Programming in R - Basic Concepts

Version 2.1

Zambach, Sine

Document Version
Final published version

Publication date:
2023

License
CC BY-NC-SA

Citation for published version (APA):
Zambach, S. (2023). *Programming in R - Basic Concepts: Version 2.1*. Copenhagen Business School, CBS.

Link to publication in CBS Research Portal

Programming in R - basic concepts

Version 2.1

Sine Zambach, 2023

2023-11-30

Contents

Chapter 1

About

This book is an introduction to R for the very beginner. It can also be a nice place to start if you have coded a bit but need more insight into what you are actually doing. After reading this compendium you are ready for more advances introductions to R, which often assume that you know some programming.

You can read the compendium in one long run or you can split it up. I recommend you to actually copy the code and run it in your own script or rmd-file (more on that in the next chapter). It should not take a long time as there is plenty with space - even if you run all the code as well.

Of cause you will not learn to really program before you have done some tasks using the tools you get here, either solving some exercises or - even better - working on your own data.

Chapter 7 on Regular expression is maybe not so important for the very beginner, so you can jump over it, and return to it when you need to work with text.

The material is developed by Sine Zambach, but some of the ideas - and even some of the text of this compendium is based on material from my old colleague, Irfan Kanat, used with his consent.

If you find errors, or have ideas for developing additional material, please don't hesitate to contact me.

Good luck with programming R!

Sine Zambach, `sz.digi@cbs.dk`

Copenhagen Business School, Department of Digitalization

December, 2022

Chapter 2

Programming for people

Today, many learn R as their first programming language. They learn statistics, data wrangling, visualization or even machine learning with R as a tool on the side. However, R is a programming language, and it becomes much easier to learn, if you learn to program *before* you start to wrangle, analyze, and visualize data.

There exists a lot of good material on how to use advanced or semi-advanced R-programming, data wrangling, etc, but I have missed material, that introduces the absolute beginner to programming in R. That is why I have created this compendium for you.

My students are normally not computer scientists, but the might have had some introduction to statistics or maybe not. And though most have used Excel and other spread sheet programmes to some extent, they are learning R as the first "write code in a script"-language.

So when I have been planning R-courses, I found that it was very hard to find adequate material for the first few lectures "Introduction to programming using R", and therefore I have developed these notes.

My philosophy is, that when you have learned a bit of programming, it is much easier to learn all the fun stuff you can use R for. You will also have an easier time in progress to Python and other programming languages, should that be important to you.

I have tried to keep it short and concise. However, I have also included sections with narratives for some of the abstract concepts that can be hard to grasp without an illustrative example.

If you already have some programming skills, you can go directly to for instance Hadley Wickhams [R for Data Science]{https://r4ds.had.co.nz/}

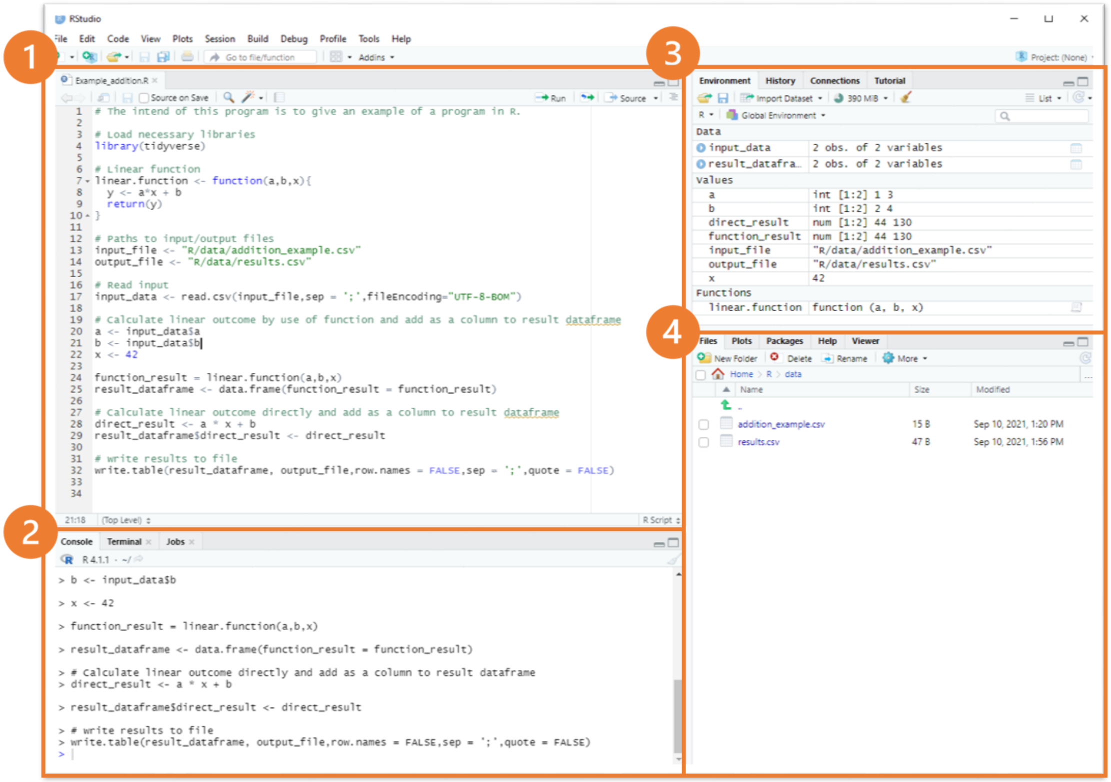
2.1 About this chapter

This chapter is mostly concerned with practical issues on how to work with R, and which nice tools you have to make your programming more comfortable. The actual coding concepts will be introduced in the next chapter.

2.2 RStudio

One of the reasons behind R's ongoing success is probably RStudio. That is the development-environment where you code, view your files and packages, and it provides a lot of help for programming.

The four panes in RStudio comprise: 1. Your script, in which you write your program. 2. The Console where the program code is run. Nothing is executed before you have run it there. 3. The Environment, where you have an overview of your data, variables and functions. 4. A pane with several important tabs, such as: Plots, for your plots, Files, where you can see in which folder you are operating, and Help, where you can get help, using a search function.



```
# The intend of this program is to give an example of a program in R.

# Load necessary libraries
library(tidyverse)

# Linear function
linear.function <- function(a,b,x){
  y <- a*x + b
  return(y)
}

# Paths to input/output files
input_file <- "R/data/addition_example.csv"
output_file <- "R/data/results.csv"

# Read input
input_data <- read.csv(input_file,sep = ';',fileEncoding="UTF-8-BOM")

# Calculate linear outcome by use of function and add as a column to result dataframe
a <- input_data$a
b <- input_data$b
x <- 42

function_result = linear.function(a,b,x)
result_dataframe <- data.frame(function_result = function_result)

# Calculate linear outcome directly and add as a column to result dataframe
direct_result <- a * x + b
result_dataframe$direct_result <- direct_result

# write results to file
write.table(result_dataframe, output_file,row.names = FALSE,sep = ';',quote = FALSE)
```

```
> b <- input_data$b

> x <- 42

> function_result = linear.function(a,b,x)

> result_dataframe <- data.frame(function_result = function_result)

> # Calculate linear outcome directly and add as a column to result dataframe
> direct_result <- a * x + b

> result_dataframe$direct_result <- direct_result

> # write results to file
> write.table(result_dataframe, output_file,row.names = FALSE,sep = ';',quote = FALSE)
>
```

```
Data
input_data        2 obs. of 2 variables
result_datafra…   2 obs. of 2 variables
Values
  a               int [1:2] 1 3
  b               int [1:2] 2 4
  direct_result   num [1:2] 44 130
  function_result num [1:2] 44 130
  input_file      "R/data/addition_example.csv"
  output_file     "R/data/results.csv"
  x               42
Functions
  linear.function function (a, b, x)
```

Name	Size	Modified
..		
addition_example.csv	15 B	Sep 10, 2021, 1:20 PM
results.csv	47 B	Sep 10, 2021, 1:56 PM

Figure 2.1: Credit to Katrine Hagedorn

2.2.1 Working in the console

To many, the command driven interface looks a bit intimidating, however there are good reasons to stick to it.

First of all, a command driven interface is very flexible and it is often better suited for data manipulation and analysis tasks. A graphical user interface with similar functionality would be hard to navigate and harder to use. Just imagine the number of menus, sub menus, check boxes and radio buttons needed to run analysis provided by over 10.000 packages R provides. This does not mean, you can never use the buttons and menus of R, but on the long run, you should convert all the button-clicking to commands.

Second, a command driven interface allows scripting. In any serious data analysis task, reproducibility is a key concern.. A command based analysis environment makes replication of results easier, as it is far easier to record every detail of the configuration used in a script(=program).

I have many friends and colleagues who have been working in a hell of linked spreadsheets and a recipe on the side with tasks to do manually. In R, everything within the process can be collected in the script. We will go through the standard structure of your script in chapter 6.

Throughout this compendium you will see two types of code blocks. First group (exact representation may vary depending on media type) is blue text in light gray boxes. Those are R commands you can type in to the R console. Second group is the black text on white background, preceded by `## [1]`. Which is the R output of said command. Below is an example.

```r
# This is a comment, anything after a # sign won't be evaluated by R.
print("Hello Students")
```

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

Right now, do not worry too much about syntax. Just know that if you type in what you see in the gray box, R will evaluate that command and return an output.

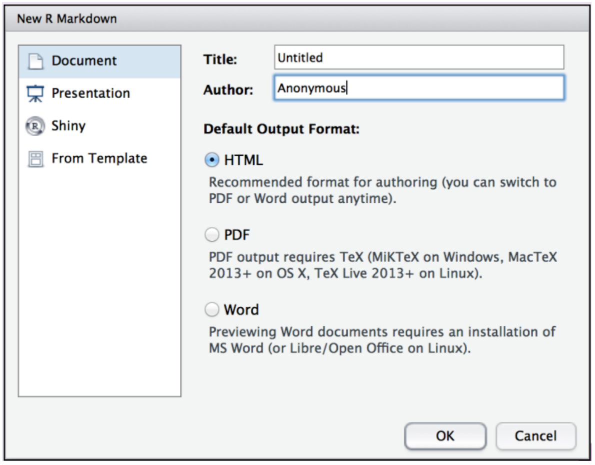
Having a basic knowledge of programming or command line interface may be helpful at this stage **but it is absolutely not necessary**. There may be times when you may get confused about what is going on. Note your question and keep going, most of your questions will answer themselves as you progress. For the remaining questions, you can always ask your instructors - or your fellow students - during exercise sessions.

2.3 R Markdown

You can always do with a simple script in R, called something like *script.R*. However, you can step one step up, and create your code in R Markdown. By doing this, you are creating your dynamic report while you are coding and it can easily be put into slides, PDF, word etc. This manuscript is also made in R Markdown.

A new feature in R Markdown is that you can shift between source code and a visual interface, the latter which might appeal to most people.

You create a new document by **File > New File > R Markdown** (Here it could also have been **R-Script**). Then you can chose to create a HTML, PDF or Word-file, and write title and author in it:



New R Markdown

- Document
- Presentation
- Shiny
- From Template

Title: Untitled

Author: Anonymous

Default Output Format:

● HTML

Recommended format for authoring (you can switch to PDF or Word output anytime).

○ PDF

PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

○ Word

Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

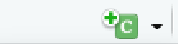
For your report, a Word file is nice, as you can use the program to do the final polishing and control of the size of charts, etc.

R Markdown consists of text, that you will have to mark (like html or latex, if you have worked with these formats). Most important tags are:

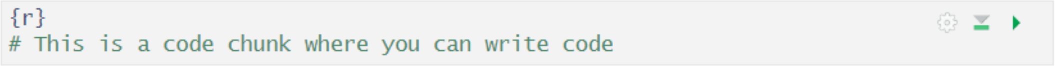
- `#` that is put before a header in the R Markdown text (but is used differently in R-cunks)
- `*` used in: `*italics*` and `**bold**` or in the beginning of a list. You use `+` for subitems.

You can even include embedded R-code in your document text by using ‘ r *code* ‘ around the code part.

In addition, you have your code-chunks in which you work within the R-language. You create them by clicking on the green C-button:



And you will end up having code like this:



```
{r}
# This is a code chunk where you can write code
```

2.3.1 Or use the script...

If it is too much with two new and different syntaxes, you can just work in a script file, called something like `script.R`, it is white and plain, and you only need to worry about the R-code.

2.4 Before you can start coding

Before we start with the basic programming concepts, you should learn about creating projects, reading files and reading in packages.

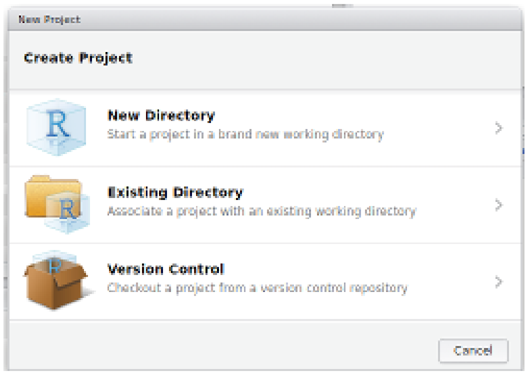
2.4.1 Creating a "project"

When you work with a new project, you can create a new project in R Studio (in the file menu). A project is a way to collect related data, analysis, documentation and code together in a tidy package. It makes your life MUCH easier, when you return to your project. Particularly if you are working with others in a common drive.

From the menu bar, create a new project for this module. Follow the steps outlined below:

1. Click File
2. Click New Project
3. In the new project window, select New Directory.
4. Select New Project.
5. Enter Directory Name (e.g. Module1).
6. Click create project.

Make sure you know where the project directory is. I recommend you place your project directory to a place you can easily access and remember (like <Course folder, Documents folder or Home folder).

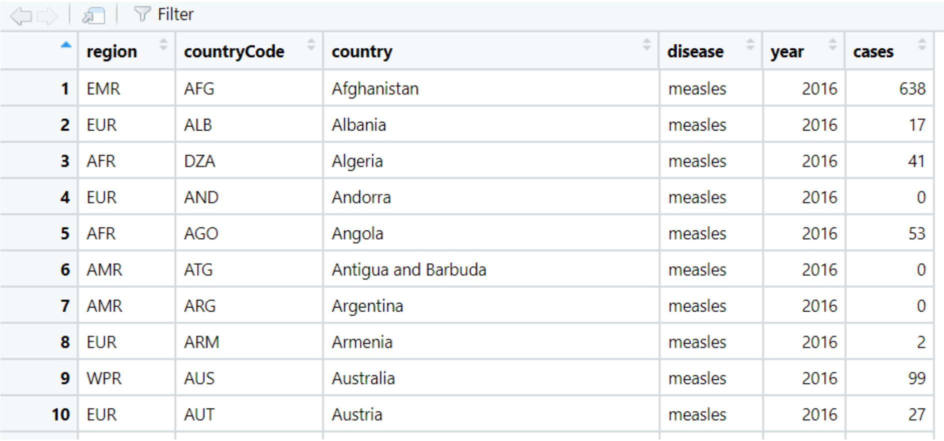

New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

Following these steps ensure we can keep building on the same work space over time.

2.4.2 Reading in files

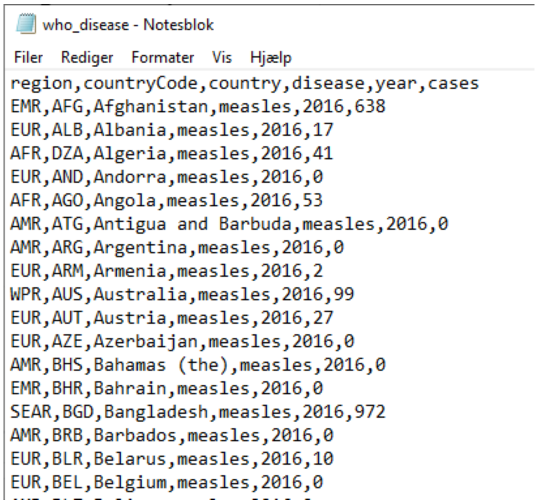
Of all the basic programming tasks I have introduced in my classes, reading files is the most time consuming topic. Therefore, I will give you a couple of options, and you can use the one that suits you the best.

The files below all refer to data-files, typically in tabular format, i.e:



	region	countryCode	country	disease	year	cases
1	EMR	AFG	Afghanistan	measles	2016	638
2	EUR	ALB	Albania	measles	2016	17
3	AFR	DZA	Algeria	measles	2016	41
4	EUR	AND	Andorra	measles	2016	0
5	AFR	AGO	Angola	measles	2016	53
6	AMR	ATG	Antigua and Barbuda	measles	2016	0
7	AMR	ARG	Argentina	measles	2016	0
8	EUR	ARM	Armenia	measles	2016	2
9	WPR	AUS	Australia	measles	2016	99
10	EUR	AUT	Austria	measles	2016	27

Sometimes when you open them in notepad, they look like this:



```
region,countryCode,country,disease,year,cases
EMR,AFG,Afghanistan,measles,2016,638
EUR,ALB,Albania,measles,2016,17
AFR,DZA,Algeria,measles,2016,41
EUR,AND,Andorra,measles,2016,0
AFR,AGO,Angola,measles,2016,53
AMR,ATG,Antigua and Barbuda,measles,2016,0
AMR,ARG,Argentina,measles,2016,0
EUR,ARM,Armenia,measles,2016,2
WPR,AUS,Australia,measles,2016,99
EUR,AUT,Austria,measles,2016,27
EUR,AZE,Azerbaijan,measles,2016,0
AMR,BHS,Bahamas (the),measles,2016,0
EMR,BHR,Bahrain,measles,2016,0
SEAR,BGD,Bangladesh,measles,2016,972
AMR,BRB,Barbados,measles,2016,0
EUR,BLR,Belarus,measles,2016,10
EUR,BEL,Belgium,measles,2016,0
```

It is typically because ther are csv-files or regular text-files. They are typically with a tab, ; or , between each column.

Typical file formats you might need to read in:

- file.txt (text format - often as tabular separated)
- file.csv (text format - columns separated by , or ;)
- file.xlsx (classical speadsheet format form Microsoft. Works suprisingly well in R)
- file.dat (STATA)
- file.sas7bdat (SAS)
- file.sav (SPSS)

You can either read them in using the "Import Dataset" in Environment pane, or you can try to figure out the right reading function and the right path.

I often start using the Import Dataset, and then I add the code in the beginning of my R-script or R markdown script.

And here is how they can be read in:

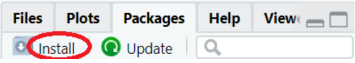
- `read_delim()` my go-to function for reading txt or csv-files. It can deal with tab-sepatation, semicolon, comma, or whatever the filemaker have used.

- `read_excel()` Needs the library `library(readxl)` to work.
- …

2.4.3 Packages

R is a language that has a lot of built in functions, and we call this "Base-R". However - over time, many have been working out different bundles of smart functions that makes your life as R-programmer much easier. These are also called packages.

To use a package you first need to install it, by using either the command: `install.packages("tidyverse")` (here the package tidyverse is installed), or you can use the install button in the lower right pane (No 4):



Before you can use your new package, you need to load it into your program, using `library(tidyverse)`.

Chapter 3

Basic concepts

In the last chapter, we mainly talked about the R interface. So you must be eager to get started with the code.

In this chapter, you will get an introduction to the foundations of programming in R. R is built very much for data handling and statistics, and therefore often files are read in as tables or data.frames. This is also what makes R a nice beginner language for people who are already used to work with spreadsheet programs such as Excel, Google sheet, etc.

We go through different types of data in this chapter as well as simple operations, similar to what you probably know how to do in a spreadsheet. This does not mean that everything is easy to grasp, but it is the building blocks for being able to perform and understand more advanced programming concepts in the next chapters.

These operations are very useful for working with all kinds of data - and particularly - for understanding your data.

3.1 Getting Your Feet Wet

We will use Area 1 (lower left pane) in R Studio, that is where R console resides.

R console is quite flexible. You can use it for a number of purposes. Below are some basic algebra examples that you might also know from your calculator or excel. They all output numeric values like 6 or 2.5 with a `## [1]` in front:

```
5 + 1 # Addition
```

```
## [1] 6
```

```
5 - 9 # Substraction
```

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

```
5 * 2 # Multiplication
```

```
## [1] 10
```

```r
5 / 2 # Division
```

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

```r
5^2 # Square
```

```
## [1] 25
```

```r
log(10) # Log
```

```
## [1] 2.302585
```

```r
sqrt(5) # Square root
```

```
## [1] 2.236068
```

You can also have logic statements with *relational operators*, that gives a `TRUE` or `FALSE` output instead of a number. That will prove very important in programming conditions later on:

```r
3 == 4 # is 3 equal to 4? Notice double equal signs
```

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

```r
3 != 4 # is 3 NOT equal to 4? The ! is used as "not" in logic statements
```

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

```r
3 < 4 # is 3 less than 4?
```

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

At this point you may be wondering about the `[1]` prepended to all the results. That is the index of the first item displayed in a row of results. Below I will ask R to output numbers from 1 to 30.

```r
1:30 # Sequential numbers from 1 to 30
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

As you can see, in the second row the index shifts to `[26]` indicating the 26th number in the series.

3.2 Data types

You have probably already noted that there are different data types. Basically there are numbers and non-numbers

Numbers

- Numeric data type is covering all types of numbered data types:
    - Integer. Any positive, whole numbers. You can declare a number an Integer by putting a L in the end, i.e. `2203L`.
    - Double. Any numbers, including negative numbers and decimal numbers.
- Dates is a special numeric data type which follows clock an calendar logic.

Non-numbers

- Characters. Any string, and can both contain test, numbers and a mix
- Factors. When data is categorical, i.e. has a limited number of instances such as colours, countries or sometimes survey responses. You can order factors, such that you can use them later for certain statistical tests.
- Logical. Contains the values: `TRUE`, `FALSE` or `NA`

3.2.0.1 `NULL`

There is a special object called NULL, meaning empty object. It is used whenever there is a need to indicate or specify that an object is absent. It should not be confused with a vector or list of zero length. To test for `NULL` use `is.null()`.

3.2.0.2 `NA`

`NA` is generally interpreted as a missing value, does not exist or "not available". You can test for missing values (`NA`) using `is.na()`.

3.3 Variables

If you want to save anything in the environment to use later; be it a scalar (single number), larger set of data, or results, you can use the assignment operator '<-'. Using '<-' will create a *variable* or object.

You can also use=for assigning, but the direction of the arrow, '<-', makes it more visually easy to interpret.

There are various variable naming conventions (camelCase, snake_case and so on) and you can use any one of them. I recommend using one consistently to prevent confusion. Most of my variable names will be snake_case and look like this: hi_there, your_variable, the_cool_variable...

```
A <- 5
```

Notice there is no output. R saved 5 into memory as A. and you can now see it in the Environment (the upper right pane, 3).

So A is a shortcut (variable) for 5. We can use it in ways displayed below.

```
A
```

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

```
A + 2
```

```
## [1] 7
```

```
A * 2
```

```
## [1] 10
```

Here it is time to point out a feature of R console. It is case sensitive.

```
"a" == "A"
```

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

If you tried to print a you would get an error message

```
# a
# gives:
# Error: object 'a' not found
```

3.4 Data Structures

The data does not just float around, but is always structured in R. As Excel structures data in spread sheets, R has different ways of structuring data. Above, data is a number in a variable, "A". But for realistic purposes, you will normally need more structure, for instance when you have read in a data set from a csv-file or similar.

3.4.1 Vectors

We can store a vector (array of numbers or text strings such as words) in a variable. The variable A from above, is basically just a vector of length 1, which we sometimes call a scalar.

Here I am saving numbers 1 to 10 in `B` of length 10.

```
B <- 1:10
```

The same vector can be made with the `c()` function.

```
B <- c(1,2,3,4,5,6,7,8,9,10)
```

Though the first notation is shorter and easier, it is smart to know the `c()` which can be used in all kind of vector creations. Note that `c()` is a function call to the *combine*-function.

You can carry out arithmetic with scalars (single number) and other vectors.

```
B
```

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

```
B * A
```

```
##  [1]  5 10 15 20 25 30 35 40 45 50
```

```
C <- B + A # let us save B+A into a third variable C
C
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15
```

A vector is a data structure that only takes one data type. So if you mix numbers with words, all elements will be perceived as characters. For instance this vector will end up with all elements being characters, indicated by the “ ” in output:

```
M <- c(1,2,3,"hello", FALSE)
M
```

```
## [1] "1"     "2"     "3"     "hello" "FALSE"
```

A bit like if you mix dry stuff such as flour with wet stuff such as milk in a bowl, you end up with everything being wet.

If you need an array of different data types, you should use a *list*, which is explained later.

3.4.2 Indexes

Nice thing about vectors (and data.frames as you will later explore) is that you can access variables in a vector through indexes. We know C has numbers from 6 to 15. If I want to access the 3rd element of the vector I can do this as follows:

```
C[3]
```

```
## [1] 8
```

Here the number between the square braces “[]” is the index. It tells R to extract only the third element.

You can also refer to more than one element at once. Below I refer to elements from second to fourth.

```
C[2:4]
```

```
## [1] 7 8 9
```

Similarly, we can refer to just 3rd and 5th elements, using our `c()` function.

```
C[c(3, 5)]
```

```
## [1]  8 10
```

You can use logic operators to refer to specific elements. This will come in handy later when you are trying to extract all information before a specific date or all transactions of a customer.

We can see if elements of C are smaller than 10.

```
C < 10
```

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

We can use this information to extract only elements of C less than 10. Pay heed here, as we will use this in some subsetting scenarios.

```
C[C < 10]
```

```
## [1] 6 7 8 9
```

3.4.3 Data frames

In R, `data.frame` is the typical storage of tabular data. You can think of it as a matrix (two dimensional data structure) that allows different types of variables to be combined. Below I am generating a data.frame. Do not worry too much about how it works at this stage. The important part is how we will refer to elements of the data.frame.

```
letters <- sample(LETTERS[1:3], 10, replace = TRUE)
D <- data.frame(1, 1:10, letters)
D
```

```
##    X1 X1.10 letters
## 1   1     1       B
## 2   1     2       A
## 3   1     3       B
## 4   1     4       A
## 5   1     5       C
## 6   1     6       B
## 7   1     7       A
## 8   1     8       B
## 9   1     9       C
## 10  1    10       C
```

As you can see, the data.frame has three variables (X1, X1.10, and letters) in columns and ten observations of each in rows. This is a two dimensional data structure. Meaning we can refer to rows and columns.

Each column can only have one data type, but the different columns can have different data types, as here, where X1 and X1.10 contains numeric data types, and letter contain character data types.

3.4.4 Subsetting

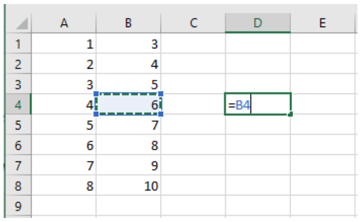
Often you want to look at part of the data.frame. This is called subsetting, and there are many smart operations to help you with this.

Let us get the first row in the third column. **Notice the ',' between the square brackets. The part before the comma refers to rows, and the part after refers to columns.**

```
# the syntax is "dataframe[row,column]"
D[1,3]
```

```
## [1] "B"
```

If you are used to spread sheets, you might notice that this index corresponds to when you want to point at a cell using e.g. B4:



	A	B	C	D	E
1	1	3			
2	2	4			
3	3	5			
4	4	6		=B4	
5	5	7			
6	6	8			
7	7	9			
8	8	10			
9					

Back to R! Let us get the first three rows.

```
D[1:3, ]
```

```
##   X1 X1.10 letters
## 1  1     1       B
## 2  1     2       A
## 3  1     3       B
```

Now let us get just the second and third column.

```
D[, 2:3]
```

```
##    X1.10 letters
## 1      1       B
## 2      2       A
## 3      3       B
## 4      4       A
## 5      5       C
## 6      6       B
## 7      7       A
## 8      8       B
## 9      9       C
## 10    10       C
```

You can refer to rows and columns at the same time.

```
D[1:3, 2:3]
```

```
##   X1.10 letters
## 1     1       B
## 2     2       A
## 3     3       B
```

You can use column names to refer to columns.

```
D[, "letters"]
```

```
##  [1] "B" "A" "B" "A" "C" "B" "A" "B" "C" "C"
```

```
# OR Alternately note the use of $ operator
D$letters
```

```
##  [1] "B" "A" "B" "A" "C" "B" "A" "B" "C" "C"
```

Notice, that what we see above is actually a vector!

Now think about a situation, where the third column (letters) is the user ID and you want all transactions of a specific user. We can use the logic operators we learned about earlier to obtain just that. Let us extract the observations relating to user C .

```
# This translates to "From D, select all rows where column letters equals 'C'".
D[D$letters == "C", ]
```

```
##    X1 X1.10 letters
## 5   1     5       C
## 9   1     9       C
## 10  1    10       C
```

I want to emphasize this example once more. This is important as it will be used often in the future to filter data.frames. Let us pay attention to what is going on by running parts of the code in isolation.

We are passing a logical operation `(D$letters=='C')` to row index. Let us see what the logical operation produces

```
D$letters
```

```
##  [1] "B" "A" "B" "A" "C" "B" "A" "B" "C" "C"
```

```
D$letters == "C"
```

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

It produces a series of TRUE/FALSE values. Compare output of `D$letters` and `D$letters=='C'`, the series returns TRUE when value in letters is C. When this series is fed into the row index, it will spit out the rows where the value of the series is TRUE.

3.4.5 Other types of data structures that we will meet

You have now learned about scalars (single numbers), vectors (a sequence of data) and data.frame (a table). You can do almost everything with just these three (or actually two) data types

There are other classes, that I will present briefly:

- List. Like a vector, but with the option of having mixed data types: `L <- list(1,"hello", FALSE)`. This can be nice. as all data types will not be overrun by character in this case:

```
L <- list(1,"hello", "world")
L > 0 # here we see that the first element is evaluated, and characters are not
```

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

- Matrix. Like a data.frame, but with only one datatype in all of the table (just like a vector). In some arithmetic calculations, a matrix is better used for speed of the operation. `M <- matrix(c(0,3,3,0), nrow=2)`
- Tibble. Like a data.frame but even more flexible and used in the popular *tidyverse* package. Can also contain list of lists.

You can try out these different data structures.

3.5 Inspecting data

In general you can find whether your variable is a vector, a list or data.frame data type by using the `class()`-command. By using `str()`, you can find out both data type and what type the content of your variable is:

```
str(D)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ X1      : num  1 1 1 1 1 1 1 1 1 1
##  $ X1.10   : int  1 2 3 4 5 6 7 8 9 10
##  $ letters: chr  "B" "A" "B" "A" ...
```

You can find the length of a vector using `length()`, and you can find the number of rows and columns by using `dim()`.

```
length(C)
```

```
## [1] 10
```

```
dim(D)
```

```
## [1] 10  3
```

You already know that you can inspect your variable by just writing the name. But you can also see the first 6 or the last 6 elements by using `head()` and `tail()`

```
head(D)
```

```
##   X1 X1.10 letters
## 1  1     1       B
## 2  1     2       A
## 3  1     3       B
## 4  1     4       A
## 5  1     5       C
## 6  1     6       B
```

```
tail(D)
```

```
##    X1 X1.10 letters
## 5   1     5       C
## 6   1     6       B
## 7   1     7       A
## 8   1     8       B
## 9   1     9       C
## 10  1    10       C
```

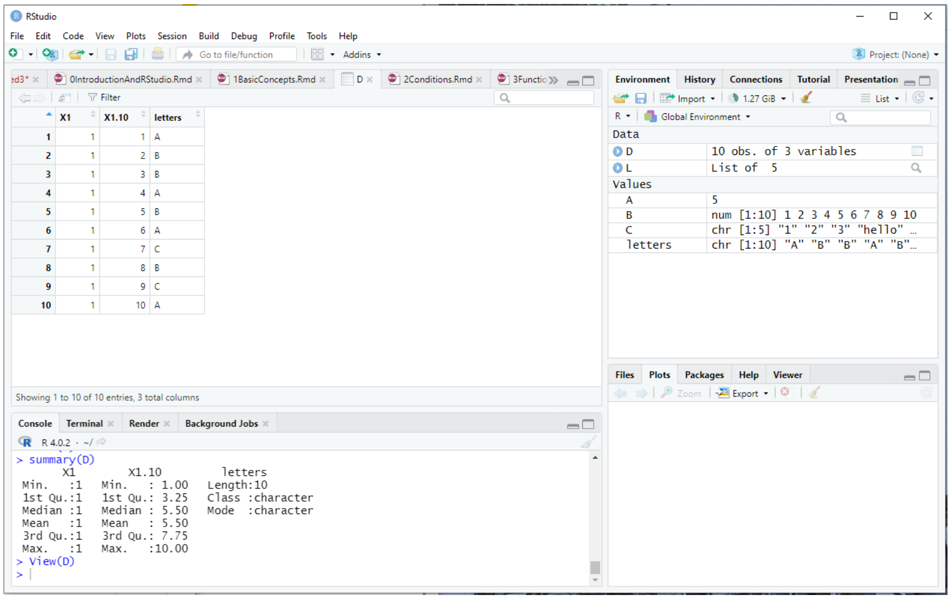
Finally, you can get an overview of the content by using `summary()`. Notice the different format of columns with numeric and character data types, respectively.

```
summary(D)
```

```
##        X1        X1.10         letters
##  Min.   :1   Min.   : 1.00   Length:10
##  1st Qu.:1   1st Qu.: 3.25   Class :character
##  Median :1   Median : 5.50   Mode  :character
##  Mean   :1   Mean   : 5.50
##  3rd Qu.:1   3rd Qu.: 7.75
##  Max.   :1   Max.   :10.00
```

Finally, you can use the `View()` command, which will show the table in the upper left pane (where the script are) similar to that of a spread sheet. In this table you can scroll and sort by the different column, while you are running more code and it is not “hidden” in the Console.

```
View(D)
```



RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
ed3*
0IntroductionAndRStudio.Rmd
1BasicConcepts.Rmd
D
2Conditions.Rmd
3Functio
Filter
Environment History Connections Tutorial Presentation
Import
1.27 GiB
List
R
Global Environment
Files Plots Packages Help Viewer
Zoom
Export
Console Terminal Render Background Jobs
R 4.0.2 · ~/

	X1	X1.10	letters
1	1	1	A
2	1	2	B
3	1	3	B
4	1	4	A
5	1	5	B
6	1	6	A
7	1	7	C
8	1	8	B
9	1	9	C
10	1	10	A

Showing 1 to 10 of 10 entries, 3 total columns

Data	
D	10 obs. of 3 variables
L	List of 5
Values	
A	5
B	num [1:10] 1 2 3 4 5 6 7 8 9 10
C	chr [1:5] "1" "2" "3" "hello" …
letters	chr [1:10] "A" "B" "B" "A" "B"…

```
> summary(D)
       X1         X1.10          letters
 Min.   :1   Min.   : 1.00   Length:10
 1st Qu.:1   1st Qu.: 3.25   Class :character
 Median :1   Median : 5.50   Mode  :character
 Mean   :1   Mean   : 5.50
 3rd Qu.:1   3rd Qu.: 7.75
 Max.   :1   Max.   :10.00
> View(D)
> 
```

Chapter 4

Conditions

You have now learned about the basic data types, variables and operators - both arithmetic operators and logic statements.

And logic is one of the core bones of how a computer works. It does not understand pragmatic messages such as *Hmm - the grass is a little high now.* If we want the lawnmower robot to cut the grass we need to say it explicitly - and we need to set logic boundaries. *If the grass is above 5 cm high, then cut it.* Maybe we want to extend the sentence: *If the grass is above 5 cm high AND it is not raining, then cut it. Else, stay in the station.*

4.1 What is a condition

What we need to make a program that can handle these kinds of sentences are both the logical operators and a way to formulate *if*, *then* and *else*. This is called conditions, and are present in all modern programming languages. Languages as HTML does not contain these, while e.g. Excel has the option in a clumsy and very slow implementation.

4.2 How to run a condition

An `if`-statement allows you to conditionally execute code. It looks like this:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

You don't need the `else`-statement for the expression to work. So if it is only *if-then* you want to express, you simply omit the else.

A simple example is the following where we test if x is larger than 10. As we learned earlier `x > 10` is the condition formulated as a logic statement.

```
x <-9
if (x > 10) {
  print("x is larger than 10")
} else {
  print("x is NOT larger than 10")
}
```

```
## [1] "x is NOT larger than 10"
```

The `condition` must evaluate to either `TRUE` or `FALSE` (boolean values). In the above example, the condition is evaluation a logic statement. You can also use variables that have contains a boolean value. Note that we can both write `TRUE` and `T` or `FALSE` and `F`, and that we do not use `else` here.

```
you_are_happy <- T
you_know_it <- T

if (you_are_happy && you_know_it) {
  "clap your hands"
}
```

```
## [1] "clap your hands"
```

4.3 Long conditions

What you saw above was the logic (or boolean) operator `&&`. These are used in long conditions, which we will often need, as seen in the example of the robot lawnmower.

4.3.1 Boolean Logic operators

We start with the concept of boolean (logical) operators like `&&` (AND) and `!` (NOT), which can be a bit hard to grasp, when you have mainly used operators such as `+`,`-`, etc. They can be used to connect two different conditions, as above *1) the grass is above 5 cm high AND 2) it is not raining.*

Here are some logic operators. It is a good idea to test them in with different values to understand how they work:

- `!` means NOT (a logical negation)
- `&&` means AND (both conditions needs to be true for the output to be true). The same does `&`, but it has different output when working on a vector of objects.
- `||` means OR (either one, the other or both conditions should be true for the output to be true). The same does `|`, but it has different output when working on a vector of objects.

An illustration of the difference can be seen here:

```
y <- c(1,2,3)
y > 1 & y < 3 # each element in the vector is checked and a boolean value for each is returned
```

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

```
y > 1 && y < 3 # gives an error
```

Using && will work on atomic logical conditions

```
length(y)==3 && y[1]<2 # returns TRUE as both conditions are true
```

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

4.3.2 Example of long conditions

Now we have the building blocks and can in principle build never ending conditions. To be able to read the code later, it is normally not feasible to include too many logical operators in one if-statement. Here we use our old variables, recalling that `you_are_happy` and `you_know_it` are both `TRUE` and `x` is `9`, and `F` is short for `FALSE`:

```
if ((you_are_happy && you_know_it || x > 10) && F) {
  "clap your hands"
}
```

Although first part of the condition is true, the last element, `F` combined with the `&&`-operator results in a `FALSE` statement and we will not clap our hands.

4.3.3 Multiple conditions using nesting

You can also expand your conditions by nesting your if-statements. If statements can be nested to handle multiple conditions:

```
if (this) {
  # do something
} else {
  if (that) {
    # do something else
  } else {
    # now do something completely different
  }
}
```

Including our conditions from before:

```
if (!(you_are_happy && you_know_it)) {
  print("Dont clap!")
} else {
  if (x > 10) {
    print("x is larger than 10")
  } else {
    print("clap you hands")
  }
}
```

Although this is maybe a bit spectacular example, it includes different ways of ending in the same, namely the boolean values `TRUE` or `FALSE`.

4.4 Other ways to write conditions

There are other ways of writing conditions in R.

```r
ifelse(condition, yes, no)
```

If the the first argument (condition) returns true, then the second argument (yes) is returned, and if not, the third argument is returned (no).

Here is the hand clapping example:

```r
ifelse(you_are_happy && you_know_it, "clap your hands", "Dont clap your hands")
```

```
## [1] "clap your hands"
```

And with a negation included:

```r
ifelse(you_are_happy && !you_know_it, "clap your hands", "Dont clap your hands")
```

```
## [1] "Dont clap your hands"
```

These can also be nested:

```r
ifelse(you_are_happy && you_know_it, "clap your hands", ifelse(x>10, "x is large", "x is small"))
```

```
## [1] "clap your hands"
```

Chapter 5

Functions

So now you know about assignment operators (<-, =), logical operators (==, <, >, …), data types, and variables with data structures such as vectors, lists and data.frames. Learning about functions will enable us to explore more exciting dimensions of analytics with R.

5.1 What is a function and how do we access it?

A function is a set of instructions to be executed by the computer. This is how you tell the computer what to do.

To make a silly, but illustrative example: When you build a sandwich, you need ingredients and you need to actively build the thing before you can eat it. The ingredients are the *input* and the burger is the *output*, and the build-*function*, basically takes ingredients and build them together in a sensible order.

A basic R function is just a word followed by *parentheses*. Between the brackets are the *input* (or arguments or parameters) of the function. The function below prints R license information to the screen. This is the *output* of the `license()`-function. It takes no parameters.

```r
license() #Source of the license
```

```
##
## This software is distributed under the terms of the GNU General
## Public License, either Version 2, June 1991 or Version 3, June 2007.
## The terms of version 2 of the license are in a file called COPYING
## which you should have received with
## this software and which can be displayed by RShowDoc("COPYING").
## Version 3 of the license can be displayed by RShowDoc("GPL-3").
##
## Copies of both versions 2 and 3 of the license can be found
## at https://www.R-project.org/Licenses/.
##
## A small number of files (the API header files listed in
## R_DOC_DIR/COPYRIGHTS) are distributed under the
## LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later.
## This can be displayed by RShowDoc("LGPL-2.1"),
## or obtained at the URI given.
## Version 3 of the license can be displayed by RShowDoc("LGPL-3").
##
## 'Share and Enjoy.'
```

If you want to see the source code (the code that creates the function) of any function, just type its name without the braces. This way you can see how existing functions work, and hopefully later debug, and even develop your own functions. Here is one of the functions you used to read in files in the first chapter.

```
read.delim
```

```
## function (file, header = TRUE, sep = "\t", quote = "\"", dec = ".",
##     fill = TRUE, comment.char = "", ...)
## read.table(file = file, header = header, sep = sep, quote = quote,
##     dec = dec, fill = fill, comment.char = comment.char, ...)
## <bytecode: 0x0000019d26654db8>
## <environment: namespace:utils>
```

Some functions (the most useful ones) take *arguments* (what we earlier called input).

Here I am asking R to calculate the mean of numbers in vector C. C serves as an argument to tell the mean function over which numbers to calculate the mean, and it is here created by the function `c()`. Note that this functions consist of a small c!

```
C <- c(6,7,8,9,10,11,12,13,14,15)
mean(C)
```

```
## [1] 10.5
```

You can directly use the output of one function with another. Here I will calculate the mean by first summing the C vector then dividing the sum by number of elements, using functions `sum()` and `length()`.

```
sum(C) / length(C)
```

```
## [1] 10.5
```

Functions can take more than one parameter. Here I am instructing R to calculate the mean, but also removing NA-values or NULL-values, that often occur in real-life data.

```
C <- c(6,7,8,9,10,11,12,13,14,15,NA)
mean(C)
```

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

```
mean(C, na.rm=T) # to avoid NA simply because there is a single missing data point.
```

```
## [1] 10.5
```

If you want to learn what a function does, or what parameters are available, use help function.

```
help(mean) # It will open the topic in the right lower pane. You can try it out!
```

5.2 Create your own function

There are many useful functions already made in R. Both those build into R (Base R) and those that are made for you to use when you need them, using a `library()`-call.

However, sometimes you might need to do the same thing in your script several times, and then it might be convenient to make your own function.

You can create your own functions by saving them in a variable. It will not turn blue, as the build in function `return()` below, but it is basically the same species. Below I declare a hello function that takes a string, or number as input and prints out a hello message as output.

I have used return() to return the output from the function, e.g. `return(x)`. You could also use `print(x)` or just `x`.

```
# Declare function
hello <- function(x){
  return( paste("Hello ", x) )
  }
# Use function
hello("you!")
```

```
## [1] "Hello  you!"
```

```
# View source
hello
```

```
## function(x){
##   return( paste("Hello ", x) )
##   }
```

Here is a function that is a bit more complicated.

I declare two parameters with the *default value* of the second parameter set to 1. If you add only one argument (here “World”), it puts in this parameter into x. But if you add to arguments, the other one will be y, and will denote the number of times “Hello” “World” will be printed. (We have not learned about the `for()` function yet, so don’t worry if you do not fully understand the contents of the function.)

```
hello <- function(x, y = 1){
  for(i in 1:y){
    print(paste("Hello ", x, i))
    }
}

hello("World")
```

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

```
hello("World", 3)
```

```
## [1] "Hello  World 1"
## [1] "Hello  World 2"
## [1] "Hello  World 3"
```

Note that here we used `print()` to print the output. If we had used `return()`, only one "Hello World" would have been printed, as return also terminate or ends the function after printing.

```
hello <- function(x, y = 1){
  for(i in 1:y){
    return(paste("Hello ", x, i))
    }
}

hello("World", 3)
```

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

5.3 Brackets

So far we have used different brackets without calling too much attention to them. But now we have been trying them all, so here they are explained a bit more in detail.

In R we use different brackets

- () Round brackets/Parentheses
- {} Curly brackets
- [] Square brackets

5.3.1 () is for grouping and function-arguments

`()` is used to group as in math, such that for example the below becomes 60 and not 510!

```
(500+100)/10
```

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

It is also used to enclose arguments to a function. E.g. the following function, in which the arguments are a string a start-position and a stop position within the string:

```
substr("hello - how are you doing",1, 5)
```

```
## [1] "hello"
```

(You will learn more about these kinds of functions when we talk about regular expressions.)

5.3.2 {} - as "then" in conditions

{} is used in the executing part of conditions, loops and functions. It can be read as "then do:". For instance, the conditional expression below can be read: *If* 10 is larger than 3, *then do* print "Hello World".

```
if(10>3){
  print("Hello World")
}
```

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

5.3.3 [] - to capture element index

[] is used to capture an index-element from e.g. a vector, list, data frame, etc. We also learned about indeces in the introduction to the basics, but here they are again:

```
my_vector[1] # The first item
my_dataframe[1, ] # The first row in the dataframe
```

With double brackets [[]] any element names are not displayed - else it is the same.

Chapter 6

Repititions - loops and apply()

Quite often in data science you need to do the same thing multiple times.

Maybe you need to do the same operation on every row in the data set. Maybe you need to do a special operation on all the individuals with the label "A"?

Then you might not

6.1 Loops

A loop in programming is a way to do the same operation mutiple times on for instance a list, vector or a data.frame.

6.1.1 for-loops in R

R is mainly build for using functions rather than loops. E.g. the `apply()` functions ar made for avoiding too much looping. However, loops can be quite useful, and are used a lot in other programming languages.

```
for (element in sequence){
  # Do some stuff
}
```

Here is how you create a very simple loop in R. It is looping over all elements in `my_vector` and printing each element.

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

for(i in x){
  print(i)
  }
```

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

We often use i as a counter or an index. The above reads, *for each element i in the sequence x, print the element.*

You can also loop over a sequence of numbers, here 1-3, and you can output, e.g. the squareroot

```
for(i in 1:3){
  print(sqrt(i))
  }
```

```
## [1] 1
## [1] 1.414214
## [1] 1.732051
```

Quite often, since R-objects are often tabular or vectorised, you can simply add the function directly to the vector:

```
x <- 1:10 # Create a vector of numbers from 1-10
sqrt(x) # take the square root of each of them
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

We cal the `i` *index* and sometimes the *counter*. And it can be used withing the squared brackets [], if you loop over a data frame. In principle it could be called anything, but it is a nice convention to know that index counting is done using i!

Recall:

```
letters <- sample(LETTERS[1:3], 10, replace = TRUE)
D <- data.frame(1, 1:10, letters)
D
```

```
##    X1 X1.10 letters
## 1   1     1       C
## 2   1     2       B
## 3   1     3       C
## 4   1     4       A
## 5   1     5       A
## 6   1     6       B
## 7   1     7       A
## 8   1     8       A
## 9   1     9       B
## 10  1    10       C
```

We can easily perform a simple operation. Here we create a new column called NewCol and take the squareroot of the column X1.10 and put it into our new column:

```
D$NewCol <- sqrt(D$X1.10)
head(D)
```

```
##   X1 X1.10 letters   NewCol
## 1  1     1       C 1.000000
## 2  1     2       B 1.414214
## 3  1     3       C 1.732051
## 4  1     4       A 2.000000
## 5  1     5       A 2.236068
## 6  1     6       B 2.449490
```

However, sometimes we have tasks that are more difficult. If we for instance want to insert the number times 10 instead of square root, in the rows with the letter “A”, we might need something more sophisticated. Here we use both loop, and the index-brackets using i as index:

```
for(i in 1:nrow(D)){
  if(D[i,"letters"]=="A"){
    D[i,"NewCol"] <- D[i,"X1.10"]*10
  }
}
head(D)
```

```
##   X1 X1.10 letters    NewCol
## 1  1     1       C  1.000000
## 2  1     2       B  1.414214
## 3  1     3       C  1.732051
## 4  1     4       A 40.000000
## 5  1     5       A 50.000000
## 6  1     6       B  2.449490
```

this reads *for each row of the dataframe D, if the value of the letter-column is A, then times 10 with the value in the X1.10-column.*

6.1.2 ‘while- loops

There is another type of loop, the `while`-loop, that is not dependent on you knowing the size of the data ahead. Instead it repeats as long as some test-condition is true.

In R, you will always know the size, as you read it in at once, but the `while` loop can be quite useful in other languages, where you might read in data one line at a time.

It is called the `while`-loop, and looks like this:

```
while (test_expression){
  # Do stuff
  # Count up
}
```

Here is an example to show the difference, and where the while loop is slightly shorter:

```
day_of_week <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
```

The `for`-loop as we know it:

```
for (i in day_of_week){ # the counter is build in
  if (i != "Friday"){
    print("Work")
  } else{
    print("Weekend!")
  }
}
```

```
## [1] "Work"
## [1] "Work"
## [1] "Work"
## [1] "Work"
## [1] "Weekend!"
```

A `while`-loop:

```
i <- 1  # Here you create the counter
while (day_of_week[i] != "Friday"){
  print("Work")
  i = i + 1  # here you count one up
}
```

```
## [1] "Work"
## [1] "Work"
## [1] "Work"
## [1] "Work"
```

```
print("Weekend")
```

```
## [1] "Weekend"
```

If we included “Saturday” and “Sunday”, the `for`-loop would go on, while the `while`-loop already stopped at “Friday. So the two loops are not exactly the same.

6.2 The apply() family

As I mentioned in the beginning, R is in its DNA function based. For repetitions, you can write a version of the code above using apply-family, which sometimes runs faster, and is considered more elegant by many R-users.

We start with a simple function `sapply()` which works on a list, vector or data frame. It takes as argument:

`sapply(X, FUN, ...)`

where

- `X` is the input vector or list
- `FUN` is the function you apply
- `...` is eventual extra variables for the function (look it up in the "Help-pane).

`lapply()` is the same structure, but gives a list as output.

I have made an example with some apply-functions based on the `while`-loop above

```
day_of_week <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

is_it_weekend <- function(X){
  if(X=="Friday"){
    return("Weekend")
  }else{
    return("Work")
  }
}

sapply(day_of_week, is_it_weekend)
```

```
##    Monday   Tuesday Wednesday  Thursday    Friday
##    "Work"    "Work"    "Work"    "Work" "Weekend"
```

```
lapply(day_of_week, is_it_weekend) # lapply is like sapply, but output has list structure
```

```
## [[1]]
## [1] "Work"
##
## [[2]]
## [1] "Work"
##
## [[3]]
## [1] "Work"
##
## [[4]]
## [1] "Work"
##
## [[5]]
## [1] "Weekend"
```

Maybe you want more columns as arguments from the same data frame, like the `for`-loop we looked at before. We would then use `mapply()`.

```
mapply(FUN, X, ...)
```

where * FUN is the function you apply * ARG is a vector or a list which is also the first argument of the function * … is eventual extra arguments for the function

Then we might need to use `mapply()` on the homemade function `times()` that takes two arguments:

```
times <- function(X,Y){
  if(X=="A"){
    return(Y*10)
  }else{
    return(sqrt(Y))
  }
}

D$NewCol2 <- mapply(times, X=D$letters, Y=D$X1.10)
head(D)
```

```
##   X1 X1.10 letters    NewCol   NewCol2
## 1  1     1       C  1.000000  1.000000
## 2  1     2       B  1.414214  1.414214
## 3  1     3       C  1.732051  1.732051
## 4  1     4       A 40.000000 40.000000
## 5  1     5       A 50.000000 50.000000
## 6  1     6       B  2.449490  2.449490
```

It can also be used as the single-argument `is_it_weekend()` function from before:

```
mapply(is_it_weekend, day_of_week) # Notice: in mapply, the function is first
```

```
##    Monday   Tuesday Wednesday  Thursday    Friday
##    "Work"    "Work"    "Work"    "Work" "Weekend"
```

Chapter 7

Strings and regular expressions

Some times you might need to do something with strings, such as sentences or other kind of text.

Regular expressions is about matching text and it is almost a language in itself. You might not need a whole lot of the functions from this section in your everyday life with R, but a few string handles can be quite convenient to have on your backbone. The rest, you can use as a reference work when you need it.

In section 1 you learned that characters are stored as strings. Contrary to numbers, which you can easily build conditions around, strings are more messy and harder to access. But of cause R can handle strings as well.

The notion regular expression is often called regex, and roughly means "pattern which can be explained symbolic or directly".

7.1 Use cases

Regular expressions are widely popular within bioinformatics, language technology and other types of information areas in which you have sequences of text that you want to get meaning from. However, even in basic data wrangle that you will work with, you will often need some kind of simple or more sophisticated string-modification in Data cleaning and filtering.

For instance,

- You might need to extract the first 5 letters
- You might want to put together a full-name string from two columns firstName and lastName
- You might want to split a full name?
- You might need to find all data-rows in your data.frame in which the name-column contains "Johnson"

You can also use it in programs for other, more sophisticated tasks. These could be:

- Validation of emails, Date, Password etc.
- Web scraping
- getting information out of "codes"

7.2 Basic string handling

Here we start with the basic string handling methods, that I can almost garantee you will meet in your project or in your later life as data analyst.

- Concatenating strings
- Splitting strings
- Substituting strings
- Simple matching

In the section about the grammar of regular expression and examples, you will get some tools that are a bit more advanced. But these four principles will make a good beginning.

7.2.1 Concatenation - gluing text together

It is often usefull to be able to concatenating (gluing) strings in R. For this we have three functions that are build into R, and that you will see often, `paste()`, `paste0()` and `cat()`

For instance, we can look at three different ways to connect `shout` and `name`

```
shout <- "Hello"
name <- "Sine"
```

The `paste()` functions concatenate strings into one string:

```
paste(shout, name) # introduces a space in the concatenation
```

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

```
paste(shout, name, sep = ";") # you can also use something else as seperator,
```

```
## [1] "Hello;Sine"
```

```
paste0(shout, name) # glues the two strings directly together without space
```

```
## [1] "HelloSine"
```

The `cat()` function is more similar to the `print()`-function you already have used, but it also takes newline “\n” as a potential separator:

```
cat(shout, name)
```

```
## Hello Sine
```

```
cat(shout, name, sep = "\n")
```

```
## Hello
## Sine
```

7.2.2 Splitting strings into smaller parts

Maybe, you want to split a string onto more objects - the opposite of concatenation, you could say. For this we use the command `str_split()` from the `tidyverse` package.

```
library(tidyverse)
```

```
## Warning: pakke 'tidyverse' blev bygget under R version 4.3.2
```

We here try to split a full name into two different objects separated by ” “:

```
full_name <- "John Doe"
full_name_two <- str_split(full_name, " ", simplify=TRUE)
full_name_two[1] # We now look at the first element of the string
```

```
## [1] "John"
```

We use the `simplify=TRUE` to create a vector from the splitting instead of a list of lists from which you would need `full_name_two[[1]][1]` to get the first element (John) out of the variable *full_name_two*..

7.2.3 Substituting strings - “search and replace”

If we want to replace something, like when you do search and replace in texts, you have the functions `sub()` and `gsub()`.

You have the options to either replace the first element or all elements. The easiest way is to use `sub()` (only first element) or `gsub()`(all elements, *g* stands for global).

Input is:

- `sub(pattern, replacement, string/vector)`

For example:

```
full_name <- "John Johnson Doe"
name1 <- sub("John", "Sue", full_name)
name1 # Only the first "John" is replaced
```

```
## [1] "Sue Johnson Doe"
```

```
# Or a global option
name2 <- gsub("John", "Sue", full_name)
name2 # Now all "John"s are replaced in the string
```

```
## [1] "Sue Sueson Doe"
```

The function `grep()` is also used widely for finding matches. It is used to match a pattern or word:

```
full_name <- "John Johnson Doe"
grep("John", full_name)          # Returning index-position in vector
```

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

```
grep("John", full_name, value=T) # Returning the matching name
```

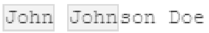
```
## [1] "John Johnson Doe"
```

```
grepl("John", full_name)         # Logic grep, returning TRUE/FALSE
```

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

Finally, a tool that can be useful for looking at parns - both simple and the more advanced that you will see in a minute is `str_view(x, pattern)`

```
library(htmlwidgets)
library(htmltools)

str_view_all(full_name, "John")
```


John Johnson Doe

7.3 Other string handles

What you have seen so far are some very basic string handles. There are examples of other string handles from the `tidyverse` package (just like `str_view()`):

- `str_length()` gives the length of the string(s)
- `str_sub(x, startpos, stop)` take the part of the string from a start position to a stop position
- `str_sort(x)` orders the strings in the list/vector alphabetical or numeric
- `str_extract(x, "regex")` returns only the part of the string that matches the regex

Recall that regex roughly means "pattern which can be explained symbolic or directly".

7.4 The grammar of regular expression

As R has functions as well as different data types and structures as it's grammar, so does regular expression. We will first present the elements og the grammar and then give some examples.

You can find most of the tools in the stringr-cheatsheet

7.4.1 Elements in matching grammar

Here are some of the most basic expressions strings.

Expr.	Meaning	Expr.	Meaning
abc	A sequence of characters	\\s	Whitespace character (space, tab, newline…)
[abc]	Any character from a set of characters	\\d	Any digit
[^abc]	Any character *not* in a set of characters	\\w	An alphanumeric character
[0-9]	Any number in a range of number	\\t	Tabulator
a| b| c	Any one of several patterns(|=OR)	\\S	Any non-whitespace character
(abc)	A group	\\D	Any non-digit character
.	Any character except for newline	\\W	A non-alphanumeric character

7.4.2 Anchors and Repititions

Expr.	Meaning	Expr.	Meaning
+	Element before repeated once or more times	^	Start of input (anchor)
*	Element before repeated zero or more times	$	End of input (anchor)
?	Element before occur zero times or one time. Nongreedy	(?<=a)x	Anything (x) preceeded by a
{2,4}	Two to four repetitions	x(?=a)	Anything (x) followed by a
{3}	Exactely three repetitions		

7.5 Examples using the grammar of regular expression

Here are a few examples to explain regular expressions. We use some very basic examples.

7.5.1 Test and match using symbolic patterns

We now go beyond the John Doe example above.

We start with creating a vector `x`, and a pattern,looking for digits in our vector:

```
x <- c("John", "Doe", "40 years", "30", "#SuperDad")

grep("\\d",x) # Notice we need to make a "\" before the pattern element
```

```
## [1] 3 4
```

Element 3 and 4 contained a digit, and therefore, element 3 turned out as a hit.

If you want to work with it in a dataframe or a vector you can utilize the index. This will become very usefull later on:

```
x[grep("\\d",x)]
```

```
## [1] "40 years" "30"
```

Maybe you want something that ends with a digit, using the `$`-anchor? Notice we now use `value=TRUE` such that we can easier check the match.

```
grep("\\d$",x, value=TRUE)
```

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

Or something that *starts* with a digit and *ends* with a non-digit character?

```
grep("^\\d.*?\\D$", x, value=TRUE)
```

```
## [1] "40 years"
```

You can also grep all that start with some letter or number (\w) or if it does NOT start with a letter or number

```
grep("^\\w", x, value=TRUE) # start with some letter?
```

```
## [1] "John"     "Doe"      "40 years" "30"
```

```
grep("^\\W", x, value=TRUE) # Start wih a non-letter
```

```
## [1] "#SuperDad"
```

7.5.2 Extraction based on match

You can make more specific matching by using the function `str_extract()`, that takes the arguments: `str_extract(string, pattern)`. On our string from before we now have the following constraints: It has to start with a digit and we only want to extract until there is a non-letter character (space, punctuation, hashtag, or similar):

```
x <- c("John", "Doe", "40 years", "30", "#SuperDad")
str_extract(x, "^\\d.*?\\w")
```

```
## [1] NA   NA   "40" "30" NA
```

It now stopped after 40, in the object "40 years" as there is a space.

7.5.3 Context-based extraction

Sometimes you need to extract based on the context and not on the target-pattern itself.

A relatively simple example here is that we want to capture what is followed by "_" and preceded by "_" (what is in between two underscores). We can then use the `(?<=x)` and `(?=y)`, where `x` and `y` are underscores:

```
code <- "vndi_123V_33"
str_extract(code, "(?<=_)\\w.+?(?=_)")
```

```
## [1] "123V"
```

This can also be done (maybe) more elegant using brackets `()` to encapsulate, `str_match()`, and index to capture the group [,2]. Mind the first element, which is basically all that is captured by the full expression:

```
str_match(code, "_(\\w+?)_")
```

```
##      [,1]     [,2]
## [1,] "_123V_" "123V"
```

```
str_match(code, "_(\\w+?)_")[,2]
```

```
## [1] "123V"
```

```
# Even more groups can be encapsulated.
str_match(code, "(\\w+?)_(\\w+?)_(\\d+)")
```

```
##      [,1]           [,2]   [,3]   [,4]
## [1,] "vndi_123V_33" "vndi" "123V" "33"
```

7.5.4 Grouping

A final example, that might be a bit complex, but very illustrative and from real-life on *grouping*, is gene-discovery in DNA. In DNA, you have some rules to find a sequence that can become a protein.

Very roughly, you need to find a sequence that - starts with "ATG", the "start-codon" - then have groups of 3 letters - stops as soon as it meet a "stop-codon", which is either "TAA", "TAG" or "TGA".

This can be formalized as follows

```
dna <- "TATGCATGTTTAGTAGCTTTTAG"

str_extract(dna, "ATG([ATCG]{3})+?(TAA|TAG|TGA)")
```

```
## [1] "ATGCATGTTTAG"
```

```
str_extract(dna, "ATG(\\w{3})+?(TAA|TAG|TGA)") # Another way to write the same
```

```
## [1] "ATGCATGTTTAG"
```

If you did not use the non-greedy "?" here, the sequence would not have stopped by the first stop-codon but by the last one in the sequence.

```
str_extract(dna, "ATG(\\w{3})+(TAA|TAG|TGA)") # greedy version
```

```
## [1] "ATGCATGTTTAGTAG"
```

7.5.5 More help on regex

- stringr-cheatsheet

Chapter 8

Code Structure

When you read the code of another person (or your own code after two months) you will find out that it is really important to structure it wisely. Other people feel the same.

Likewise, if you have a uniform work flow that is similar to your other codes, it is easier to find what you are looking for, if you are browsing your script for some function you once write or a smart hack! (A hack is here understood as *some code that does something smart* and not an attempt to attack the university servers in order to improve your grade!)

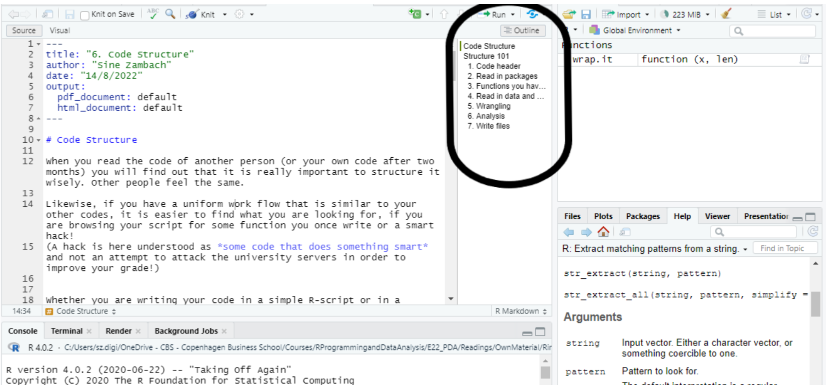
Whether you are writing your code in a simple R-script or in a markdown file, there are a few good tips and tricks, that I will go through in this chapter.

8.1 Structure 101

First, you get a brief overview over how you can/should organise your script.

1. Start with yourself in code header
2. Read in packets
3. Your own functions
4. Read in data
5. Wrangling
6. Analysis
7. Writing files

You can check your outline of your RMD-file if you click the “outline button:



```
---
title: "6. Code Structure"
author: "Sine Zambach"
date: "14/8/2022"
output:
  pdf_document: default
  html_document: default
---

# Code Structure

When you read the code of another person (or your own code after two months) you will find out that it is really important to structure it wisely. Other people feel the same.

Likewise, if you have a uniform work flow that is similar to your other codes, it is easier to find what you are looking for, if you are browsing your script for some function you once write or a smart hack!
(A hack is here understood as *some code that does something smart* and not an attempt to attack the university servers in order to improve your grade!)


Whether you are writing your code in a simple R-script or in a
```

Code Structure
Structure 101
1. Code header
2. Read in packages
3. Functions you hav…
4. Read in data and …
5. Wrangling
6. Analysis
7. Write files

Functions

wrap.it	function (x, len)

R: Extract matching patterns from a string.

```
str_extract(string, pattern)

str_extract_all(string, pattern, simplify =
```

Arguments

string	Input vector. Either a character vector, or something coercible to one.
pattern	Pattern to look for.

R 4.0.2 · C:/Users/sz.digi/OneDrive - CBS - Copenhagen Business School/Courses/RProgrammingandDataAnalysis/E22_PDA/Readings/OwnMaterial/Rir

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
```

8.1.1 1. Code header

Start with writing your header, saying at least what year it is, your name and purpose or title.

In R Markdown it looks like:

```
---
title: "Strings - 101"
author: "Sine Zambach"
date: "2021"
output:
  theme: united
---
```

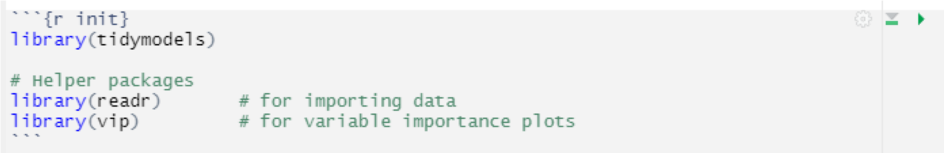
In a .r-script it would look like

```
# Script for looking at Strings
# Sine Zambach 2021
```

8.1.2 2. Read in packages

After the header you load in all packages that you need.

In RMD you would do this in a code chunk called init:



~~~
```{r init}
library(tidymodels)

# Helper packages
library(readr)       # for importing data
library(vip)         # for variable importance plots
```
~~~

In a script-file you have a secion like ## Load packages---- such that you can see it in your outline.

```
## Load packages----

library(tidyverse)
library(ggplot2)
library(readxl)
library(esquisse)
...
```

In the lecture we sometimes read in the libraries during the scripting. That is for showing you which packages are used when, but it is not wise to have it different places in your script as you loose the overview.

8.1.3 3. Functions you have created or borrowed (if any)

After reading in packages, you add functions you hvae created yourself or borrowed from the interned. For instance, I have borrowed this wrapper function from [Marc Schwartz] {https://stat.ethz.ch/pipermail/r-help/2005-April/069566.html} for labels in my figures:

```
####### Functions----
# Core wrapping function
wrap.it <- function(x, len)
{
  sapply(x, function(y) paste(strwrap(y, len),
                              collapse = "\n"),
         USE.NAMES = FALSE)
}
```

8.1.4 4. Read in data and check it

Now you can read in data and control it before wrangle it. You can in comments say something about what you expect to see.

```
####### Read in data ----
BOO <-  read_excel("Code/Data/BagsOfOranges.xlsx")

str(BOO) # should consist of some character columns and weight and price (num)
head(BOO) # Check Data Frame
```

8.1.5 5. Wrangling

After this starts the data wrangle which means using the neccesary functions to adjust data such that it is ready for the analysis and visualization. We have not covered much of wrangling principles in this compendium, but other materials goes deeply into this, e.g. [Wickhams R for data science]{https://r4ds.had.co.nz/}

```
####### Data cleaning and wrangling ----

BOO <- left_join(BOO, Geo_dim, by = c("origin" = "Country"))
BOO <- mutate(BOO, ppk = prize/weight)
BOO <- select(BOO, -prize, -weight)
na.omit(BOO)

newfile <- BOO
```

You can split your wrangling into more sections if it make sence.

8.1.6 6. Analysis

Now you can have a section particularly for the statistical investigations. This is where interesting models and insights come fro your data: your visualizations, your statistical tests, your models, etc.

```
####### Analysis ----

# What can explain the price of oranges...?
lmfit(price ~ ., data=newfile)
...
```

8.1.7 7. Write files

Some times - the output of a script is a new file that you want to write out. It could look like:

```
write.table(my_file, file="myfile.txt",sep = "\t", row.names = FALSE)
```

- There exists many "write" functions! Also related to visualizations, that you might also want to save as images.

8.2 Working together

Many things are more fun when you are two - also programming. There are some ways you can work together.

Pair programming is a technique, where one is coding and the other one tells what to code. After a while (say 15 min), the roles shifts. In this way, one will not end up coding everything, but both will get the experience of programming, and instructing others in programming. Which you learn a lot from.

When you work together, you will often work on each of your computers, working on different files. Here you should

8.2.1 Github

You can use GitHub {https://github.com/} to share you code. This is made for code sharing and can be very nice for this. You can find more information by following the link.

8.2.2 Start code that helps finding where you are

Often you will have different paths to the same drive, whether it is OneDrive, DropBox or similar. Maybe your data is confidential, and you don't want it on GitHub.

Then, you can save this with a smart code in the beginning of the script:

```
# Sourcing script with packages
file_path <- paste0(dirname(rstudioapi::getSourceEditorContext()$path), "/")
```

Then you use the file path when you read in files, and you can access the same drive from almost everywhere:

```
BagsOfOranges <- read_excel(paste0(file_path,"Data/BagsOfOranges.xlsx"))
```

8.3 Debugging - correcting errors

The word computer bug is coined by programmer Grace Hopper and comes from the 40'es where actual bugs could fly into the machine and create an error in the programme. This was the old days with punch cards in the computers and of cause real bugs will not create problems in our computers today.

However, the word is nice and illustrative to talk about finding errors and correcting them, we talk about errors as *bugs* in computer science, and when we have eliminated them, we have *debugged* the program.

There are sophisticated functions in R that can help you, but I will present you for the most basic tools that are very useful for me.

First - if an error message comes, I will google the message + R. Very often, I end in a forum called "stack overflow" or in an R-group, where the problem is discussed and people have suggested fixes.

If you are uncertain about the output, you can use the print function (or just write the variable). Then you can follow what it should be, and what it actually is. It is always useful to inspect your data, but in e.g. oops it can be very useful to print.

For instance you can see that something is wring here, as the loop prints `NULL` instead of 11,12,13, etc.

```
x <- c(1,2,3,"hello")

for(i in x){
  if (is.numeric(i)){
    y <- i+ 10
  } else{y <- NULL}
  print(y)
}
```

```
## NULL
## NULL
## NULL
## NULL
```

This is of cause because of the "hello", which assign element 4 to a string. And since `x` is a vector, all elements will then be strings.

Instead it should in this case be a list to work properly:

```
x <- list(1,2,3,"hello")

for(i in x){
  if (is.numeric(i)){
    y <- i+ 10
  } else{y <- NULL}
  print(y)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## NULL
```

Chapter 9

Coding with AI

Since November 2022, there is a new great tool, namely the so called large language models, implemented in tools such as ChatGPT, Bing and Google Bard.

Where we in the old days of summer 2022 would google a problem and try to find a page that described a solution, you can now simply ask the chat bot for getting some code that can help you creating a solution to your problem.

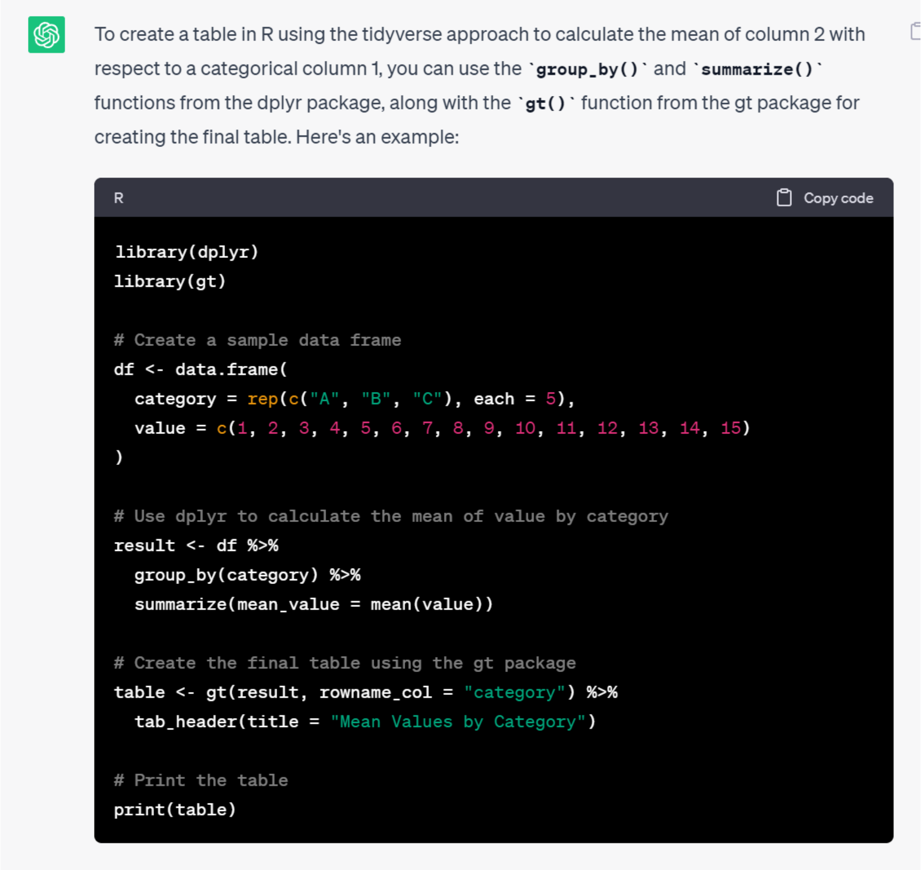
Although the models sometimes have problems with being 100% correct, they used to be good at helping with programming code, and if you already know some R, your "prompt" can be even more powerful with respect to getting the code you need. On the other hand - if you have not learned the basics, you will not be able to use it efficiently, and you might end up not understanding why it will not work.

Additionally, you can use it to help you debug your code and it can help you clarify concepts and operators and even creating datas set. I will give a few examples below, using ChatGPT based on GPT 3.5. but it could just as well have been Bing, Bard, etc.

9.1 Creating Code

In principle, you can ask ChatGPT to provide any code for you. In practice, you will have to understand the basics to understand what you get out of it. Underneath is an example where I do not provide the data in the prompt. For privacy reasons, this is the safest way to use it as we are not 100% certain about what happens to out prompt.

How do I create a table in r where I take the mean of column 2 with respect to column 1 which is categorical using tidýverse



To create a table in R using the tidyverse approach to calculate the mean of column 2 with respect to a categorical column 1, you can use the `` `group_by()` `` and `` `summarize()` `` functions from the dplyr package, along with the `` `gt()` `` function from the gt package for creating the final table. Here's an example:

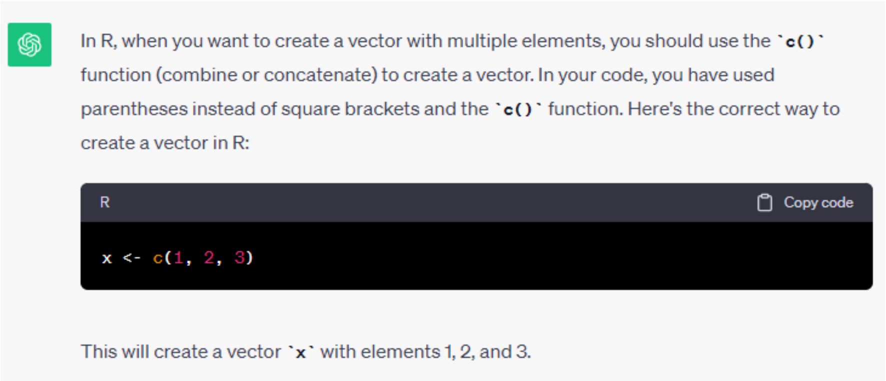
```r
library(dplyr)
library(gt)

# Create a sample data frame
df <- data.frame(
  category = rep(c("A", "B", "C"), each = 5),
  value = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
)

# Use dplyr to calculate the mean of value by category
result <- df %>%
  group_by(category) %>%
  summarize(mean_value = mean(value))

# Create the final table using the gt package
table <- gt(result, rowname_col = "category") %>%
  tab_header(title = "Mean Values by Category")

# Print the table
print(table)
```

9.2 Help debugging code

This is a nice feature. As already mentioned, we are used to google our problem, and next, we might need to read quite a lot of pages before we find a solution. Language models can act as your own tutor.

This might be a toy example, but it often works for even larger problems.

Why does this not work in r? x <- (1,2,3)



In R, when you want to create a vector with multiple elements, you should use the `` `c()` `` function (combine or concatenate) to create a vector. In your code, you have used parentheses instead of square brackets and the `` `c()` `` function. Here's the correct way to create a vector in R:

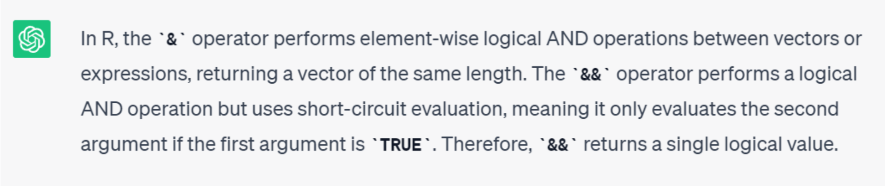
```r
x <- c(1, 2, 3)
```

This will create a vector `` `x` `` with elements 1, 2, and 3.

9.3 Concept clarification

If you are wondering about something conceptual in your code, large language models can also help:

What is the difference between & and && in R?



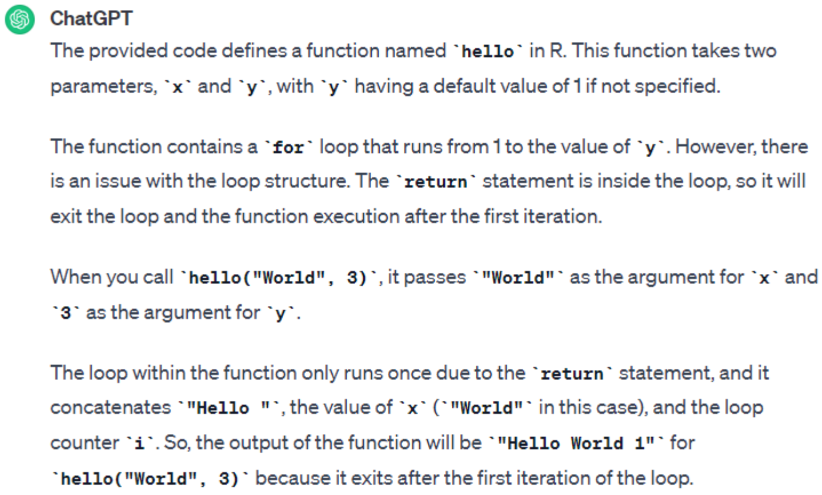
In R, the `&` operator performs element-wise logical AND operations between vectors or expressions, returning a vector of the same length. The `&&` operator performs a logical AND operation but uses short-circuit evaluation, meaning it only evaluates the second argument if the first argument is `TRUE`. Therefore, `&&` returns a single logical value.

Interestingly, this logic operator (&&) changed with the R 4.3.0-release from August 2023, and thus was not updated. This is of cause always a pitfall using large language models as most are not fully updated on new changes.

9.4 Coding explanation

Maybe somebody has written some code at work and is now ill. And you need the code, but your colleague uses a syntax that is a bit different than you. What do you do? You ask a large language model!

What does this code do? hello <- function(x, y = 1){ for(i in 1:y){ return(paste(“Hello”, x, i)) } } hello(“World”, 3)



ChatGPT

The provided code defines a function named `hello` in R. This function takes two parameters, `x` and `y`, with `y` having a default value of 1 if not specified.

The function contains a `for` loop that runs from 1 to the value of `y`. However, there is an issue with the loop structure. The `return` statement is inside the loop, so it will exit the loop and the function execution after the first iteration.

When you call `hello("World", 3)`, it passes `"World"` as the argument for `x` and `3` as the argument for `y`.

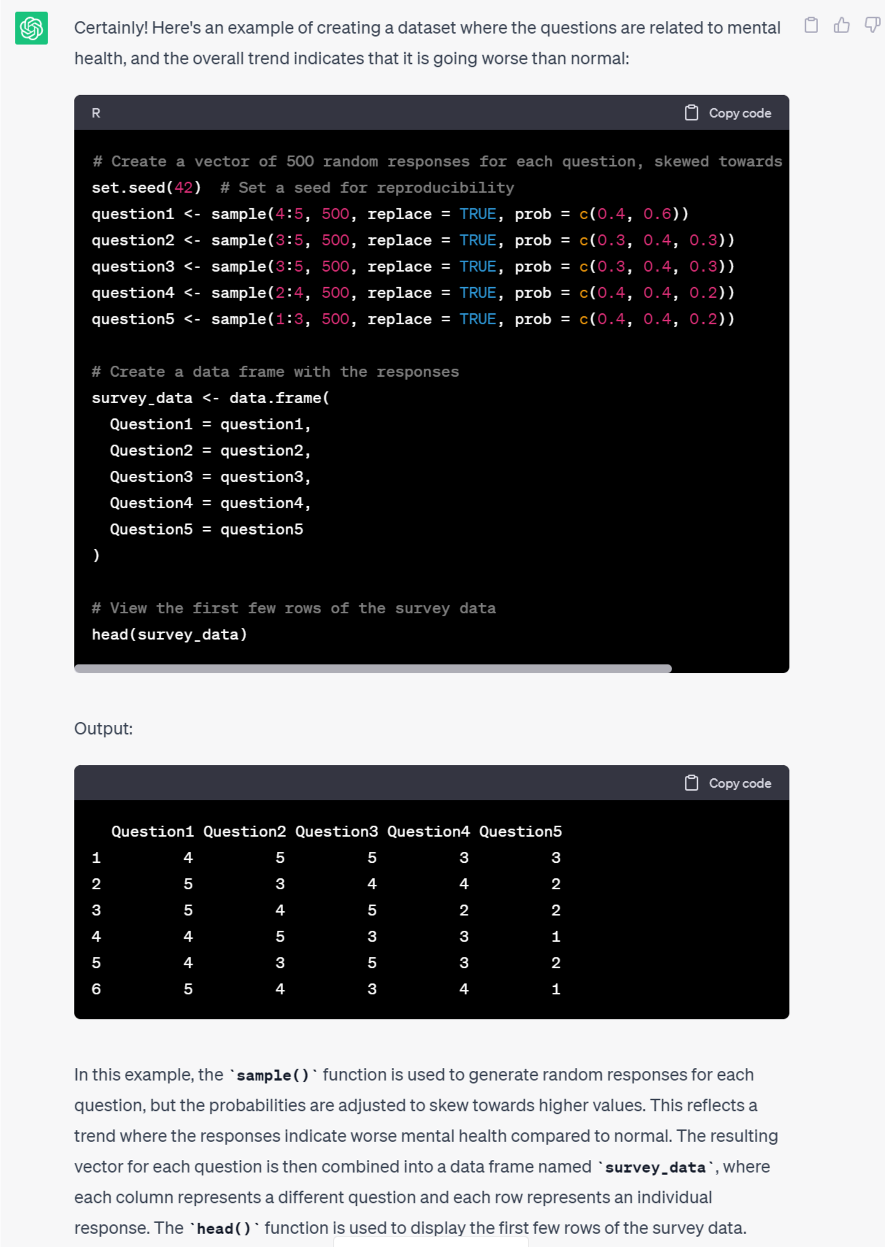
The loop within the function only runs once due to the `return` statement, and it concatenates `"Hello "`, the value of `x` (`"World"` in this case), and the loop counter `i`. So, the output of the function will be `"Hello World 1"` for `hello("World", 3)` because it exits after the first iteration of the loop.

Notice that this is code from chapter 5, in which I explain the same thing in another way.

9.5 Creating a synthetic data set

Finally, AI could also help you creating a data set. The one below is quite precise, but it can be asked even more sloppy.

Create a dataset from a survey with a 5-likert scale questions as columns asking into mental health. There should be 500 row, corresponding to 500 answers. The overall trend is that it is going worse than normal?



Certainly! Here's an example of creating a dataset where the questions are related to mental health, and the overall trend indicates that it is going worse than normal:

```r
# Create a vector of 500 random responses for each question, skewed towards
set.seed(42)  # Set a seed for reproducibility
question1 <- sample(4:5, 500, replace = TRUE, prob = c(0.4, 0.6))
question2 <- sample(3:5, 500, replace = TRUE, prob = c(0.3, 0.4, 0.3))
question3 <- sample(3:5, 500, replace = TRUE, prob = c(0.3, 0.4, 0.3))
question4 <- sample(2:4, 500, replace = TRUE, prob = c(0.4, 0.4, 0.2))
question5 <- sample(1:3, 500, replace = TRUE, prob = c(0.4, 0.4, 0.2))

# Create a data frame with the responses
survey_data <- data.frame(
  Question1 = question1,
  Question2 = question2,
  Question3 = question3,
  Question4 = question4,
  Question5 = question5
)

# View the first few rows of the survey data
head(survey_data)
```

Output:

```
  Question1 Question2 Question3 Question4 Question5
1         4         5         5         3         3
2         5         3         4         4         2
3         5         4         5         2         2
4         4         5         3         3         1
5         4         3         5         3         2
6         5         4         3         4         1
```

In this example, the `sample()` function is used to generate random responses for each question, but the probabilities are adjusted to skew towards higher values. This reflects a trend where the responses indicate worse mental health compared to normal. The resulting vector for each question is then combined into a data frame named `survey_data`, where each column represents a different question and each row represents an individual response. The `head()` function is used to display the first few rows of the survey data.

9.6 Other options

If you are ready to leave the safe environment of RStudio, you can also use the dedicated AI-support, Copilot, that works in the Visual Studio Code, which is a general code editor, that works for more languages.