Data manipulation and visualization with R

1.2 Introduction to the R language

1.3 Introduction to the R language

Overview

In this chapter we will learn the basics of the R language.

While learning the basics of a new (natural or programming) language can happen rather quickly, the process of mastering it will require several years of dedication and hard work. After reading this chapter, you will not be a master in R programming. But you will have a good grasp on the fundamental concepts of the language. Our goal is to give an introduction to all central topics that you will eventually come across along the way.

First, we will look at the syntax and semantic of the R language: the basic grammar. As R is used for many different types of statistical data analyses, we will explore all relevant types of data and how they are structured. We will look at methods to iterate over elements of a dataset, using for-loops, and the principle of conditional statements to check the validity of statements and propositions. Finally, we will learn how to use packages that will provide advanced or more specified functions to the programming language.

Most of these topics will also reappear in later chapters, where we will take a closer look at them.

Syntax and semantic

Syntax

Programming is a communication process between a user and a computer. The computer can only make sense of our commands when we use the right **syntax**. This term refers to “the way in which linguistic elements (such as words) are put together to form constituents (such as phrases or clauses)” (https://www.merriam-webster.com/dictionary/syntax). Any typo or incorrect use of signs and characters will leave our code incomprehensible to the computer. The following table showcases a few instances where incorrect syntax is used in English as well as in R. The three shown types of syntax errors belong to the major reasons why code may not work as intended by the programmer.

Syntax error	**Natural language example**	**Programming language example**
Wrong punctuation	I went. to bed.	Correct: as.numeric(5) Incorrect: asnumeric(5)
Wrong word order	I to bed went.	Correct: as.numeric(5) Incorrect: 5.as.numeric( )
Typos	I went to bad.	Correct: as.numeric(5) Incorrect: as.numerc(5)

Syntax and semantic

Semantic

Semantic relates to the "meaning in language". A sentence without grammatical mistakes or typos can be void of any meaning or be uninterpretable. For example, the following sentence by Noam Chomsky, although grammatically correct, has no comprehensible meaning:

"Colorless green ideas sleep furiously."

Semantic errors in coding can be hard to spot as the computer may be able to execute the code when the command has correct syntax. The result, however, may be nonsensical or simply an answer to a question the programmer never asked.

The following coding example uses correct syntax and returns a correct result. Nonetheless, the command is rather pointless: The mean of the mean of the mean of 5 (equivalent to: (((5/1)/1)/1)) is always 5 itself.

```
mean(mean(mean(5)))
```

Syntax and semantic

Case sensitivity

In the syntax of R, uppercase and lowercase letters are distinct, e.g. "R" and "r" are treated as entirely different characters.

The first example shows an incorrect way to call the "mean( )" functions. The vast majority of functions use lowercase letters in their names. Calling a function by a wrong name will not trigger it.

This principle also applies to the declaration of variables. "X" and "x" would be stored as two different variables without any connection to each other. A consistent way of naming variables is recommended to avoid confusion.

For variables and functions with longer names, "camelCasing" is often used, where a capital letter indicates the beginning of a new word.

```
# example 1: wrong use of uppercase letters in functions
mean(...)                # the mean( ) function uses all lowercase letters
Mean(...)                # uppercase "M"
MEAN(...)                # all uppercase letters


# example 2: wrong use of uppercase letters in variables
x = 5                    # save the value 5 as a variable called "x"
print(x)                 # display the value of x on the screen
print(X)                 # error: object "X" not found


# example 3: how to give variables proper names
numberofevilexes = 7     # variable is difficult to read
numberOfEvilExes = 7     # variable name is easier to read
addDataFrame(...)        # example of function using camelCasing
```

Syntax and semantic

Common mistakes and oversights

One of the most common syntax mistakes concerns brackets: ( ), { }, [ ]. All three kinds of brackets are used in a different way and are not interchangeable. We will discuss all of them in later chapters. Every bracket opened must be closed: “function1(function2(function3(data)))”. Mixed brackets close in reversed order of opening: “function(data[selection])”.

The letters “c”/”C”, “s”/“S” and “x”/”X” are often difficult to tell apart. Variables and functions should use uppercase and lowercase letters consistently to avoid confusion. For example, one variable called “x” and one called “Y” will eventually lead to the programmer calling “X” or “y” by mistake.

Variables should not contain any special signs like “/”, “*”, or “\” as they may trigger something unintended. Underscores “_” are often used by some programmers instead of camelCasing. Underscoring makes individual names in functions and variables more obvious, but also leads to longer names to type. This is mostly a matter of preference, though we recommend sticking to one prefered coding style. For example, the variables “x_Pos” and “positionY” will lead to confusion while coding; better to call them “xPosition” and “yPosition” (or “x_Position” and “y_Position”) for clarity. When copying code from the internet (e.g. stackoverflow.com), the code should be adjusted to go along with the coding style already used to avoid the implementation of inconsistencies.

Data types and classes

Introduction

R is an object-oriented programming language (like Java or Python), where each data used within the program has a specific class and a type. The exact definitions of “data class” and “data type” are not relevant for this course and, hence, will be omitted. Simply put: All data has to be put into a specific container for the program to know how to handle it. For example, numbers can be added and multiplied, combinations of letters to words cannot.

Atomic or primitive data types are the simplest building blocks of the programming language. They can either be individual numbers, letters or logical states (TRUE / FALSE). In this course we will look at the following atomic data types: logical, integer, double and character. There are further types like complex numbers which will not be discussed in this course.

These building blocks can be combined within data structures to objects containing data, like lists of names or tables of numbers. Data structures are containers that can include types of atomic data. In this course we will look at: vectors, lists, factors, matrices and data.frames.

Data types and classes

Atomic data types: Logicals

At first we will look at logicals. In other programming languages they are also known as Booleans, named after George Boole (1815-1864). They represent two binary states: TRUE / FALSE.

Logicals are mainly used in conditional programming: DO something IF condition is fulfilled or to receive a feedback for a specific condition or comparison.

Please note that R uses logicals in all caps: TRUE and FALSE while other languages use them written with minor letters. R also accepts the short versions T and F. However, this is not recommended as the user may introduce variables named "T" or "F" which will then break the code unintentionally.

To compare two statements "==" is used.

```
# Check validity of a statement
# Is 1 equal to 1?
1 == 1
# output: TRUE (because both sides are equal)

# Is 1 equal to 2?
1 == 2
# output: FALSE (because both sides are different)

# Logicals can be compared to other logicals:
T==T # output: TRUE (because both sides are equal)
F==F # output: TRUE (because both sides are equal)
F==T # output: FALSE (because both sides are different)
T==F # output: FALSE (because both sides are different)
```

Data types and classes

Atomic data types: Logicals

Logical comparisons can also include mathematical operations.

```
1+1 == 2
# output: TRUE (1 plus 1 is equal to 2)

1+1 == 2+2
# output: FALSE (2 is not equal to 4)
```

Functions like identical( ) check two variables and return a logical.

```
identical(1, 1)
# output: TRUE (1 and are identical)

identical(1+1, 2)
# output: TRUE (1+1 is identical to 2)
```

Logical comparisons can also be made using variables.

```
a <- TRUE
b <- TRUE
c <- FALSE

a==b
# output: TRUE (TRUE is equal to TRUE)

a==c
# output: FALSE (TRUE is not equal to FALSE)
```

Logicals and related functions will be discussed in greater detail later in this chapter.

Data types and classes

Atomic data types: Double and integer

In most cases, numbers have either the type double or integer.

Doubles are numbers that might have a decimal precision, e.g. 1.5 or 3.14. In other languages they are also known as floats. While the type of a decimal number is “double”, its class is “numeric”. In this course we will use these terms synonymously.

Integers are numbers without decimal places (“whole numbers”).

The different types were introduced when the computers’ memories were more limited. As an integer would not require bits to store decimal places, it was more efficient to store values that will always be whole numbers (such as years) as integer types. Nowadays this division is less meaningful.

```
# check types of numbers
# If not explicitly declared, all numbers will be created as doubles.
typeof(3.1) # output: “double”
typeof(3) # output: “double”

# Integers can be created using “as.integer(x)”
# if necessary rounded down to the next lowest whole number
as.integer(3.5) # output: 3
typeof(as.integer(3.5)) # output: “integer”

# Numbers with quotation marks are not recognized as either type
typeof("3.5") # output: “character”
typeof(as.integer("3.5")) # output: “integer”
```

Data types and classes

Atomic data types: Character

Characters are individual letters or numbers or combinations thereof. They are identified by surrounding quotation marks. A group of letters and numbers within a single character container is called a string.

```
# create character string assigned to the variable called sentence
sentence <- "The quick brown fox jumps over the lazy dog"

typeof(sentence) # output: "character"
```

Everything put between quotation marks will be will be converted into a character string: “this string of characters”, “R2D2”, “4”, “1+1”. Mathematical operations on character are not possible.

Please note that R does not recognize cursive quotation marks as used in Word or PowerPoint. They will break the code.

```
# Trying to do operations with characters:
1+1       # output: 2
1+"1"    # output: error (non-numeric argument)
"1"+"1" # output: error (non-numeric argument)

# To combine characters we use paste( )
# The separator (sep) includes separations
paste("Oh", "hi", "Mark", sep="")  # output: "OhhiMark" (no sep)
paste0("Oh", "hi", "Mark", )  # output: 2: "OhhiMark" (no sep)
paste("Oh", "hi", "Mark", , sep=" ")  # output: "Oh hi Mark" (spaces)
paste("Oh", "hi", "Mark", , sep=",")  # output: "Oh,hi,Mark" (commas)

paste(paste("Oh", "hi", sep=" "), "Mark", sep=", ")
# output: "Oh hi, Mark" (spaces and commas)
```

Data types and classes

Type Coercion

Type coercion refers to a forced conversion of a data type to another.

The user can explicitly force a certain type onto data:

```
as.integer(3.5) # output: 3 (type integer)
as.character(3.5) # output: "3.5" (type character)
as.logical(1) # output: TRUE (type logical)
as.logical(0) # output: FALSE (type logical)
```

Coercion can lead to loss of information. Doubles are the most specific type. Integers can be doubles but do not gain more decimal precision. Doubles converted to integers lose all decimal places. Logicals can be expressed as 0 or 1 if converted to numbers. Characters can be converted only if they include pure numbers.

	as.double	**as.integer**	**as.logical**	**as.character**
Double (3.5)	3.5	3	TRUE*	“3.5”
Integer (5)	5	5	TRUE*	“5”
Logical (TRUE) **Logical (FALSE)**	1 0	1 0	TRUE FALSE	“TRUE” “FALSE”
Character (“hello”) **Character (“3.5”)**	NA** 3.5	NA** 3	NA** NA**	“Hello” “3.5”

Figure X: Type coercions can be applied to every type. However, information might get lost.
* every number converted to a logical results in TRUE
** coercing characters into anything else results in NA, a missing value

Data structures

Vectors

The first data structure we will discuss are Vectors. Vectors are one-dimensional mono-class data containers.

Unlike tables that have rows and columns, vectors are one-dimensional, i.e. they only consist of one line of data, that is neither a row nor a column. The function c( ) ("concatenate") combines atomic data to a vector:

```
c(1, 2, 3, 4) # a vector containing four numbers
```

A vector can only contain one type of atomic data and will force a coercion if necessary.

```
c(1, "2" ,3, 4) # coercion to characters: c("1", "2", "3", "4")
c(1, FALSE, 3, 4) # coercion to doubles: c(1, 0, 3, 4)
```

```
# Usually the c( ) function generates vectors
a <- c(1,2,3)     # assign vector with three numbers to variable "a"

# Colons can be used to create a series of numbers
1:10 # same as c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# c( ) can combine multiple series or vectors to one vector.
c(1:3, 5:10)  # same as c(1, 2, 3, 5, 6, 7, 8, 9, 10)

a <- c(1,2,3)
b <- c(4,5,6)
c <- c("4","5","6") # note that a variable can also be called "c"

c(a,b) # same as c(1, 2, 3, 4, 5, 6)
c(a,c) # coercion to characters: c("1", "2", "3", "4", "5", "6")
```

Data structures

Factors

Factors contain data but store them within a finite set of categories. Any vector can be converted to a factor by using as.factor( ).

Calling a factor in the console, will display all “levels”. This can be regarded as a mathematical set that contains each distinct entry once. The example on the side shows that factor displays the levels 1 to 6.

Factors are useful when dealing with categorical data (e.g. genders, school grades). However, even though they can contain numbers, mathematical operations on them are impossible as they are not converted as doubles or integers. They can be converted back to vectors by using as.vector( ) or, preferably, a type-specific conversion like as.double( ) or as.character( ) to ensure they receive the desired type format.

```
# Creating  a vector of numbers
a <- c(1,1,2,1,4,5,6,2,4,5,6,3,3,1,4)
sum(a)  # output: 48

# Conversion to factor
b <- as.factor(a)
#output:
[1] 1 1 2 1 4 5 6 2 4 5 6 3 3 1 4; Levels: 1 2 3 4 5 6

sum(b) # output: error (‘sum’ not meaningful for factors)

# Conversion back to numbers
a <- as.double(b)
sum(a) # output: 48
```

Data structures

Factors

While some categorical data are unordered (e.g. genders could be M/F or F/M) other categorical data need an order to have meaning (e.g. “low” / “med” / “high” rankings or school grades).

The example on the side creates a vector of received school grades in a factor format. By calling levels(data) the levels are presented in an alphabetical order, which, like here, does not always make sense.

The second example therefore specifies which levels the data should include and whether they should be ordered or not. Here they are brought in a logical ranking order.

With the function table( ) the frequency of each level can be displayed. The table also respects the order we established for the factor.

```
# creating a factor vector of school grades
a <- factor(c("good", "very good", "excellent","good","good","fail",
"satisfactory","very good", "fail"))
levels(a)
#output: "excellent"  "fail"  "good"  "satisfactory"  "very good"

a <- factor(
          c("good", "very good", "excellent","good","good","fail",
          "satisfactory","very good", "fail"),
          levels=c("excellent", "very good", "good", "satisfactory", "fail"),
          ordered=TRUE)

levels(a) #output: "excellent"  "very good"  "good"  "satisfactory"  "fail"
nlevels(a) #output: there are 5 different levels
table(a) #output:
  excellent  very good  good  satisfactory  fail
        1           2        3         1            2
```

Data structures

Matrices

Matrices (plural of matrix) are multi-dimensional mono-class data containers.

A matrix is two dimensional: It has rows and columns. Any vector can be converted to a matrix (by default having one column and as many rows as the vector has data).

```
as.matrix(c(1, 2, 3, 4)) # a matrix containing a column of numbers
```

A matrix can only contain one type of atomic data and will force a coercion if necessary.

```
# as.matrix( ) converts vectors to matrices.
a <- c(1,2,3)
b <- c(4,5,6)
A <- as.matrix(c(a,b))

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

# Numbers in square brackets indicate the row and column index.
# [x,] are rows, [,x] are columns
```

Data structures

Matrices

The matrix( ) function has arguments to create a desired number of rows and columns, as shown on the side.

An “m by n” matrix has m rows and n columns. It is crucial to memorize which number indicates which dimension, especially for data selection tasks. For example, A[1,2] selects the first row and the second column of matrix A. This will be further explained in the chapter 2.2 about data selection.

A mnemonic aid can be found in the word “**R**E**C**TANGLE” where the “R” (for “row”) comes before the “C” (for “column”). A matrix will always have a rectangular shape.

```
# matrix( ) converts vectors to matrices.
a <- c(1,2,3)
b <- c(4,5,6)

# specify number of rows:
matrix(c(a,b), nrow=2)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

# specify number of columns:
matrix(c(a,b), ncol=2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Data structures

Matrices

Many functions can be applied to matrices to gain summaries about values and parameters. Some of them work the same for vectors and matrices, but a few will only work with matrices.

Function	**Description**	**Vectors**	**Matrices**
length( )	Returns number of elements	yes	yes
dim( )	Returns dimensions as m by n (rows by columns)	no	yes
nrow( )	Returns number of rows	no	yes
ncol( )	Returns number of columns	no	yes

```
# matrix( ) converts vectors to matrices.
a <- c(1,2,3)
b <- c(4,5,6)
A <- matrix(c(a,b), nrow=2)

length(A)  # output: 6 (there are six elements in the matrix)
length(a)  # output: 6 (there are six elements in the vector)

dim(A)  # output: 2 3 (indicating a 2 by 3 matrix)
dim(a)  # output: NULL

nrow(A)  # output: 2 (there are two rows)
nrow(a)  # output: NULL

ncol(A)  # output: 3 (there are three columns)
ncol(a)  # output: NULL
```

Data structures

Dataframes (DF)

Dataframes (DF) are multi-dimensional multi-class data containers. Unlike matrices, each column of a DF can contain a different type of data. Most data tables will be formatted as dataframes.

The example on the side shows the creation of a matrix and a DF from two similar vectors a and b. As a and b have different types, the matrix coerces the type of all data to characters.

The DF is created by combining columns (cbind( )) of equal lengths (the two vectors). Its representation in the console differs. The columns are given the names of the vectors used to create them.

```
a <- c(1,2,3)       # vector of doubles
b <- c("x","y","z")  # vector of characters

# creating a matrix from two vectors:
matrix(c(a,b), ncol=2)
     [,1] [,2]
[1,] "1"  "x"
[2,] "2"  "y"
[3,] "3"  "z"

# creating a DF from two vectors:
data.frame(cbind(a,b))
  a b
1 1 x
2 2 y
3 3 z
```

Data structures

Dataframes (DF)

If a DF is assigned to a variable name (in this case A), the columns can be accessed individually by calling dataframe$column (e.g. A$a). The output is a vector of the column's data type.

By default, the columns are often formatted as factors depending on the way and function used to create them. It is therefore crucial to check the columns' types before using them in more complex operations.

A column can be overwritten by another vector of equal length. Thus, it is possible to overwrite a vector by a type-converted version of itself, as shown in the example. This can also be used to create new columns of equal length:

```
A$c <- c(TRUE, FALSE, TRUE) # creating new column c
```

```
# creating a DF from two vectors:
A <- data.frame(cbind(a,b))
  a b
1 1 x
2 2 y
3 3 z

# Checking columns
A$a # output: [1] 1 2 3; Levels: 1 2 3
A$b # output: [1] x y z; Levels: x y z

# Manual conversion
A$a <- as.double(A$a)
A$b <- as.character(A$b)

# Checking columns
A$a # output: [1] 1 2 3
A$b # output: [1] "x" "y" "z"
```

Data structures

Dataframes (DF)

DFs have many advantages over matrices in regards to data analyses. The columns are easier to navigate as they have specific names instead of only indices.

The example on the right shows a DF with two columns “a” and “b”. By calling names(DF) the names of all columns are listed. This can be handy when data tables become large. However, this function can also be used to overwrite existing column names by manually defined ones, as shown in the example. Now the columns are labeled A$Numbers and A$Letters.

The str( ) function (short for “structure”) shows a summary for each column. In this case, both columns are formatted as factors with three levels each.

```
# creating a DF from two vectors:
a <- c(1,2,3,3)        # vector of doubles
b <- c("x","y","z","x")  # vector of characters
A <- data.frame(cbind(a,b))
  a b
1 1 x
2 2 y
3 3 z
3 3 x

names(A) # output: "a" "b"

names(A) <- c("Numbers", "Letters")

str(A) # output:
'data.frame':      4 obs. of  2 variables:
 $ Numbers: Factor w/ 3 levels "1","2","3": 1 2 3 3
 $ Letters: Factor w/ 3 levels "x","y","z": 1 2 3 1
```

Data structures

Lists

The last data structure we will discuss are lists which are containers for other data containers. A list can combine vectors, matrices, DF and factors into a single object.

The list( ) command can take any number of elements as inputs:

```
list_name <- list(element1, element2, …) # creating new column c
```

The list gives indices to all elements and all their items. Elements can be accessed by double square brackets (list[[index_element]]) and their respective items with additional single square brackets (list[[index_element]][index_item]).

```
list_name[[1]] # output: [1] 1 2 3 4 5 6 (entire first element)
list_name[[3]][2] # output:  "y" (second item of third element)
```

```
# creating three different vectors
a <- c(1,2,3,4,5,6)
b <- c(TRUE, FALSE, TRUE, FALSE)
c <- c("x", "y", "z")

# creating a list from three different vectors
list_1 <- list(a,b,c)

# output
[[1]] # first element
[1] 1 2 3 4 5 6

[[2]] # second element
[1]  TRUE FALSE  TRUE FALSE

[[3]] # third element
[1] "x" "y" "z"
```

for-loops

Introduction

Loops are a core principle in all programming languages. Unfortunately, all languages use loops with a slightly different syntax. A loop repeats certain actions as long as a condition is fulfilled.

This can be used to trigger a function several times in a row. For example, calling a function that returns the current time a hundred times, would only require one loop instead of 100 individual lines of code.

In R, loops are often used to iterate over elements of a data container. The command “for(i in vector){ }” would iterate over each element of the vector, e.g. to show each element on screen, to tally up all elements to a sum, or to search a container for a specific value. Round brackets contain the condition, while swirly brackets contain the action.

```
# template: for each element i in the dataContainer, apply function
“doSomething( )” to that element i
for(i in dataContainer){
        doSomething(i)
}

# print all element i to the screen
vector <- c(1:5)          # all numbers from 1 to 5
sum <- 0                  # empty container for the sum
for(i in vector){         # for each element in vector
  print(i)                # output print: 1 … 2 … 3 … 4 … 5
  sum <- sum + i          # add the element to the sum
}
print(sum)                # output sum: 15
```

for-loops

Looping over indices

A common way to iterate over elements of a container is the index method. The index indicates the position of an element within the container from position 1 to n, where n is equal to the length of the container (i.e. the number of its elements): “i in 1:length(data)”

Please note, that R starts counting indices with 1, while many other language start with index 0.

The index of an element is not necessarily equal to its value. In the following example, the value of index 1 is 100, the value of index 2 is 200, and so on.

```
vector <- c(100, 200, 400, 700, 800)          # values of vector
index 1: 100, index 2: 200, index 3: 400, …   # indices of vector
```

```
# template: for each index i from 1 to length of dataContainer, apply
the index to the function doSomething(i)
for(i in 1:length(dataContainer)){
        doSomething(i)
}
```

```
# print all element with index i to the screen
vector <- c(11:15)               # vector values: 11, 12, 13, 14, 15
for(i in 1:length(vector)){      # indices i: 1, 2, 3, 4, 5
   print(vector[i])              # print value of index i
}
# output: 11 … 12 … 13 … 14 … 15
```

for-loops

Looping over elements

The loop can also apply functions directly to the elements of a data container. This can be useful when the elements are not stored in a particular order and the creation of an index would have no further use.

When working with matrices or dataframes, the user might want to make the index part of the outgoing data structure, e.g. by storing the index within an “Index” column, so that element n is also given index n. If the index number itself is not needed for any further calculations or declarations, the loop can access each element directly.

The example was already shown on the introduction slide.

```
# template: for each element i in the dataContainer, apply function
“doSomething( )” to that element i
for(i in dataContainer){
            doSomething(i)
}


# print all element i to the screen
vector <- c(1:5)          # all numbers from 1 to 5
sum <- 0                  # empty container for the sum
for(i in vector){         # for each element in vector
   print(i)               # output print: 1 … 2 … 3 … 4 … 5
   sum <- sum + i         # add the element to the sum
}
print(sum)                # output sum: 15
```

for-loops

Nested loops

We speak of “nested loops” when loops contain further loops.

By convention, the first loop iterates over elements i, while a second loop iterates over elements j. A nested loop can have many individual layers. However, if more than two loops are nested, the elements should be named more clearly (e.g. “family in house” and “member in family”).

The loop initiated last, will complete first. The example on the side will start with i=1 and iterate through j=3 … 4 before changing to i=2 and then iterate through j=3 … 4 again. The total number of iterations is the product of all elements. In this case: length(i)*length(j)=5*2=10. A nested loop with three layers of 100 elements each needs a million iterations (100 * 100 * 100), which might take some time. Hence, be aware of the needed iterations before starting a nested loop!

```
# template: for each element i in the dataContainer, apply function
“doSomething( )” to that element i
for(i in dataContainer){
            doSomething(i)
}
```

```
for(i in 1:5){      # for each element of value 1 … 2 … 3 … 4 … 5
   for(j in 3:4){   # for each element of value 3 … 4 … 5
      print(i+j)    # print the sum of i+j … 1+3=4, 1+4=5, 2+3=5, ...
   }
}
```

Conditional statements

Truth tables, AND, OR, negation

The core principle of conditional statements and conditional programming is to test whether a statement is TRUE or FALSE and which action(s) should be done in either case.

The table on the right is called a truth table. It contains the propositions X and Y, which are either TRUE or FALSE and each possible combination of their values.

Logical operators can be applied to these statements. The logical AND (&) is TRUE when both X and Y are TRUE (conjunction). The logical OR (|) is TRUE when either X or Y (or both) are TRUE (disjunction). In R the "!" in front of logical variables or statements indicates a negation: every TRUE becomes FALSE and vice versa.

X	**Y**	**AND: X&Y**	**OR: X | Y**	**Not X: !X**	**Not Y: !Y**
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE	TRUE

```
X <- c(T, T, F, F) # creating logical vector X
Y <- c(T, F, T, F) # creating logical vector Y

X & Y # output: TRUE FALSE FALSE FALSE
X | Y # output: TRUE TRUE TRUE FALSE
!X # output: FALSE FALSE TRUE TRUE
```

Conditional statements

Comparisons and data queries

Comparisons are indicated by a "==" and are similar to the logical AND: only if both sides of the comparison are the same, the output value will be TRUE. A negated comparison "!=" (not equal) is TRUE when both sides are different. R also allows comparisons:

- bigger than (x>y),
- smaller than (x<y),
- bigger or equal (x>=y) and
- smaller or equal (x<=y)

```
2 == 1+1 # output: TRUE (2 is the same as 1+1)
1 == 2 # output: FALSE (1 is not the same as 2)
1 != 2 # output: TRUE (1 is not the same as 2)
2 >= 1+1 # output: TRUE (2 is bigger or equal to 1+1)
```

Comparisons can also be used for data queries, e.g. does this data contain this specific value?

```
numbers <- c(1,3,4,3,4,5,6,1) # creating a vector of numbers

numbers  == 1 # checking each entry if it is a 1 or not
# output:  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE

any(numbers  == 1) # does the vector contain the ANY number 1?
# output:  TRUE (if a clear short statement is wanted)

sum(numbers == 1) # how many 1's does the vector contain?
# output: 2 (as each TRUE is counted as 1 and each FALSE as 0).

mean(numbers == 1) # What is the probability of 1's?
# output: 0.25 (2*TRUE, 6*FALSE;  2 of 8 equals 25%).
```

Conditional statements

If-clauses

If-clauses are another principle of all programming languages. A command will only be executed when a certain condition is fulfilled.

- if(TRUE) {do something}
- if(FALSE) {don’t do something}

If-clauses are used when an action should only be executed under specific circumstances:

- if(doorbell rings) {open the door}
- if(I need new shoes) {buy new shoes}
- if(I don’t need new shoes) {don’t buy new shoes}

Similar to for-loops, we use round brackets for the condition and swirly brackets for the actions.

```
# example 1: execute when a logical is TRUE
bool <- TRUE
if(bool){
   print("The value is true.")
} # output: “The value is true.”


# example 2: do not execute when a logical is FALSE
bool <- FALSE
if(bool){
   print("The value is true")
} # no output

# example 3: execute when an equation or comparison is true
if(1+1 == 2){
   "The equation is correct."
} # output: “The equation is correct.”
```

Conditional statements

If- / else-clauses

When an if-clause is not triggered, usually nothing happens. However, an alternative case can be defined that is always executed when the if-condition is not fulfilled: the else-clause.

This is useful when we still want a feedback, that something did not fulfill the condition. For example:

```
if(testing for disease is positive) {give patient response A}
else {give patient response B}
```

Naturally, the patient wants to know about the result in either case. Only the response will be different.

Please note that the else-clause does not require a condition like the if-clause, because it is automatically triggered when the if-condition is FALSE.

```
# example: if else statement to cover true and false conditions
# if-clause triggered
if(1+1 < 5){
   "The equation is correct."
}else{
   "The equation is wrong."
}
# output: “The equation is correct.”


# else-clause triggered
if(1+1 > 5){
   "The equation is correct."
}else{
   "The equation is wrong."
}
# output: “The equation is wrong.”
```

Conditional statements

If- / else- / else-if-clauses

An else-if-clause also gives the else-clause a condition. This is useful when more than one condition could be fulfilled, but a prioritization needs to happen. For example:

- if(wife wants to go to restaurant A) {go to restaurant A}
- else if(wife wants to go to restaurant B) {go to restaurant B}
- else if(wife wants to order food) {order food}
- else {starve}

Naturally, only one option will be chosen. The order matters: If one if- or else-if-condition is fulfilled, all following else-clauses are automatically denied.

In these examples we show that the condition and the action can be put in the same line, when the action only consists of one line of code.

```
# example 1: both conditions are TRUE, both are triggered.
if(TRUE){print("First condition is TRUE")}
if(TRUE){print("Second condition is TRUE")}
# output: "First condition is TRUE" … "Second condition is TRUE"


# example 2: both conditions are TRUE, only first is triggered
if(TRUE){print("First condition is TRUE")
}else if(TRUE){print("Second condition is TRUE")}
# output: "First condition is TRUE"


# example 3: both conditions are FALSE, else is triggered
if(FALSE){print("First condition is TRUE")
}else if(FALSE){print("Second condition is TRUE.")
}else{print("No condition is TRUE.")}
# output: "No condition is TRUE"
```

Conditional statements

Nested if-clauses

```
"1 is neither divisible by 2 or 3."
"2 is divisible by 2."
"3 is divisible by 3."
"4 is divisible by 2."
"5 is neither divisible by 2 or 3."
"6 is divisible by 2 and 3."
```

Finally, the following example brings everything together. We want to check whether a number in a vector is divisible by either 2 or 3 or both.

The “modulo” operator “%%” returns the remainder of a division, e.g. “3%%2” is 1. A remainder of 0 indicates that the first number is a multiple of the second one. paste( ) is used to combine variables and strings. Thus, we get a dynamic feedback based on i.

We first test whether a number is divisible by 2 OR 3. If not, the else-clause is triggered. We then test whether the number is a multiple of both 2 AND 3. If yes, the else-if-clauses will be skipped and the next number is selected. If a number is divisible by 2 or 3 but not both, we test if it is a multiple of 2 or 3 (here, the order does not matter, as we know, that exactly one of them must be triggered.) Please take note of the code’s indentations indicating the different layers of conditions and loops.

```
# example: nested if-, else- and else-if-clauses
# Which numbers are divisible by 2, 3 or both?
vector <- c(1:12) # all numbers from 1 to 12

for(i in vector){              # for each element in vector (1 to 12)
   if(i %% 2 == 0 | i %% 3 == 0){              # if i is divisible by 2 OR 3
      if(i %% 2 == 0 & i %% 3 == 0){           # if i is divisible by 2 AND 3
         print(paste(i, "is divisible by 2 and 3."))
      } else if(i %% 2 == 0){                  # else if i is div. only by 2
         print(paste(i, "is divisible by 2."))
      } else if(i %% 3 == 0){                  # else if i is div. only by 3
         print(paste(i, "is divisible by 3."))
      }
   } else {print(paste(i, "is neither divisible by 2 or 3."))} # else
}
```

Packages

Packages

A package is a downloadable extension that provides functions not implemented in the standard set of functions and commands. Programmers often call software without extensions implemented by the developers and without user customization "Vanilla" in reference to, probably, the most basic ice cream flavor.

Packages can be downloaded manually or directly via R code executed in RStudio. The following line of code installs the specified package from the internet and all packages it needs to work:

```
install.packages("package_name")
```

Once a package is installed, it can be activated by this command:

```
library("package_name")
```

In this course we will mainly deal with the packages dplyr (manipulation of dataframes) and ggplot2 (data visualization).

Both packages are part of the package tidyverse, which includes several packages designed for data science. Check out the collection on its official website:

https://www.tidyverse.org/packages/

Other packages enable the import and manipulation of geospatial data, Excel and XML files or various Machine Learning methods. However, this exceeds this course's scope.

1.2 Introduction to the R language

What have we learned?

In this chapter we learned about many core principles and methods used in the R language.

We learned about the language's basic syntax and semantic, so we can have a smooth conversion with the program while coding.

Learning about the numerous data types and data structures is a big part of getting familiar with the language. While you do not need to memorize many things by heart, it is recommended that you have a firm understanding of the data types and structures before you proceed. The manipulation of data will be the focus of the next chapters.

The different conditional classes and loops will not be the main focus of the following chapters. But we will meet them again when we encounter a good situation to make use of them.

Eventually, two chapters will be dedicated entirely to two specific packages, "dplyr" (for data manipulation) and "ggplot2" (for data visualization).

We did not go into many details with most of the topics. But this will change in the following chapters!