Exercise Number 1: Introduction to R

Christopher Diebel & Jan-Hendrik Schmidt - ISE Darmstadt

Helpful Links

- :house_with_garden: Home
- :open_book: Information about R
- :open_book: `dplyr` documentation
- :open_book: Further Reading R for Data Science, R für Einsteiger or Einführung in R (This is what this first exercise is largely based on)

Preparation

Please work through the Prerequisites section to be prepared for the first exercise.

What is R?

R is a free software environment for statistical analysis.

Graphical User Interface

Your opened RStudio should look something like the following:

The RStudio GUI consists of several areas. In the console area, R code can be entered, which is then interpreted (executed) by R. The `>` character is the R prompt (prompt character).

In the upper right corner, besides the History, which lists all the commands you have executed so far, we find the Environment section. There you will find all the variables, data sets and functions that have been defined.

Operators

Arithmetic Operators

Command	Meaning	Example (if appropriate)
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	
^ or **	Exponentiation	
x %*% y	Matrix Multiplication	c(1,4) %*% c(3,5) == 23
%/%	Whole Number Division	6 %/% 4 == 1
%%	Modulo (Remainder of a Division; x mod y)	6 %% 4 == 2

Logic Operators

Command	Meaning	Example (if appropriate)
==	Equal	
!=	Unequal	
<	Smaller than	
>	Greater than	
<=	Smaller equal	
>=	Greater equal	
&	Logical AND	(x & y)
|	Logical OR	(x | y)
!	Logical NOT	!x
xor(x, y)	Exclusive OR	Either in x or y, not both

The following graphic shows the use of the logical operators by means of Venn diagrams. `x` always refers to the left circle, `y` to the right. The selected area is always shown in dark. Source: R for Data Science

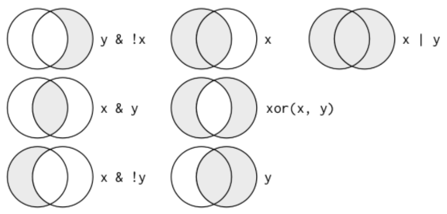

Figure 1: Illustration of the Logical Operators

R Functions

In the following we will look at some of the standard R functions that can be used and that make up the language in the first place. Each function can be called in the documentation (even if it was inserted via a library - more to that later). Just call the command `help(fct-name)` or `?fct` (just type it into the console). There you get a first overview about the function and its parameters.

```
help(seq)

?round
```

Numerical Functions

Command	Meaning	Example (if appropriate)
abs(x)	Absolute Value	
sqrt(x)	Square root	
ceiling(x)	Round Up	ceiling(4.687) is 5
floor(x)	Round Off	floor(4.687) is 4
round(x, digits = n)	Rounding	round(4.687, digits = 2) is 4.69
log(x)	Natural Logarithm	
log2(x)	Logarithm to base 2	
log10(x)	Logarithm to base 10	
exp(x)	Exponential Function	e^x

Statistical Functions

These functions all have the argument `na.rm`, which by default takes the value `FALSE`. This allows missing values to be taken into account, i.e. missing values (na = not available) are not removed in this case (rm = remove). These functions can all be applied to a vector (see below).

Command	Meaning
mean(x, na.rm = FALSE)	Mean Value
sd(x)	Standard Deviation
var(x)	Variance
median(x)	Median
quantile(x, probs)	Quantiles of X (probs: Vector with Probabilities)
sum(x)	Sum
min(x)	Minimum Value (x_min)
max(x)	Maximum Value (x_max)
range(x)	x_min and x_max

Further Useful Functions

Print Function

The `print()` function writes the output to the console or output file (batch mode).

```r
print(sqrt(2))
```

```
## [1] 1.414214
```

```r
print(sqrt(2), digits = 4)
```

```
## [1] 1.414
```

```r
print(sqrt(2) + 10, digits = 4)
```

```
## [1] 11.41
```

Generation of a Vector

```r
c(2, 5, 7, 2, 6)
```

```
## [1] 2 5 7 2 6
```

Vectors can be called as you like: `vector = c(2,5,7,2,6)` or `vector <- c(2,5,7,2,6)`

Character Vectors

```r
text <- c("these are", "some strings")
text
```

```
## [1] "these are"    "some strings"
```

By using the `paste()` function we can merge character vectors:

```r
paste(text[1], text[2], sep = " ")
```

```
## [1] "these are some strings"
```

```r
# Special case with sep = ""
abc <- c("a", "b", "c")
paste0(1:3, abc)
```

```
## [1] "1a" "2b" "3c"
```

```r
first_name <- "Luke"
last_name <- "Skywalker"
paste("My name is", first_name, last_name, sep = " ")
```

```
## [1] "My name is Luke Skywalker"
```

```r
number <- 9
# Although number is an integer, it becomes a character by using paste():
paste(number, "is an integer", sep = " ")
```

```
## [1] "9 is an integer"
```

Generation of a Sequence

`seq(from, to, by)`

```r
seq(1, 5, 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Colon Operator

`:` generates a regular sequence (i.e., sequence in steps of one)

```r
1:6
```

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

Repetition of X

`rep(x, times, each)`

- times: the sequence is repeated n times
- each: each element is repeated n times

```r
rep(1:6, times=2)
```

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

```r
rep(1:6, each=2)
```

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

```r
rep(1:6, times=2, each=2)
```

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

Display of the n first elements of x

`head(x, n = 6)`

```r
x = c(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144)
```

```r
head(x, n = 6)
```

```
## [1] 1 1 2 3 5 8
```

Display of the n last elements of x

`tail(x, n = 6)`

```r
tail(x, n = 6)
```

```
## [1]  13  21  34  55  89 144
```

```r


c(1, 2, 3, 4, 5, 6)

mean(c(1, 2, 3, 4, 5, 6))

mean(c(1, NA, 3, 4, 5, 6), na.rm = TRUE)

mean(c(1, NA, 3, 4, 5, 6), na.rm = FALSE)

sd(c(1, 2, 3, 4, 5, 6))

sum(c(1, 2, 3, 4, 5, 6))

min(c(1, 2, 3, 4, 5, 6))

range(c(1, 2, 3, 4, 5, 6))

seq(from = 1, to = 6, by = 1)

1:6

rep(1:6, times = 2)

rep(1:6, each = 2)

rep(1:6, times = 2, each = 2)
```

Examples

R Variables

So far, we have not saved the results of our calculations. Of course, we can define variables in R, and assign a value to them.

Variables are defined in R like this: `my_var <- 4`. `<-` here is a special assignment arrow, and consists of a `<` character and a `-`. There is a key combination `ALT` + `-` in RStudio for this. So here we assign the value 4 to the new variable `my_var`.

However, it is also possible to assign a variable with `=` instead of `<-` However, since `=` is also the symbol for the assignment of arguments (for functions), misunderstandings can potentially arise here. R purists therefore prefer to use `<-`.

Variable Names

A variable must have a name - this consists of `letters`, `numbers` and the characters `_` and/or `.` A variable name must start with a letter and must not contain spaces.

There are a few conventions to follow to make R code readable and understandable - especially when sharing it with others. We recommend using `snake_case` for naming, i.e., we separate words within a name with an underscore: `my_var`.

If we have defined a variable:

```r
my_var <- 9
```

we can display their value in the console:

```r
print(my_var)
```

```
## [1] 9
```

```r
# or simply

my_var
```

```
## [1] 9
```

Variables exist in the Global Environment, but only as long as the current R session remains. If you restart R, these variables are no longer present. Therefore you should always record what you have done in an R script/notebook.

Data Types

So far we have worked with vectors. These represent the fundamental data type. All further data types build on this. Vectors themselves can be divided into the following types:

- **Numeric Vectors**: These are what we have been dealing with so far. Numeric vectors are further divided into `integer` (whole numbers) and `double` (real numbers).
- **Character Vectors**: The elements of this type consist of characters surrounded by quotation marks (either `'` or `"`), e.g. `'word'` or `"word"`.

- **Logical Vectors**: The elements of this type can take only 3 values: `TRUE`, `FALSE` or `NA`.

Vectors must consist of the same elements, i.e., we cannot mix `logical` and `character` vectors. Vectors have 3 properties:

- Type: `typeof()`: What is it?
- Length: `length()`: How many elements?
- Attributes: `attributes()`: Additional information (metadata)

Vectors are created either with the `c()` (abbreviation for combine) function, or with special functions like `seq()` or `rep()`.

Subsetting of Vectors

We can select the individual elements of a vector with `[]` (this is called `subsetting` in technical jargon):

```
#Numeric vectors consist of numbers. These are either natural numbers (integer) or real numbers (double

numbers <- c(1, 2.5, 4.5)
```

```
# First Element:
numbers[1]
```

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

```
# Second Element:
numbers[2]
```

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

```
#Third Element:
numbers[3]
```

```
## [1] 4.5
```

```
# Alternatively, the last element - here the third - can be identified by the length of the vector:
numbers[length(numbers)]
```

```
## [1] 4.5
```

```
# With - (minus) we can omit an element, e.g., all elements except the first one:
numbers[-1]
```

```
## [1] 2.5 4.5
```

```r
# We can use a sequence, e.g., the first two elements:
numbers[1:2]
```

```
## [1] 1.0 2.5
```

```r
# We can omit the first and third elements:
numbers[-c(1, 3)]
```

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

```r
# Or let us output only the elements that are greater than a threshold:
numbers[numbers>2]
```

```
## [1] 2.5 4.5
```

Matrices As mentioned before, actually everything in R is a vector. A scalar is a vector of length 1. A matrix is in principle also a vector, but one with a `dim` (dimension) attribute:

```r
m <- matrix(1:8, nrow = 2, ncol = 4)
m
```

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

There are two more functions we should get to know: `cbind()` and `rbind()`. These are used to join vectors or matrices.

`cbind()` combines the columns (column-bind) of vectors/matrices to one object:

```r
x1 <- 1:3
x1
```

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

```r
x2 <- 10:12
x2
```

```
## [1] 10 11 12
```

```r
m1 <- cbind(x1, x2)
m1
```

```
##      x1 x2
## [1,]  1 10
## [2,]  2 11
## [3,]  3 12
```

This results in a matrix `m1` with the output vectors as columns.

`rbind()` combines the rows (row-bind) of vectors/ matrices to one object:

```
m2 <- rbind(x1, x2)
m2
```

```
##    [,1] [,2] [,3]
## x1    1    2    3
## x2   10   11   12
```

Matrices can also be indexed with `[]`. But we have to specify here which row(s) and column(s) we want to get, using `[rownumber, columnnumber]`.

Missing Values Missing values are declared with `NA`.

```
numbers <- c(12, 13, 15, 11, NA, 10)
numbers
```

```
## [1] 12 13 15 11 NA 10
```

The `is.na()` function can be used to test whether something is actually a missing value:

```
is.na(numbers)
```

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

The `na.omit()` function removes all incomplete cases of a data object.

```
na.omit(numbers)
```

```
## [1] 12 13 15 11 10
## attr(,"na.action")
## [1] 5
## attr(,"class")
## [1] "omit"
```

```
#The output above consists of all cases that are not NA. However, the output also consists of additiona

numbers_omit <- as.numeric(na.omit(numbers))
numbers_omit
```

```
## [1] 12 13 15 11 10
```

Data Frames

Data Frames can be considered as the most important objects in R. Data sets are represented in R by data frames. A data frame consists of rows and columns and corresponds to a data set in SPSS.

Technically, a Data Frame is a list whose elements are equal-length vectors. The vectors themselves can be `numeric`, `logical` or `character` vectors, or also factors. A Data Frame is a 2-dimensional structure, and can be indexed like a vector on the one hand (more precisely: like a matrix), and like a list on the other hand.

Traditionally, Data Frames are defined in R with the function `data.frame()`.

```
trad_df <- data.frame(gender = c("male", "female", "male", "male", "female"), age = c(22, 45, 33, 27, 3
trad_df
```

```
##   gender age  hometown
## 1   male  22 Darmstadt
## 2 female  45 Frankfurt
## 3   male  33 Offenbach
## 4   male  27 Frankfurt
## 5 female  30 Darmstadt
```

You can do a lot with data frames, e.g., we will talk about the sub setting in a moment. But already on such initially created data frames we can apply some functions, among them the well-known `print()` function to output the data frame, but also the `str()` function to display the structure of the data frame or a statistical summary and nature of the data by applying `summary()` function.

```
print(trad_df)
```

```
##   gender age  hometown
## 1   male  22 Darmstadt
## 2 female  45 Frankfurt
## 3   male  33 Offenbach
## 4   male  27 Frankfurt
## 5 female  30 Darmstadt
```

```
str(trad_df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ gender  : chr  "male" "female" "male" "male" ...
##  $ age     : num  22 45 33 27 30
##  $ hometown: chr  "Darmstadt" "Frankfurt" "Offenbach" "Frankfurt" ...
```

```
summary(trad_df)
```

```
##     gender               age         hometown
##  Length:5           Min.   :22.0   Length:5
##  Class :character   1st Qu.:27.0   Class :character
##  Mode  :character   Median :30.0   Mode  :character
##                     Mean   :31.4
##                     3rd Qu.:33.0
##                     Max.   :45.0
```

Excurse: Packages

Almost every software has extensions of some kind. Some have extensions, some have plug-ins, some have add-ons. Different terminology for the same principle: more features through other people's extensions. In R, such extensions are called *Packages*.

Packages provide additional functions that are not included in the base R package. We first install a collection of packages for data manipulation (`tidyr`, `dplyr`, `forcats`), for importing data files (`readr`) and for graphics (`ggplot2`). These can all be installed together with this command, which we enter in the console:

```
install.packages("tidyverse")
```

This installation needs to run only once!

After that we can load the packages like this:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
## v purrr   0.3.4
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

You need to run this every time you open RStudio again!

Back to Data Frames

In the `tidyverse` package, data frames are recently also called `tibbles` or `tbl`. tibbles `are` defined with the `tibble()` function, and are just a modern variant of a data frame. They make working with data sets easier.

A Data Frame is defined like this:

```
df <- tibble(gender = c("male", "female", "male", "male", "female"), age = c(22, 45, 33, 27, 30), homet

df
```

```
## # A tibble: 5 x 3
##   gender   age hometown
##   <chr>  <dbl> <chr>
## 1 male      22 Darmstadt
## 2 female    45 Frankfurt
## 3 male      33 Offenbach
## 4 male      27 Frankfurt
## 5 female    30 Darmstadt
```

`df` is now a Data Frame with two variables, `gender` and `age`. In the Environment section of RStudio, data frames appear under Data.

A data frame has the attributes `names()`, `colnames()` and `rownames()`, where `names()` and `colnames()` mean the same thing.

```
attributes(df)
```

```
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
##
## $names
## [1] "gender"   "age"      "hometown"
```

The length of a Data Frame is the length of the list, i.e., it corresponds to the number of columns. This can be queried with `ncol()`; `nrow()` gives the number of rows.

```
ncol(df)
```

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

```
nrow(df)
```

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

Data Frame Subsetting

As mentioned above, a data frame can be indexed like a list, or like a matrix.

- Like a list: the individual columns can be selected with `$`.
- Like a matrix: the elements can be selected with `[ ]`.

```
df$gender
```

```
## [1] "male"   "female" "male"   "male"   "female"
```

```
df$age
```

```
## [1] 22 45 33 27 30
```

```
df$hometown
```

```
## [1] "Darmstadt" "Frankfurt" "Offenbach" "Frankfurt" "Darmstadt"
```

```
df["gender"]
```

```
## # A tibble: 5 x 1
##   gender
##   <chr>
## 1 male
## 2 female
## 3 male
## 4 male
## 5 female
```

```r
df["age"]
```

```
## # A tibble: 5 x 1
##     age
##   <dbl>
## 1    22
## 2    45
## 3    33
## 4    27
## 5    30
```

```r
df["hometown"]
```

```
## # A tibble: 5 x 1
##   hometown
##   <chr>
## 1 Darmstadt
## 2 Frankfurt
## 3 Offenbach
## 4 Frankfurt
## 5 Darmstadt
```

```r
# Select by position
df[1]
```

```
## # A tibble: 5 x 1
##   gender
##   <chr>
## 1 male
## 2 female
## 3 male
## 4 male
## 5 female
```

```r
df[2]
```

```
## # A tibble: 5 x 1
##     age
##   <dbl>
## 1    22
## 2    45
## 3    33
## 4    27
## 5    30
```

```r
df[3]
```

```
## # A tibble: 5 x 1
##   hometown
##   <chr>
## 1 Darmstadt
## 2 Frankfurt
## 3 Offenbach
## 4 Frankfurt
## 5 Darmstadt
```

Similar to matrices, rows and columns can be selected, again with `[row number, column number]`.

```
# First row, first column
df[1, 1]
```

```
## # A tibble: 1 x 1
##   gender
##   <chr>
## 1 male
```

```
# First row, all columns
df[1, ]
```

```
## # A tibble: 1 x 3
##   gender   age hometown
##   <chr>  <dbl> <chr>
## 1 male      22 Darmstadt
```

```
# All rows, first column
df[, 1]
```

```
## # A tibble: 5 x 1
##   gender
##   <chr>
## 1 male
## 2 female
## 3 male
## 4 male
## 5 female
```

```
# Also, we can use sequences
# First three rows, all columns
df[1:3, ]
```

```
## # A tibble: 3 x 3
##   gender   age hometown
##   <chr>  <dbl> <chr>
## 1 male      22 Darmstadt
## 2 female    45 Frankfurt
## 3 male      33 Offenbach
```

Since the columns are vectors, we can also index them:

```r
df$gender[1]
```

```
## [1] "male"
```

```r
# or

df$age[2:3]
```

```
## [1] 45 33
```

Pipe Operator

We've already noticed that code can quickly become cluttered when we perform a sequence of operations. This leads to nested function calls.

Example: We have a numerical vector of n = 10 measured values (generated here for training purposes with `rnorm()` from normally distributed random numbers) and want to center these first, then calculate the standard deviation, and finally round to two decimal places.

```r
set.seed(1283)
random_sample <- rnorm(10, 24, 5)
random_sample
```

```
##  [1] 24.74984 21.91726 23.98551 19.63019 23.96428 22.83092 18.86240 19.08125
##  [9] 23.76589 21.88846
```

We can perform the desired calculation of the rounded standard deviation of the centered values as nested function calls:

```r
round(sd(scale(random_sample,
               center = TRUE,
               scale = FALSE)),
      digits = 2)
```

```
## [1] 2.19
```

The `scale()`, `sd()` and `round()` functions are now executed in sequence (from the inside out), in such a way that the output of one function is passed as input to the next function.

The `scale()` and `round()` functions have additional arguments: `center = TRUE`, `scale = FALSE`, and `digits = 2`, respectively. This is efficient, but leads to code that is difficult to read.

An alternative to this would be to store the intermediate steps as variables:

```r
random_sample_z <- scale(random_sample, center = TRUE, scale = FALSE)

sd_random_sample_z <- sd(random_sample_z)

sd_random_sample_z_rounded <- round(sd_random_sample_z, digits = 2)

sd_random_sample_z_rounded
```

```
## [1] 2.19
```

This way, each of the substeps is on a separate line and we understand the code without any problems. However, this method requires us to define two variables that we don't actually need.

But there is a very elegant method to call functions one after the other without having to write them nested: we use the `pipe` operator. This is provided by the package `dplyr` and looks like this:

```
library(dplyr)
```

```
%>%
```

and is defined as an *Infix* operator. This means that it stands *between two* objects, similar to a mathematical operator. The name `pipe` is to be understood in the way that we "pass" or "pass on" an object to a function.

This `pipe` operator is used so often that it already has its own key combination: `Cmd+Shift+M` (macOS) or `Ctrl+Shift+M` (Windows and Linux).

Our example from above:

```
round(sd(scale(random_sample,
               center = TRUE,
               scale = FALSE)),
      digits = 2)
```

```
## [1] 2.19
```

becomes with the `%>%` operator to:

```
library(dplyr)
random_sample %>%
    scale(center = TRUE, scale = FALSE) %>%
    sd() %>%
    round(digits = 2)
```

```
## [1] 2.19
```

This code is to be read like this:

1. We start with the object `random_sample` and pass it with `%>%` as argument to the function `scale()`
2. We apply scale(), with the additional arguments `center = TRUE`, `scale = FALSE` to it, and pass the output as an argument to the function `sd()`
3. We apply `sd()` (without further arguments) and pass the output as argument to `round()`
4. `round()`, with the further argument `digits = 2`, is executed. Since no further `pipe` follows, the output is written to the console.

So it is clear: if we want to use the result further, we have to assign it to a variable:

```r
sd_random_sample_z_rounded <- random_sample %>%
  scale(center = TRUE, scale = FALSE) %>%
  sd() %>%
  round(digits = 2)

sd_random_sample_z_rounded
```

```
## [1] 2.19
```

So we pass an object to a function with `%>%`. If we do not specify anything else, this object is the first argument of the function.