

# Polish Language(s) and Digital Humanities Using R

G. Moroz

2020



# Contents

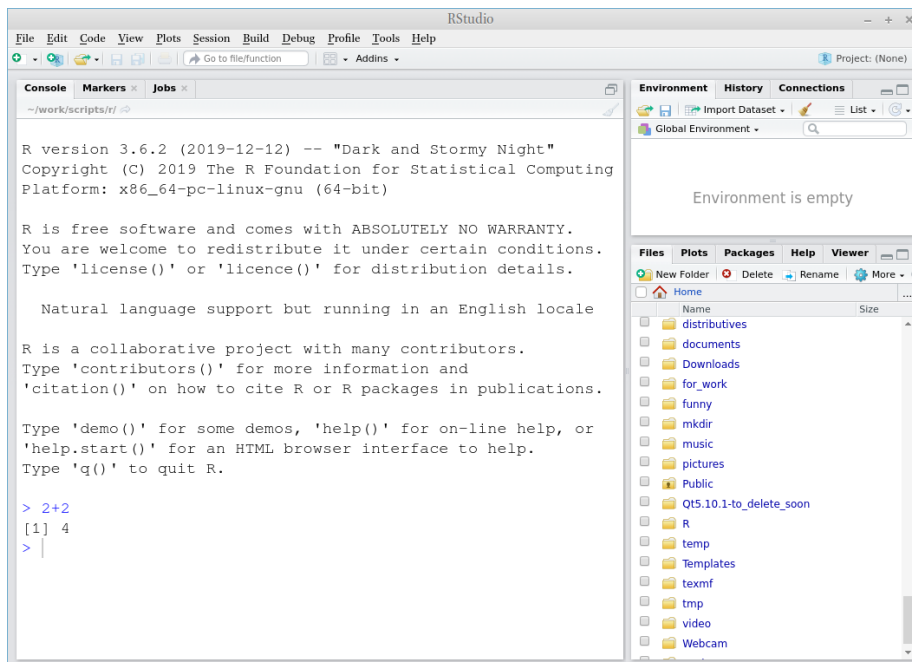


# Chapter 1

## Prerequisites

Before the classes I would like to ask you to follow the instructions mentioned below to prepare your device for the class work:

- install **R** from the following link: <https://cloud.r-project.org/>
- install **RStudio** from the following link: <https://rstudio.com/products/rstudio/download/#download> (FREE version, no need to pay!)
- after the installation run the RStudio program, type `2+2`, and press **Enter**.



If you see something like this, then you are well prepared for classes.

- Go to the <https://rstudio.cloud/> website and sign up there. This is optional, but it will be a backup version, if something will not work on your computer.

Special thanks to Helena Link for the workshop organisation and for correcting typos in this text.

## Chapter 2

# Introduction to R and RStudio

### 2.1 Introduction

#### 2.1.1 Why data science?

Data science is a new field that is actively developing lately. This field merges computer science, mathematics, statistics, and it is hard to say how much science in data science. In many scientific fields a new data science paradigm arises and even forms a new sub-field:

- Bioinformatics
- Crime data analysis
- Digital humanities
- Data journalism
- Data driven medicine
- ...

There are a lot of new books “Data Science for ...”:

- psychologists (?)
- immunologists (?)
- business (?)
- public policy (?)
- fraud detection (?)
- ...

Data scientists need to be able to:

- gather data
- transform data

- visualize data
- create a statistical model based on data
- share and represent the results of this work
- organize the whole workflow in a reproducible way

### 2.1.2 Why R?

R (?) is a programming language with a big infrastructure of packages that helps to work in different fields of science and computer technology.

There are several alternatives:

- Python (??)
- Julia (?)
- bash (?)
- java (?)
- ...

You can find some R answers here:

- R for data science (?), it is online
- R community
- stackoverflow
- any search engine you use
- ...

## 2.2 Introduction to RStudio

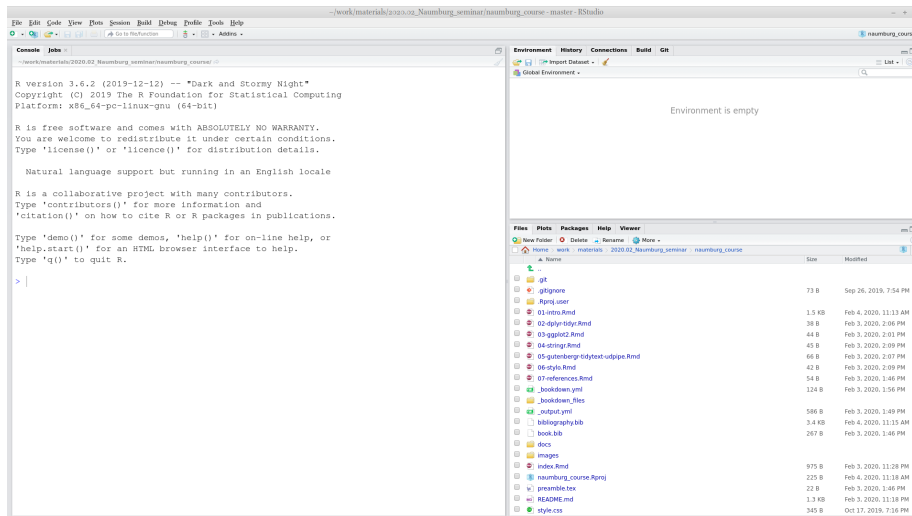
R is the programming language. RStudio is the most popular IDE (Integrated Development Environment) for R language.

When you open RStudio for the first time you can see something like this:

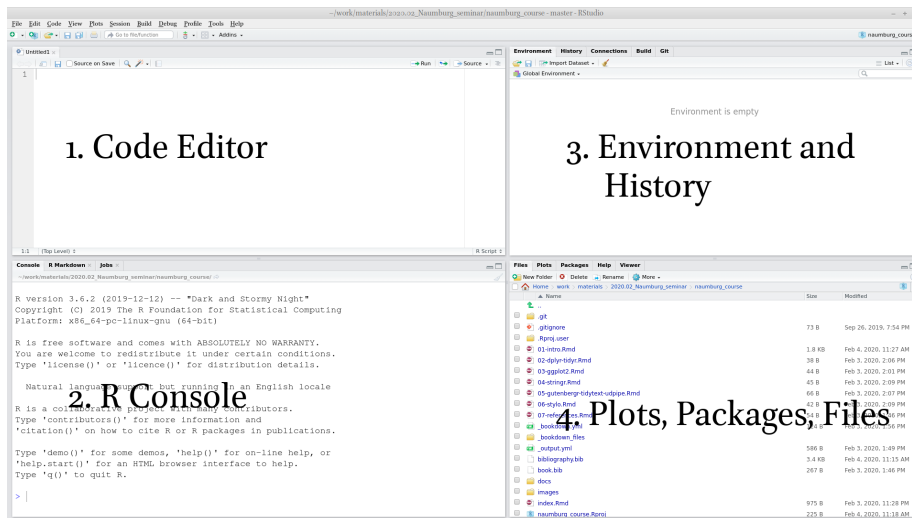


## 2.3. R AS A CALCULATOR

5



When you press button at the top of the left window you will be able to see all four panels of RStudio.



## 2.3 R as a calculator

Lets first start with the calculator. Press in R console

```
2+9
```

```
## [1] 11
```

```
50*(9-20)
```

```
## [1] -550
```

```
3^3
```

```
## [1] 27
```

```
9^0.5
```

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

```
9+0.5
```

```
## [1] 9.5
```

```
9+.5
```

```
## [1] 9.5
```

```
pi
```

```
## [1] 3.141593
```

Remainder after division

```
10 %% 3
```

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



So you are ready to solve some really hard equations (round it four decimal places):

$$\frac{\pi + 2}{2^{3-\pi}}$$

list of hints

Are you sure that you rounded the result? I expect the answer to be rounded to four decimal places: 0.87654321 becomes 0.8765.

Are you sure you didn't get into the brackets trap? Even though there isn't any brackets in the mathematical notation, you need to add them in R, otherwise the operation order will be wrong.

## 2.4 Comments

Any text after a hash # within the same line is considered a comment.

```
2+2 # it is four
```

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

```
# you can put any comments here  
3+3
```

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

## 2.5 Functions

The most important part of R is functions: here are some of them:

```
sqrt(4)
```

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

```
abs(-5)
```

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

```
sin(pi/2)
```

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

```
cos(pi)
```

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

```
sum(2, 3, 9)
```

```
## [1] 14
```

```
prod(5, 3, 9)
```

```
## [1] 135
```

```
sin(cos(pi))
```

```
## [1] -0.841471
```

Each function has a name and zero or more arguments. All arguments of the function should be listed in parenthesis and separated by comma:

```
pi
```

```
## [1] 3.141593
```

```
round(pi, 2)
```

```
## [1] 3.14
```

Each function's argument has its own name and serial number. If you use names of the function's arguments, you can put them in any order. If you do not use names of the function's arguments, you should put them according the serial number.

```
round(x = pi, digits = 2)
```

```
## [1] 3.14
```

```
round(digits = 2, x = pi)
```

```
## [1] 3.14
```

```
round(x = pi, d = 2)
```

```
## [1] 3.14
```

```
round(d = 2, x = pi)
```

```
## [1] 3.14
```

```
round(pi, 2)
```

```
## [1] 3.14
```

```
round(2, pi) # this is not the same as all previous!
```

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

There are some functions without any arguments, but you still should use parenthesis:

```
Sys.Date() # correct
```

```
## [1] "2020-02-12"
```

```
Sys.Date # wrong
```

```
## function ()
## as.Date(as.POSIXlt(Sys.time()))
## <bytecode: 0x61b8c7bc0fd0>
## <environment: namespace:base>
```

Each function in R is documented. You can read its documentation typing a question mark before the function name:

```
?Sys.Date
```



Explore the function `log()` and calculate the following logarithm:

$$\log_3(3486784401)$$

list of hints

A-a-a! I don't remember anything about logarithms... The logarithm is the inverse function to exponentiation. That means the logarithm of a given number  $x$  is the exponent to which another fixed number, the base  $b$ , must be raised, to produce that number  $x$ .

$$10^n = 1000, \text{ what is } n?$$

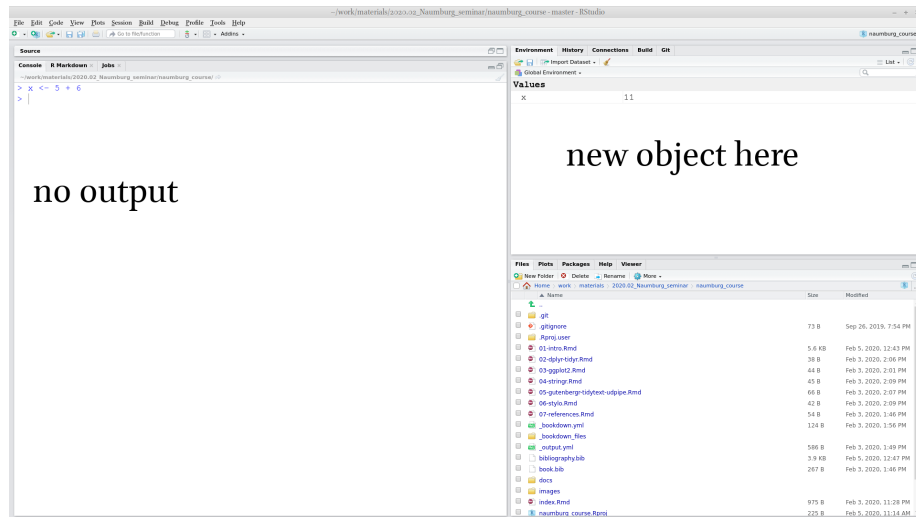
$$n = \log_{10}(1000)$$

What does this small 3 in the task mean? This is the base of the logarithm. So the task is: what is the exponent to which another fixed number, the base 3, must be raised, to produce that number *3486784401*.

## 2.6 Variables

Everything in R can be stored in a variable:

```
x <- 5 + 6
```



As a result, no output in the Console, and a new variable `x` appear in the Environment window. From now on I can use this new variable:

```
x + x
```

```
## [1] 22
```

```
sum(x, x, 7)
```

```
## [1] 29
```

All those operations don't change the variable value. In order to change the variable value you need to make a new assignment:

```
x <- 5 + 6 + 7
```

The fast way for creating `<-` in RStudio is to press **Alt -** on your keyboard.

It is possible to use equal sign `=` for assignment operation, but the recommendations are to use arrow `<-` for the assignment, and equal sign `=` for giving arguments' value inside the functions.

For removing vector you need to use the function `rm()`:

```
rm(x)
x
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

### 2.6.1 Variable comparison

It is possible to compare different variables

```
x <- 18
x > 18
```

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

```
x >= 18
```

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

```
x < 100
```

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

```
x <= 18
```

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

```
x == 18
```

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

```
x != 18
```

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

Operator ! can work by itself changing logical values into reverse:

```
!TRUE
```

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

```
!FALSE
```

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

### 2.6.2 Variable types

There are several types of variables in R. In this course the only important types will be `double` (all numbers), `character` (or strings), and `logical`:

```
x <- 2+3
typeof(x)
```

```
## [1] "double"
```

```
y <- "Cześć"
typeof(y)
```

```
## [1] "character"
```

```
z <- TRUE
typeof(z)
```

```
## [1] "logical"
```

## 2.7 Vector

An R object that contains multiple values of the same type is called **vector**. It could be created with the command `c()`:

```
c(3, 0, pi, 23.4, -53)
```

```
## [1] 3.000000 0.000000 3.141593 23.400000 -53.000000
```

```
c("Kraków", "Warszawa", "Cieszyn")
```

```
## [1] "Kraków" "Warszawa" "Cieszyn"
```

```
c(FALSE, FALSE, TRUE)
```

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

```
a <- c(2, 3, 4)
b <- c(5, 6, 7)
c(a, b)
```

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



For the number sequences there is an easy way:

```
1:10
```

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

```
3:-5
```

```
## [1] 3 2 1 0 -1 -2 -3 -4 -5
```

From now on you can understand that everything we have seen before is a vector of length one. That is why there is `[1]` in all outputs: it is just an index of elements in a vector. Have a look here:

```
1:60
```

```
## [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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
## [51] 51 52 53 54 55 56 57 58 59 60
```

```
60:1
```

```
## [1] 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
## [26] 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11
## [51] 10 9 8 7 6 5 4 3 2 1
```

There is also a function `seq()` for creation of arithmetic progressions:

```
1:20
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
seq(from = 1, to = 20, by = 1)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
seq(from = 2, to = 100, by = 13)
```

```
## [1] 2 15 28 41 54 67 80 93
```



Use the argument `length.out` of function `seq()` and create an arithmetic sequence from  $\pi$  to  $2\pi$  of length 50.

There are also some built-in vectors:

```
letters
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
month.name
```

```
## [1] "January" "February" "March" "April" "May" "June"
## [7] "July" "August" "September" "October" "November" "December"
```

```
month.abb
```

```
## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

### 2.7.1 Vector coercion

Vectors are R objects that contain multiple values of **the same type**. But what if we merged together different types?

```
c(1, "34")
```

```
## [1] "1" "34"
```

```
c(1, TRUE)
```

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

```
c(TRUE, "34")
```

```
## [1] "TRUE" "34"
```

It is clear that there is a hierarchy: strings > double > logical. It is not universal across different programming languages. It doesn't correspond to the amount of values of particular type:

```
c(1, 2, 3, "34")
```

```
## [1] "1" "2" "3" "34"
```

```
c(1, TRUE, FALSE, FALSE)
```

```
## [1] 1 1 0 0
```

The same story could happen during other operations:

```
5+TRUE
```

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

### 2.7.2 Vector operations

All operations, that we discussed earlier, could be done with vectors of the same length:

```
1:5 + 6:10
```

```
## [1] 7 9 11 13 15
```

```
1:5 - 6:10
```

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

```
1:5 * 6:10
```

```
## [1] 6 14 24 36 50
```

There are operations where the vector of any length and vector of length one is involved:

```
1:5 + 7
```

```
## [1] 8 9 10 11 12
```

```
1:5 - 7
```

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

```
1:5 / 7
```

```
## [1] 0.1428571 0.2857143 0.4285714 0.5714286 0.7142857
```

There are a lot of functions in R that are **vectorised**. That means that applying this function to a vector is the same as applying this function to each element of the vector:

```
sin(1:5)
```

```
## [1] 0.8414710 0.9092974 0.1411200 -0.7568025 -0.9589243
```

```
sqrt(1:5)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
abs(-5:3)
```

```
## [1] 5 4 3 2 1 0 1 2 3
```

### 2.7.3 Indexing vectors

How to get some value or bunch of values from a vector? You need to index them:

```
x <- c(3, 0, pi, 23.4, -53)
y <- c("Kraków", "Warszawa", "Cieszyn")

x[4]
```

```
## [1] 23.4
```

```
y[2]
```

```
## [1] "Warszawa"
```

It is possible to have a vector as index:

```
x[1:2]
```

```
## [1] 3 0
```

```
y[c(1, 3)]
```

```
## [1] "Kraków" "Cieszyn"
```

It is possible to index something that you **do not** want to see in the result:

```
y[-2]
```

```
## [1] "Kraków" "Cieszyn"
```

```
x[-c(1, 4)]
```

```
## [1] 0.000000 3.141593 -53.000000
```

It is possible to have other variables as an index

```
z <- c(3, 2)
x[z]
```

```
## [1] 3.141593 0.000000
```

```
y[z]
```

```
## [1] "Cieszyn" "Warszawa"
```

It is possible to index with a logical vector:

```
x[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 3.000000 3.141593 23.400000
```

That means that we could use TRUE/FALSE-vector produced by comparison:

```
x[x > 2]
```

```
## [1] 3.000000 3.141593 23.400000
```

It works because `x > 2` is a vector of logical values:

```
x > 2
```

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

It is possible to use `!` operator here changing all `TRUE` values to `FALSE` and vice versa.

```
x[!(x > 2)]
```

```
## [1] 0 -53
```



How many elements in the vector `g` if expression `g[pi < 1000]` does not return an error?

### 2.7.4 NA

Sometimes there are some missing values in the data, so it is represented with `NA`

```
NA
```

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

```
c(1, NA, 9)
```

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

```
c("Kraków", NA, "Cieszyn")
```

```
## [1] "Kraków" NA "Cieszyn"
```

```
c(TRUE, FALSE, NA)
```

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

It is possible to check, whether there are missing values or not

```
x <- c("Kraków", NA, "Cieszyn")
y <- c("Kraków", "Warszawa", "Cieszyn")
is.na(x)
```

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

```
is.na(y)
```

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

Some functions doesn't work with vectors that contain missed values, so you need to add argument `na.rm = TRUE`:

```
x <- c(1, NA, 9, 5)
mean(x)
```

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

```
mean(x, na.rm = TRUE)
```

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

```
min(x, na.rm = TRUE)
```

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

```
max(x, na.rm = TRUE)
```

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

```
median(x, na.rm = TRUE)
```

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

```
range(x, na.rm = TRUE)
```

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

## 2.8 Packages

The most important and useful part of R is hidden in its packages. Everything that we discussed so far is basic R functionality invented back in 1979. Since then a lot of different things changed, so all new practices for data analysis, visualisation and manipulation are packed in packages. During our class we will learn the most popular “*dialect*” of R called **tidyverse**.

In order to install packages you need to use a command. Let's install the **tidyverse** package:

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

For today we also will need the **readxl** package:

```
install.packages("readxl")
```

After you have downloaded packages nothing will change. You can not use any functionality from packages unless you load the package with the `library()` function:

```
library("tidyverse")
```

Not loading package is the most popular mistake of my students. So remember:

- `install.packages("...")` is like you are buying a screwdriver set;
- `library("...")` is like you are stusing art your screwdriver.



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

`library("...")`

For the further lectures we will need `tidyverse` package.



Please install `tidyverse` package and load it.

### 2.8.1 tidyverse

The `tidyverse` is a set of packages:

- `tibble`, for tibbles, a modern re-imagining of data frames — analogue of tables in R
- `readr`, for data import
- `dplyr`, for data manipulation
- `tidyr`, for data tidying (we will discuss it later today)
- `ggplot2`, for data visualisation
- `purrr`, for functional programming



## 2.9 Dataframe (tibble)

A data frame is a collection of variables of the same number of rows with unique row names. Here is an example dataframe with the Tomm Moore filmography:

```
moore_filmography <- tibble(title = c("The Secret of Kells",
                                     "Song of the Sea",
                                     "Kahlil Gibran's The Prophet",
                                     "The Breadwinner",
                                     "Wolfwalkers"),
                           year = c(2009, 2014, 2014, 2017, 2020),
                           director = c(TRUE, TRUE, TRUE, FALSE, TRUE))

moore_filmography
```

```
## # A tibble: 5 x 3
##   title                year director
##   <chr>              <dbl> <lgl>
## 1 The Secret of Kells    2009 TRUE
## 2 Song of the Sea       2014 TRUE
## 3 Kahlil Gibran's The Prophet 2014 TRUE
## 4 The Breadwinner       2017 FALSE
## 5 Wolfwalkers           2020 TRUE
```

There are a lot of built-in dataframes:

```
mtcars
```

```
##           mpg  cyl  disp  hp drat   wt  qsec vs  am  gear  carb
## Mazda RX4      21.0   6  160.0  110 3.90 2.620 16.46 0   1    4    4
## Mazda RX4 Wag  21.0   6  160.0  110 3.90 2.875 17.02 0   1    4    4
## Datsun 710     22.8   4  108.0   93 3.85 2.320 18.61 1   1    4    1
## Hornet 4 Drive  21.4   6  258.0  110 3.08 3.215 19.44 1   0    3    1
## Hornet Sportabout 18.7   8  360.0  175 3.15 3.440 17.02 0   0    3    2
## Valiant        18.1   6  225.0  105 2.76 3.460 20.22 1   0    3    1
## Duster 360     14.3   8  360.0  245 3.21 3.570 15.84 0   0    3    4
## Merc 240D      24.4   4  146.7   62 3.69 3.190 20.00 1   0    4    2
## Merc 230       22.8   4  140.8   95 3.92 3.150 22.90 1   0    4    2
## Merc 280       19.2   6  167.6  123 3.92 3.440 18.30 1   0    4    4
## Merc 280C      17.8   6  167.6  123 3.92 3.440 18.90 1   0    4    4
## Merc 450SE     16.4   8  275.8  180 3.07 4.070 17.40 0   0    3    3
## Merc 450SL     17.3   8  275.8  180 3.07 3.730 17.60 0   0    3    3
## Merc 450SLC    15.2   8  275.8  180 3.07 3.780 18.00 0   0    3    3
## Cadillac Fleetwood 10.4   8  472.0  205 2.93 5.250 17.98 0   0    3    4
## Lincoln Continental 10.4   8  460.0  215 3.00 5.424 17.82 0   0    3    4
## Chrysler Imperial 14.7   8  440.0  230 3.23 5.345 17.42 0   0    3    4
```

```
## Fiat 128          32.4  4  78.7  66 4.08 2.200 19.47  1  1  4  1
## Honda Civic      30.4  4  75.7  52 4.93 1.615 18.52  1  1  4  2
## Toyota Corolla   33.9  4  71.1  65 4.22 1.835 19.90  1  1  4  1
## Toyota Corona    21.5  4 120.1  97 3.70 2.465 20.01  1  0  3  1
## Dodge Challenger  15.5  8 318.0 150 2.76 3.520 16.87  0  0  3  2
## AMC Javelin      15.2  8 304.0 150 3.15 3.435 17.30  0  0  3  2
## Camaro Z28       13.3  8 350.0 245 3.73 3.840 15.41  0  0  3  4
## Pontiac Firebird  19.2  8 400.0 175 3.08 3.845 17.05  0  0  3  2
## Fiat X1-9        27.3  4  79.0  66 4.08 1.935 18.90  1  1  4  1
## Porsche 914-2    26.0  4 120.3  91 4.43 2.140 16.70  0  1  5  2
## Lotus Europa     30.4  4  95.1 113 3.77 1.513 16.90  1  1  5  2
## Ford Pantera L   15.8  8 351.0 264 4.22 3.170 14.50  0  1  5  4
## Ferrari Dino     19.7  6 145.0 175 3.62 2.770 15.50  0  1  5  6
## Maserati Bora    15.0  8 301.0 335 3.54 3.570 14.60  0  1  5  8
## Volvo 142E       21.4  4 121.0 109 4.11 2.780 18.60  1  1  4  2
```

```
iris
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1           3.5           1.4           0.2   setosa
## 2           4.9           3.0           1.4           0.2   setosa
## 3           4.7           3.2           1.3           0.2   setosa
## 4           4.6           3.1           1.5           0.2   setosa
## 5           5.0           3.6           1.4           0.2   setosa
## 6           5.4           3.9           1.7           0.4   setosa
## 7           4.6           3.4           1.4           0.3   setosa
## 8           5.0           3.4           1.5           0.2   setosa
## 9           4.4           2.9           1.4           0.2   setosa
## 10          4.9           3.1           1.5           0.1   setosa
## 11          5.4           3.7           1.5           0.2   setosa
## 12          4.8           3.4           1.6           0.2   setosa
## 13          4.8           3.0           1.4           0.1   setosa
## 14          4.3           3.0           1.1           0.1   setosa
## 15          5.8           4.0           1.2           0.2   setosa
## 16          5.7           4.4           1.5           0.4   setosa
## 17          5.4           3.9           1.3           0.4   setosa
## 18          5.1           3.5           1.4           0.3   setosa
## 19          5.7           3.8           1.7           0.3   setosa
## 20          5.1           3.8           1.5           0.3   setosa
## 21          5.4           3.4           1.7           0.2   setosa
## 22          5.1           3.7           1.5           0.4   setosa
## 23          4.6           3.6           1.0           0.2   setosa
## 24          5.1           3.3           1.7           0.5   setosa
## 25          4.8           3.4           1.9           0.2   setosa
## 26          5.0           3.0           1.6           0.2   setosa
```

## 27	5.0	3.4	1.6	0.4	setosa
## 28	5.2	3.5	1.5	0.2	setosa
## 29	5.2	3.4	1.4	0.2	setosa
## 30	4.7	3.2	1.6	0.2	setosa
## 31	4.8	3.1	1.6	0.2	setosa
## 32	5.4	3.4	1.5	0.4	setosa
## 33	5.2	4.1	1.5	0.1	setosa
## 34	5.5	4.2	1.4	0.2	setosa
## 35	4.9	3.1	1.5	0.2	setosa
## 36	5.0	3.2	1.2	0.2	setosa
## 37	5.5	3.5	1.3	0.2	setosa
## 38	4.9	3.6	1.4	0.1	setosa
## 39	4.4	3.0	1.3	0.2	setosa
## 40	5.1	3.4	1.5	0.2	setosa
## 41	5.0	3.5	1.3	0.3	setosa
## 42	4.5	2.3	1.3	0.3	setosa
## 43	4.4	3.2	1.3	0.2	setosa
## 44	5.0	3.5	1.6	0.6	setosa
## 45	5.1	3.8	1.9	0.4	setosa
## 46	4.8	3.0	1.4	0.3	setosa
## 47	5.1	3.8	1.6	0.2	setosa
## 48	4.6	3.2	1.4	0.2	setosa
## 49	5.3	3.7	1.5	0.2	setosa
## 50	5.0	3.3	1.4	0.2	setosa
## 51	7.0	3.2	4.7	1.4	versicolor
## 52	6.4	3.2	4.5	1.5	versicolor
## 53	6.9	3.1	4.9	1.5	versicolor
## 54	5.5	2.3	4.0	1.3	versicolor
## 55	6.5	2.8	4.6	1.5	versicolor
## 56	5.7	2.8	4.5	1.3	versicolor
## 57	6.3	3.3	4.7	1.6	versicolor
## 58	4.9	2.4	3.3	1.0	versicolor
## 59	6.6	2.9	4.6	1.3	versicolor
## 60	5.2	2.7	3.9	1.4	versicolor
## 61	5.0	2.0	3.5	1.0	versicolor
## 62	5.9	3.0	4.2	1.5	versicolor
## 63	6.0	2.2	4.0	1.0	versicolor
## 64	6.1	2.9	4.7	1.4	versicolor
## 65	5.6	2.9	3.6	1.3	versicolor
## 66	6.7	3.1	4.4	1.4	versicolor
## 67	5.6	3.0	4.5	1.5	versicolor
## 68	5.8	2.7	4.1	1.0	versicolor
## 69	6.2	2.2	4.5	1.5	versicolor
## 70	5.6	2.5	3.9	1.1	versicolor
## 71	5.9	3.2	4.8	1.8	versicolor
## 72	6.1	2.8	4.0	1.3	versicolor

## 73	6.3	2.5	4.9	1.5 versicolor
## 74	6.1	2.8	4.7	1.2 versicolor
## 75	6.4	2.9	4.3	1.3 versicolor
## 76	6.6	3.0	4.4	1.4 versicolor
## 77	6.8	2.8	4.8	1.4 versicolor
## 78	6.7	3.0	5.0	1.7 versicolor
## 79	6.0	2.9	4.5	1.5 versicolor
## 80	5.7	2.6	3.5	1.0 versicolor
## 81	5.5	2.4	3.8	1.1 versicolor
## 82	5.5	2.4	3.7	1.0 versicolor
## 83	5.8	2.7	3.9	1.2 versicolor
## 84	6.0	2.7	5.1	1.6 versicolor
## 85	5.4	3.0	4.5	1.5 versicolor
## 86	6.0	3.4	4.5	1.6 versicolor
## 87	6.7	3.1	4.7	1.5 versicolor
## 88	6.3	2.3	4.4	1.3 versicolor
## 89	5.6	3.0	4.1	1.3 versicolor
## 90	5.5	2.5	4.0	1.3 versicolor
## 91	5.5	2.6	4.4	1.2 versicolor
## 92	6.1	3.0	4.6	1.4 versicolor
## 93	5.8	2.6	4.0	1.2 versicolor
## 94	5.0	2.3	3.3	1.0 versicolor
## 95	5.6	2.7	4.2	1.3 versicolor
## 96	5.7	3.0	4.2	1.2 versicolor
## 97	5.7	2.9	4.2	1.3 versicolor
## 98	6.2	2.9	4.3	1.3 versicolor
## 99	5.1	2.5	3.0	1.1 versicolor
## 100	5.7	2.8	4.1	1.3 versicolor
## 101	6.3	3.3	6.0	2.5 virginica
## 102	5.8	2.7	5.1	1.9 virginica
## 103	7.1	3.0	5.9	2.1 virginica
## 104	6.3	2.9	5.6	1.8 virginica
## 105	6.5	3.0	5.8	2.2 virginica
## 106	7.6	3.0	6.6	2.1 virginica
## 107	4.9	2.5	4.5	1.7 virginica
## 108	7.3	2.9	6.3	1.8 virginica
## 109	6.7	2.5	5.8	1.8 virginica
## 110	7.2	3.6	6.1	2.5 virginica
## 111	6.5	3.2	5.1	2.0 virginica
## 112	6.4	2.7	5.3	1.9 virginica
## 113	6.8	3.0	5.5	2.1 virginica
## 114	5.7	2.5	5.0	2.0 virginica
## 115	5.8	2.8	5.1	2.4 virginica
## 116	6.4	3.2	5.3	2.3 virginica
## 117	6.5	3.0	5.5	1.8 virginica
## 118	7.7	3.8	6.7	2.2 virginica

```
## 119      7.7      2.6      6.9      2.3 virginica
## 120      6.0      2.2      5.0      1.5 virginica
## 121      6.9      3.2      5.7      2.3 virginica
## 122      5.6      2.8      4.9      2.0 virginica
## 123      7.7      2.8      6.7      2.0 virginica
## 124      6.3      2.7      4.9      1.8 virginica
## 125      6.7      3.3      5.7      2.1 virginica
## 126      7.2      3.2      6.0      1.8 virginica
## 127      6.2      2.8      4.8      1.8 virginica
## 128      6.1      3.0      4.9      1.8 virginica
## 129      6.4      2.8      5.6      2.1 virginica
## 130      7.2      3.0      5.8      1.6 virginica
## 131      7.4      2.8      6.1      1.9 virginica
## 132      7.9      3.8      6.4      2.0 virginica
## 133      6.4      2.8      5.6      2.2 virginica
## 134      6.3      2.8      5.1      1.5 virginica
## 135      6.1      2.6      5.6      1.4 virginica
## 136      7.7      3.0      6.1      2.3 virginica
## 137      6.3      3.4      5.6      2.4 virginica
## 138      6.4      3.1      5.5      1.8 virginica
## 139      6.0      3.0      4.8      1.8 virginica
## 140      6.9      3.1      5.4      2.1 virginica
## 141      6.7      3.1      5.6      2.4 virginica
## 142      6.9      3.1      5.1      2.3 virginica
## 143      5.8      2.7      5.1      1.9 virginica
## 144      6.8      3.2      5.9      2.3 virginica
## 145      6.7      3.3      5.7      2.5 virginica
## 146      6.7      3.0      5.2      2.3 virginica
## 147      6.3      2.5      5.0      1.9 virginica
## 148      6.5      3.0      5.2      2.0 virginica
## 149      6.2      3.4      5.4      2.3 virginica
## 150      5.9      3.0      5.1      1.8 virginica
```

You can find information about them:

```
?mtcars
?iris
```

Dataframe consists of vectors that could be called using \$ sign:

```
moore_filmography$year
```

```
## [1] 2009 2014 2014 2017 2020
```

```
moore_filmography$title
```

```
## [1] "The Secret of Kells"      "Song of the Sea"
## [3] "Kahlil Gibran's The Prophet" "The Breadwinner"
## [5] "Wolfwalkers"
```

It is possible to add a vector to an existing dataframe:

```
moore_filmography$producer <- c(TRUE, TRUE, FALSE, TRUE, TRUE)
moore_filmography
```

```
## # A tibble: 5 x 4
##   title                year director producer
##   <chr>              <dbl> <lgl>    <lgl>
## 1 The Secret of Kells    2009 TRUE     TRUE
## 2 Song of the Sea        2014 TRUE     TRUE
## 3 Kahlil Gibran's The Prophet 2014 TRUE     FALSE
## 4 The Breadwinner        2017 FALSE    TRUE
## 5 Wolfwalkers            2020 TRUE     TRUE
```

There are some useful functions that tell you something about a dataframe:

```
nrow(moore_filmography)
```

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

```
ncol(moore_filmography)
```

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

```
summary(moore_filmography)
```

```
##      title                year      director      producer
## Length:5             Min.   :2009   Mode :logical   Mode :logical
## Class :character     1st Qu.:2014   FALSE:1         FALSE:1
## Mode  :character     Median :2014   TRUE :4          TRUE :4
##                               Mean  :2015
##                               3rd Qu.:2017
##                               Max.   :2020
```

```
str(moore_filmography)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':  5 obs. of  4 variables:
## $ title   : chr  "The Secret of Kells" "Song of the Sea" "Kahlil Gibran's The Prophet" "The E
## $ year    : num  2009 2014 2014 2017 2020
## $ director: logi  TRUE TRUE TRUE FALSE TRUE
## $ producer: logi  TRUE TRUE FALSE TRUE TRUE
```

We will work exclusively with dataframes. But it is not the only data structure in R.



How many rows are in the `iris` dataframe?



How many columns are in the `mtcars` dataframe?

### 2.9.1 Indexing dataframes

Since dataframes are two-dimensional objects it is possible to index its rows and columns. Rows are the first index, columns are the second index:

```
moore_filmography[3, 2]
```

```
## # A tibble: 1 x 1
##   year
##   <dbl>
## 1  2014
```

```
moore_filmography[3,]
```

```
## # A tibble: 1 x 4
##   title                                year director producer
##   <chr>                                <dbl> <lgl>    <lgl>
## 1 Kahlil Gibran's The Prophet    2014 TRUE     FALSE
```

```
moore_filmography[,2]
```

```
## # A tibble: 5 x 1
##   year
##   <dbl>
## 1  2009
## 2  2014
## 3  2014
## 4  2017
## 5  2020
```

```
moore_filmography[,1:2]
```

```
## # A tibble: 5 x 2
##   title          year
##   <chr>         <dbl>
## 1 The Secret of Kells      2009
## 2 Song of the Sea         2014
## 3 Kahlil Gibran's The Prophet 2014
## 4 The Breadwinner         2017
## 5 Wolfwalkers             2020
```

```
moore_filmography[, -3]
```

```
## # A tibble: 5 x 3
##   title          year producer
##   <chr>         <dbl> <lgl>
## 1 The Secret of Kells      2009 TRUE
## 2 Song of the Sea         2014 TRUE
## 3 Kahlil Gibran's The Prophet 2014 FALSE
## 4 The Breadwinner         2017 TRUE
## 5 Wolfwalkers             2020 TRUE
```

```
moore_filmography[, -c(1:2)]
```

```
## # A tibble: 5 x 2
##   director producer
##   <lgl>    <lgl>
## 1 TRUE     TRUE
## 2 TRUE     TRUE
## 3 TRUE     FALSE
## 4 FALSE    TRUE
## 5 TRUE     TRUE
```

```
moore_filmography[, "year"]
```

```
## # A tibble: 5 x 1
##   year
##   <dbl>
## 1 2009
## 2 2014
## 3 2014
## 4 2017
## 5 2020
```



```
moore_filmography[,c("title", "year")]
```

```
## # A tibble: 5 x 2
```

	title	year
##	<chr>	<dbl>
## 1	The Secret of Kells	2009
## 2	Song of the Sea	2014
## 3	Kahlil Gibran's The Prophet	2014
## 4	The Breadwinner	2017
## 5	Wolfwalkers	2020

```
moore_filmography[moore_filmography$year > 2014,]
```

```
## # A tibble: 2 x 4
```

	title	year	director	producer
##	<chr>	<dbl>	<lgl>	<lgl>
## 1	The Breadwinner	2017	FALSE	TRUE
## 2	Wolfwalkers	2020	TRUE	TRUE

## 2.10 Data import

### 2.10.1 .csv files

A .csv files (comma-separated values) is a delimited text file that uses a comma (or other delemeters such as tabulation or semicolon) to separate values. It is broadly used because it is possible to parse such a file using computers and people can edit it in the Office programs (Microsoft Excel, LibreOffice Calc, Numbers on Mac). Here is our `moore_filmography` dataset in the .csv format:

```
title,year,director,producer
The Secret of Kells,2009,TRUE,TRUE
Song of the Sea,2014,TRUE,TRUE
Kahlil Gibran's The Prophet,2014,TRUE,FALSE
The Breadwinner,2017,FALSE,TRUE
Wolfwalkers,2020,TRUE,TRUE
```

Let's create a variable with this file:

```
our_csv <- "title,year,director,producer
The Secret of Kells,2009,TRUE,TRUE
Song of the Sea,2014,TRUE,TRUE
Kahlil Gibran's The Prophet,2014,TRUE,FALSE
The Breadwinner,2017,FALSE,TRUE
Wolfwalkers,2020,TRUE,TRUE"
```

Now we are ready to use `read_csv()` function:

```
read_csv(our_csv)
```

```
## # A tibble: 5 x 4
##   title                year director producer
##   <chr>                <dbl> <lgl>    <lgl>
## 1 The Secret of Kells    2009 TRUE     TRUE
## 2 Song of the Sea        2014 TRUE     TRUE
## 3 Kahlil Gibran's The Prophet 2014 TRUE     FALSE
## 4 The Breadwinner        2017 FALSE    TRUE
## 5 Wolfwalkers            2020 TRUE     TRUE
```

It is also possible to read files from your computer. Download this file on your computer (press `Ctrl S` or `Cmd S`) and read into R:

```
read_csv("C:/path/to/your/file/moore_filmography.csv")
```

```
## # A tibble: 5 x 4
##   title                year director producer
##   <chr>                <dbl> <lgl>    <lgl>
## 1 The Secret of Kells    2009 TRUE     TRUE
## 2 Song of the Sea        2014 TRUE     TRUE
## 3 Kahlil Gibran's The Prophet 2014 TRUE     FALSE
## 4 The Breadwinner        2017 FALSE    TRUE
## 5 Wolfwalkers            2020 TRUE     TRUE
```

It is also possible to read files from the Internet:

```
read_csv("https://raw.githubusercontent.com/agricolamz/2020.02_Naumburg_R/master/data/1")
```

```
## Parsed with column specification:
```

```
## cols(
##   title = col_character(),
##   year = col_double(),
##   director = col_logical(),
##   producer = col_logical()
## )
```

```
## # A tibble: 5 x 4
##   title                year director producer
##   <chr>                <dbl> <lgl>    <lgl>
## 1 The Secret of Kells    2009 TRUE     TRUE
## 2 Song of the Sea        2014 TRUE     TRUE
## 3 Kahlil Gibran's The Prophet 2014 TRUE     FALSE
```

```
## 4 The Breadwinner      2017 FALSE   TRUE
## 5 Wolfwalkers          2020  TRUE   TRUE
```



Because of the 2019–20 Wuhan coronavirus outbreak the city of Wuhan is on media everywhere. In Russian for some reason Wuhan is sometimes masculine and sometimes it is feminine. I looked into other Slavic languages and recorded obtained data into the `.csv` file. Download this files to R. What variables does it have?

All file manipulations in R are somehow connected with space on your computer via working directory. You can get information about your current working directory using `getwd()` function. You can change your working directory using `setwd()` function. If a file you want to read is in the working directory you don't need to write the whole path to file:

```
read_csv("moore_filmography.csv")
```

The same simple function will create your `.csv` file:

```
write_csv(moore_filmography, "moore_filmography_v2.csv")
```

Sometimes reading `.csv` files into Microsoft Excel is complicated, please follow the following instructions.

### 2.10.2 `.xls` and `.xlsx` files

There is a package `readxl` that allows to open and save `.xls` and `.xlsx` files. Install and load the package:

```
library(readxl)
```

Here is a test file. Download it to your computer and put it to your working directory:

```
read_xlsx("moore_filmography.xlsx")
```

```
## # A tibble: 5 x 4
##   title                year director producer
##   <chr>                <dbl> <chr>    <chr>
## 1 The Secret of Kells  2009 TRUE     TRUE
## 2 Song of the Sea      2014 TRUE     TRUE
## 3 Kahlil Gibran's The Prophet 2014 TRUE     FALSE
## 4 The Breadwinner      2017 FALSE    TRUE
## 5 Wolfwalkers          2020 TRUE     TRUE
```

.xls and .xlsx files could have multiple tables on different sheets:

```
read_xlsx("moore_filmography.xlsx", sheet = "iris")
```

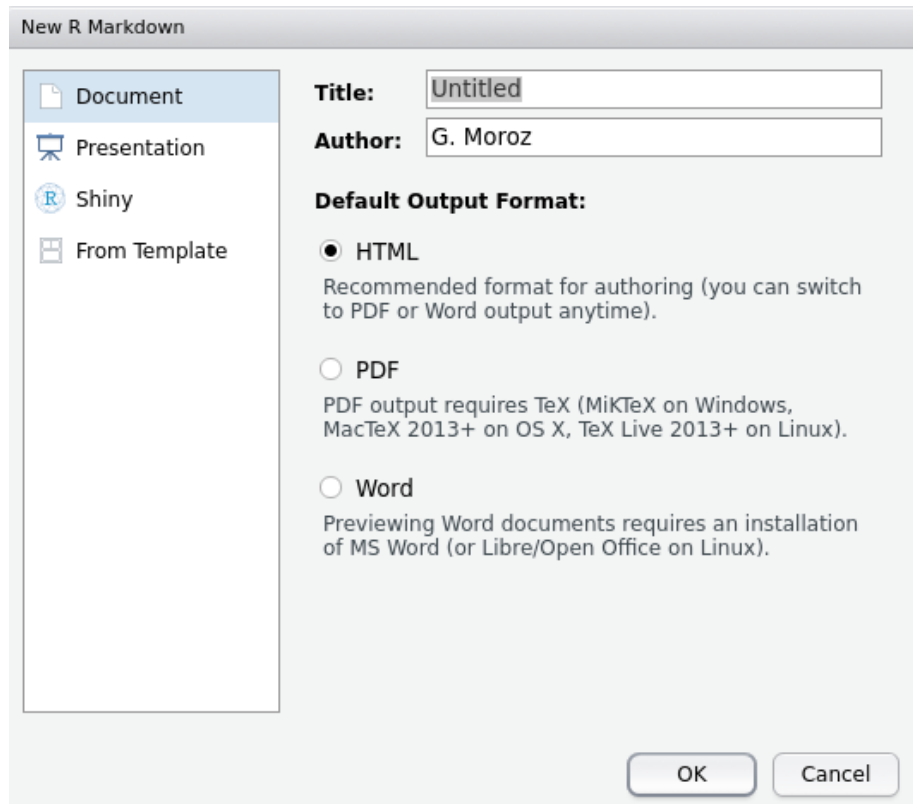
```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##         <dbl>         <dbl>         <dbl>         <dbl> <chr>
## 1         5.1         3.5         1.4         0.2 setosa
## 2         4.9         3         1.4         0.2 setosa
## 3         4.7         3.2         1.3         0.2 setosa
## 4         4.6         3.1         1.5         0.2 setosa
## 5         5         3.6         1.4         0.2 setosa
## 6         5.4         3.9         1.7         0.4 setosa
## 7         4.6         3.4         1.4         0.3 setosa
## 8         5         3.4         1.5         0.2 setosa
## 9         4.4         2.9         1.4         0.2 setosa
## 10        4.9         3.1         1.5         0.1 setosa
## # ... with 140 more rows
```

## 2.11 Rmarkdown


If you press **Ctrl S** or **Cmd S** then you will save your script. There is also another useful type of coding in R: **rmarkown**. First install this package:

```
install.packages("rmarkdown")
```

Then it will be possible to create a new file: **File > New File > R Markdown....**



Press OK in the following menu and you will get the template of your R Mark-

down file. You can modify it, then press  and the result file will be created in your working directory. `rmarkdown` package is a really popular and well developed package that creates output into:

- markdown
- html
- docx
- pdf
- beamer presentation
- pptx presentation
- epub
- ...
- multiple templates for different scientific journals (package `rticles` and `papaja`)
- ...



## Chapter 3

# Data manipulation: dplyr

First, load the library:

```
library(tidyverse)
```

### 3.1 Data

In this chapter we will use the following datasets.

#### 3.1.1 Misspelling dataset

This dataset I gathered after some manipulations with data from The Gyllenhaal Experiment By Russell Goldenberg and Matt Daniels for pudding. They analysed mistakes in spellings of celebrities during the search.

```
misspellings <- read_csv("https://raw.githubusercontent.com/agricolamz/2020.02_Naumburg_R/master/
```

```
## Parsed with column specification:
## cols(
##   correct = col_character(),
##   spelling = col_character(),
##   count = col_double()
## )
```

```
misspellings
```

```
## # A tibble: 15,477 x 3
##   correct  spelling    count
```

```
##      <chr>      <chr>      <dbl>
##  1 deschanel  deschanel  18338
##  2 deschanel  dechanel   1550
##  3 deschanel  deschannel  934
##  4 deschanel  deschenel   404
##  5 deschanel  deshanel   364
##  6 deschanel  dechannel   359
##  7 deschanel  deschanelle 316
##  8 deschanel  dechanelle  192
##  9 deschanel  deschanell  174
## 10 deschanel  deschenal   165
## # ... with 15,467 more rows
```

There are the following variables in this dataset:

- `correct` — correct spelling
- `spelling` — user's spelling
- `count` — number of cases of user's spelling

### 3.1.2 diamonds

`diamonds` — is the dataset built-in in `tidyverse` package.

```
diamonds
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23 Ideal    E     SI2    61.5   55   326  3.95  3.98  2.43
##  2 0.21 Premium  E     SI1    59.8   61   326  3.89  3.84  2.31
##  3 0.23 Good     E     VS1    56.9   65   327  4.05  4.07  2.31
##  4 0.290 Premium  I     VS2    62.4   58   334  4.2   4.23  2.63
##  5 0.31 Good     J     SI2    63.3   58   335  4.34  4.35  2.75
##  6 0.24 Very Good J     VVS2   62.8   57   336  3.94  3.96  2.48
##  7 0.24 Very Good I     VVS1   62.3   57   336  3.95  3.98  2.47
##  8 0.26 Very Good H     SI1    61.9   55   337  4.07  4.11  2.53
##  9 0.22 Fair     E     VS2    65.1   61   337  3.87  3.78  2.49
## 10 0.23 Very Good H     VS1    59.4   61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

```
?diamonds
```

## 3.2 dplyr

Here and here is a cheatsheet on `dplyr`.



### 3.2.1 filter()



This function filter rows by some condition.

How many wrong spellings that were used by less then 10 users?

```
misspellings %>%
  filter(count < 10)
```

```
## # A tibble: 14,279 x 3
##   correct spelling count
##   <chr>    <chr>    <dbl>
## 1 deschanel deshanael      9
## 2 deschanel daychanel      9
## 3 deschanel deschaneles    9
## 4 deschanel dashenel      9
## 5 deschanel deschenael     9
## 6 deschanel deechanel      9
## 7 deschanel deichanel      9
## 8 deschanel dechantel      9
## 9 deschanel deychanel      9
## 10 deschanel daschenell     9
## # ... with 14,269 more rows
```

%>% it is **pipe**. It allow to chain operations, puting the output of one function into the input of another:

```
sort(sqrt(abs(sin(1:22))), decreasing = TRUE)
```

```
## [1] 0.9999951 0.9952926 0.9946649 0.9805088 0.9792468 0.9554817 0.9535709
## [8] 0.9173173 0.9146888 0.8699440 0.8665952 0.8105471 0.8064043 0.7375779
## [15] 0.7325114 0.6482029 0.6419646 0.5365662 0.5285977 0.3871398 0.3756594
## [22] 0.0940814
```

```
1:22 %>%
  sin() %>%
  abs() %>%
  sqrt() %>%
  sort(., decreasing = TRUE) # why do we need a dot here?
```

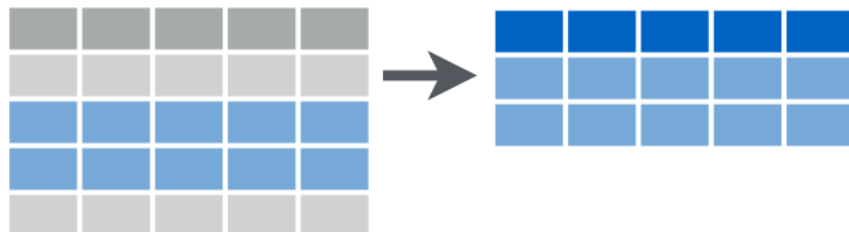
```
## [1] 0.9999951 0.9952926 0.9946649 0.9805088 0.9792468 0.9554817 0.9535709
## [8] 0.9173173 0.9146888 0.8699440 0.8665952 0.8105471 0.8064043 0.7375779
## [15] 0.7325114 0.6482029 0.6419646 0.5365662 0.5285977 0.3871398 0.3756594
## [22] 0.0940814
```

Pipes that are used in `tidyverse` are from the package `magrittr`. Sometimes pipe could work not well with functions outside the `tidyverse`.



### 3.2.2 slice()

This function filter rows by its index.



```
misspellings %>%
  slice(3:7)
```

```
## # A tibble: 5 x 3
##   correct spelling count
##   <chr>    <chr>   <dbl>
## 1 deschanel deschannel  934
## 2 deschanel deschenel  404
```

```
## 3 deschannel deshanel      364
## 4 deschannel dechannel     359
## 5 deschannel deschanelle   316
```

### 3.2.3 select()

This functions for choosing variables from dataframe.



```
diamonds %>%
  select(8:10)
```

```
## # A tibble: 53,940 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1  3.95  3.98  2.43
## 2  3.89  3.84  2.31
## 3  4.05  4.07  2.31
## 4  4.2   4.23  2.63
## 5  4.34  4.35  2.75
## 6  3.94  3.96  2.48
## 7  3.95  3.98  2.47
## 8  4.07  4.11  2.53
## 9  3.87  3.78  2.49
## 10 4     4.05  2.39
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(color:price)
```

```
## # A tibble: 53,940 x 5
##   color clarity depth table price
##   <ord> <ord>   <dbl> <dbl> <int>
## 1 E     SI2     61.5    55    326
## 2 E     SI1     59.8    61    326
```

```
## 3 E      VS1      56.9    65    327
## 4 I      VS2      62.4    58    334
## 5 J      SI2      63.3    58    335
## 6 J      VVS2     62.8    57    336
## 7 I      VVS1     62.3    57    336
## 8 H      SI1      61.9    55    337
## 9 E      VS2      65.1    61    337
## 10 H     VS1      59.4    61    338
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(-carat)
```

```
## # A tibble: 53,940 x 9
##   cut      color clarity depth table price      x      y      z
##   <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 Ideal    E      SI2     61.5    55    326  3.95  3.98  2.43
## 2 Premium E      SI1     59.8    61    326  3.89  3.84  2.31
## 3 Good     E      VS1     56.9    65    327  4.05  4.07  2.31
## 4 Premium I      VS2     62.4    58    334  4.2   4.23  2.63
## 5 Good     J      SI2     63.3    58    335  4.34  4.35  2.75
## 6 Very Good J      VVS2     62.8    57    336  3.94  3.96  2.48
## 7 Very Good I      VVS1     62.3    57    336  3.95  3.98  2.47
## 8 Very Good H      SI1     61.9    55    337  4.07  4.11  2.53
## 9 Fair     E      VS2     65.1    61    337  3.87  3.78  2.49
## 10 Very Good H      VS1     59.4    61    338  4     4.05  2.39
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(-c(carat, cut, x, y, z))
```

```
## # A tibble: 53,940 x 5
##   color clarity depth table price
##   <ord> <ord>   <dbl> <dbl> <int>
## 1 E      SI2     61.5    55    326
## 2 E      SI1     59.8    61    326
## 3 E      VS1     56.9    65    327
## 4 I      VS2     62.4    58    334
## 5 J      SI2     63.3    58    335
## 6 J      VVS2     62.8    57    336
## 7 I      VVS1     62.3    57    336
## 8 H      SI1     61.9    55    337
## 9 E      VS2     65.1    61    337
## 10 H     VS1     59.4    61    338
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(cut, depth, price)
```

```
## # A tibble: 53,940 x 3
##   cut      depth price
##   <ord>    <dbl> <int>
## 1 Ideal      61.5   326
## 2 Premium    59.8   326
## 3 Good       56.9   327
## 4 Premium    62.4   334
## 5 Good       63.3   335
## 6 Very Good  62.8   336
## 7 Very Good  62.3   336
## 8 Very Good  61.9   337
## 9 Fair       65.1   337
## 10 Very Good 59.4   338
## # ... with 53,930 more rows
```

### 3.2.4 arrange()

This function order rows in dataframe (numbers — by order, strings — alphabetically).

```
misspellings %>%
  arrange(count)
```

```
## # A tibble: 15,477 x 3
##   correct spelling count
##   <chr>    <chr>    <dbl>
## 1 deschanel deschil      1
## 2 deschanel deshauneil    1
## 3 deschanel deshmuel     1
## 4 deschanel deshannle    1
## 5 deschanel deslanges    1
## 6 deschanel deshoenel    1
## 7 deschanel dechadel     1
## 8 deschanel dooschaney    1
## 9 deschanel dishana      1
## 10 deschanel deshaneil    1
## # ... with 15,467 more rows
```

```
diamonds %>%
  arrange(desc(carat), price)
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  5.01 Fair      J      I1      65.5   59 18018 10.7  10.5  6.98
## 2  4.5  Fair      J      I1      65.8   58 18531 10.2  10.2  6.72
## 3  4.13 Fair      H      I1      64.8   61 17329 10    9.85  6.43
## 4  4.01 Premium   I      I1      61     61 15223 10.1  10.1  6.17
## 5  4.01 Premium   J      I1      62.5   62 15223 10.0  9.94  6.24
## 6  4    Very Good I      I1      63.3   58 15984 10.0  9.94  6.31
## 7  3.67 Premium   I      I1      62.4   56 16193 9.86  9.81  6.13
## 8  3.65 Fair      H      I1      67.1   53 11668 9.53  9.48  6.38
## 9  3.51 Premium   J      VS2      62.5   59 18701 9.66  9.63  6.03
## 10 3.5  Ideal      H      I1      62.8   57 12587 9.65  9.59  6.03
## # ... with 53,930 more rows
```

```
diamonds %>%
  arrange(-carat, price)
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  5.01 Fair      J      I1      65.5   59 18018 10.7  10.5  6.98
## 2  4.5  Fair      J      I1      65.8   58 18531 10.2  10.2  6.72
## 3  4.13 Fair      H      I1      64.8   61 17329 10    9.85  6.43
## 4  4.01 Premium   I      I1      61     61 15223 10.1  10.1  6.17
## 5  4.01 Premium   J      I1      62.5   62 15223 10.0  9.94  6.24
## 6  4    Very Good I      I1      63.3   58 15984 10.0  9.94  6.31
## 7  3.67 Premium   I      I1      62.4   56 16193 9.86  9.81  6.13
## 8  3.65 Fair      H      I1      67.1   53 11668 9.53  9.48  6.38
## 9  3.51 Premium   J      VS2      62.5   59 18701 9.66  9.63  6.03
## 10 3.5  Ideal      H      I1      62.8   57 12587 9.65  9.59  6.03
## # ... with 53,930 more rows
```

### 3.2.5 distinct()

This function return only unique rows from an input dataframe.

```
misspellings %>%
  distinct(correct)
```

```
## # A tibble: 15 x 1
##   correct
##   <chr>
## 1 deschanel
## 2 mclachlan
```

```
## 3 galifianakis
## 4 labeouf
## 5 macaulay
## 6 mcconaughey
## 7 minaj
## 8 morissette
## 9 poehler
## 10 shyamalan
## 11 kaepernick
## 12 mcgwire
## 13 palahniuk
## 14 picabo
## 15 johansson
```

```
misspellings %>%
  distinct(spelling)
```

```
## # A tibble: 15,462 x 1
##   spelling
##   <chr>
## 1 deschanel
## 2 dechanel
## 3 deschannel
## 4 deschenel
## 5 deshanel
## 6 dechannel
## 7 deschanelle
## 8 dechanelle
## 9 deschanell
## 10 deschenal
## # ... with 15,452 more rows
```

```
diamonds %>%
  distinct(color, cut)
```

```
## # A tibble: 35 x 2
##   color cut
##   <ord> <ord>
## 1 E     Ideal
## 2 E     Premium
## 3 E     Good
## 4 I     Premium
## 5 J     Good
## 6 J     Very Good
## 7 I     Very Good
```

```
## 8 H      Very Good
## 9 E      Fair
## 10 J     Ideal
## # ... with 25 more rows
```



In built-in dataset `starwars` filter those characters that are higher then 180 (`height`) and weigh less then 80 (`mass`). Then get a unique names of their homeworlds (`homeworld`).

### 3.2.6 `mutate()`

This functions creates new variables.



```
misspellings %>%
  mutate(misspelling_length = nchar(spelling),
         id = 1:n())
```

```
## # A tibble: 15,477 x 5
##   correct spelling    count misspelling_length  id
##   <chr>    <chr>    <dbl>             <int> <int>
## 1 deschanel deschanel 18338             9      1
## 2 deschanel dechannel 1550              8      2
## 3 deschanel deschannel 934             10      3
## 4 deschanel deschenel 404              9      4
## 5 deschanel deshanel 364              8      5
## 6 deschanel dechannel 359              9      6
## 7 deschanel deschanelle 316             11      7
## 8 deschanel dechanelle 192             10      8
## 9 deschanel deschanell 174             10      9
## 10 deschanel deschenal 165              9     10
## # ... with 15,467 more rows
```



Create a variable with body mass index  $\text{BMI} = \frac{\text{mass}}{\text{height}^2}$  for all characters from `starwars` dataset. How many charachters have obesity (have body mass index greater 30)? (Don't forget to convert height from centimetres to metres).



### 3.2.7 `group_by(...)` `%>% summarise(...)`

This function allows to group variables by some columns and get some descriptive statistics (maximum, minimum, last value, first value, mean, median etc.)



```
misspellings %>%
  summarise(min(count), mean(count))
```

```
## # A tibble: 1 x 2
##   `min(count)` `mean(count)`
##       <dbl>       <dbl>
## 1           1         21.8
```

```
misspellings %>%
  group_by(correct) %>%
  summarise(mean(count))
```

```
## # A tibble: 15 x 2
##   correct `mean(count)`
##   <chr>       <dbl>
## 1 deschanel      25.9
## 2 galifianakis    8.64
## 3 johansson      74.8
## 4 kaepernick     29.1
## 5 labeouf        61.2
## 6 macaulay       17.6
## 7 mcconaughey     7.74
## 8 mcgwire        55.3
## 9 mclachlan      14.8
## 10 minaj         140.
## 11 morissette    55.2
## 12 palahniuk     10.2
## 13 picabo        23.2
## 14 poehler       65.3
## 15 shyamalan     16.9
```

```
misspellings %>%
  group_by(correct) %>%
  summarise(my_mean = mean(count))
```

```
## # A tibble: 15 x 2
##   correct      my_mean
##   <chr>      <dbl>
## 1 deschanel    25.9
## 2 galifianakis  8.64
## 3 johansson    74.8
## 4 kaepernick   29.1
## 5 labeouf      61.2
## 6 macaulay     17.6
## 7 mcconaughey   7.74
## 8 mcgwire      55.3
## 9 mclachlan    14.8
## 10 minaj       140.
## 11 morissette  55.2
## 12 palahniuk    10.2
## 13 picabo      23.2
## 14 poehler     65.3
## 15 shyamalan    16.9
```

If you need to calculate number of cases, use the function `n()` in `summarise()` or the `count()` function:

```
misspellings %>%
  group_by(correct) %>%
  summarise(n = n())
```

```
## # A tibble: 15 x 2
##   correct      n
##   <chr>    <int>
## 1 deschanel  1015
## 2 galifianakis 2633
## 3 johansson   392
## 4 kaepernick   779
## 5 labeouf     449
## 6 macaulay    1458
## 7 mcconaughey 2897
## 8 mcgwire     262
## 9 mclachlan   1054
## 10 minaj      200
## 11 morissette  478
## 12 palahniuk  1541
```

```
## 13 picabo          460
## 14 poehler         386
## 15 shyamalan       1473
```

```
misspellings %>%
  count(correct)
```

```
## # A tibble: 15 x 2
##   correct      n
##   <chr>      <int>
## 1 deschanel  1015
## 2 galifianakis 2633
## 3 johansson   392
## 4 kaepernick   779
## 5 labeouf      449
## 6 macaulay    1458
## 7 mcconaughey 2897
## 8 mcgwire      262
## 9 mclachlan   1054
## 10 minaj       200
## 11 morissette  478
## 12 palahniuk   1541
## 13 picabo      460
## 14 poehler     386
## 15 shyamalan   1473
```

It is even possible to sort the result, using `sort` argument:

```
misspellings %>%
  count(correct, sort = TRUE)
```

```
## # A tibble: 15 x 2
##   correct      n
##   <chr>      <int>
## 1 mcconaughey 2897
## 2 galifianakis 2633
## 3 palahniuk   1541
## 4 shyamalan   1473
## 5 macaulay    1458
## 6 mclachlan   1054
## 7 deschanel   1015
## 8 kaepernick   779
## 9 morissette  478
## 10 picabo      460
## 11 labeouf      449
```

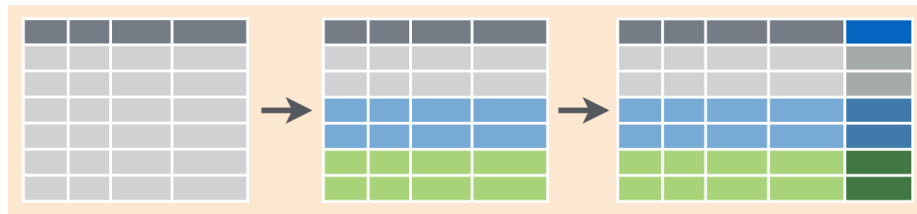
```
## 12 johansson      392
## 13 poehler        386
## 14 mcgwire         262
## 15 minaj           200
```

In case you don't want to have any summary, but an additional column, just replace `summarise()` with `mutate()`

```
misspellings %>%
  group_by(correct) %>%
  mutate(my_mean = mean(count))
```

```
## # A tibble: 15,477 x 4
## # Groups:   correct [15]
##   correct spelling count my_mean
##   <chr>    <chr>    <dbl> <dbl>
## 1 deschanel deschanel 18338 25.9
## 2 deschanel dechannel 1550 25.9
## 3 deschanel deschannel 934 25.9
## 4 deschanel deschenel 404 25.9
## 5 deschanel deshanel 364 25.9
## 6 deschanel dechannel 359 25.9
## 7 deschanel deschanelle 316 25.9
## 8 deschanel dechanelle 192 25.9
## 9 deschanel deschanell 174 25.9
## 10 deschanel deschenal 165 25.9
## # ... with 15,467 more rows
```

Here is a scheme:



In the `starwars` dataset create a variable that contain mean height value for each species.

## 3.3 Merging dataframes

### 3.3.1 `bind_...`

This is a family of functions that make it possible to merge dataframes together:

```
my_tbl <- tibble(a = c(1, 5, 2),
                 b = c("e", "g", "s"))
```

Here is how to merge two datasets by row:

```
my_tbl %>%
  bind_rows(my_tbl)
```

```
## # A tibble: 6 x 2
##       a b
##   <dbl> <chr>
## 1     1 e
## 2     5 g
## 3     2 s
## 4     1 e
## 5     5 g
## 6     2 s
```

In case there is an absent column, values will be filled with NA:

```
my_tbl %>%
  bind_rows(my_tbl[, -1])
```

```
## # A tibble: 6 x 2
##       a b
##   <dbl> <chr>
## 1     1 e
## 2     5 g
## 3     2 s
## 4    NA e
## 5    NA g
## 6    NA s
```

In order to merge dataframes by column you need another function:

```
my_tbl %>%
  bind_cols(my_tbl)
```

```
## # A tibble: 3 x 4
##       a b       a1 b1
##   <dbl> <chr> <dbl> <chr>
## 1     1 e         1 e
## 2     5 g         5 g
## 3     2 s         2 s
```

In case there is an absent row, this function will return an error:

```
my_tbl %>%
  bind_cols(my_tbl[-1,])
```

```
## Error: Argument 2 must be length 3, not 2
```

### 3.3.2 `.._join()`

These functions allow to merge different datasets by some column (or columns in common).

```
languages <- data_frame(
  languages = c("Selkup", "French", "Chukchi", "Polish"),
  countries = c("Russia", "France", "Russia", "Poland"),
  iso = c("sel", "fra", "ckt", "pol")
)
languages
```

```
## # A tibble: 4 x 3
##   languages countries iso
##   <chr>      <chr>   <chr>
## 1 Selkup    Russia   sel
## 2 French    France   fra
## 3 Chukchi   Russia   ckt
## 4 Polish    Poland   pol
```

```
country_population <- data_frame(
  countries = c("Russia", "Poland", "Finland"),
  population_mln = c(143, 38, 5))
country_population
```

```
## # A tibble: 3 x 2
##   countries population_mln
##   <chr>          <dbl>
## 1 Russia          143
## 2 Poland           38
## 3 Finland           5
```

```
inner_join(languages, country_population)
```

```
## Joining, by = "countries"
```

```
## # A tibble: 3 x 4
```

```
## languages countries iso population_mln
## <chr> <chr> <chr> <dbl>
## 1 Selkup Russia sel 143
## 2 Chukchi Russia ckt 143
## 3 Polish Poland pol 38
```

```
left_join(languages, country_population)
```

```
## Joining, by = "countries"
## # A tibble: 4 x 4
## languages countries iso population_mln
## <chr> <chr> <chr> <dbl>
## 1 Selkup Russia sel 143
## 2 French France fra NA
## 3 Chukchi Russia ckt 143
## 4 Polish Poland pol 38
```

```
right_join(languages, country_population)
```

```
## Joining, by = "countries"
## # A tibble: 4 x 4
## languages countries iso population_mln
## <chr> <chr> <chr> <dbl>
## 1 Selkup Russia sel 143
## 2 Chukchi Russia ckt 143
## 3 Polish Poland pol 38
## 4 <NA> Finland <NA> 5
```

```
anti_join(languages, country_population)
```

```
## Joining, by = "countries"
## # A tibble: 1 x 3
## languages countries iso
## <chr> <chr> <chr>
## 1 French France fra
```

```
anti_join(country_population, languages)
```

```
## Joining, by = "countries"
## # A tibble: 1 x 2
## countries population_mln
```

```
##   <chr>                <dbl>
## 1 Finland                5
```

```
full_join(country_population, languages)
```

```
## Joining, by = "countries"
```

```
## # A tibble: 5 x 4
##   countries population_mln languages iso
##   <chr>          <dbl> <chr>    <chr>
## 1 Russia          143 Selkup    sel
## 2 Russia          143 Chukchi   ckt
## 3 Poland           38 Polish    pol
## 4 Finland           5 <NA>     <NA>
## 5 France           NA French    fra
```

a

x1	x2
A	1
B	2
C	3

+
b

x1	x3
A	T
B	F
D	T

=

### Mutating Joins

<table border="1" style="width: 100%; text-align: center;"> <thead><tr><th>x1</th><th>x2</th><th>x3</th></tr></thead> <tbody> <tr><td>A</td><td>1</td><td>T</td></tr> <tr><td>B</td><td>2</td><td>F</td></tr> <tr><td>C</td><td>3</td><td>NA</td></tr> </tbody> </table>	x1	x2	x3	A	1	T	B	2	F	C	3	NA	<p><b>dplyr::left_join(a, b, by = "x1")</b> Join matching rows from b to a.</p>			
x1	x2	x3														
A	1	T														
B	2	F														
C	3	NA														
<table border="1" style="width: 100%; text-align: center;"> <thead><tr><th>x1</th><th>x3</th><th>x2</th></tr></thead> <tbody> <tr><td>A</td><td>T</td><td>1</td></tr> <tr><td>B</td><td>F</td><td>2</td></tr> <tr><td>D</td><td>T</td><td>NA</td></tr> </tbody> </table>	x1	x3	x2	A	T	1	B	F	2	D	T	NA	<p><b>dplyr::right_join(a, b, by = "x1")</b> Join matching rows from a to b.</p>			
x1	x3	x2														
A	T	1														
B	F	2														
D	T	NA														
<table border="1" style="width: 100%; text-align: center;"> <thead><tr><th>x1</th><th>x2</th><th>x3</th></tr></thead> <tbody> <tr><td>A</td><td>1</td><td>T</td></tr> <tr><td>B</td><td>2</td><td>F</td></tr> </tbody> </table>	x1	x2	x3	A	1	T	B	2	F	<p><b>dplyr::inner_join(a, b, by = "x1")</b> Join data. Retain only rows in both sets.</p>						
x1	x2	x3														
A	1	T														
B	2	F														
<table border="1" style="width: 100%; text-align: center;"> <thead><tr><th>x1</th><th>x2</th><th>x3</th></tr></thead> <tbody> <tr><td>A</td><td>1</td><td>T</td></tr> <tr><td>B</td><td>2</td><td>F</td></tr> <tr><td>C</td><td>3</td><td>NA</td></tr> <tr><td>D</td><td>NA</td><td>T</td></tr> </tbody> </table>	x1	x2	x3	A	1	T	B	2	F	C	3	NA	D	NA	T	<p><b>dplyr::full_join(a, b, by = "x1")</b> Join data. Retain all values, all rows.</p>
x1	x2	x3														
A	1	T														
B	2	F														
C	3	NA														
D	NA	T														



### 3.4 tidyR package

Here is a dataset with number of speakers of some language of India according census 2001 (data from Wikipedia):

```
langs_in_india_short <- read_csv("https://raw.githubusercontent.com/agricolamz/2020.02_Naumburg_F")
```

```
## Parsed with column specification:
## cols(
##   language = col_character(),
##   n_L1_sp = col_double(),
##   n_L2_sp = col_double(),
##   n_L3_sp = col_double(),
##   n_all_sp = col_double()
## )
```

- Short format

```
langs_in_india_short
```

```
## # A tibble: 12 x 5
##   language    n_L1_sp n_L2_sp n_L3_sp n_all_sp
##   <chr>      <dbl>   <dbl>   <dbl>   <dbl>
## 1 Hindi      422048642 98207180 31160696 551416518
## 2 English      226449 86125221 38993066 125344736
## 3 Bengali      83369769 6637222 1108088 91115079
## 4 Telugu       74002856 9723626 1266019 84992501
## 5 Marathi      71936894 9546414 2701498 84184806
## 6 Tamil        60793814 4992253 956335 66742402
## 7 Urdu         51536111 6535489 1007912 59079512
## 8 Kannada      37924011 11455287 1396428 50775726
## 9 Gujarati     46091617 3476355 703989 50271961
## 10 Odia        33017446 3272151 319525 36609122
## 11 Malayalam   33066392 499188 195885 33761465
## 12 Sanskrit     14135 1234931 3742223 4991289
```

- Long format

```
## # A tibble: 48 x 3
##   language type    n_speakers
##   <chr>   <chr>      <dbl>
## 1 Hindi   n_L1_sp    422048642
## 2 Hindi   n_L2_sp     98207180
## 3 Hindi   n_L3_sp     31160696
## 4 Hindi   n_all_sp    551416518
## 5 English n_L1_sp      226449
```

```
## 6 English n_L2_sp 86125221
## 7 English n_L3_sp 38993066
## 8 English n_all_sp 125344736
## 9 Bengali n_L1_sp 83369769
## 10 Bengali n_L2_sp 6637222
## # ... with 38 more rows
```

- Short format → Long format: `tidyr::pivot_longer()`

```
langs_in_india_short %>%
  pivot_longer(names_to = "type", values_to = "n_speakers", n_L1_sp:n_all_sp)->
  langs_in_india_long

langs_in_india_long
```

```
## # A tibble: 48 x 3
##   language type      n_speakers
##   <chr>      <chr>      <dbl>
## 1 Hindi     n_L1_sp    422048642
## 2 Hindi     n_L2_sp    98207180
## 3 Hindi     n_L3_sp    31160696
## 4 Hindi     n_all_sp   551416518
## 5 English   n_L1_sp     226449
## 6 English   n_L2_sp    86125221
## 7 English   n_L3_sp    38993066
## 8 English   n_all_sp   125344736
## 9 Bengali   n_L1_sp    83369769
## 10 Bengali  n_L2_sp    6637222
## # ... with 38 more rows
```

- Long format → Short format: `tidyr::pivot_wider()`

```
langs_in_india_long %>%
  pivot_wider(names_from = "type", values_from = "n_speakers")->
  langs_in_india_short

langs_in_india_short
```

```
## # A tibble: 12 x 5
##   language  n_L1_sp n_L2_sp n_L3_sp n_all_sp
##   <chr>      <dbl>   <dbl>   <dbl>   <dbl>
## 1 Hindi    422048642 98207180 31160696 551416518
## 2 English    226449 86125221 38993066 125344736
## 3 Bengali   83369769 6637222 1108088 91115079
## 4 Telugu    74002856 9723626 1266019 84992501
## 5 Marathi   71936894 9546414 2701498 84184806
## 6 Tamil     60793814 4992253 956335 66742402
```

```
## 7 Urdu      51536111 6535489 1007912 59079512
## 8 Kannada   37924011 11455287 1396428 50775726
## 9 Gujarati  46091617 3476355 703989 50271961
## 10 Odia     33017446 3272151 319525 36609122
## 11 Malayalam 33066392 499188 195885 33761465
## 12 Sanskrit 14135 1234931 3742223 4991289
```



Here is data, that contain information about villages of Daghestan in .xlsx format. Data separated by different sheets and contain the following variables (data obtained from different sources, so they have suffixes `_s1` – first source and `_s2` – second source):

- `id_s1` – (s1) identification number from first source;
- `name_1885` – (s1) name of the village according the 1885 census
- `census_1885` – (s1) population according the 1885 census
- `name_1895` – (s1) name of the village according the 1895 census
- `census_1895` – (s1) population according the 1895 census
- `name_1926` – (s1) name of the village according the 1926 census
- `census_1926` – (s1) population according the 1926 census
- `name_2010` – (s1) name of the village according the 2010 census
- `census_2010` – (s1) population according the 2010 census
- `language_s1` – (s1) language name according the first source
- `name_s2` – (s2) village name according the second source
- `language_s2` – (s2) language name according the second source
- `Lat` – (s2) latitude
- `Lon` – (s2) longitude
- `elevation` – (s2) altitude

First, merge all sheets fromt the .xlsx file:

```
## # A tibble: 6 x 15
##   id_s1 name_1885 census_1885 name_1895 census_1895 name_1926 language_s1
##   <dbl> <chr>         <dbl> <chr>         <dbl> <chr>         <chr>
## 1    15      (...         122      (...         141      Avar
## 2    17      ...         169      ...         190      ... Avar
## 3    19      ...         102      ...         97      Avar
## 4    21      - ...         581      ...         550      ... Avar
## 5    23      ...         159      ...         137      ... Avar
## 6    25      ( ...         557      ( ...         595      Avar
## # ... with 8 more variables: census_1926 <dbl>, name_2010 <chr>,
## #   census_2010 <dbl>, name_s2 <chr>, language_s2 <chr>, Lat <dbl>, Lon <dbl>,
## #   elevation <dbl>
```



Second, caculate how many times language name is the same in both sources.



Third, calculate mena altitude for languages from the first source. Which is the highest?



Fourth, calculate population for languages from the second source in each census. Show the values obtained for the Lak language:

```
## # A tibble: 25 x 5
##   language_s2 `s_1885` <- sum(...) `s_1895` <- sum(...) `s_1926` <- sum(...)
##   <chr>          <dbl>          <dbl>          <dbl>
## 1 Aghul             6577             6813             7886
## 2 Akhvakh           3535             3229             2697
## 3 Andi              4600             4543             4583
## 4 Archi              804              765              126
## 5 Avar             110191           123363           103565
## 6 Bagvalal          2807             2625             3049
## 7 Bezhta            2330             2546             1270
## 8 Botlikh           1383             1323             1346
## 9 Chamalal          3731             3742             2714
## 10 Chechen           396              344              524
## # ... with 15 more rows, and 1 more variable: `s_2010` <- sum(census_2010)` <dbl>
```

## Chapter 4

# Data visualisation: ggplot2

```
library("tidyverse")
```

### 4.1 Why visualise data?

#### 4.1.1 The Anscombe's Quartet

In Anscombe, F. J. (1973). “Graphs in Statistical Analysis” there were the following dataset:

```
quartet <- read_csv("https://raw.githubusercontent.com/agricolamz/2020.02_Naumburg_R/master/data/quartet")
```

```
## # A tibble: 44 x 4
##       id dataset      x      y
##   <dbl>   <dbl> <dbl> <dbl>
## 1     1     1      1    10  8.04
## 2     2     1      2    10  9.14
## 3     3     1      3    10  7.46
## 4     4     1      4     8  6.58
## 5     5     2      1     8  6.95
## 6     6     2      2     8  8.14
## 7     7     2      3     8  6.77
## 8     8     2      4     8  5.76
## 9     9     3      1    13  7.58
## 10    10     3      2    13  8.74
## # ... with 34 more rows
```

```

quartet %>%
  group_by(dataset) %>%
  summarise(mean_X = mean(x),
            mean_Y = mean(y),
            sd_X = sd(x),
            sd_Y = sd(y),
            cor = cor(x, y),
            n_obs = n()) %>%
  select(-dataset) %>%
  round(2)

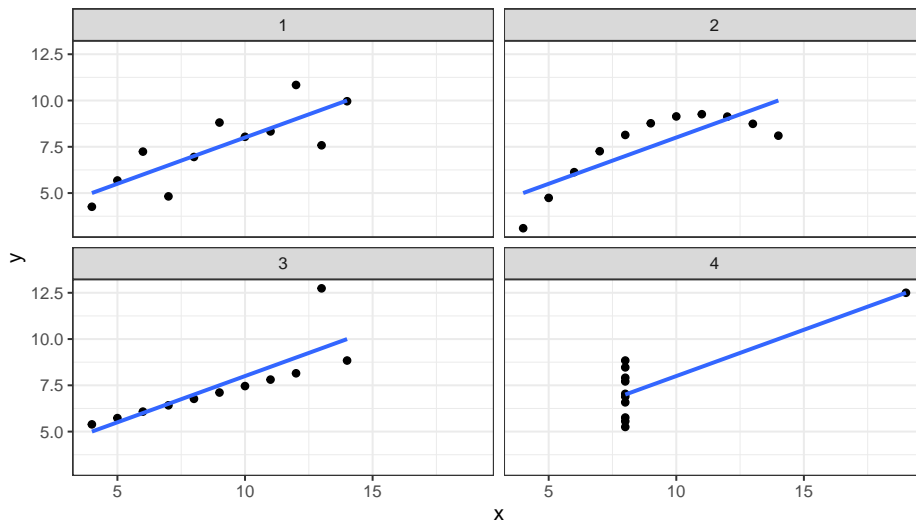
```

```

## # A tibble: 4 x 6
##   mean_X mean_Y sd_X sd_Y cor n_obs
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     9     7.5  3.32  2.03  0.82    11
## 2     9     7.5  3.32  2.03  0.82    11
## 3     9     7.5  3.32  2.03  0.82    11
## 4     9     7.5  3.32  2.03  0.82    11

```

Lets visualise those datasets:



### 4.1.2 The DataSaurus

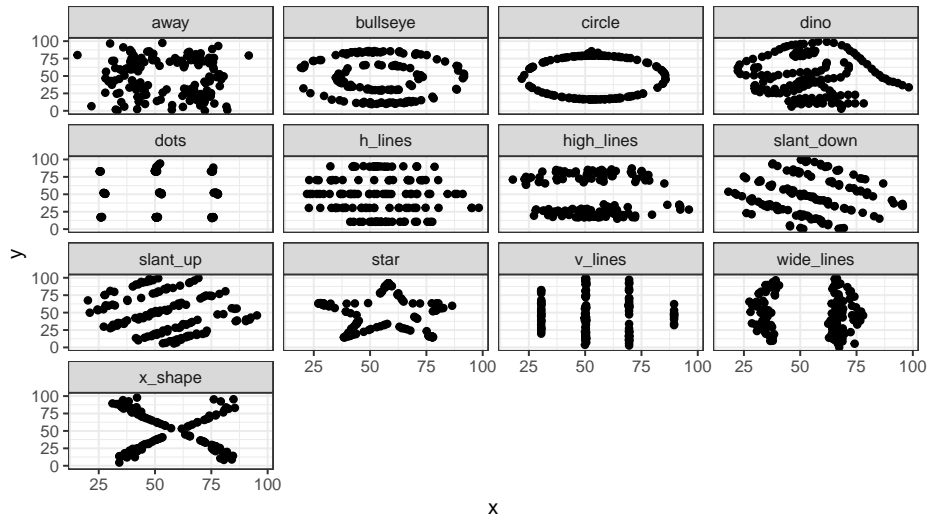
In Matejka and Fitzmaurice (2017) “Same Stats, Different Graphs” there are the following datasets:

```

datasaurus <- read_csv("https://raw.githubusercontent.com/agricolamz/2020.02_Naumburg_1")
datasaurus

```

```
## # A tibble: 1,846 x 3
##   dataset      x      y
##   <chr>    <dbl> <dbl>
## 1 dino      55.4  97.2
## 2 dino      51.5  96.0
## 3 dino      46.2  94.5
## 4 dino      42.8  91.4
## 5 dino      40.8  88.3
## 6 dino      38.7  84.9
## 7 dino      35.6  79.9
## 8 dino      33.1  77.6
## 9 dino      29.0  74.5
## 10 dino     26.2  71.4
## # ... with 1,836 more rows
```



And... all descriptive statistics are the same!

```
datasaurus %>%
  group_by(dataset) %>%
  summarise(mean_X = mean(x),
            mean_Y = mean(y),
            sd_X = sd(x),
            sd_Y = sd(y),
            cor = cor(x, y),
            n_obs = n()) %>%
  select(-dataset) %>%
  round(1)
```

```
## # A tibble: 13 x 6
```

```
##      mean_X mean_Y sd_X sd_Y cor n_obs
##      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1    54.3   47.8  16.8  26.9 -0.1   142
##  2    54.3   47.8  16.8  26.9 -0.1   142
##  3    54.3   47.8  16.8  26.9 -0.1   142
##  4    54.3   47.8  16.8  26.9 -0.1   142
##  5    54.3   47.8  16.8  26.9 -0.1   142
##  6    54.3   47.8  16.8  26.9 -0.1   142
##  7    54.3   47.8  16.8  26.9 -0.1   142
##  8    54.3   47.8  16.8  26.9 -0.1   142
##  9    54.3   47.8  16.8  26.9 -0.1   142
## 10    54.3   47.8  16.8  26.9 -0.1   142
## 11    54.3   47.8  16.8  26.9 -0.1   142
## 12    54.3   47.8  16.8  26.9 -0.1   142
## 13    54.3   47.8  16.8  26.9 -0.1   142
```

## 4.2 Basic ggplot2

ggplot2 is a modern tool for data visualisation. There are a lot of extensions for ggplot2. There is also a cheatsheet on ggplot2. There is also a whole book about ggplot2 (?).

Every ggplot2 plot has three key components:

- data,
- A set of aesthetic mappings between variables in the data and visual properties, and
- At least one layer which describes how to render each observation. Layers are usually created with a `geom_...()` function.

### 4.2.1 Scatterplot

I downloaded a Polish dictionary from here. I removed all abbreviations and proper names and took only one form from the paradigm. After all this I calculated the number of syllables (simply by counting vowels, combinations of *i* and other vowels I counted as one) and number of symbols in each word. Here is the result dataset.



Download this dataset to the variable `polish_dictionary`. How many words are there?

So this data could be visualised using the following code:

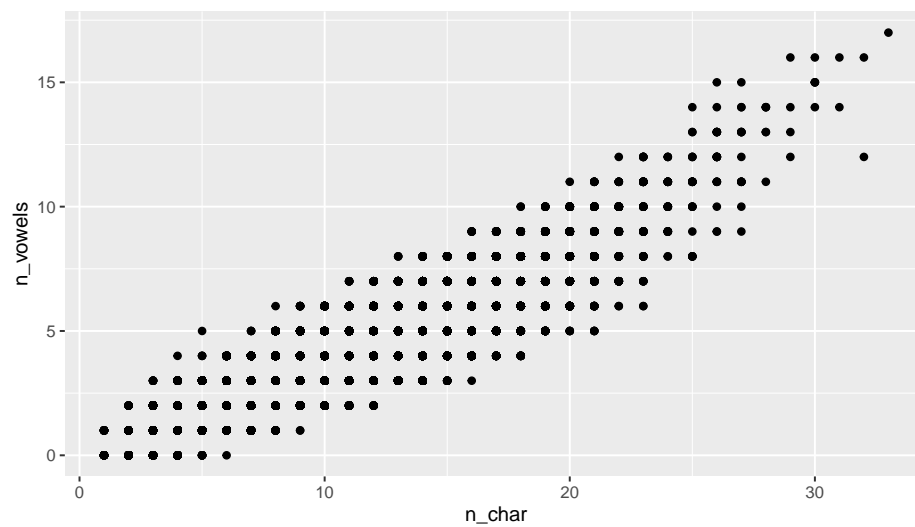
- `ggplot2`



```
ggplot(data = polish_dictionary, aes(n_char, n_vowels)) +  
  geom_point()
```

- dplyr, ggplot2

```
polish_dictionary %>%  
  ggplot(aes(n_char, n_vowels)) +  
  geom_point()
```





## Chapter 5

### Strings manipulation: `stringr`



## Chapter 6

Text manipulation:  
gutenbergr, tidytext,  
udpipe



## Chapter 7

### Stylometric analysis: `stylo`

