.3.1 Tutorial: Compute the mean

In order to illustrate the point of how functions work in R and how we can write our own functions in R, the following code-example illustrates how to implement a function that computes the mean/average value, given a numeric vector.

First, we initiate a simple numeric vector which we then use as an example to test the function. Whenever you implement a function, it is very useful to first define a simple example of an input for which you know what the output should be.

```
# a simle integer vector, for which we want to compute the Mean
a <- c(5.5, 7.5)
# desired functionality and output:
# my_mean(a)
# 6.5
```

In this example, we would thus expect the output to be `6.5`. Later, we will compare the output of our function with this in order to check whether our function works as desired.

In addition to defining a simple example and the desired output, it makes sense to also think about *how* the function is expected to produce this output. When implementing functions related to statistics (such as the mean), it usually makes sense to have a look at the mathematical definition:

$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1+x_2+\cdots+x_n}{n}$.

Now, we can start thinking about how to implement the function based on built-in R functions. From looking at the mathematical definition of the mean ($\bar{x}$), we recognize that there are two main components to computing the mean:

- $\sum_{i=1}^{n} x_i$: the sum of all the elements in vector $x$
- $n$: the number of elements in vector $x$.

Once we know how to get these two components, computing the mean is straightforward. In R there are two built-in functions that deliver exactly these two components:

- `sum()` returns the sum of all the values in i ts arguments (i.e., if `x` is a numeric vector, `sum(x)` returns the sum of all elements in `x`).
- `length()` returns the length of a given vector.

With the following short line of code we thus get the mean of the elements in vector `a`.

[3] Of course, on the very low level, everything that happens in a microprocessor can in the end be expressed in some formal way using math. However, the point here is, that at the level we work with R, a function could simply process different text strings (i.e., stack them together). Thus for us as programmers, R functions do not necessarily have to do anything with arithmetic and numbers but could serve all kind of purposes, including the parsing of HTML code, etc.

```
sum(a)/length(a)
```

```
## [1] 6.5
```

All that is left to do is to pack all this into the function body of our newly defined `my_mean()` function:

```
# define our own function to compute the mean, given a numeric vector
my_mean <- function(x) {
     x_bar <- sum(x) / length(x)
     return(x_bar)
}
```

Now we can test it based on our example:

```
# test it
my_mean(a)
```

```
## [1] 6.5
```

Moreover, we can test it by comparing it with the built-in `mean()` function:

```
b <- c(4,5,2,5,5,7)
my_mean(b) # our own implementation
```

```
## [1] 4.666667
```

```
mean(b) # the built_in function
```

```
## [1] 4.666667
```

## References

Matter, Ulrich. 2018. "A Brief Introduction to Programming with R." Lecture notes. St. Gallen: University of St. Gallen.