Exercise 1

A note on R Markdown

This document is written in a format called R Markdown. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com. It allows us to easily create a pdf file with the exercises, some readable code to give to you, and a nice html file with the solutions and outputs. It will also give you an idea on how to print nice html or pdf reports with R.

An R Markdown file contains text chunks (for humans to read) and code chunks (for R to evaluate and print the results). The R code for the solutions inside the code chunks is delimited by "‘. When you click the **Knit** button in RStudio ("Knit HTML"), a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. If you want to create a pdf file, you will need to install LaTeX to your computer.

We hope that this will be both easier for you to read in RStudio and easier to print. When you write code for yourself, you may find it more convenient to save it in a standard .R file.

What is R ?

R is a *statistical programming language*, which means that the focus of its application is statistical analysis (including data science) and visualization.

RStudio is one of several *integrated development environments (IDE)* developed for R (and support for other languages). It adds a nice interface and some functionality that makes working with R even easier and more convenient.

Why R ?

R and RStudio have several advatanges when compared to other popular statistical software such as Microsoft Office Excel, SPSS Statistics, MATLAB, STATA, or SAS.

1. R (together with Python) is now the most common languages used for business and data analytics, both in academia and practice. R (and Python) are also free, so you can practice at home.
2. The R community is large and new functionality is constantly being developed. Many state-of-the-art methods are available in the form of tested *packages* on the *comprehensive R archive network* (CRAN) with excellent documentation. There exists numerous tutorials and particular questions can be searched or asked on *stackoverflow.com*.
3. RStudio includes package mangement, project management and Github integration, among other features.

R Download: https://cran.r-project.org/

R Studio download: https://www.rstudio.com/products/rstudio/

Basics

Math

```
# R can do math
# Aside: Press CTRL/Cmd + Enter to run the selected line
#        Press CTRL/Cmd + Shift + Enter to run the current chunk of code
#            and display the results below
1 + 4
```

```r
1 - 1 / 2
1 - (1 / 2)
2 ^ 2
```

Comments

The following code shows how to comment single rows and multiple lines without influencing the code. Use # to comment a single line. We use comments to explain the code below or to the left of the comment. Here is an example

```r
# Welcome to this R tutorial. This line is just a comment
# The code below substracts 2 by 2 and should return 0.
2 - 2
# 1 + 1 This line is not evaluated, because it is commented out"

# In RStudio, press Ctrl/Cmd + Shift + C to quickly comment or uncomment the selected lines
```

Objects

In R, both = and <- can be used to store something in an object. This can be simple values, e.g. a number, but also more complex types of information, e.g. tables or functions. In R, there is no need to specify what type of information is saved in the variable and the type can be changed as convenient. When we use an object in the code, it is *evaluated* when the code is run.
Think of objects as different kinds of boxes. When you tell R to look into a box, it will find whatever you have *last put into it*.

```r
a <- 5 # The number 5 is assigned to a.
b = 2.5 # This is equivalent and assigns value 2.5 to b.

# Calling an object prints its content.
b
# We can use the objects as we would the numbers that they contain.
a + b
# We can also save the result to a new object.
x <- a + b
x

# The content and content type can be changed.
b <- "hello"
b
```

Functions

Functions relate an input to an output. The input is specified as arguments in brackets delimited by a comma (if there is more than one).
Conceptually, functions are just another object type that saves the instructions how to transform the input.
Think of functions as little helpers: You give them input, either in the right order or by explicitely telling them which argument is which. They remember what to do with that input and can also call on other functions to help. When they are done they will give you back some output.

```r
# exp(x) calculates the exponential function output to input value x.
exp(0)

# The output can of course be saved to an object
a <- exp(0)
```

```r
a

# We can save our own instructions into an object and use them like any other function.
# To do so, we first use function function() and provide a list of placeholders
# There are some conventions, but we can essentially use any name.
division <- function(numerator, denominator) numerator / denominator
# Use a function by order of arguments...
division(5, 2)
# ...or by naming them specifically... (recommended!)
division(denominator = 2, numerator = 5)
# ...or with a mix of both (be careful to keep the order for the non-named)
division(denominator = 2, 5)
# We can also pass objects to functions
x <- 10
division(numerator = x, denominator = 2)

# It's easy to look into the function 'box' to see what's going on inside it
# Just leave out the parenthesis ()
division
```

Help

The help function shows the documention of a function, which is usually very helpful in R. Use **?** to look for a specific function or **??** to look for a term in the documentation text.

```r
# ?mean will give us information about the mean() function, which computes the Arithmetic Mean.
?mean
# Or alternatively
help("mean")

# ??mean searches for the keyword in the documentation of all functions
# It can be useful when you don't know the name of a function, e.g. for the standard deviation
??deviation
??"standard deviation"
```

More help

There are many ways to solve a task in programming. There are also many ways in which things can go wrong. You therefore must now how to find out how to solve a problem by experimenting or looking for it online. The homework exercise will guide you through the process for practice, but here is an example.

```r
# Simple multiplication
x <- 1
#y <- 5x
# The above lines produces the error:
# Error: unexpected symbol in "Error: unexpected symbol in "y <- 5x""

# As is often the case, we don't know what the exact problem is
```

1.Step: Look into the R help

```r
?"*"
# This tells us how to do multiplication, but it's not clear what our problem is.

# A quick experiment may help us to narrow down the issue
```

```r
y <- 5 # Assignment works without error
y <- x # Assignment of a variable works without error
```

2.Step: Google

Let's google the error message. Most likely someone has had the same problem. We'll leave out the specific part, the 5x, and add the language, i.e. "R Error: unexpected symbol in"

The first result, from the amazing page stackoverflow.com has a whole list of common problems causing the error. And among them we find "Missing * when doing multiplication"! This will help us to avoid the error in the future.

Object classes I

We have already used three object classes above: *numeric* values, *character* strings, and *functions*. We can use function *class()* to find out the type of an R object. Simply speaking, *class()* tells us what kind of thing is in the box. We can also check with **is.{some class}()** if an object has the class that we expect. **as.{some class}()** can be used to transform one type to another, but only if the content allows the transformation.

Numeric and character

```r
# Numbers are called 'integer' or 'numeric' (fractions aka floats)
a <- 2.5
class(a)
is.numeric(a)
# Text is called 'character' and is specified and displayed with quotation marks.
b <- "hello"
class(b)
b <- "10"
class(b)

# There's things that you can and cannot do, depending on the class
# Here, we can't sum a number and a word
# a + b # You can uncomment this, it will give an error
# "Error in a + b : non-numeric argument to binary operator"

# But since "10" is a number formatted as a word, we could use as.numeric()
a + as.numeric(b)
# ...but only where it makes sense
as.numeric("chicken")
```

Logical

A special class of objects have the logical values **TRUE**/**FALSE** (they have to be written in all caps in R). They will be important later to *index*, i.e. specify which values to take out of another objects, or to control the flow of the code with **if** ... **then** statements.
Many functions in R output a logical value, e.g. $>$ (bigger than), $<=$ (smaller or equal), $==$ (equal), or **!**$=$ (not equal).
Logical operations can be used on logical statements, e.g. **!** (Not), **&** (and), or | (or).

```r
# Logical output
3 > 4
3 <= 4
```

```
# Don't forget to use capital letters
a <- TRUE
b <- FALSE

# Logical operators
a & b
a | !b

# We will see later how to use these to control the code with this.
x = 42
if(x == 42) print("That's the answer to everything")
```

Vectors

Just as in geometry, the objects that we have seen can represent more than one value. They can be structured as a *vector* or *matrix*, which can be used more efficiently. The important thing to remember about these is that they only save values of one class, usually either numeric or character, but not mixed. Values are *concatenated* or combined used function **c()**.

```
# Create a vector by concatenating several values
v <- c(1, 2, 3, 4, 5)
v
# Many functions return a vector directly
v1 <- seq(1, 5)
v1
# ... or even more simple
v2 <- 1:5
v2
```

Many functions take vectors as input. Depending on their purpose they might return another vector or a single value. Let's look at **sort()** that sorts the values in a given vector and **length()** to find out the number of elements.

```
unsorted <- c(3,5,1,7,11,105,13,12) # This vector is unsorted.
sort(unsorted) # the return sorts the numeric values cardinally
length(unsorted)

v1 <- 1:6
v2 <- 6:1

# Element-wise
v1 >= v2
v1 + v2
v1 * v2 # This is not the dot product
```

Matrices

Matrices are best understood as vectors with more than one dimensions. They can also take only one type of content (number/string).
By default, algebraic operations are done element-wise for vectors. Additionally, matrix algebra is available to be used with vectors and matrices, e.g. %%$** (matrix multiplication) or **t()** (transpose).
Vectors can be combined with **rbind()** (row-wise) or **cbind()** column-wise to form a matrix.

```
# Dimensions
# The values are saved as a vector, with the dimensions
```

```r
# saved as an attribute that describes how the vector is
# structured.
dim(v2) # note that function dim() returns NULL for vectors; it is designed to work with matrices.
        # You can use function length() to identify the number of elements in a vector.

# It is possible to assign a new structure without changing the values.
dim(v2) <- c(3,2)
v2

# Matrix algebra
v1 %*% v1
m <- v1 %*% t(v1) # Multiply vector with its transpose and save to object
m
m * 2
m %*% cbind(v1, v1, v1, v1) # column-wise binding together 4 times
matrix(v1, ncol = 2) # using function matrix() with argument 'ncol'
```

Indexing

When we have objects that contain more than one value, we often want to extract only a part. This is called *indexing* and R uses square brackets to specify the position of values to be extracted. The standard ways to index are to specify the position of the element by a number or to specify for each element if it should be extracted by a logical value (TRUE/FALSE).
The dimensions are separated by a comma. If we want to specify more than one index, we therefore have to pass in a *vector* of row indices. Otherwise, R would confuse the comma between our row indices and the comma between the dimensions.

```r
# Indexing by position
v <- 10:15
v[2]
v[1:2]

# Logical indexing
v > 11
v[v > 11] # For each value specify if larger than 11 and select
v[v > 11 & v < 14]

# More-dimensional indexing by position
m <- matrix(c(1:9), ncol = 3)
m
m[,]
m[2, 2] # a specific value
m[c(1,2), 2] # the value in row 1 and 2, column 2
m[3, ] # all values in row 3
m[, 2:3] # all values in column 2 and 3
```