Data Handling: Import, Cleaning and Visualisation

Lecture 2:
Programming with Data

Prof. Dr. Ulrich Matter, updated by Dr. Aurélien Sallin

Fetch Lecture Materials from GitHub

(optional, for those who want to download the materials directly)

1. Open Nuvolos/RStudio
2. Move to the Terminal window
3. Type/copy-paste the following

```
git clone https://github.com/ASallin/datahandling-lecture2023.git
```

4. Hit enter

Basic Programming Concepts

Values, vectors, lists, variables

- Values: a quantity of information, that is, in the simplest case, a letter (`"a"` , “character”), integer (`2`), fraction (`1.5`)
  - There are four main types of values in R: characters (`"a"`), integers (`2L`), numeric (or doubles, `2.3`), and logical (or booleans, `TRUE`).
  - To know which type of value a particular value is, we use the function `class()`
- Vector: a simple R object containing one or a collection of values (of the same data type)
- List: a R object containing one or a collection of values or other objects
- Variables: abstract storage location with a name. With `<-` we can assign a value to a variable.

Values and variables

```
# assign values to variables
a <- "Hello"
print(a)
```

```
## [1] "Hello"
```

```
class(a)
```

```
## [1] "character"
```

```
b <- FALSE
print(b)
```

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

```
class(b)
```

```
## [1] "logical"
```

```
# re-use variables
savings_t0 <- 2500
interest_rate <- 0.015
savings_t1 <- savings_t0 + (savings_t0 * 0.015)
savings_t1
```

```
## [1] 2537.5
```

Vectors

```
# initiate a character (text) vector
some_names <- c("Andy", "Betty", "Claire")
some_names
```

```
## [1] "Andy"   "Betty"  "Claire"
```

```
# how many elemnts are in a vector?
length(some_names)
```

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

```
# initiate an integer vector
some_numbers <- c(30, 50, 60)
some_numbers
```

```
## [1] 30 50 60
```

```
other_numbers <- 1:10
other_numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# merge vectors together
some_other_numbers <- c(some_numbers, other_numbers)
```

Vectors indexation

```
# Access the first value of a vector
some_numbers[1]
```

```
## [1] 30
```

```
# Access the second and third value of a vector
some_numbers[c(2,3)]
```

```
## [1] 50 60
```

```
# Remove the second value from a vector
some_numbers[-2]
```

```
## [1] 30 60
```

Vectorization in R

- Basic R functions (like math operators) are vectorized (execute the core function on each of the elements of a vector).
- Advantages: fast, clean code, optimal for writing analytics scripts.
- Note: **“In R, everything is a vector”!**

```
some_numbers + 1
```

```
## [1] 31 51 61
```

```
some_numbers > 3
```

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

```
some_numbers * 5
```

```
## [1] 150 250 300
```

Matrices R

Matrices are vectors bound together, either as column vectors or as row vectors. To bind row vectors together, we use `rbind` and to bind column vectors together we use `cbind`.

```
v1 <- c(1,2,3)
v2 <- c(3,4,5)

mat1 <- rbind(v1, v2)
mat2 <- cbind(v1, v2)

mat1
```

```
##    [,1] [,2] [,3]
## v1    1    2    3
## v2    3    4    5
```

```
mat2
```

```
##      v1 v2
## [1,]  1  3
## [2,]  2  4
## [3,]  3  5
```

To index matrices in r, we use square brackets. The first position in the square bracket is the row, and the second position is the column. The comma must be written.

```
# First column
mat1[, 1]
```

```
## v1 v2
##  1  3
```

```
# First row
mat1[1,]
```

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

```
# Element in second column, second row
mat1[2,2]
```

```
## v2
##  4
```

Math operators in R

```
# 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
```

To look up the most common math operators in R and get more details about how to use them type

```
?`+`
```

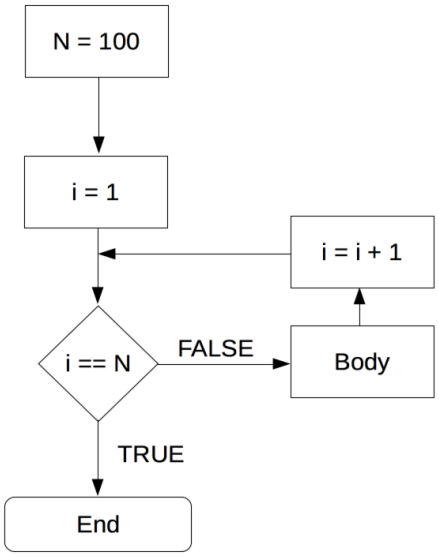
```
## starte den http Server für die Hilfe fertig
```

in the R console and hit enter.

Loops

- Repeatedly execute a sequence of commands.
- Known or unknown number of iterations.
- Types: ‘for-loop’ and ‘while-loop’.
    - ‘for-loop’: number of iterations typically known.
    - 'while-loop: number of iterations typically not known.

for-loop



`for` -loop in R

```
# number of iterations
n <- 100
# start loop
for (i in 1:n) {

     # BODY
}
```

`for` -loop in R

```
# vector to be summed up
numbers <- c(1,2,3,4,5)
# initiate total
total_sum <- 0
# number of iterations
n <- length(numbers)
# start loop
for (i in 1:n) {
     total_sum <- total_sum + numbers[i]
}
```

Nested `for` -loops (not covered in course)

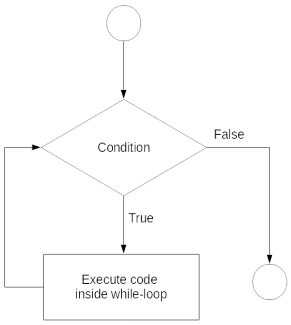
```
# matrix to be summed up
numbers_matrix <- matrix(1:20, ncol = 4)
numbers_matrix
```

```
##      [,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
```

Nested `for` -loops (not covered in course)

```
# 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)
     }
```

while-loop



`while`-loop in R

```
# initiate variable for logical statement
x <- 1
# start loop
while (x == 1) {

     # BODY
}
```

`while`-loop in R

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

Booleans and logical statements

```
2+2 == 4
3+3 == 7
4!=7
6>3
6<7
6<=6
```

Control statements

- What to do at a certain point in the program?
- Check conditions, decide on path.
- “If this is the case then do that…”
- “else do the other…”

```
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!"
```

R functions

- $f : X \to Y$
- ‘Take a variable/parameter value X as input and provide value Y as output’
- For example, $2 \times X = Y$.

- R functions take ‘parameter values’ as input, process those values according to a predefined program, and ‘return’ the results.

R functions in packages

- Many functions are provided with R.
- More can be loaded by installing and loading packages.

```
# install a package
install.packages("<PACKAGE NAME>")
# load a package
library(<PACKAGE NAME>)
```

Write functions

The basic R-syntax to write a function is as follows.

```
myfun <- function(){

}
```

Functions have three elements:

1. *formals()*, the list of arguments that control how you call the function
2. *body()*, the code inside the function
3. *environment()*, the data structure that determines how the function finds the values associated with the names (not the focus of this course)

For instance,

```
myfun <- function(x, y){

  # BODY
  z <- x + y

  # What the function returns
  return(z)
}
```

```
formals(myfun)
```

```
## $x
##
##
## $y
```

```
body(myfun)
```

```
## {
##     z <- x + y
##     return(z)
## }
```

```
environment(myfun)
```

```
## <environment: R_GlobalEnv>
```

Advanced: the “apply” family

In R, loops are generally avoided in favor of faster functions from the “base” R family of functions: the `apply()`, `lapply()`, `sapply()`, `tapply()`, `mapply()` and `aggregate()`. These functions are not faster than for-loops, but are more readable and more concise than larger loops.

`apply` applies a function to margins of an array or matrix. It loops over rows (`MARGIN = 1`) or columns (`MARGIN = 2`) of a matrix.

```
# Create an empty matrix with 2 rows and 4 columns
mymatrix <- matrix(c(1,2,3, 11,12,13, 1,10),
                   nrow = 2,
                   ncol = 4)
print(mymatrix)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3   12    1
## [2,]    2   11   13   10
```

```
# With for-loops
for (i in 1:ncol(mymatrix)){
  print(mean(mymatrix[, i], 2))
}

# With apply
apply(mymatrix, MARGIN = 2, mean)
```

`lapply` applies a function over a list ("list-" apply). It loops over each element of a list to execute a function.

```
mylist <- list(1,2,5,6,90)

# With a for loop
for (i in mylist){
  mean(i)
}

# With lapply
lapply(mylist, mean)
```

Tutorial: A Function to Compute the Mean

Preparation

1. Open a new R-script and save it in your `code` -directory as `my_mean.R` .
2. In the first few lines, use `#` to write some comments describing what this script is about.

Preparation

```
#####################################
# Mean Function:
# Computes the mean, given a
# numeric vector.
```

Preparation

1. Open a new R-script and save it in your `code` -directory as `my_mean.R` .
2. In the first few lines, use `#` to write some comments describing what this script is about.
3. Also in the comment section, describe the function argument (input) and the return value (output)

Preparation

```
#####################################
# Mean Function:
# Computes the mean, given a
# numeric vector.
# x, a numeric vector
# returns the arithmetic mean of x (a numeric scalar)
```

Preparation

1. Open a new R-script and save it in your `code` -directory as `my_mean.R` .
2. In the first few lines, use `#` to write some comments describing what this script is about.
3. Also in the comment section, describe the function argument (input) and the return value (output)
4. Add an example (with comments), illustrating how the function is supposed to work.

Preparation

```
# Example:
# a simlpe numeric vector, for which we want to compute the mean
# a <- c(5.5, 7.5)
# desired functionality and output:
# my_mean(a)
# 6.5
```

1. Know the concepts/context!

- Programming a function in R means telling R how to transform a given input (x).
- Before we think about how we can express this transformation in the R language, we should be sure that we understand the transformation per se.

1. Know the concepts/context!

- Programming a function in R means telling R how to transform a given input (x).
- Before we think about how we can express this transformation in the R language, we should be sure that we understand the transformation per se.

Here, we should be aware of how the mean is defined:

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

2. Split the problem into several smaller problems

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
- and n, the *number of elements* in vector x.

3. Address each problem step-by-step

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 total number of elements in a given vector (the vector's 'length').

4. Putting the pieces together

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

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

5. Define the function

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)
}
```

6. Test it with the pre-defined example

```
# test it
a <- c(5.5, 7.5)
my_mean(a)
```

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

6. Test it with other implementations

Here, compare 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
```