Exercise session 1

How to submit your exercises

During this exercise session, you will be asked to write code in R and submit the solution. Write your solutions in a file R. Then, click **ctrl + shift + k** to generate a Report. Choose *PDF* as Report format. This will open a new pdf file that you can save and submit.

Introduction

In this exercise session we will learn the basis of R, covering the first lecture of the course. We start by familiarizing ourselves with R and RStudio:

```
Data<-mtcars[,3:5]
head(Data)
plot(Data)
```

Copy these three lines of code in the top-left panel (the **R-file**). Don't worry if you don't understand what the code means: it will be goal of this course to understand it. You can now run the code: highlight the code and press the button Run on the top of the screen. Alternatively, press Ctrl+Enter (for Windows and Linux users) and Command + Enter (for Mac users).

Let's see what happened.

First of all, we can see that the code has appeared in the **console** as well (bottom-left panel). The console has some extra text though: after the line *head(Data)* the console printed a list of values. This is because we used a function (**head**) and in the console we see the result of said function (in this example, **head** shows the first 6 rows of a dataset. Data is a dataset, so it does just that!)

We can see in the lines of code that we used the symbol **<-**: this is the way we save an object in R. For example, in the first line we are saving a dataset (mtcars) with a new name, Data. Now if we write Data in your console (bottom-left panel) and press Enter, the full dataset will be printed in the console.

Look at the top right panel: under **environment**, we will find the object Data. Environment is the area in RStudio where we can see every object that we saved up until now. Pay attention to it while we keep working today.

Lastly, the bottom right panel will display a figure. This is because one of the lines we ran contains the function **plot**. This function creates a scatterplot of the dataset Data. This panel has more functions than just showing plots, but we will study it in more details later on in the course.

An important question now is: what is the difference between **console** and **R file**? We saw that the console returns the code *and* the results of the functions (as **head** in the example). Why not work directly in the console?

The answer is simple: the R file acts as a manuscript that can be saved (by going to File->Save, or pressing the **save** icon). Anything that is written on the console will be lost from one session to the other: if we want to reuse the lines we are writing, we must be sure that we are coding in the R file panel and that the file has been saved!

Now we know what every panel does; we can start learning.

We start with something simple: we try to compute a sum, say 2+2. In R we use the basic operation symbols:

```
+ , -, *, /
```

Write it in the Rfile and run the line. As before, the console returns both the line 2+2 and the solution, 4. We can define the object *a* as this sum. Write

```
a<-2+2
```

in the console.

Now, if you write *a* and run it, the console will print the value 4. We can also find *a* in the **Environment**, in the top right panel. The object *a* has been defined and stored, and R will remember it. Try to sum *a* to 5, for example, and see the result. Be careful! If we define a new object (e.g. 3+2) with the same name, we will rewrite the previous object. We cannot have two objects with the same name.

Exercise 1.

Define two objects, *first* and *second*, respectively as the sum of 23 and 16 and the difference between 11 and 45. Define a third object, *third*, as the quotient between *first* and *second*. Print out the result.

Vectors

In this section, we will study how to create and use vectors. A **vector** is a one-dimensional array. For example, we can construct a vector that contains *first*, *second*, and *third* from the previous exercise. The command is the following:

```
c(first, second, third)
```

```
## [1]  39.000000 -34.000000  -1.147059
```

We can call this vector *our_vector*

```
our_vector<-c(first, second, third)
```

Now, the object *our_vector* is stored in the **environment**.

Vectors can hold numeric data (as in the example above), but also logical data (that is, TRUE or FALSE) or character data (for example, we can construct a vector of letters).

```
logical_vector<-c(TRUE, FALSE, FALSE)
logical_vector
```

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

```
character_vector<-c("a", "b", "c")
character_vector
```

```
## [1] "a" "b" "c"
```

We can also give names to the single elements of the vector. For example, we want to give names to the three elements of *our_vector*; we will call them first, second, and third respectively. We create a character vector with these names, and call it *vector_of_names*:

```
vector_of_names<-c("first", "second", "third")
```

We use the function **names** in the following way:

```
names(our_vector)<-vector_of_names
our_vector
```

```
##      first     second      third
##  39.000000 -34.000000  -1.147059
```

Now, when we print *our_vector*, each element has the name given by *vector_of_names*. Observe that *our_vector* and *vector_of_names* must have the same length (we are giving a name to each element in the vector).

We can select a subset of the vector. For example, we might be interested in the element called *first* in *our_vector*. We use square parenthesis:

```
our_vector["first"]
```

```
## first
##    39
```

We can also select multiple elements, using a vector with the names of the objects we are interested in:

```
our_vector[c("first", "third")]
```

```
##     first     third
## 39.000000 -1.147059
```

We can also select the first element of the vector in the following way:

```
our_vector[1]
```

```
## first
##    39
```

Note that the number 1 isn't in quotation marks: in fact, it is not name, but the position of the element in the vector. In this way, we are selecting only the first element. As before, we can select more than one object by using vectors in the square parenthesis.

In order to select elements in your vector, we can also use **comparison operators**. These are:

== for equal to each other

!= for not equal to each other

< for less than

> for more than

<= for less or equal

>= for more or equal

When we use these operators on a vector, the output is logical vector, whose elements are TRUE and FALSE depending on whether each element satisfies the condition asked. Let us see an example:

```
our_vector
```

```
##      first     second      third
##  39.000000 -34.000000  -1.147059
```

```
our_vector<0
```

```
##  first second  third
##  FALSE   TRUE   TRUE
```

With the second command, we asked if the elements of *our_vector* are smaller than 0. The first element is 34, which is positive; hence, it returns FALSE. Both the second and third element in *our_vector* are negative, so the operator returns TRUE for both.

We can select elements of *our_vector* by using comparison operators in the square parenthesis:

```
our_vector[our_vector<0]
```

```
##     second      third
## -34.000000  -1.147059
```

Whenever we use logical vector in square parenthesis, we are telling R to only print out the TRUE elements (in our case, only the second and third element).

We can also perform computations on vectors. For example, assume we want to multiply each element of *our_vector* by 3:

```
our_vector
```

```
##      first     second      third
##  39.000000 -34.000000  -1.147059
```

```
3*our_vector
```

```
##       first      second       third
##  117.000000 -102.000000   -3.441176
```

The notation $3*$*our_vector* tells R to multiply each element in the vector by three.

We can use the operations $+, -, *, /$ on vectors and R will perform the operation on each element.

Exercise 2.

During this assignment, you will be analysing monthly temperatures in Bergen. All the data can be found on the website yr.no, following the link: https://www.yr.no/place/Norway/Vestland/Bergen/Bergen/statistics.html

During the year of 2019, the average monthly temperature in Bergen were, respectively: 2.6, 5.3, 4.6, 9.8, 9.4, 14.1, 16.2, 16.4, 11.6, 8.0, 3.0, and 5.0. Construct a vector in R, called *temperature_2019*, containing these values. Then, using the function **names**, give the vector the names of the months, from January to December.

Then, create two new vectors: *first_half*, that contains the first six elements of the vector temperature_2019, and *second_half*, that contains the remainder elements.

Tip: if you want to define a vector containing all integers from 1 to 6, you can use colons:

```
c(1:6)
```

```
## [1] 1 2 3 4 5 6
```

Exercise 3.

Consider the vector *temperature_2019* and select only the elements larger than 10. Save this new object as *high_temperature*.

Create a second vector, *low_temperature*, which is the complementary of *high_temperature*.

Exercise 4.

You are asked by a colleague in the USA to send her the information collected in *temperature_2019*. Before sending the data, you want to transform the temperatures from Celsius to Farenheit.The formula is: F=C*1.8 + 32.

Matrices

A **matrix** is a rectangular array containing numerical, logical, or character data arranged in rows and columns.

```
our_matrix<-matrix(c(1:6), ncol=3, byrow = TRUE)
our_matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

In the example above, we constructed a 2x3 matrix (i.e., a matrix with 2 rows and 3 columns) containing the numbers from 1 to 6.

In order to construct a matrix, we use the function **matrix** in R. As input, it takes a vector (in the example above, c(1:6)) and a number of columns (in the example above, 3). The argument **byrow** tells us if the element of the vector must be organized following rows or columns:

```
matrix(c(1:6), ncol=3, byrow = FALSE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Try and play with the function **matrix**.

Since a matrix is a rectangular array, in order to select one element in it we must specify both its row and column position. For example:

```
our_matrix[1,2]
```

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

selects the element in the first row, second column.

We can also select one entire column or one entire row by leaving blank the other spot in the square parenthesis:

```
our_matrix[1,]
```

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

```
our_matrix[,1]
```

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

We can also use comparison operators on matrices. It will return a logical matrix.

```
our_matrix<3
```

```
##       [,1]  [,2]  [,3]
## [1,]  TRUE  TRUE FALSE
## [2,] FALSE FALSE FALSE
```

One of the very basic statistics function implemented in R is **mean**, which computes the average of a collection of numbers. For example, we can compute the mean of *our_vector*:

```
mean(our_vector)
```

```
## [1] 1.284314
```

We can also compute the average of a matrix, or a subset of the matrix:

```
mean(our_matrix)
```

```
## [1] 3.5
```

```
mean(our_matrix[,1])
```

```
## [1] 2.5
```

```
mean(our_matrix[2,])
```

```
## [1] 5
```

The first line computes the mean of all the values in *our_matrix*. The second line computes the mean of the first column, and the third line computes the mean of the second row.

We can also add rows and columns to an already existing matrix. The functions we use are called **rbind** and **cbind** respectively.

```
rbind(our_matrix, c(7,8,9))
```

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

```
cbind(our_matrix, c(3.5,6.5))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3  3.5
## [2,]    4    5    6  6.5
```

The length of the vector we want to add must be consistent with the length of the row or column, respectively. Note that the order of the objects in the function rbind makes a difference! Try to write rbind(c(7,8,9), our_matrix) and see what happens.

Exercise 5.

We know that the average monthly temperatures for 2018 in Bergen were 2.7, 0.9, 0.8, 7.5, 14.7, 14.6, 17.3, 14.5, 12.0, 8.4, 6.7, 4.2 respectively. Construct a vector containing these values, called *temperature_2018*. Assign a name to each element as done in Exercise 2.

Create now a matrix having the vectors temperature_2018 and temperature_2019 as columns. Call it *temperature_matrix*.

Exercise 6.

Using the function **mean** compute the average temperature of 2018 and 2019 separately. Construct a vector containing these two values, then add it as an extra row to *temperature_matrix*.

Then, compute the average monthly temperature in *temperature_matrix* (that is, compute the mean between the temperature in 2018 and 2019 for each month, plus the mean between the means in the last row). Construct a vector with these values, and add it as a column to temperature_matrix. Important: do not rewrite the object temperature_matrix in this exercise. Save the new matrix with a different name.

Exercise 7.

You want to send *temperature_matrix* to your american colleague. Transform all the measurements in Fahrenheit (Hint: operations on matrices work in the same way as they did on vectors).

Factors

A group of friends is asked to choose which color they prefer among red (R), yellow (Y), and green (G). This is the result:

```
colors<-c("Y", "G", "G", "R", "R", "G")
```

By construction, the elements of this vector can only assume three values: R, G, and Y. In this small example, counting the number of G's, Y's, and R's is relatively easy, but once we increase the length of the vector, it can become more complicated.

For this reason, we want to create a **factor**:

```
colors_factor<-factor(colors)
colors_factor
```

```
## [1] Y G G R R G
## Levels: G R Y
```

Now R recognized G, R and Y as levels.

We look at the levels and realize that we might forget what the letters stand for; we want to change the names of the levels. We do not need to recreate the whole vector from scratch: we can use the function **levels** to assign new names to the levels.

```
levels(colors_factor)<-c("Green", "Red", "Yellow")
colors_factor
```

```
## [1] Yellow Green  Green  Red    Red    Green
## Levels: Green Red Yellow
```

R has changed the name to each element both in the levels and in the vector itself. Be careful to assign the correct name to the levels, and not to switch them up!

A very important function for factors is **summary**:

```
summary(colors_factor)
```

```
##  Green    Red Yellow
##      3      2      1
```

As the line above shows, **summary** identifies the different levels and counts the elements in each such level.

Note that green, red, and yellow are not **ordered factors**: there is not a natural way to compare them. We can compare factors called, for example, "low", "medium", "high": there is a natural order to them, low < medium < high.

In this case, we define the vector as follows:

```
ordered_factor<-factor(c("h", "h", "l", "m", "h", "m", "m"), ordered=TRUE,
                       levels=c("l", "m", "h"))
ordered_factor
```

```
## [1] h h l m h m m
## Levels: l < m < h
```

If we want to order the factors, we must set the argument **ordered** as TRUE, and we must specify the levels and their order (l < m < h).

We can still use summary:

```
summary(ordered_factor)
```

```
## l m h
## 1 3 3
```

We can extract elements from the factor vector using the square parenthesis and, if the factor is ordered, we can compare different elements:

```
ordered_factor[1] < ordered_factor[3]
```

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

It returns FALSE, because high > low.

Exercise 8.

We want to look at the vector *temperature_2019* as a factor. In order to do that, we define three different levels: below 5 (l), between 5 and 15 (m), and above 15 (h). Then, construct the vector *factor_temperature_2019* with the above described levels. Then, count the elements in each level.

Repeat the same procedure with the vector *temperature_2018*. Then, compare the two new factors (Note: these factors must be ordered).

Data Frames

In R, to construct a data frame we use the function **data.frame**. As argument, it can have vectors or matrices. For example:

```
df_matr<-data.frame(our_matrix)
df_matr
```

```
##   X1 X2 X3
## 1  1  2  3
## 2  4  5  6
```

We can create data frames made with vectors of different types. The important thing is that the length of the different vectors is the same.

```
df_vec<-data.frame(logical_vector, character_vector)
df_vec
```

```
##   logical_vector character_vector
## 1           TRUE                a
## 2          FALSE                b
## 3          FALSE                c
```

The function **str** allows us to analyse the structure of the dataframe:

```
str(df_vec)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ logical_vector  : logi  TRUE FALSE FALSE
##  $ character_vector: Factor w/ 3 levels "a","b","c": 1 2 3
```

str tells us the number of rows (**obs.**), the number of columns (**variables**), and summarizes the data (in our case, it tells us that the vectors are factors, and their levels).

Try to see what **str** returns for *df_matr*.

We can extract elements from a data frame in the same way as with a matrix:

```
df_vec[1,2]
```

```
## [1] a
## Levels: a b c
```

We will now work a bit on a dataset already implemented in R, called *mtcars*. To find out what this dataset contains, we write in the console:

```
?mtcars
```

A new file will appear in the bottom-right panel, under **help**.

As we can read in the help file, *mtcars* has 32 observations in 11 variables. To have an idea of what the dataset looks like without printing it completely, we can use the funciton **head**, which shows only the first 6 rows of a dataset.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

We can use **tail**, instead, to show the last six rows:

```
tail(mtcars)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

There are two different ways to display the full dataset. The first one is to simply write its name:

```
mtcars
```

In this case, the full dataset will be printed in the console. *mtcars* is not a big dataset, but imagine if there were hundreds, or thousands, of lines. If would be quite difficult to work on the console to study the dataset. Another way to look at it is using the funcion **View**:

```
View(mtcars)
```

This function will open a new R file containing the dataset. In this way we can look at it whenever we want without needing to print it to the console each time. We will study later on in the course some summarizing functions which give a good idea of what the dataset contains without printing it.

Let us go back to selection. As with matrices and vectors, we use square parenthesis to select elements from a dataset. If we look at *mtcars*, we see that every column and every row has a name. We can use those for our selection:

```
mtcars[, "hp"]
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

```
mtcars["Maserati Bora",]
```

```
##               mpg cyl disp  hp drat   wt qsec vs am gear carb
## Maserati Bora  15   8  301 335 3.54 3.57 14.6  0  1    5    8
```

The first line prints the column called *hp*, the second the row called *Maserati Bora*. We find their intersection by writing:

```
mtcars["Maserati Bora", "hp"]
```

```
## [1] 335
```

Another (faster) way to select a column in a dataset is by using **$**. This does not work for rows.

```
mtcars$hp
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

An important selection tool is the function **subset**. It allows us to impose a condition the elements in the dataset need to satisfy to be printed. For example, we see that the column *vs* (Engine configuration) has two possible outcomes: 0 and 1. We want to select only cars with V-shaped engines, that is:

```
subset(mtcars, subset=vs==0)
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
```

```
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

To define the condition, we use comparison operators. For example, try to select only the cars with more than three gears. Play around with the function and the dataset.

Last important thing we want to learn about datasets, is how to order them. Let us study the function **order**:

```
our_vector
```

```
##      first     second      third
##  39.000000 -34.000000  -1.147059
```

```
order(our_vector)
```

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

order indicates the new position the elements must have to be ordered increasingly (in the example, the second element of *our_vector* must be positioned as first, being the smallest one; and so forth).

This means that we can reorganize the vector using **order** and the square parenthesis:

```
our_vector[order(our_vector)]
```

```
##     second      third      first
## -34.000000  -1.147059  39.000000
```

One of the arguments in the function order is **decreasing**, which is by default set on FALSE, but can be changed if we want the vector to be arranged in decreasing order.

To apply **order** to a dataset, we need to choose a column (for example, let us consider the variable mpg, which is continuous). Then, we write:

```
head(mtcars[order(mtcars$mpg),])
```

```
##                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Maserati Bora       15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
```

Now the dataset is showed in ascending order with respect to mpg. If we think we might use *order(mtcars$mpg)* more than once in our code, we can assign it a name using <-.

```
mpg_position<-order(mtcars$mpg)
head(mtcars[mpg_position,])
```

```
##                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Maserati Bora       15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
```

The function **order** must be used in the rows space in the square parenthesis, because what you are ordering are the rows.

Exercise 9.

Construct a dataset containing *temperature_matrix*, *factor_temperature_2019* and *factor_temperature_2018*. Order the dataset according to *temperature_2019* and save it under *increasing_2019*. Do the same for *temperature_2018*. Then, in both these datasets, select only the elements where *factor_temperature_2019* (or 2018 in the second one) is *h*.

Lists

Lists allow to group elements of different types and different dimensions in the same object (for example, matrices and vectors). To construct a list, we use the function **list**:

```
our_list<-list(our_vector, our_matrix, df_matr)
our_list
```

```
## [[1]]
##      first     second      third
##  39.000000 -34.000000  -1.147059
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
##
## [[3]]
##   X1 X2 X3
## 1  1  2  3
## 2  4  5  6
```

We can give names to the components in a list using the function **names** (as for dataframes). For example, we can change the names in *our_list*:

```
names(our_list)<-c("vector", "matrix", "dataframe")
our_list
```

```
## $vector
##      first     second      third
##  39.000000 -34.000000  -1.147059
##
## $matrix
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
##
## $dataframe
##   X1 X2 X3
## 1  1  2  3
## 2  4  5  6
```

In order to select elements from a list, the syntax is a bit different than before. We need first to select a component (in our example above, *our_vector*, *our_matrix*, or *df_matr*) and then select the elements inside the chosen component. We first use double square parenthesis:

```
our_list[[2]]
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

What we selected here is the second component, that is *our_matrix* (which is now called *matrix*).

If we want to choose, for example, the second column of *our_matrix*, we use the standard matrix notation afterwards:

```
our_list[[2]][,2]
```

```
## [1] 2 5
```

Be careful! Depending on the class of the component that we choose, we might need to use different syntax to select an element (for example, we need to specify only one position to select an object from a vector).

Another way we can select a component in our list is by using $ and using the name of said component:

```
our_list$matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
our_list$matrix[,2]
```

```
## [1] 2 5
```

Experiment with your list, and try to select different elements in it!

Exercise 10.

Construct a list containing *temperature_matrix*, *factor_temperature_2019*, and *factor_temperature_2018*. How is it different from the dataset we created in exercise 9?

Extract from the list the temperature of May 2019 and, separately, its respective factor. Do the same for May 2018.
