Data Analysis Coursebook

Marcell Granat & Zoltan Madari

Contents

Welcome

I hope this message finds you well.

This is an online bookdown file, which we plan to *update regularly* with the class material. We suggest that you to write the code simultaneously with us at seminar, but bugs can always occur out of the blue… If you missed something or just want to revisit the topic with additional comments (probably before the exam day) this page is here to help.

At the end of the course you should know:

- What are data types, data quality, and data preprocessing?
- What are the components of `tidyverse` and what are their advantage?
- What are density, distribution function, quantile functions?
- What are data clustering techniques?
- What are the main techniques for association analysis?

We will work with `R`, one the most loved statistical programming language. Do not be afraid if you do not have any programming experience, this is a beginner course. However, by the end of the program, we hope you will find useful the concept and practical tips we offer, and you will be able to solve your own real life data analysis issues.

Syllabus

0.1 Topics

- **Basic R knowledge (Week 1)**
  - Data categorize, sampling, importing-exporting
  - Types, tables, selection, objects, functions
- **Data manipulation in Tidyverse (Week 2)**
  - Filter, group_by, arrange, summarize commands
  - `%>%`
  - Join (mutating, filtering)
  - tidy data (longer, wider)
- **Visualization with ggplot2 (Week 3)**
  - Layers, facets, geoms
  - Descriptive statistics in R
  - Summary statistics, variability, correlation, covariance
  - Extreme values, problem of missing values
- **Statistical estimation (Week 4)**
  - Distributions
  - Sample techniques, confidence intervals, standard error
- **Hypothesis testing I (Week 5)**
  - Inductive statistics in R
  - Null and alternative hypothesis, t-test, p-value, fals positive/negative, Type I and II error

- **Hypothesis testing II (Week 6)**
  - Relation testing in R
- **Project presentations (Week 7)**

0.2 Requirements

- **Test** (30%): 90 min task solution in R + explanation (exam week)
- **Project** (25%): groups (3-4 students), one detailed report (essay) and one - **presentation**
- **Homeworks** and essays (45%)
- **Extra** tasks (+5%)

0.2.1 Project

By the end of the course you have to make a statistical report about your own research project in small groups. You have to find a proper dataset (kaggle, eurostat, UCI datasets etc.) or conduct an own survey in optional topic.

At week 7 you will hold a presentation about your findings and results. At the end of the week you will upload your detailed essay about your project work.

0.2.2 Homework

During the Study period you get 3-5 homeworks. The number of homeworks depends ont he material progress. It could be R code writing or R code writing and analysis (short essay about methods and results).

Table 1: Grades

0-50%	Fail (1)
51-62%	Pass (2)
63-74%	Satisfactory (3)
75-86%	Good (4)
87-100%	excellent (5)

0.3 Recommended (compulsory) reading

- **Grolemund G & Wickham H: R for Data Science**
- Gábor Békés & Gábor Kézdi: Data Analysis: Patterns, Prediction and Causality

Week 1

1 Lecture 1

2 Introduction to R

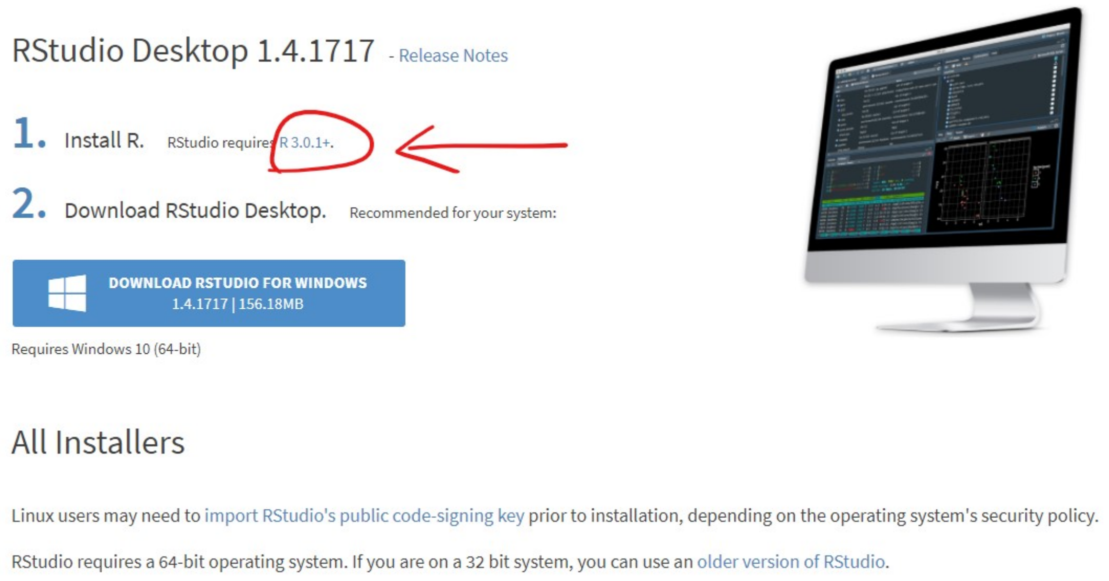
2.1 Why R?

In this chapter we will discuss the basics of R programming. R is a free software, used by millions in the field of statistics, data science, economics and many others.

The R programming language is an important tool for data related tasks, but it is much more. Just like other programming languages, R has many additional packages, which can extend its basic functionality. R has a great (probably the best) graphical tools to create your charts, and with shiny, you can easily build your minimalist web applications. We will learn about data manipulation, analysis and how to create awesome reports, like dashboards.

2.2 Setup

You can download R and RStudio from the official site of RStudio. Please install the appropriate version based on your OS, and do not forget that you also have to install R as well.



RStudio Desktop 1.4.1717 - Release Notes

1. Install R. RStudio requires R 3.0.1+.

2. Download RStudio Desktop. Recommended for your system:

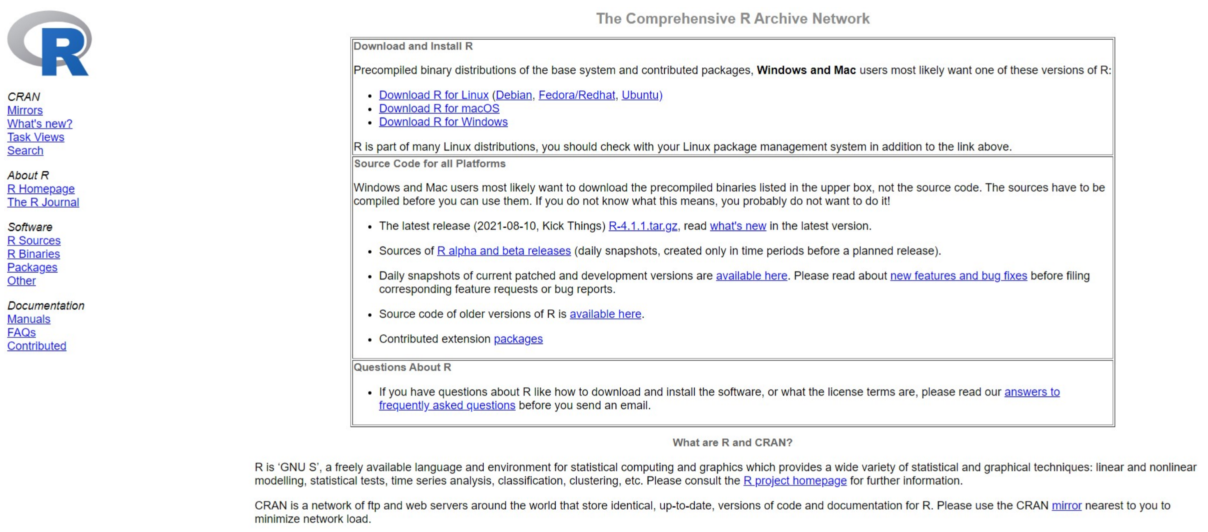
DOWNLOAD RSTUDIO FOR WINDOWS
1.4.1717 | 156.18MB

Requires Windows 10 (64-bit)

All Installers

Linux users may need to import RStudio's public code-signing key prior to installation, depending on the operating system's security policy.

RStudio requires a 64-bit operating system. If you are on a 32 bit system, you can use an older version of RStudio.



CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2021-08-10, Kick Things) R-4.1.1.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

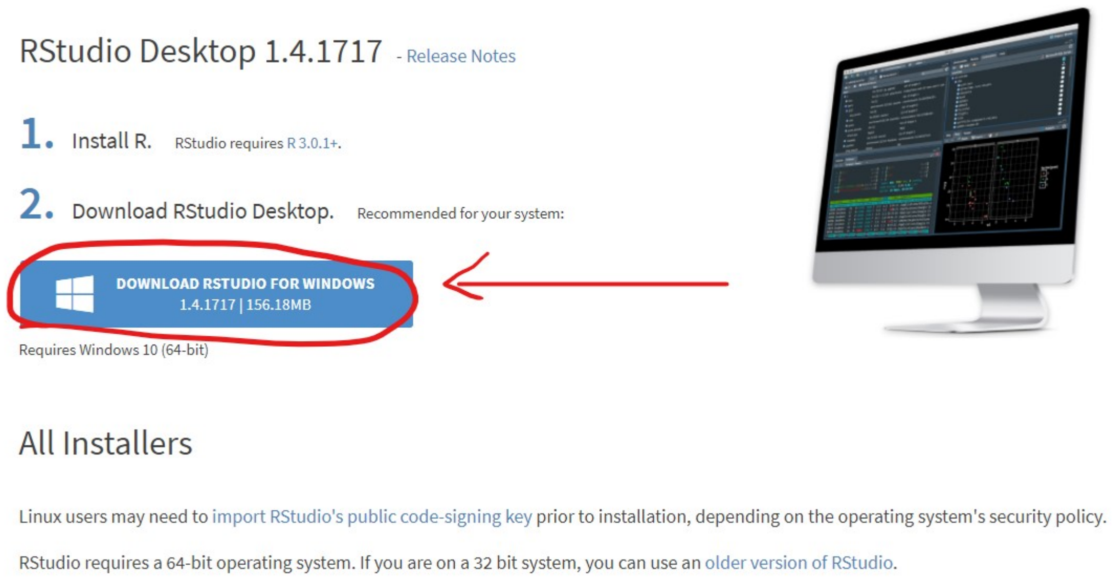
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.

Run R's installer file after the downloading process is finished. Next, we will also need the RStudio.


RStudio Desktop 1.4.1717 - Release Notes

1. Install R. RStudio requires R 3.0.1+.

2. Download RStudio Desktop. Recommended for your system:

DOWNLOAD RSTUDIO FOR WINDOWS
1.4.1717 | 156.18MB

Requires Windows 10 (64-bit)

All Installers

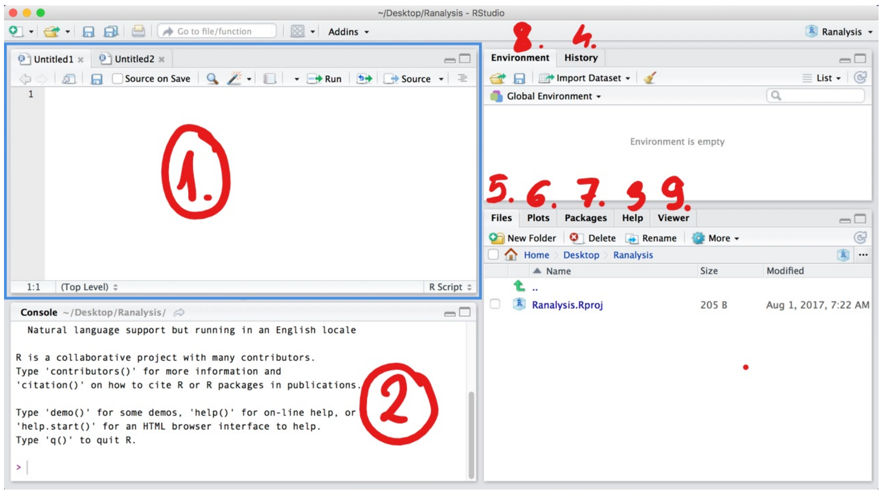
Linux users may need to import RStudio's public code-signing key prior to installation, depending on the operating system's security policy.

RStudio requires a 64-bit operating system. If you are on a 32 bit system, you can use an older version of RStudio.

If the installation process of R and RStudio is finished, then we can open RStudio and start to learn the software.

2.3 Our first meet with R

RStudio is dedicated IDEE for R, which means, that it will make our life much simplier. In stead of writing each line of code ourself, RStudio has many built-in functions to help us. We see some panes if we open RStudio:



```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1: Panes in RStudio

1. Source
   - We will write here our codes, which we would like to save. The basic extension of our codes are `.R`, but this is not the only possibility (we will cover this later). Once you save your code for later use, you can open your script also with a simple text editor (like Notepad), since this is only plain text. If you hit `enter` your code wont be executed, you will just simply start a new line. If you want to run your code hit `ctrl + enter` to execute a single line, and `ctrl+shift+enter` to execute your full script.

2. Console
    - Here you find the executed codes, and the response to that. For example, if you type `2 + 2` and hit `enter`, R will execute the expression, and response that it is 4.

```
2 + 2
#> [1] 4
```

3. Help
    - You can use this pane if you are not familier with a function. For example, you want to know what input you can specify while using `mean`, you can type `?mean` on the console, or use the search field on this pane. The description of the function will be presented on this pane. (This pane is super useful on the exam)
4. History
5. Files
    - You can see the list of your files which are in the current working directory. Working directory is the folder, from where R want currently read the files. If you want to import a dataset, just click on a file on this pane.
    - I highly recommend you to set a project folder for the class and any later job. This means that, R creates a folder and puts an `.Rproj` file into it. You can always click on this `.Rproj` file to return your unfinished work. You can customise if R should put the variables into your environtent as you left them last time, you have a history about the used codes, and you see all the data you copy + paste into this folder.
6. Plots
7. Packages
    - You can install packages from this pane. If you need a given package, click on install, and start typing its name. After that, you have to activate packages each time you open R again with the `library(eurostat)` command. You can also use a function from a package if you just simly type `eurostat::get_eurostat()`.
8. Environment
    - Here you can see the list of the variables you have already created. For example you can type `x = 3` on the console. Now and x variable will appear in the environment pane, and you can check its value if you type `x` on the console. You can also save these variables into an `.RData` data format if you wish.
9. Viewer

2.4 Data types

Lets see first, what kind of datatypes exist in R. Lets assign a variable called `x`.

```
x <- 4
```

So, what is the type of `x`? We can use the `class` command to answer this.

```
class(x)
#> [1] "numeric"
```

Its numeric[1]. This means that you can use `+`, `-`, `*` operators on it.

Lets see other types.

[1]Integer and double also exist in R, but these are not the default, and variables will be always coerced automatically

```
y <- "blue"
class(y)
#> [1] "character"
```

Its a character, basically can contain any kind of letter, digits, or white space.

```
does_it_rain <- TRUE
class(does_it_rain)
#> [1] "logical"
```

Its a logical value. It can be `TRUE` or `FALSE`

2.4.1 vectors

We can create a vector with the `c` function. (combine)

```
x <- c(11, 201, 301)
x
#> [1]  11 201 301
```

We can asses a given element of it by:

```
x[2]
#> [1] 201
```

Or we can use functions on it:

```
sum(x)
#> [1] 513
```

We can also easily create sequence with the syntax `start:stop`

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

If we combine characters, I mentiont that we can convert this vector to **factor** type. This is useful if we can enclose an order to the vector or we want to control for the possible values. Lets see a minimal example

```
my_vector <- c("First", "Second", "Third", "Fourth")
sort(my_vector)
#> [1] "First"  "Fourth" "Second" "Third"
```

If we want to sort the vector, we see that *Fourth* comes right after *First*. It is because character vectors are sorted in alphabetical order. We can solve it with `factor`

```
my_vector2 <- factor(my_vector, ordered = TRUE, levels = c("First", "Second", "Third", "Fourth"))
sort(my_vector2)
#> [1] First  Second Third  Fourth
#> Levels: First < Second < Third < Fourth
```

We can merge these vectors into a data.frame, which is basically like an excel table. Each column is a variable (with a header), and each row is an observation.

```
avengers_df <- data.frame(name = c("Captain America", "Hulk", "Dr. Strange"),
           color = c("blue", "green", NA))

avengers_df
#>              name color
#> 1 Captain America  blue
#> 2            Hulk green
#> 3      Dr. Strange  <NA>
```

NA stands for "not available", so these values are missing. Most of the times we will work with data.frames (similarly like pandas in python), so it is the most important data type we learn.

Storing more complex data, you can use the `list`. To use `data.frame` you need vectors with equal length. If this does not hold, or a more frequent case, you want to store a collection of data.frames, then `list` is a perfect solution! It is not a rare issue, big panel dataset are usually stored in separated files (a different file to each year, like: `cis_survey2016.csv`, `cis_survey2017.csv`). In this situations its suggested to store your data in a list.

```
mylist <- list(avengers_df, my_vector, x)
```

Now `mylist` stores a data.frame and two vector. You can access the components with a `[[ ]]`. For example, the first element:

```
mylist[[1]]
#>              name color
#> 1 Captain America  blue
#> 2            Hulk green
#> 3      Dr. Strange  <NA>
```

2.5 Data manipulation

2.5.1 Import data into R

We mentioned formely that the easiest way to import your data is to click on it in the files pane. However, this manual step is useful if you have to import and analyse the data once, but probably you want to use your data next time as well. That is way it is a good idea to copy and paste the code for importing the data into your script.

In fact, if the data is in your working directory, you can refer to it with "**relative referencing**". This means that you have to type only the name of the file, not the full path, because R will automatically start to look for the file in the working directory[2].

```
library(readr)
df <- read_delim("da_q.csv", delim = ";", escape_double = FALSE, trim_ws = TRUE)
```

```
df <- read_delim(str_c(WD, "/data/da_q.csv"), delim = ";", escape_double = FALSE, trim_ws = TRUE)
```

Now we have imported a tidy dataset. Each column is variable, and each row is an observation. Lets see how to select specific data from that. If you want to analyse only one column of it, you can use $ operator.

```
pizza <- df$`How many slices of pizza can you it at once?`

pizza
#>  [1] "8"
#>  [2] "12"
#>  [3] "Depends on size. Can be up to 5 slices of the medium pizza"
#>  [4] "2"
#>  [5] "3"
#>  [6] "4"
#>  [7] "4"
#>  [8] "4"
#>  [9] "3"
```

[2]you can download this example dataset from the GitHub page of this bookdown

```
#> [10] "2"
#> [11] "4"
#> [12] "3"
#> [13] "3"
#> [14] "2"
#> [15] "2"
#> [16] "4 and It depends how much I am hungry"
#> [17] "4"
#> [18] "3"
#> [19] "2"
#> [20] "6"
#> [21] "3"
```

The output `pizza` is a character vector currently, because some of the answers contain letters. We have to options here:

1. Using `as.numeric` function to force R using the values as numerical data.

```
as.numeric(pizza)
#> Warning: NAs introduced by coercion
#>  [1]  8 12 NA  2  3  4  4  4  3  2  4  3  3  2  2 NA  4  3  2  6  3
```

We got a warning message. Where letters appear R cannot convert the values to numbers, so this values became NA (Not Available) values.

2. Remove the letters from the answers and convert the vector to the correct datatype.

To manage this, we have to use the syntax called **regular expressions**. I want to show you 4 expressions now and a function. The function `gsub` will detect a given letter in a character and replace it with something. Lets see how!

```
gsub(x = "Awesome 12", pattern = "\\w", replacement = "B") # every non-white space
#> [1] "BBBBBBB BB"
gsub(x = "Awesome 12", pattern = "\\s", replacement = "B") # every white space
#> [1] "AwesomeB12"
gsub(x = "Awesome 12", pattern = "\\d", replacement = "B") # every digit
#> [1] "Awesome BB"
gsub(x = "Awesome 12", pattern = "\\D", replacement = "B") # every non-digit value
#> [1] "BBBBBBBB12"
```

So we can use the last example to solve our problem.

```
pizza_only_digits <- gsub(x = pizza, pattern = "\\D", replacement = "")

pizza_only_digits
#>  [1] "8"  "12" "5"  "2"  "3"  "4"  "4"  "4"  "3"  "2"  "4"  "3"  "3"  "2"  "2"
#> [16] "4"  "4"  "3"  "2"  "6"  "3"

as.numeric(pizza_only_digits)
#>  [1]  8 12  5  2  3  4  4  4  3  2  4  3  3  2  2  4  4  3  2  6  3
```

2.6 Conditional statements

We offen use conditional statement in programming. It has a clean concept: **If the condition is TRUE, then evaluate the following task.**

If you want to write an if else statement in R, I highly recomment you to use the snippet for that. Snippet means, that when you type `if` and press `shift + tab`, then R will automaticly write the framework you have to use:

```
if (condition) {

}
```

As a condition you have to use a logical value as input, or a condition. You can use conditions with the following operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, `is.na`, `%in%`, `stringr::str_detect()`.

```
4 < 5
#> [1] TRUE
5 <= 5
#> [1] TRUE
4 > 5
#> [1] FALSE
5 >=4
#> [1] TRUE
2 == 3 # equal?
#> [1] FALSE
(2 + 2) == 4
#> [1] TRUE
(2 + 2) != 4 # not equal?
#> [1] FALSE
3 != 3
#> [1] FALSE
is.na(4)
#> [1] FALSE
is.na(NA)
#> [1] TRUE
3 %in% c(1, 2, 3)
#> [1] TRUE
stringr::str_detect(string = "this function is awesome!", pattern = "some")
#> [1] TRUE
stringr::str_detect(string = "this function is awesome!", pattern = "none")
#> [1] FALSE
```

You can also specify the task R has to do, if the statement is false.

```
if (2>3) {
  print("Print this")
} else {
  print("Print that")
}
#> [1] "Print that"
```

2.7 Loops

2.7.1 While

You can also use while loop to specify a task R has to do until a condition is TRUE.

```
x <- 1

while (x < 15) {
  cat(paste0(x, "^2=")) # cat = print, just into the same line
  cat(x^2)
  cat("\n") # force R to create a new line
  x <- x + 1 # if you miss this step then R will repeat the task infinit times
}
#> 1^2=1
#> 2^2=4
#> 3^2=9
#> 4^2=16
#> 5^2=25
#> 6^2=36
#> 7^2=49
#> 8^2=64
#> 9^2=81
#> 10^2=100
#> 11^2=121
#> 12^2=144
#> 13^2=169
#> 14^2=196
```

2.7.2 For

With this framewrok you can specify a task, that R has to do x times. For example, print a message 10 times.

```
for (i in 1:10) {
  print("You R amazing!")
}
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
#> [1] "You R amazing!"
```

And you can use `i` inside the `{` parenthesis.

```
for (i in 1:5) {
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

2.8 Functions

We offen work with functions in R, but you can also write your own. You have to use the `function` word and specify the input variables.

```
my_first_function <- function(x) {
  # removed all non-digit characters from x, and take the squared of it.
  as.numeric(gsub(x, pattern = "\\D", replacement = ""))^2
}

my_first_function("Depends on, maybe 5 slices")
#> [1] 25
```

2.9 Apply family

This family contains 3 functions, which I want to show you (There are more complex ones, those are not covered in this bookdown).

The function `apply` tells R to use a function on each row or column of a data.frame. So the its frist argument is the `data.frame`, the third is the function which shoul use and the second is the margin: - margin = 2: apply the given function on each of the COLUMNS - margin = 1: apply the given function on each of the ROWS

```
non_na <- function(x) {
  # how many numeric observation are in the vector
  sum(!is.na(as.numeric(x)))
}
```

Number of numeric answers by quetions:

```
apply(df, 2, non_na)
#>                                                                                ID
#>                                                                                21
#>                    What is your zodiac? (https://www.astrology-zodiac-signs.com/)
#>                                                                                 0
#>                                                       Do you prefer dogs or cats?
#>                                                                                 0
#>                             What experiences do have on related to R programming?
#>                                                                                 0
#>                                      How many slices of pizza can you it at once?
#>                                                                                19
#>                                                             Do you wear glasses?2
#>                                                                                 0
#>                                       How many countries have you been to so far?
#>                                                                                21
#>   How many instagram followers do you have? (zero, if you do not have an account)
#>                                                                                20
#>                                        How many brothers and sisters do you have?
#>                                                                                19
#>                             What is your batteries current charge level? (0-100)
#>                                                                                19
#> What is the traditional food in your country? (you can mention more, if you wish)
#>                                                                                 0
```

Number of numeric answers by participant:

```
apply(df, 1, non_na)
#>  [1] 6 6 4 5 6 6 6 6 6 6 5 6 6 6 6 4 6 6 6 6 5
```

`Lapply` is similar but with list objects.

```
mylist <- list(
  first_vector = c(1, 2, 3),
  second_vector = letters # built in character vector, contains all the letters
)
```

```
mylist
#> $first_vector
#> [1] 1 2 3
#>
#> $second_vector
#>  [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"
```

We are interested in the number of observation (the `length`) of each vector:

```
out <- lapply(mylist, length)

out
#> $first_vector
#> [1] 3
#>
#> $second_vector
#> [1] 26

class(out)
#> [1] "list"
```

But the output is still a list. `sapply` is the solution if we want to convert it into vector.

```
sapply(mylist, length)
#>  first_vector second_vector
#>             3            26
```

Week 2

3 Lecture 2

4 Markdown & Tidyverse

4.1 Markdown

Last week we wrote or codes on the `Source` pane, and if you save the codes and revisit the file with your file explorer, you can see that the extension of the file is `.R`. You can open the file with any other text editor (like Notepad) and see the codes. In a `.R` file you can write your codes and comments, and if you hit `ctrl + shift + enter` then all lines will be evaluated. Most of the times the goal to use `.R` files is this, we want to reuse the codes frequently (like downloading data from a website).

Another possible extension for your files is the `.Rmd`, which stands for *R MarkDown*. In this file format you can combine text, R codes and their output into one single document. from now, We will use `RMarkdown` during the class. Please visit the following website to find useful examples: https://rmarkdown.rstudio.com/articles_intro.html

4.2 Introduction to the tidyverse

We will work with the tidyverse today. You can simply install it from the CRAN (go to `Packages` pane and click the install button). But tidyverse is not a simple package. "The tidyverse is a set of packages that work in harmony because they share common data representations […]"

`Tidyverse` contains the most important packages that you're likely to use in everyday data analyses:

- ggplot2, for data visualisation.
- dplyr, for data manipulation.
- tidyr, for data tidying.
- readr, for data import.
- purrr, for functional programming.
- tibble, for tibbles, a modern re-imagining of data frames.
- stringr, for strings.
- forcats, for factors.

After you installed the packages, you can load in the core packages with 'library' command. You will receive a warning message, but do not worry, this is fine.

```
library(tidyverse)
```

4.3 The %>% operator

This operator is written `ctrl + shift + m`, it is denoted as %>%, but we pronounce as **pipe**. To understand its relevance, you should think about it like *... then*. This opreator forward the value of the previous expression into the next function as the first unspecified input. Lets see an example:

We first generate a numerical vector as trajectories of standard normal distribution.

```
norm_sample <- rnorm(n = 1000, mean = 0, sd = 1) # standard normal
```

AND THEN we visualize its distribution with a histogram.

```
hist(norm_sample)
```

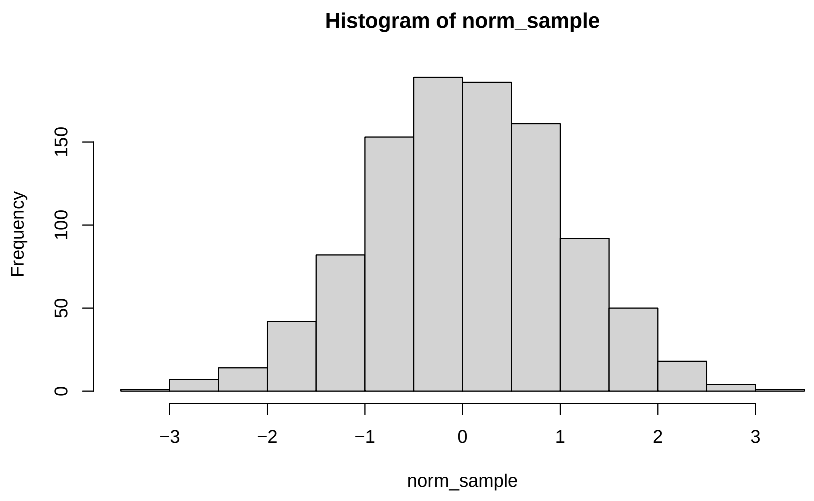


Figure 2: Histogram of standard normal distribution

Now we had to assign the `norm_sample` object to use only for a single graph. If you are working on a project it will be confusing to have tons of one time used objects in your `environment` (Like `DataFrameAfterCleaningStep1`, `DataFrameAfterCleaningStep2`, `DataFrameAfterCleaningStep3`, etc.).

With the pipe operator we can embed several steps into one single workflow. This way we do not have to assign the `norm_sample` object. We simple generate the random values AND THEN draw their distribution.

```
rnorm(n = 1000, mean = 0, sd = 1) %>%
  hist()
```

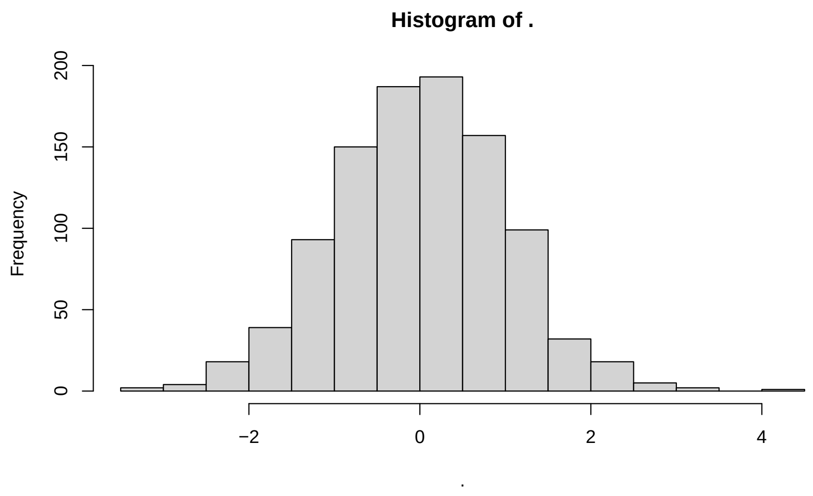


Figure 3: Histogram of standard normal distribution using the pipe

You may see that the title of plot has changed (probably the graph also changed a bit, since chances are low that we generate exactly the 1000 random values). This is because pipe and many other tidyverse function works with a *lambda like* function framework. This means that you can refer to the input value (only if you have only one single input) with `.`.

Example:

```
(2 + 2) %>%
  {. * .}
#> [1] 16
```

The result is 16, since 2 + 2 = 4, and 4 * 4 equals 16.

The pipe may seem unrelevant first, but this is one of the most powerful tool in R if you have complex data manipulating steps. We will cover the core functions of `dplyr` in this chapter, which appear in the following video (37:30 - 38:30), but when you are familier with them, you will see the motivation behind the pipe.

4.4 Tibble, count, filter, select, arrange

4.4.1 Tibble

Let's import a dataset from the OECD webpage. Download the data in csv format and paste it into your working directory.

```
fertility_df <- read_csv("DP_LIVE_22092021161631568.csv")
```

`read_csv` is importad from the `readr` package, thus shares its output is adequat to the tidyverse principles. `fertility_df` is a data.frame, but it is a `tibble`, which means that it will be printed nicely on the console: only the first 10 rows appears, and only the amount of columns that can be printed without a linebreak (similar to pandas dataframe in python).

```
fertility_df
#> # A tibble: 3,193 x 8
#>    LOCATION INDICATOR SUBJECT MEASURE   FREQUENCY  TIME Value `Flag Codes`
#>    <chr>    <chr>     <chr>   <chr>     <chr>     <dbl> <dbl> <lgl>
#>  1 AUS      FERTILITY TOT     CHD_WOMAN A          1960  3.45 NA
#>  2 AUS      FERTILITY TOT     CHD_WOMAN A          1961  3.55 NA
#>  3 AUS      FERTILITY TOT     CHD_WOMAN A          1962  3.43 NA
#>  4 AUS      FERTILITY TOT     CHD_WOMAN A          1963  3.34 NA
#>  5 AUS      FERTILITY TOT     CHD_WOMAN A          1964  3.15 NA
#>  6 AUS      FERTILITY TOT     CHD_WOMAN A          1965  2.97 NA
#>  7 AUS      FERTILITY TOT     CHD_WOMAN A          1966  2.89 NA
#>  8 AUS      FERTILITY TOT     CHD_WOMAN A          1967  2.85 NA
#>  9 AUS      FERTILITY TOT     CHD_WOMAN A          1968  2.89 NA
#> 10 AUS      FERTILITY TOT     CHD_WOMAN A          1969  2.89 NA
#> # ... with 3,183 more rows
```

4.4.2 Count

With the `count` function you can simple *count* the number of appearancesof the levels of a given column. Example:

```
count(fertility_df, LOCATION)
#> # A tibble: 54 x 2
#>    LOCATION     n
#>    <chr>    <int>
#>  1 ARG         60
#>  2 AUS         60
#>  3 AUT         61
#>  4 BEL         60
#>  5 BGR         60
#>  6 BRA         60
#>  7 CAN         60
#>  8 CHE         61
#>  9 CHL         60
#> 10 CHN         60
#> # ... with 44 more rows
```

Now we can see that we have 60 observation from ARG. If you are interested in for which country we have the most datapoints, you can use the `sort = TRUE` option.

```
count(fertility_df, LOCATION, sort = TRUE)
#> # A tibble: 54 x 2
#>    LOCATION     n
#>    <chr>    <int>
#>  1 AUT         61
#>  2 CHE         61
#>  3 CZE         61
#>  4 DNK         61
#>  5 EST         61
#>  6 FIN         61
#>  7 FRA         61
#>  8 LUX         61
#>  9 NOR         61
#> 10 NZL         61
#> # ... with 44 more rows
```

You can also use the pipe here[^It would not be a good solution to assign a new dataframe to each count table]

```
fertility_df %>% # %>% = the 1st argument of count function is the fertility_df data.frame
  count(LOCATION)
#> # A tibble: 54 x 2
#>    LOCATION     n
#>    <chr>    <int>
#>  1 ARG         60
#>  2 AUS         60
#>  3 AUT         61
#>  4 BEL         60
#>  5 BGR         60
#>  6 BRA         60
#>  7 CAN         60
#>  8 CHE         61
#>  9 CHL         60
#> 10 CHN         60
#> # ... with 44 more rows
```

You may specify multiple columns, this way the frequency of all value combinaton appears.

```
fertility_df %>%
  count(LOCATION, TIME, sort = T)
#> # A tibble: 3,193 x 3
#>    LOCATION  TIME     n
#>    <chr>    <dbl> <int>
#>  1 ARG       1960     1
#>  2 ARG       1961     1
#>  3 ARG       1962     1
#>  4 ARG       1963     1
#>  5 ARG       1964     1
#>  6 ARG       1965     1
#>  7 ARG       1966     1
#>  8 ARG       1967     1
#>  9 ARG       1968     1
#> 10 ARG       1969     1
#> # ... with 3,183 more rows
```

Now we see that we have only one observation for each country at a time, so this data frame was originally tidy (this is usually not the case). Let's see for example the Q GDP data from the OECD webpage.

```
gdp_df <- read_csv("DP_LIVE_22092021163414835.csv")
```

Using the `count` function, we may find that `MEASURE`, `SUBJECT` & `FREQUENCY` columns have multiple levels.

```
gdp_df %>%
  count(MEASURE, SUBJECT, FREQUENCY)
#> # A tibble: 5 x 4
#>   MEASURE  SUBJECT FREQUENCY     n
#>   <chr>    <chr>   <chr>     <int>
#> 1 IDX      VOLIDX  A          1764
#> 2 IDX      VOLIDX  Q          7127
#> 3 PC_CHGPP TOT     A          2329
#> 4 PC_CHGPP TOT     Q          9472
#> 5 PC_CHGPY TOT     Q          9385
```

Let's say we are interested only in the observations where the `SUBJECT` is equal to TOT (total), `MEASURE` is equal to PC_CHGPY (percentage change to previous value) and the `FREQUENCY` is not equal to A (only quarterly observations). *You can select rows by using the* `filter` function. Its 1st argument is the dataframe you want to modify, and the other arguments are conditional statements. If all the statements are TRUE, then the row will be selected.

```
filter(gdp_df, SUBJECT == "TOT", MEASURE == "PC_CHGPY", FREQUENCY != "A")
#> # A tibble: 9,385 x 8
#>    LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY TIME     Value `Flag Codes`
#>    <chr>    <chr>     <chr>   <chr>    <chr>     <chr>    <dbl> <chr>
#>  1 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q1  7.45  E
#>  2 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q2  5.00  E
#>  3 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q3  3.58  E
#>  4 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q4  3.00  E
#>  5 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q1  3.47  E
#>  6 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q2  5.11  E
#>  7 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q3  5.47  E
#>  8 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q4  4.76  E
#>  9 DEU      QGDP      TOT     PC_CHGPY Q         1963-Q1 -0.372 E
#> 10 DEU      QGDP      TOT     PC_CHGPY Q         1963-Q2  3.02  E
#> # ... with 9,375 more rows
```

Just like the `count`, we use this function with `%>%` operator, and use `&` instead of `,`, but if you wish to use *"OR"*, then you can use `|` between two statement.

```
gdp_df %>%
  filter(SUBJECT == "TOT" & MEASURE == "PC_CHGPY" & FREQUENCY != "A")
#> # A tibble: 9,385 x 8
#>    LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY TIME     Value `Flag Codes`
#>    <chr>    <chr>     <chr>   <chr>    <chr>     <chr>    <dbl> <chr>
#>  1 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q1  7.45  E
#>  2 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q2  5.00  E
#>  3 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q3  3.58  E
#>  4 DEU      QGDP      TOT     PC_CHGPY Q         1961-Q4  3.00  E
#>  5 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q1  3.47  E
#>  6 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q2  5.11  E
#>  7 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q3  5.47  E
#>  8 DEU      QGDP      TOT     PC_CHGPY Q         1962-Q4  4.76  E
#>  9 DEU      QGDP      TOT     PC_CHGPY Q         1963-Q1 -0.372 E
#> 10 DEU      QGDP      TOT     PC_CHGPY Q         1963-Q2  3.02  E
#> # ... with 9,375 more rows
```

4.4.3 Mutate

In the `gdp_df` dataframe you can find the `TIME` column and that its class character. Lets say we want to change this to date. Functions related to date are in the `lubridate` package. Since the values of the `TIME` column consist of a *y*ear and a *q*uarter, we will use the `yq` function to change it into date. But we should not forget that this will work only with the datapoints where the `FREQUENCY` is not annual. So let's keep the previous filter. If we want to modify a given column of a dataframe we should use the `mutate` function. With the pipe operator we now combine the `filter` and the `mutate` command.

```
gdp_df %>%
  filter(SUBJECT == "TOT" & MEASURE == "PC_CHGPY" & FREQUENCY != "A") %>%
  mutate(TIME = lubridate::yq(TIME))
#> # A tibble: 9,385 x 8
#>    LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY TIME        Value `Flag Codes`
#>    <chr>    <chr>     <chr>   <chr>    <chr>     <date>      <dbl> <chr>
#>  1 DEU      QGDP      TOT     PC_CHGPY Q         1961-01-01  7.45  E
#>  2 DEU      QGDP      TOT     PC_CHGPY Q         1961-04-01  5.00  E
#>  3 DEU      QGDP      TOT     PC_CHGPY Q         1961-07-01  3.58  E
#>  4 DEU      QGDP      TOT     PC_CHGPY Q         1961-10-01  3.00  E
#>  5 DEU      QGDP      TOT     PC_CHGPY Q         1962-01-01  3.47  E
#>  6 DEU      QGDP      TOT     PC_CHGPY Q         1962-04-01  5.11  E
#>  7 DEU      QGDP      TOT     PC_CHGPY Q         1962-07-01  5.47  E
#>  8 DEU      QGDP      TOT     PC_CHGPY Q         1962-10-01  4.76  E
#>  9 DEU      QGDP      TOT     PC_CHGPY Q         1963-01-01 -0.372 E
#> 10 DEU      QGDP      TOT     PC_CHGPY Q         1963-04-01  3.02  E
#> # ... with 9,375 more rows
```

Now that TIME is already a date column, we can add an additional condition: we want to analyze only the latest valuse (where the value of TIME is maximum).

```
gdp_df %>%
  filter(SUBJECT == "TOT" & MEASURE == "PC_CHGPY" & FREQUENCY != "A") %>%
  mutate(TIME = lubridate::yq(TIME)) %>%
  filter(TIME == max(TIME))
#> # A tibble: 49 x 8
#>    LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY TIME       Value `Flag Codes`
#>    <chr>    <chr>     <chr>   <chr>    <chr>     <date>     <dbl> <chr>
#>  1 DEU      QGDP      TOT     PC_CHGPY Q         2021-04-01  9.41 P
#>  2 IND      QGDP      TOT     PC_CHGPY Q         2021-04-01 20.9  <NA>
#>  3 TUR      QGDP      TOT     PC_CHGPY Q         2021-04-01 21.4  <NA>
#>  4 IDN      QGDP      TOT     PC_CHGPY Q         2021-04-01  7.19 <NA>
#>  5 LVA      QGDP      TOT     PC_CHGPY Q         2021-04-01 10.8  <NA>
#>  6 CZE      QGDP      TOT     PC_CHGPY Q         2021-04-01  8.18 <NA>
#>  7 BRA      QGDP      TOT     PC_CHGPY Q         2021-04-01 12.4  <NA>
#>  8 POL      QGDP      TOT     PC_CHGPY Q         2021-04-01 11.0  <NA>
#>  9 MEX      QGDP      TOT     PC_CHGPY Q         2021-04-01 19.5  P
#> 10 CAN      QGDP      TOT     PC_CHGPY Q         2021-04-01 12.7  <NA>
#> # ... with 39 more rows
```

Let's where we find the highest values. We have to change the order of the rows with the arrange function. If you want to set decreasing order, then you should put the columns name into the desc command.

```
gdp_df %>%
  filter(SUBJECT == "TOT" & MEASURE == "PC_CHGPY" & FREQUENCY != "A") %>%
  mutate(TIME = lubridate::yq(TIME)) %>%
  filter(TIME == max(TIME)) %>%
  arrange(desc(Value))
#> # A tibble: 49 x 8
#>    LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY TIME       Value `Flag Codes`
#>    <chr>    <chr>     <chr>   <chr>    <chr>     <date>     <dbl> <chr>
#>  1 GBR      QGDP      TOT     PC_CHGPY Q         2021-04-01  22.2 <NA>
#>  2 TUR      QGDP      TOT     PC_CHGPY Q         2021-04-01  21.4 <NA>
#>  3 IRL      QGDP      TOT     PC_CHGPY Q         2021-04-01  21.1 <NA>
#>  4 IND      QGDP      TOT     PC_CHGPY Q         2021-04-01  20.9 <NA>
#>  5 ESP      QGDP      TOT     PC_CHGPY Q         2021-04-01  19.8 P
#>  6 MEX      QGDP      TOT     PC_CHGPY Q         2021-04-01  19.5 P
#>  7 FRA      QGDP      TOT     PC_CHGPY Q         2021-04-01  18.7 <NA>
#>  8 HUN      QGDP      TOT     PC_CHGPY Q         2021-04-01  17.7 <NA>
#>  9 NZL      QGDP      TOT     PC_CHGPY Q         2021-04-01  17.4 <NA>
#> 10 ITA      QGDP      TOT     PC_CHGPY Q         2021-04-01  17.3 <NA>
#> # ... with 39 more rows
```

And we want to remove the unused columns. You can select columns with the `select` function.

```
gdp_df %>%
  filter(SUBJECT == "TOT" & MEASURE == "PC_CHGPY" & FREQUENCY != "A") %>%
  mutate(TIME = lubridate::yq(TIME)) %>%
  filter(TIME == max(TIME)) %>%
  arrange(desc(Value)) %>%
  select(geo = LOCATION, gdp_change = Value) # select the LOCATION & gdp_change columns
#> # A tibble: 49 x 2
#>    geo   gdp_change
#>    <chr>      <dbl>
#>  1 GBR         22.2
#>  2 TUR         21.4
#>  3 IRL         21.1
#>  4 IND         20.9
#>  5 ESP         19.8
#>  6 MEX         19.5
#>  7 FRA         18.7
#>  8 HUN         17.7
#>  9 NZL         17.4
#> 10 ITA         17.3
#> # ... with 39 more rows
```

Alternative notation with select:

```
gdp_df %>%
  select(1) # select the 1st column
#> # A tibble: 30,077 x 1
#>    LOCATION
#>    <chr>
#>  1 OECD
#>  2 OECD
#>  3 OECD
#>  4 OECD
#>  5 OECD
#>  6 OECD
#>  7 OECD
#>  8 OECD
#>  9 OECD
#> 10 OECD
#> # ... with 30,067 more rows

gdp_df %>%
  select(-1) # omit the 1st column
#> # A tibble: 30,077 x 7
#>    INDICATOR SUBJECT MEASURE  FREQUENCY TIME  Value `Flag Codes`
#>    <chr>     <chr>   <chr>    <chr>     <chr> <dbl> <chr>
#>  1 QGDP      TOT     PC_CHGPP A         1962   5.70 <NA>
#>  2 QGDP      TOT     PC_CHGPP A         1963   5.20 <NA>
#>  3 QGDP      TOT     PC_CHGPP A         1964   6.38 <NA>
#>  4 QGDP      TOT     PC_CHGPP A         1965   5.35 <NA>
#>  5 QGDP      TOT     PC_CHGPP A         1966   5.75 <NA>
#>  6 QGDP      TOT     PC_CHGPP A         1967   3.96 <NA>
#>  7 QGDP      TOT     PC_CHGPP A         1968   5.92 <NA>
#>  8 QGDP      TOT     PC_CHGPP A         1969   5.57 <NA>
#>  9 QGDP      TOT     PC_CHGPP A         1970   3.94 <NA>
#> 10 QGDP      TOT     PC_CHGPP A         1971   3.70 <NA>
#> # ... with 30,067 more rows

gdp_df %>%
  select(1:2) # select all the columns between the 1st and the 2nd
#> # A tibble: 30,077 x 2
#>    LOCATION INDICATOR
#>    <chr>    <chr>
#>  1 OECD     QGDP
#>  2 OECD     QGDP
#>  3 OECD     QGDP
#>  4 OECD     QGDP
#>  5 OECD     QGDP
#>  6 OECD     QGDP
#>  7 OECD     QGDP
#>  8 OECD     QGDP
#>  9 OECD     QGDP
#> 10 OECD     QGDP
#> # ... with 30,067 more rows

gdp_df %>%
  select(LOCATION:TIME) # select all the columns between the LOCATION and the TIME column
#> # A tibble: 30,077 x 6
#>    LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY TIME
#>    <chr>    <chr>     <chr>   <chr>    <chr>     <chr>
#>  1 OECD     QGDP      TOT     PC_CHGPP A         1962
#>  2 OECD     QGDP      TOT     PC_CHGPP A         1963
#>  3 OECD     QGDP      TOT     PC_CHGPP A         1964
#>  4 OECD     QGDP      TOT     PC_CHGPP A         1965
#>  5 OECD     QGDP      TOT     PC_CHGPP A         1966
#>  6 OECD     QGDP      TOT     PC_CHGPP A         1967
#>  7 OECD     QGDP      TOT     PC_CHGPP A         1968
#>  8 OECD     QGDP      TOT     PC_CHGPP A         1969
#>  9 OECD     QGDP      TOT     PC_CHGPP A         1970
#> 10 OECD     QGDP      TOT     PC_CHGPP A         1971
#> # ... with 30,067 more rows

gdp_df %>%
  select(TIME, LOCATION, everything())
#> # A tibble: 30,077 x 8
#>    TIME  LOCATION INDICATOR SUBJECT MEASURE  FREQUENCY Value `Flag Codes`
#>    <chr> <chr>    <chr>     <chr>   <chr>    <chr>     <dbl> <chr>
#>  1 1962  OECD     QGDP      TOT     PC_CHGPP A          5.70 <NA>
#>  2 1963  OECD     QGDP      TOT     PC_CHGPP A          5.20 <NA>
#>  3 1964  OECD     QGDP      TOT     PC_CHGPP A          6.38 <NA>
#>  4 1965  OECD     QGDP      TOT     PC_CHGPP A          5.35 <NA>
#>  5 1966  OECD     QGDP      TOT     PC_CHGPP A          5.75 <NA>
#>  6 1967  OECD     QGDP      TOT     PC_CHGPP A          3.96 <NA>
#>  7 1968  OECD     QGDP      TOT     PC_CHGPP A          5.92 <NA>
#>  8 1969  OECD     QGDP      TOT     PC_CHGPP A           5.57 <NA>
#>  9 1970  OECD     QGDP      TOT     PC_CHGPP A           3.94 <NA>
#> 10 1971  OECD     QGDP      TOT     PC_CHGPP A           3.70 <NA>
#> # ... with 30,067 more rows
  # select all the columns, but TIME & LOCATION to the first place
```

4.5 Group_by, Summary

Let's see another source for data. You can easily access Eurostat tables with the `eurostat` package.

```
eurostat::search_eurostat("birth")
#> # A tibble: 275 x 8
#>    title   code  type  `last update of~ `last table str~ `data start` `data end`
#>    <chr>   <chr> <chr> <chr>            <chr>            <chr>        <chr>
#>  1 Live b~ demo~ data~ 30.06.2021       23.02.2021       1990         2019
#>  2 Live b~ demo~ data~ 01.07.2021       01.07.2021       2013         2019
#>  3 Live b~ demo~ data~ 30.06.2021       23.02.2021       1990         2019
#>  4 Popula~ cens~ data~ 26.08.2015       08.02.2021       2011         2011
#>  5 Popula~ cens~ data~ 26.08.2015       08.02.2021       2011         2011
#>  6 Popula~ cens~ data~ 26.08.2015       08.02.2021       2011         2011
#>  7 Popula~ cens~ data~ 26.08.2015       08.02.2021       2011         2011
#>  8 Popula~ cens~ data~ 26.08.2015       08.02.2021       2011         2011
#>  9 Popula~ lfst~ data~ 10.09.2021       27.04.2021       1999         2020
#> 10 Activi~ lfst~ data~ 10.09.2021       27.04.2021       1999         2020
#> # ... with 265 more rows, and 1 more variable: values <chr>
```

Let's choose the "Live births by mother's age and NUTS 2 region" dataset.

```
livebirth_eu_df <- eurostat::get_eurostat("demo_r_fagec")

livebirth_eu_df
#> # A tibble: 456,956 x 5
#>    unit  age   geo   time       values
#>    <chr> <chr> <chr> <date>      <dbl>
#>  1 NR    TOTAL AL    2019-01-01  28561
#>  2 NR    TOTAL AL0   2019-01-01  28438
#>  3 NR    TOTAL AL01  2019-01-01   8909
#>  4 NR    TOTAL AL02  2019-01-01  12089
#>  5 NR    TOTAL AL03  2019-01-01   7440
#>  6 NR    TOTAL ALX   2019-01-01    123
#>  7 NR    TOTAL ALXX  2019-01-01    123
#>  8 NR    TOTAL AT    2019-01-01  84952
#>  9 NR    TOTAL AT1   2019-01-01  36819
#> 10 NR    TOTAL AT11  2019-01-01   2232
#> # ... with 456,946 more rows
```

First of all, we are interested in NUTS 2 reginal data. But in this dataset national aggregated values are also published (where the geo codes length is only 2 characters). Let's remove these.

```
livebirth_eu_df %>%
  filter(str_length(geo) != 2)
#> # A tibble: 416,961 x 5
#>    unit  age   geo   time       values
#>    <chr> <chr> <chr> <date>      <dbl>
#>  1 NR    TOTAL AL0   2019-01-01  28438
#>  2 NR    TOTAL AL01  2019-01-01   8909
#>  3 NR    TOTAL AL02  2019-01-01  12089
#>  4 NR    TOTAL AL03  2019-01-01   7440
#>  5 NR    TOTAL ALX   2019-01-01    123
#>  6 NR    TOTAL ALXX  2019-01-01    123
#>  7 NR    TOTAL AT1   2019-01-01  36819
#>  8 NR    TOTAL AT11  2019-01-01   2232
#>  9 NR    TOTAL AT12  2019-01-01  14652
#> 10 NR    TOTAL AT13  2019-01-01  19935
#> # ... with 416,951 more rows
```

We also should remove the aggregated values and keep only the latest one.

```
livebirth_eu_df %>%
  filter(str_length(geo) != 2) %>%
  filter(age != "TOTAL" & time == "2019-01-01") %>%
  filter(!(age %in% c("UNK", "Y_GE45", "Y_GE48", "Y_GE50", "Y_LT16")))
#> # A tibble: 14,309 x 5
#>    unit  age    geo   time       values
#>    <chr> <chr>  <chr> <date>      <dbl>
#>  1 NR    Y10-14 AL0   2019-01-01     24
#>  2 NR    Y10-14 AL01  2019-01-01      4
#>  3 NR    Y10-14 AL02  2019-01-01     13
#>  4 NR    Y10-14 AL03  2019-01-01      7
#>  5 NR    Y10-14 ALX   2019-01-01      0
#>  6 NR    Y10-14 ALXX  2019-01-01      0
#>  7 NR    Y10-14 AT1   2019-01-01      1
#>  8 NR    Y10-14 AT11  2019-01-01      0
#>  9 NR    Y10-14 AT12  2019-01-01      0
#> 10 NR    Y10-14 AT13  2019-01-01      1
#> # ... with 14,299 more rows
```

Now our dataset is clean: we have one observation to each geo and age category. Let's suppose we are interested in the total number of birth by the mothers age in EU (I know we would found a table for this). For this we want to sum the values in the `values` column by age category.

```
livebirth_eu_df %>%
  filter(str_length(geo) != 2) %>%
  filter(age != "TOTAL" & time == "2019-01-01") %>%
  filter(!(age %in% c("UNK", "Y_GE45", "Y_GE48", "Y_GE50", "Y_LT16"))) %>%
  group_by(age) %>%
  summarise(values = sum(values))
```

age	values
Y10-14	5337
Y15	13440
Y16	30049
Y17	60282
Y18	106563
Y19	172655
Y20	244849
Y21	308539
Y22	378984
Y23	450972
Y24	542341
Y25	649112
Y26	757241
Y27	867143
Y28	971229
Y29	1053152
Y30	1089894
Y31	1099205
Y32	1044856
Y33	979962
Y34	903304
Y35	811897
Y36	707466
Y37	597897
Y38	489777
Y39	385349
Y40	281512
Y41	192645
Y42	124356
Y43	73296
Y44	43848
Y45	23717
Y46	12628
Y47	7172
Y48	4354
Y49	2922

4.6 Pivot longer/wider

Let's assume we are interested in the growth rate of fertility in each country. First, we should write a function to calculate growth rates (chain index).

```r
chain_index <- function(x) {
  scales::percent(x/lag(x)-1, accuracy = .01)
  # lag: previous observation
}
```

Example:

```r
x <- c(100, 107, 105, 110)

chain_index(x)
#> [1] NA       "7.00%"  "-1.87%" "4.76%"
```

```r
fertility_df %>%
  select(LOCATION, TIME, Value) %>%
  mutate(GrowthRate = chain_index(Value))
```

(I hided the rest of the table)

LOCATION	TIME	Value	GrowthRate
AUS	1960	3.45	NA
AUS	1961	3.55	2.90%
AUS	1962	3.43	-3.38%
AUS	1963	3.34	-2.62%
AUS	1964	3.15	-5.69%
AUS	1965	2.97	-5.71%
AUS	1966	2.89	-2.69%
AUS	1967	2.85	-1.38%
AUS	1968	2.89	1.40%
AUS	1969	2.89	0.00%
AUS	1970	2.86	-1.04%
AUS	1971	2.95	3.15%
AUS	1972	2.74	-7.12%
AUS	1973	2.49	-9.12%
AUS	1974	2.32	-6.83%
AUS	1975	2.15	-7.33%
AUS	1976	2.06	-4.19%
AUS	1977	2.01	-2.43%
AUS	1978	1.95	-2.99%
AUS	1979	1.91	-2.05%
AUS	1980	1.89	-1.05%
AUS	1981	1.94	2.65%
AUS	1982	1.93	-0.52%
AUS	1983	1.92	-0.52%
AUS	1984	1.84	-4.17%
AUS	1985	1.92	4.35%
AUS	1986	1.87	-2.60%
AUS	1987	1.85	-1.07%
AUS	1988	1.83	-1.08%
AUS	1989	1.84	0.55%
AUS	1990	1.90	3.26%
AUS	1991	1.85	-2.63%
AUS	1992	1.89	2.16%
AUS	1993	1.86	-1.59%
AUS	1994	1.84	-1.08%
AUS	1995	1.82	-1.09%
AUS	1996	1.80	-1.10%
AUS	1997	1.78	-1.11%
AUS	1998	1.76	-1.12%
AUS	1999	1.76	0.00%
AUS	2000	1.76	0.00%
AUS	2001	1.73	-1.70%
AUS	2002	1.77	2.31%
AUS	2003	1.77	0.00%
AUS	2004	1.78	0.56%
AUS	2005	1.85	3.93%
AUS	2006	1.88	1.62%
AUS	2007	1.99	5.85%
AUS	2008	2.02	1.51%
AUS	2009	1.97	-2.48%
AUS	2010	1.95	-1.02%
AUS	2011	1.92	-1.54%
AUS	2012	1.93	0.52%
AUS	2013	1.88	-2.59%
AUS	2014	1.79	-4.79%
AUS	2015	1.79	0.00%
AUS	2016	1.79	0.00%
AUS	2017	1.74	-2.79%

But the problem comes at the `GrowthRate` at AUT in 1960, since now (check in the table):

$$\text{Growth}_{\text{AUT},1960} = \frac{\text{AUS}_{2019}}{\text{AUT}_{1960}}$$

We can easily solve this by transforming the structure of the table. We need to make the table **wider** in this case, with the `pivot_wider`. this function will create a new column to each level of a seleceted variable.

```
fertility_df %>%
  select(geo = LOCATION, time = TIME, fertility = Value) %>%
  pivot_wider(names_from = "geo", values_from = "fertility")
#> # A tibble: 61 x 55
#>     time   AUS   AUT   BEL   CAN   CZE   DNK   FIN   FRA   DEU   GRC   HUN   ISL
#>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  1960  3.45  2.69  2.54  3.9   2.11  2.54  2.71  2.74  2.37  2.23  2.02  4.26
#>  2  1961  3.55  2.78  2.63  3.84  2.13  2.55  2.65  2.82  2.44  2.13  1.94  3.88
#>  3  1962  3.43  2.8   2.59  3.76  2.14  2.54  2.66  2.8   2.44  2.16  1.79  3.98
#>  4  1963  3.34  2.82  2.68  3.67  2.33  2.64  2.66  2.9   2.51  2.14  1.82  3.98
#>  5  1964  3.15  2.79  2.72  3.5   2.36  2.6   2.58  2.91  2.53  2.24  1.8   3.86
#>  6  1965  2.97  2.7   2.62  3.15  2.18  2.61  2.46  2.85  2.5   2.25  1.81  3.71
#>  7  1966  2.89  2.66  2.52  2.81  2.01  2.62  2.4   2.8   2.51  2.32  1.88  3.58
#>  8  1967  2.85  2.62  2.41  2.6   1.9   2.35  2.32  2.67  2.45  2.45  2.01  3.28
#>  9  1968  2.89  2.58  2.31  2.45  1.83  2.12  2.15  2.59  2.36  2.42  2.06  3.07
#> 10  1969  2.89  2.49  2.28  2.4   1.86  2     1.94  2.53  2.21  2.36  2.04  2.99
#> # ... with 51 more rows, and 42 more variables: IRL <dbl>, ITA <dbl>,
#> #   JPN <dbl>, KOR <dbl>, LUX <dbl>, MEX <dbl>, NLD <dbl>, NZL <dbl>,
#> #   NOR <dbl>, POL <dbl>, PRT <dbl>, SVK <dbl>, ESP <dbl>, SWE <dbl>,
#> #   CHE <dbl>, TUR <dbl>, GBR <dbl>, USA <dbl>, BRA <dbl>, CHL <dbl>,
#> #   CHN <dbl>, EST <dbl>, IND <dbl>, IDN <dbl>, ISR <dbl>, RUS <dbl>,
#> #   SVN <dbl>, ZAF <dbl>, COL <dbl>, LVA <dbl>, LTU <dbl>, ARG <dbl>,
#> #   BGR <dbl>, HRV <dbl>, CYP <dbl>, MLT <dbl>, ROU <dbl>, SAU <dbl>, ...
```

If we use the `chain_index` function now on all the columns, then we can avoid the previous bug. You can use a function on all columns (except on the first) with the `mutate_at` function.

```
fertility_df %>%
  select(geo = LOCATION, time = TIME, fertility = Value) %>%
  pivot_wider(names_from = "geo", values_from = "fertility") %>%
  mutate_at(-1, chain_index)
#> # A tibble: 61 x 55
#>     time AUS   AUT   BEL   CAN   CZE   DNK   FIN   FRA   DEU   GRC   HUN   ISL
#>    <dbl> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
#>  1  1960 <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
#>  2  1961 2.90% 3.35% 3.54% -1.5~ 0.95% 0.39% -2.2~ 2.92% 2.95% -4.4~ -3.9~ -8.9~
#>  3  1962 -3.3~ 0.72% -1.5~ -2.0~ 0.47% -0.3~ 0.38% -0.7~ 0.00% 1.41% -7.7~ 2.58%
#>  4  1963 -2.6~ 0.71% 3.47% -2.3~ 8.88% 3.94% 0.00% 3.57% 2.87% -0.9~ 1.68% 0.00%
#>  5  1964 -5.6~ -1.0~ 1.49% -4.6~ 1.29% -1.5~ -3.0~ 0.34% 0.80% 4.67% -1.1~ -3.0~
#>  6  1965 -5.7~ -3.2~ -3.6~ -10.~ -7.6~ 0.38% -4.6~ -2.0~ -1.1~ 0.45% 0.56% -3.8~
#>  7  1966 -2.6~ -1.4~ -3.8~ -10.~ -7.8~ 0.38% -2.4~ -1.7~ 0.40% 3.11% 3.87% -3.5~
#>  8  1967 -1.3~ -1.5~ -4.3~ -7.4~ -5.4~ -10.~ -3.3~ -4.6~ -2.3~ 5.60% 6.91% -8.3~
#>  9  1968 1.40% -1.5~ -4.1~ -5.7~ -3.6~ -9.7~ -7.3~ -3.0~ -3.6~ -1.2~ 2.49% -6.4~
#> 10  1969 0.00% -3.4~ -1.3~ -2.0~ 1.64% -5.6~ -9.7~ -2.3~ -6.3~ -2.4~ -0.9~ -2.6~
#> # ... with 51 more rows, and 42 more variables: IRL <chr>, ITA <chr>,
#> #   JPN <chr>, KOR <chr>, LUX <chr>, MEX <chr>, NLD <chr>, NZL <chr>,
#> #   NOR <chr>, POL <chr>, PRT <chr>, SVK <chr>, ESP <chr>, SWE <chr>,
#> #   CHE <chr>, TUR <chr>, GBR <chr>, USA <chr>, BRA <chr>, CHL <chr>,
#> #   CHN <chr>, EST <chr>, IND <chr>, IDN <chr>, ISR <chr>, RUS <chr>,
#> #   SVN <chr>, ZAF <chr>, COL <chr>, LVA <chr>, LTU <chr>, ARG <chr>,
#> #   BGR <chr>, HRV <chr>, CYP <chr>, MLT <chr>, ROU <chr>, SAU <chr>, ...
```

And now let's transform the table to its original structure. You can do this with the `pivot_longer` column.

```
fertility_df %>%
  select(geo = LOCATION, time = TIME, fertility = Value) %>%
  pivot_wider(names_from = "geo", values_from = "fertility") %>%
  mutate_at(-1, chain_index) %>%
  pivot_longer(-1, names_to = "geo", values_to = "fertility")
#> # A tibble: 3,294 x 3
#>     time geo   fertility
#>    <dbl> <chr> <chr>
#>  1  1960 AUS   <NA>
#>  2  1960 AUT   <NA>
#>  3  1960 BEL   <NA>
#>  4  1960 CAN   <NA>
#>  5  1960 CZE   <NA>
#>  6  1960 DNK   <NA>
#>  7  1960 FIN   <NA>
#>  8  1960 FRA   <NA>
#>  9  1960 DEU   <NA>
#> 10  1960 GRC   <NA>
#> # ... with 3,284 more rows
```

4.7 Group_modify

Alternativly, we could solve this problem if we split the data frame into 63 individual data frames (one for each country). If you use the following syntax, you will get the correct results:

```
fertility_df %>%
  select(geo = LOCATION, time = TIME, fertility = Value) %>%
  arrange(time) %>%
  group_by(geo) %>%
  group_modify(~ mutate(.x, fertility_growth = chain_index(fertility)), .keep = F)
```

geo	time	fertility	fertility_growth
ARG	1960	3.11	NA
ARG	1961	3.10	-0.32%
ARG	1962	3.09	-0.32%
ARG	1963	3.08	-0.32%
ARG	1964	3.07	-0.32%
ARG	1965	3.06	-0.33%
ARG	1966	3.05	-0.33%
ARG	1967	3.05	0.00%
ARG	1968	3.05	0.00%
ARG	1969	3.06	0.33%
ARG	1970	3.08	0.65%
ARG	1971	3.11	0.97%
ARG	1972	3.15	1.29%
ARG	1973	3.20	1.59%
ARG	1974	3.25	1.56%
ARG	1975	3.30	1.54%
ARG	1976	3.34	1.21%
ARG	1977	3.36	0.60%
ARG	1978	3.36	0.00%
ARG	1979	3.34	-0.60%
ARG	1980	3.30	-1.20%
ARG	1981	3.25	-1.52%
ARG	1982	3.20	-1.54%
ARG	1983	3.16	-1.25%
ARG	1984	3.12	-1.27%
ARG	1985	3.10	-0.64%
ARG	1986	3.08	-0.65%
ARG	1987	3.06	-0.65%
ARG	1988	3.04	-0.65%
ARG	1989	3.02	-0.66%
ARG	1990	3.00	-0.66%
ARG	1991	2.97	-1.00%
ARG	1992	2.93	-1.35%
ARG	1993	2.88	-1.71%
ARG	1994	2.83	-1.74%
ARG	1995	2.77	-2.12%
ARG	1996	2.72	-1.81%
ARG	1997	2.67	-1.84%
ARG	1998	2.62	-1.87%
ARG	1999	2.58	-1.53%
ARG	2000	2.54	-1.55%
ARG	2001	2.51	-1.18%
ARG	2002	2.49	-0.80%
ARG	2003	2.46	-1.20%
ARG	2004	2.44	-0.81%
ARG	2005	2.42	-0.82%
ARG	2006	2.40	-0.83%
ARG	2007	2.38	-0.83%
ARG	2008	2.37	-0.42%
ARG	2009	2.36	-0.42%
ARG	2010	2.35	-0.42%
ARG	2011	2.34	-0.43%
ARG	2012	2.33	-0.43%
ARG	2013	2.32	-0.43%
ARG	2014	2.31	-0.43%
ARG	2015	2.30	-0.43%
ARG	2016	2.29	-0.43%
ARG	2017	2.28	-0.44%

4.8 Exercise

You have to get to know your classmates to solve this exercise.

1. Create a new .Rmd file
2. Import the da_q.csv data into R with relative referencing (see Chapter 2.5.1)
3. Write down the information you know about at least 3 of your teammates (Names are neccesary) as text in the `.Rmd`.
4. Write R codes to find the teammates (write also comments in the code[3]) using only base R function (tools you have learned in the 1st seminar).
5. Write R codes to find the teammates using dplyr function (tools you have learned in the 2nd seminar)

You found your teammate if the output of an R code is correct ID number of the person.

Example:

```
df
#> # A tibble: 21 x 11
#>       ID `What is your zod~ `Do you prefer ~ `What experience~ `How many slices~
#>    <dbl> <chr>              <chr>            <chr>             <chr>
#>  1     1 leo                dogs             I am familier wi~ 8
#>  2     2 taurus             Dogd             I have some expe~ 12
#>  3     3 pisces             Both             I do not have an~ Depends on size.~
#>  4     4 taurus             neither          I do not have an~ 2
#>  5     5 aquarius           dogs             I have some expe~ 3
#>  6     6 leo                Cats             I am familier wi~ 4
#>  7     7 virgo              Cats             I am familier wi~ 4
#>  8     8 virgo              Dogs             I learnt R in un~ 4
#>  9     9 taurus             dogs             I learnt R in un~ 3
#> 10    10 cancer             dogs             I do not have an~ 2
#> # ... with 11 more rows, and 6 more variables: Do you wear glasses?2 <chr>,
#> #   How many countries have you been to so far? <dbl>,
#> #   How many instagram followers do you have? (zero, if you do not have an account) <chr>,
#> #   How many brothers and sisters do you have? <chr>,
#> #   What is your batteries current charge level? (0-100) <chr>,
#> #   What is the traditional food in your country? (you can mention more, if you wish) <chr>
```

I am from Hungary, and the national food in Hungary is gulash.

```
df %>%
  filter(str_detect(`What is the traditional food in your country? (you can mention more, if you wish)`
  pull(ID)
#> [1] 1
```

So my ID number is 1.

Of course, you can not choose me. :)

[3]“A comment always writes down the why, not the what”