How to Use R for Data Science

Lecture Notes (currently under revision)

Prof. Dr. Stephan Huber

April 12, 2024

Table of contents

1. Getting started with R **4**
- 1.1. Why R? . . . 4
- 1.2. How to learn R . . . 5
- 1.3. Learning resources . . . 6
- 1.4. What are R and RStudio? . . . 7
- 1.5. How to use R and RStudio without installation . . . 10
- 1.6. Installing R and RStudio . . . 10
- 1.7. What is a function in R? . . . 10
- 1.8. What are objects in R? . . . 11
- 1.9. What are R packages? . . . 12
  - 1.9.1. Package installation . . . 12
  - 1.9.2. Package loading . . . 13
  - 1.9.3. Simplified package management with `p_load` . . . 13
- 1.10. Base R and the tidyverse universe . . . 14
- 1.11. Are there some guidelines for working with R? . . . 15

2. An interactive introduction using swirl **17**
- 2.1. Set up `swirl` . . . 17
- 2.2. swirl-it: huber-intro-1 . . . 18
- 2.3. swirl-it: huber-intro-2 . . . 22
- 2.4. swirl-it: Data analytical basics . . . 27
- 2.5. swirl-it: The `tidyverse` package . . . 27
- 2.6. Other `swirl` modules . . . 27

3. Work with R scripts **28**
- 3.1. R Scripts: Why they are useful . . . 28
- 3.2. Generate, write, and run R scripts . . . 28
- 3.3. The assignment operator: `<-` . . . 29
- 3.4. Doing calculation in scripts . . . 30
- 3.5. User-defined functions and conflicts . . . 31

4. Manage data **34**
- 4.1. Import data . . . 34
  - 4.1.1. Vectors and matrices . . . 34
  - 4.1.2. Open RData files . . . 35
  - 4.1.3. Open datasets of packages . . . 35
  - 4.1.4. Import data using public APIs . . . 36
  - 4.1.5. Import various file formats . . . 37
  - 4.1.6. Examples . . . 38
- 4.2. Data . . . 38
  - 4.2.1. Data frames and tibbles . . . 38
  - 4.2.2. Tidy data . . . 39
  - 4.2.3. Data types . . . 41
- 4.3. The pipe operator: `|>` . . . 41

4.4. Tools to manipulate data . . . 43
4.4.1. The `%in%` operator . . . 43
4.4.2. Extract operators . . . 43
4.4.3. Logical operators . . . 45
4.4.4. If statements . . . 47
4.5. Data manipulation with `dplyr` . . . 48
4.6. How to explore a dataset . . . 53

5. Visualizing data **57**

6. Collection of exercises **59**
6.1. Links to the R scripts with the solutions . . . 59
6.2. Links to the output of the scripts . . . 59
EXERCISE: Generate and drop variables . . . 59
EXERCISE: Import data . . . 60
EXERCISE: Base R or pipe . . . 60
EXERCISE: Subsetting . . . 60
EXERCISE: Data transformation . . . 61
EXERCISE: Load the Stata dataset "auto" using R . . . 61
EXERCISE: DatasauRus . . . 62
EXERCISE: Convergence . . . 63
EXERCISE: Unemployment and GDP in Germany and France . . . 63
EXERCISE: Import data and write a report . . . 67
EXERCISE: Explain the weight . . . 68
EXERCISE: Calories and weight . . . 68
EXERCISE: Bundesliga . . . 70
EXERCISE: Okun's Law . . . 73
EXERCISE: Names and duplicates . . . 77
EXERCISE: Zipf's law . . . 77

References **85**

Appendices **87**

A. Navigating the file system **87**
A.1. The file system . . . 87
A.2. Working directory . . . 87
A.3. Navigating the file system using the R console . . . 88
A.4. R Studio projects . . . 89
A.5. Why do the Windows paths use the back-slash? . . . 89

B. Troubleshooting **90**

C. Operators **91**
C.1. Assignment: . . . 91
C.2. Arithmetic: . . . 91
C.3. Relational: . . . 91
C.4. Logical: . . . 91
C.5. Others: . . . 91

D. Popular functions **92**
D.1. Help . . . 92

D.2. Package management 92
D.3. General 92
D.4. Tools 92
D.5. Data import 92
D.6. Inspect data 93
D.7. Graphics 93
D.8. Data management 93
D.9. `dplyr` functions 94
D.10. Data analysis 94
D.11. Statistical functions 94

E. Helpful shortcuts **96**

List of figures

1. Prof. Dr. Stephan Huber 2

1.1. Analogy of difference between R and RStudio 8
1.2. Icons of R versus RStudio on your computer 8
1.3. RStudio interface to R 9
1.4. The R Documentation of `q()` 11
1.5. The tidyverse universe 15

4.1. The logo of the packages readr, haven, and readxl 37
4.2. The logos of the tidyr and tibble packages 38
4.3. Features of a tidy dataset: variables are columns, observations are rows, and values are cells 39
4.4. The logo of the `dplyr` package 49

5.1. Pie charts are problematic 57

6.1. The logo of the DatasauRus package 62
6.2. Weight vs. Calories 69
6.3. Ranking history: 1. FC Kaiserslautern 72
6.4. Ranking history: 1. FC Köln 72

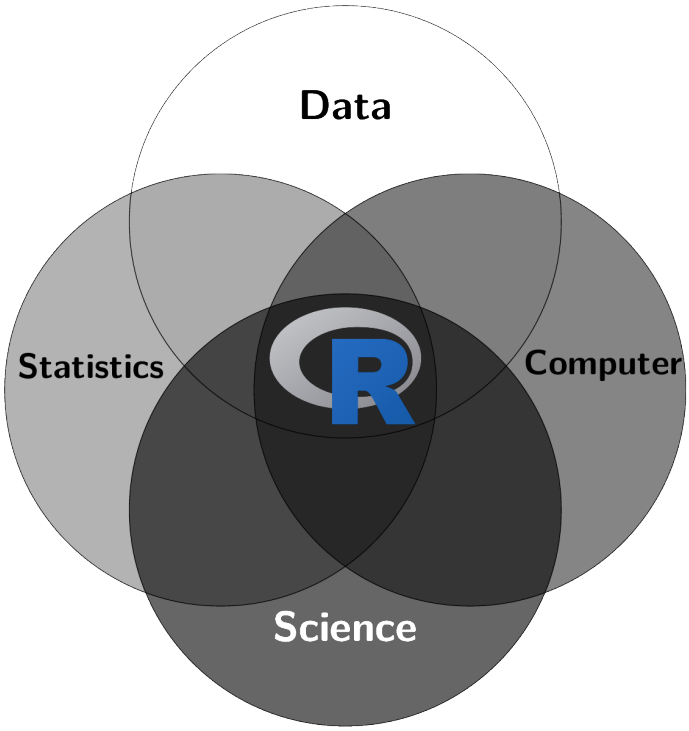
List of tables

4.1. Logical operators 45

6.1. Covid cases and deaths till August 2022 60

E.1. Table 1: Different OS, different keys 96
E.2. Table 2: Helpful shortcuts 96

Preface



About R

The programming language R enables you to handle, visualize, and analyze data. It is compatible with various operating systems (Windows, Mac, Linux) and can do a lot of things better compared to other programs like Python, Stata, Eviews, SPSS, SAS, and Excel. R is open source, extensively utilized, and there are abundant resources available for learning it. These notes are just my five cents.

About the cover of the notes

Data science is a buzzword that combines different fields of knowledge such as computer science, software engineering, informatics, database management, statistics, econometrics, business intelligence, and mathematics. However, there is no universally accepted definition of it and I think it is not important to define it precisely. Kelleher and Tierney [2018, p. 97] wrote "Data science is best understood as a partnership between a data scientist and a computer." So data science is about embracing the power of computers for scientific, commercial or social purposes. Of course, empirical models and statistics play a role in gaining meaningful insights. The graphic on the cover page may illustrate that R combines four important fields, that are, data, science, computer, and statistics.

About the notes

A pdf version of these notes is available

here.

- These notes aims to support my lecture at the HS Fresenius but are incomplete and no substitute for taking actively part in class.
- I host the notes in a GitHub repo.
- I hope you find this book helpful. Any feedback is both welcome and appreciated.
- This is work in progress so please check for updates regularly.
- These notes offer a curated collection of explanations, exercises, and tips to facilitate learning R without causing unnecessary frustration. However, these notes don't aim to rival comprehensive textbooks such as Wickham and Grolemund [2023].
- These notes are published under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. This means it can be reused, remixed, retained, revised and redistributed as long as appropriate credit is given to the authors. If you remix, or modify the original version of this open textbook, you must redistribute all versions of this open textbook under the same license. This script draws from the work of Navarro [2020], Muschelli and Jaffe [2022], Thulin [2021], and Ismay and Kim [2022] which is also published under the same license.

About the author

Contact:

Prof. Dr. Stephan Huber
Hochschule Fresenius für Wirtschaft & Medien GmbH
Im MediaPark 4c
50670 Cologne
Office: 4e OG-3
Telefon: +49 221 973199-523
Mail: stephan.huber@hs-fresenius.de
Private homepage: www.hubchev.github.io
Github: https://github.com/hubchev

Figure 1.: Prof. Dr. Stephan Huber

I am a Professor of *International Economics and Data Science* at HS Fresenius, holding a Diploma in Economics from the University of Regensburg and a Doctoral Degree (summa cum laude) from the University of Trier. I completed postgraduate studies at the Interdisciplinary Graduate Center of Excellence at the Institute for Labor Law and Industrial Relations in the

European Union (IAAEU) in Trier. Prior to my current position, I worked as a research assistant to Prof. Dr. Dr. h.c. Joachim Möller at the University of Regensburg, a post-doc at the Leibniz Institute for East and Southeast European Studies (IOS) in Regensburg, and a freelancer at Charles University in Prague.

Throughout my career, I have also worked as a lecturer at various institutions, including the TU Munich, the University of Regensburg, Saarland University, and the Universities of Applied Sciences in Frankfurt and Augsburg. Additionally, I have had the opportunity to teach abroad for the University of Cordoba in Spain, the University of Perugia in Italy, and the Petra Christian University in Surabaya, Indonesia. My published work can be found in international journals such as the Canadian Journal of Economics and the Stata Journal. For more information on my work, please visit my private homepage at hubchev.github.io.

I was always fascinated by data and statistics. For example, in 1992 I could name all soccer players in Germany's first division including how many goals they scored. Later, in 2003 I joined the introductory statistics course of Daniel Rösch. I learned among others that probabilities often play a role when analyzing data. I continued my data science journey with Harry Haupt's *Introductory Econometrics* course, where I studied the infamous Jeffrey M. Wooldridge [2002] textbook. It got me hooked and so I took all the courses Rolf Tschernig offered at his chair of Econometrics, where I became a tutor at the University of Regensburg and a research assistant of Joachim Möller. Despite everything we did had to do with how to make sense out of data, we never actually used the term *data science* which is also absent in the more 850 pages long textbook by Wooldridge [2002]. The book also remains silent about *machine learning* or *artificial intelligence*. These terms became popular only after I graduated. The *Harvard Business Review* article by Davenport and Patil [2012] who claimed that data scientist is "The Sexiest Job of the 21st Century" may have boosted the popularity.

The term "data scientist" has become remarkably popular, and many people are eager to adopt this title. Although I am a professor of *data science*, my professional identity is more like that of an applied, empirically-oriented international economist. My hesitation to adopt the title "data scientist" also stems from the deep respect I have developed through my interactions with econometricians and statisticians. Considering their in-depth expertise, I feel like a passionate amateur.

Ultimately, I poke around in data to find something interesting. Much like my ten-year-old younger self who analyzed soccer statistics to gain a deeper understanding of the sport. The only thing that has changed since then is that I know more promising methods and can efficiently use tools for data processing and data analysis.

1. Getting started with R

Content of this chapter

1. Why R?
2. How to learn R?
3. What are R and RStudio?
4. How to use R and RStudio without installation
5. How to install R and RStudio
6. How to write and run code in R
7. What are R packages?
8. What is a function in R?
9. What are objects in R?
10. Base R and the tidyverse universe
11. Are there some guidelines for working with R?

1.1. Why R?

R is a free and open-source programming language that provides a wide range of advanced statistics capabilities, state-of-the-art graphics, and powerful data manipulation capabilities. It supports larger data sets, reads any type of data, and runs on multiple platforms. R makes it easier to automate tasks, organize projects, ensure reproducibility, and find and fix errors, and anyone can contribute packages to improve its functionality. Moreover, the following points are worth to emphasize:

- **R is an artist!** Check out:
  - The R Graph Gallery
  - R CHARTS by R CODER
- **R is an employment insurance!** Programming is a core skill in research, economics, and business. If you can write code, you have plenty of opportunities to earn a decent salary. R is one of the most widely used programming languages in the world today. It is used in almost every industry such as finance, banking, medicine or manufacturing. R is used for portfolio management, risk analytics in finance and banking industries. Even if you need to learn a new programming language later, knowing R makes it much easier to pick up another one.
- **R uses the computer and computers are great!** Doing statistics on a computer is faster, easier and more powerful than doing it by hand. Computers are an extension to your brain and can do repetitive tasks better and faster without making logical errors. The only reason to do statistical calculations with pencil and paper is for learning purposes.
- **Excel is limited!** Using spreadsheets software like Microsoft Excel for research can be problematic. It's easy to lose track of operations, making the process difficult to oversee and document. Command-line programs are maybe not as easy to learn but offer a more straightforward approach that allows the results to be replicated easily.

- **R is open source!** Proprietary software expansive, support can only be provided by the copyright owner which means the software expires and you can't do anything against it. Moreover, security issues cannot be checked as the source code is not available, and possibilities for customization are limited. R is yours and everybody can contribute to its success.
- **R is big!** When you download and install R, you get some basic packages, that contain functions that allow you to do already a lot of things. Beyond that, you can write your own packages or install user-written packages that extend your possibilities. With over 20,684 packages on the CRAN repository and many more available on GitHub and other platforms, R's extensive library supports a wide variety of data science tasks. Its widespread use and open-source availability have cemented R as a standard tool in data science and ensured that there are multiple approaches to most data handling processes. These can be easily adopted.

R has weaknesses

For newcomers to programming, the learning curve is rather flat at the beginning. One reason is that R tools are spread across many packages, which can overwhelm beginners. There is no centralized support and the helpful and active online community have different backgrounds. It can be difficult for beginners to find the right solution as there are often many different ways to tackle the same problem. Moreover, R can be slower than languages like Python, MATLAB, C/C++ or Java.

1.2. How to learn R

There are many different approaches to learning R. It pretty much depends on your preferences, needs, goals, prerequisites and limitations. It is up to you to search and find a suitable way to achieve your learning goals. While I hope you find my notes helpful, I additionally provide in section Section 1.3 a list of other resources that are worth considering. To start with, I recommend my swirl courses that provide an interactive learning environment, see Chapter 2.

Make your hands dirty!

Learning a programming language can, like learning a foreign language, be daunting and frustrating. However, if you put in the effort and are not afraid to make mistakes, anybody can learn it. You don't have to be a nerd. To have a guide next to you can help and speed up your progress significantly. The key is taking action and getting involved. I mean, do write code. Try to copy the code that you read here and elsewhere. Explore what the code does on your machine. Don't be afraid to make errors. Your PC will not explode. In this paper, most of the code is written in a manner that allows you to effortlessly copy and reproduce the output on your PC. Take advantage of this opportunity and go for it! Hands-on practice is far more enjoyable than merely reading through the material.

Here are some comments that may help you to learn efficiently:

- **Computers need clear and precise instructions to work:** They can't handle mistakes or unclear directions. Even small errors like a missing comma or an unclosed bracket can cause your code to not work. Computers do exactly what you tell them, no more and no less.

- **Copy, paste, and tweak**: While learning code from scratch is sometimes essential, you can speed up your work by modifying code that already exists. I call this the *"copy, paste, and tweak"* approach. It is a good way to learn code and get a job done quick.

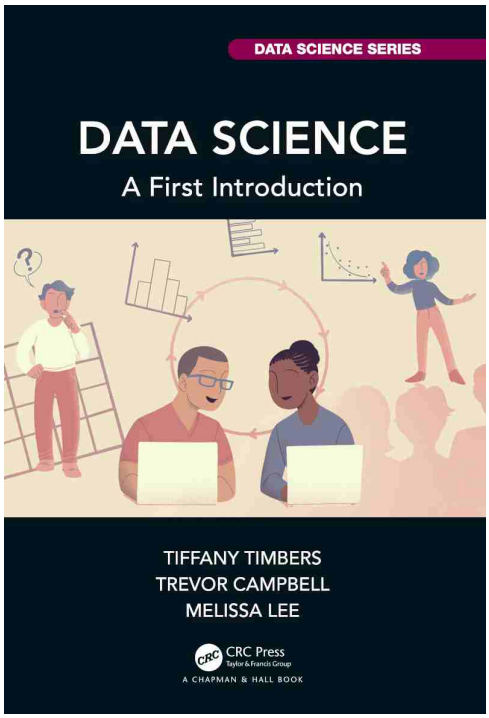
- **Have a purpose when coding**: Rather than learning to code for its own sake, it is more fun and you'll probably learn faster when you have a goal in mind. Try to analyze data that you are interested in. Another good exercise is replicating a research paper.
- **Practice is key**: The best method to improving your coding skills is through lots of practice. Consequently, these notes give you plenty of exercises.
- **Use ChatGPT**: The usage of supporting tools is not forbidden. ChatGPT can help you to understand code and brainstorm solutions. However, it's important to know that ChatGPT might suggest complex methods when there are shorter and more elegant solutions available Absolute beginners might find ChatGPT's solutions overwhelming and have difficulties to tweak the proposed sketch of a solution. So, use it thoughtfully.

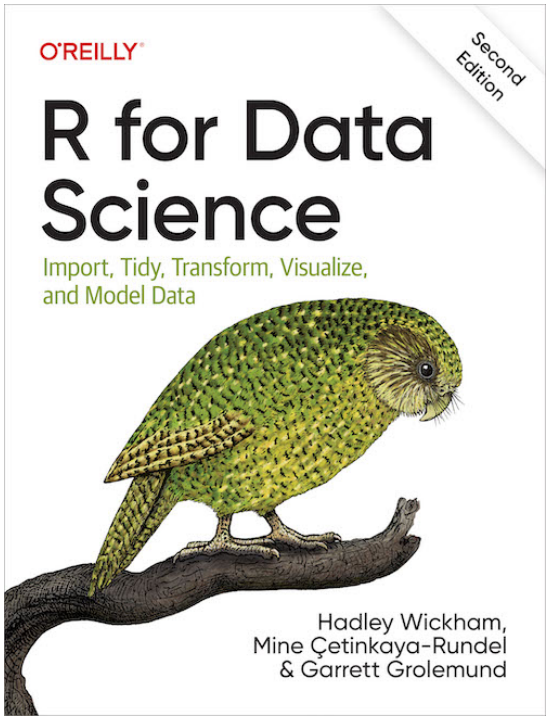
1.3. Learning resources

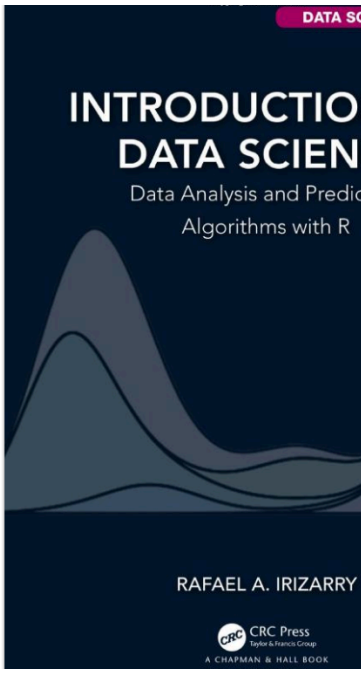
Thousand of freely available books and resources exist. bookdown.org and the Big Book of R are two vast collections of links to R books that might verify my claim.

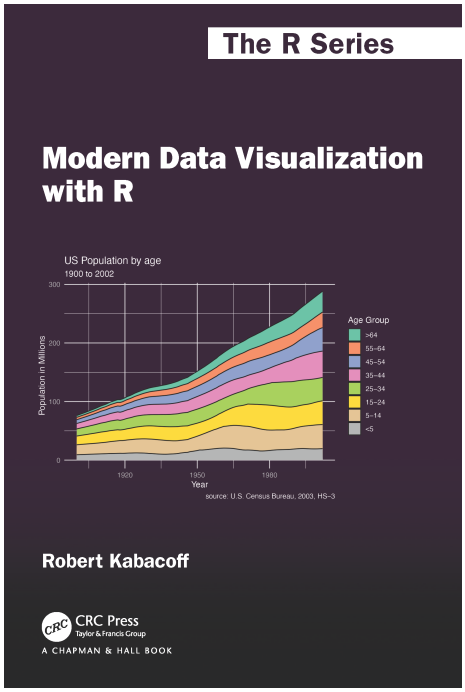
In RStudio you find in the right side at the bottom a panel that is called *Help*. There you find a lot of links, manuals, and references that offer you tons of resources to learn R for free including: education.rstudio.com and Links for Getting Help with R. At the top right of RStudio you find a panel called tutorial. Here you can install the `learnr` package that offers some nice interactive tutorials.

Since you may feel overwhelmed by the number of resources, I would like to highlight some books:









1. Timbers et al. [2022]: Data Science: A First Introduction is a free and up to date book that comes with exercises with worksheets that are available on UBC-DSCI GitHub repository
2. Wickham and Grolemund [2023]: R for Data Science: Import, Tidy, Transform, Visualize, and Model Data is the most popular source to learn R. It focuses on introducing the tidyverse package and is freely available online.
3. Irizarry [2022]: Introduction to Data Science: Data Analysis and Prediction Algorithms With R is a complete, up to date, and applied introduction.
4. Venables et al. [2022] An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics is a manual from the R Core Development Team that shows how to use R without having to install and load additional packages.
5. Neth [2023]: Data Science for Psychologists is a comprehensive introduction to R and data science for non experts of both programming and data science. It uses a variety of data types and includes many examples and exercises.
6. Kabacoff [2024]: Modern Data Visualization with R teaches how to create graphs from scratch providing a lot of examples that you can copy, paste and tweak.

Some other sources that are worth mentioning are these:

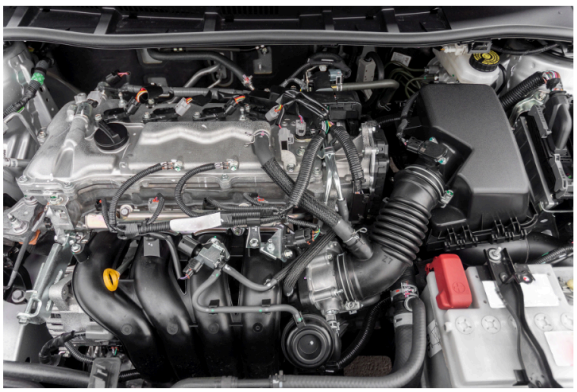
- The search engine www.rseek.org is R specific and often better than www.google.com as it only searches for content that has to do with the programming language R.
- On rdocumentation.org you can find the complete documentation of all R packages.
- Many find these cheatsheets helpful.

1.4. What are R and RStudio?

Throughout this book, I will assume that you are using R via RStudio. First time users often confuse the two. At its simplest, R is like a car's engine while RStudio is like a car's dashboard as illustrated in Figure Figure 1.1.

More precisely, R is a programming language that runs computations, while RStudio is an *integrated development environment (IDE)* that provides an interface by adding many convenient

Figure 1.1.: Analogy of difference between R and RStudio

R: Engine	RStudio: Dashboard
	

features and tools. So just as the way of having access to a speedometer, rearview mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

Much as we don't drive a car by interacting directly with the engine but rather by interacting with elements on the car's dashboard, we won't be using R directly but rather we will use RStudio's interface. After you install R and RStudio on your computer, you'll have two new *programs* (also called *applications*) you can open. We'll always work in RStudio and not in the R application. Figure Figure 1.2 shows what icon you should be clicking on your computer.

Figure 1.2.: Icons of R versus RStudio on your computer

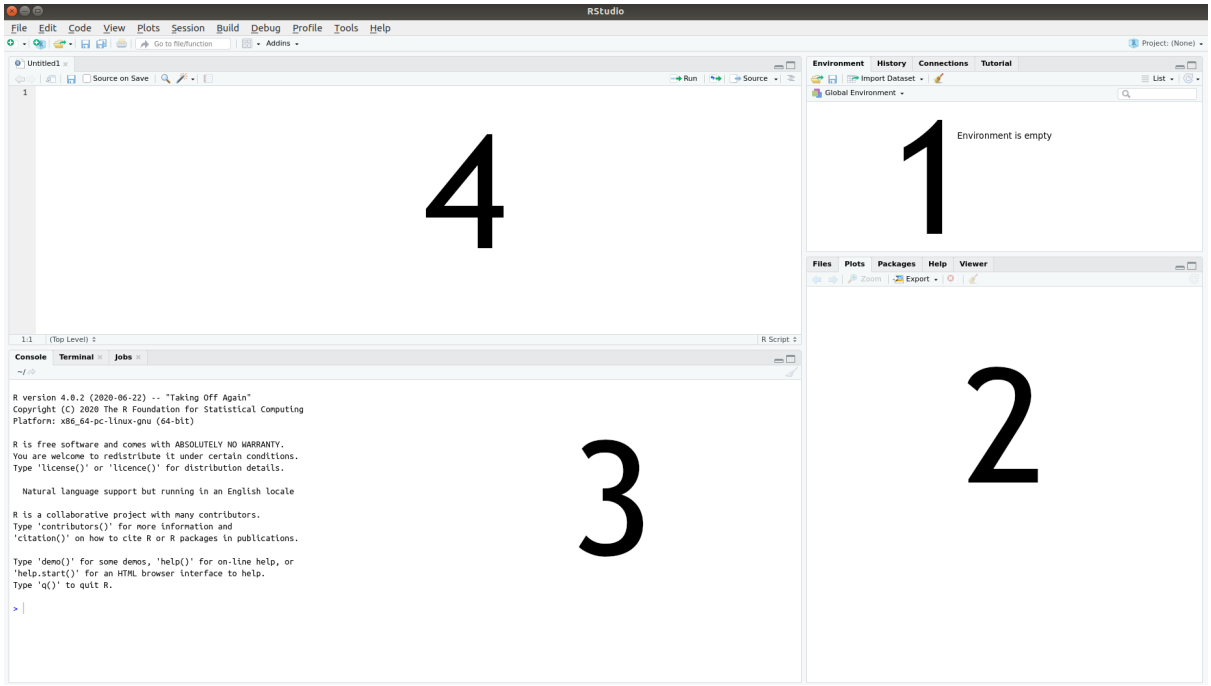
R: Do not open this	RStudio: Open this

After you open RStudio, you should see something similar to Figure Figure 1.3 where three or four panels dividing the screen.

1. The *Environment* panel, where a list of the data you have imported and created can be found.
2. The *Files*, *Plots* and *Help* panel, where you can see a list of available files, will be able to view graphs that you produce, and can find help documents for different parts of R.
3. The *Console* panel, used for running code. This is where we'll start with the first few examples.
4. The *Script* panel, used for writing code. This is where you'll spend most of your time working.

The *Console* panel will contain R's startup message, which shows information about which version of R you're running. My startup message at the time of writing was as follows:

Figure 1.3.: RStudio interface to R




```
R version 4.3.3 (2024-02-29) -- "Angel Food Cake"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

 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.
```

If you don't have panel number 4, open it by opening an existing R-script or creating a new one. You can create a new on by clicking *Ctrl+Shift+N* (alternatively, you can use the menu: File→New File→R Script).

You can resize the panels as you like, either by clicking and dragging their borders or using the minimise/maximise buttons in the upper right corner of each panel. Clicking *Ctrl++* and *Ctrl+-* allows to make the fonts larger or smaller.

1.5. How to use R and RStudio without installation

If you don't want to install R on your PC or you don't have admin rights to do so or if you want to run R on your tablet (IPad or Chromebook) or even your smartphone, you can use RStudio online doing *cloud computing* on https://posit.cloud. Posit Cloud (formerly RStudio Cloud) is a cloud-based solution that allows anyone to use RStudio online and navigate it through your web browser. It is free for individuals with some restrictions and limited capacities.

1.6. Installing R and RStudio

To install R on your personal computer (Windows, Mac, Linux), you will first need to download and install both R and RStudio (Desktop version) on your computer. It is important that you install R first and then install RStudio.

1. **Do this firstly:** Download and install R by going to https://cloud.r-project.org/.
    - If you are a Windows user: Click on "Download R for Windows", then click on "base", then click on the Download link.
    - If you are macOS user: Click on "Download R for (Mac) OS X", then under "Latest release:" click on R-X.X.X.pkg, where R-X.X.X is the version number. For example, the latest version of R as of March 29, 2024 was R-4.3.3.
    - If you are a Linux user: Click on "Download R for Linux" and choose your distribution for more information on installing R for your setup.
2. **Do this secondly:** Download and install RStudio at https://www.rstudio.com/products/rstudio/download/.
    - Scroll down to "Installers for Supported Platforms" near the bottom of the page.
    - Click on the download link corresponding to your computer's operating system.

1.7. What is a function in R?

R is a *functional programming language*. If you want R to do something, you need to use a function. Or, in the words of Chambers [2017, p. 4]:

> "Everything that happens is a function call."

For example, when you like to exit R, you do it with the function `q()`:

```
> q()
Save workspace image? [y/n/c]:
```

If you want to specify what exactly you want R to do for you, you need to refer to the arguments of a function. For example, if you don't want to be asked interactively what you want to do with your workspace (this is the place where you store all your objects, see section Section 1.8), you can do this with an argument that is part of the `q()` function:

```
> q(save = "no")
```

To learn more about a function, you can access its documentation by typing a question mark followed by the function name into the Console:

```
?q()
```

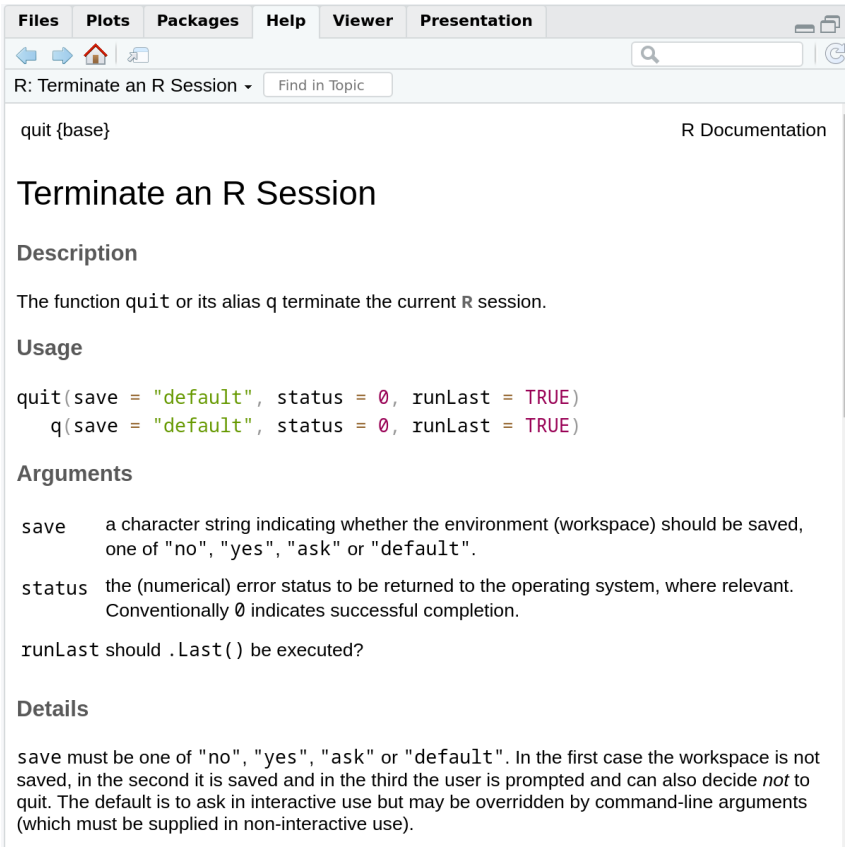
Unfortunately, the documentation can sometimes be a bit confusing for beginners in applied contexts. However, the documentation for all functions is structured similarly, typically featuring several key sections:

- **Description**: A brief overview of what the function does.
- **Usage**: How to use the function, including the function name and its arguments.
- **Arguments**: Detailed descriptions of each argument the function accepts, including what types of values are expected.
- **Details**: Additional details about the function's behavior and any important notes.
- **Examples**: Practical examples demonstrating how to use the function in various contexts.

Understanding these sections can significantly enhance your ability to navigate and utilize R.

An excerpt of the *R Documentation* for the function `q()` is shown in Figure 1.4. Here, we observe that the function has three arguments that you can manipulate. If you do not specify any of these arguments explicitly, we see that *by default*, R sets the three arguments as shown.

Figure 1.4.: The R Documentation of `q()`



1.8. What are objects in R?

R is an object oriented programming language. That means,

> "everything that exists in R is an object" [Chambers, 2017, p. 4].

Objects are the fundamental units that are used to store information. Objects can be a variety of data types, including vectors, matrices, data frames, lists, functions. Moreover, you can store empirical results, tables, figures and many more in form of so-called objects. All objects are shown in the *workspace* which is shown in the *Environment* panel.

In R, you can show the content of the workspace with `ls()`. The function `rm()` allows to remove objects and with `rm(list=ls())` you clear all objects from the workspace.

1.9. What are R packages?

A package is a collection of functions, data sets and other R objects that are all grouped together under a common name. More than 20,000 packages are available at the official repository (CRAN)[1] and many more are publicly available through the internet.

However, before we get started, there's a critical distinction that you need to understand, which is the difference between having a package **installed** on your computer, and having a package **loaded** in R. When you install R on your computer only a small number of packages come bundled with the basic R installation. The installed packages are on your computer. The critical thing to remember is that just because something is on your computer doesn't mean R can use it. In order for R to be able to *use* one of your installed packages, that package must also be *loaded*. Generally, when you open up R, only a few of these packages (about 7 or 8) are actually loaded.

> **i** Package management
>
> 1. A package must be installed before it can be loaded.
> 2. A package must be loaded before it can be used.

We only need to install a package once on our computer. However, to use the package, we need to load it every time we start a new R environment or R Studio, respectively.

1.9.1. Package installation

To install an R package you can use the GUI of R Studio or the command line. In R Studio you can click on the *Packages* tab, then on the *Install* button, then you must search for a package and click *Install*. An alternative way to install a package is by typing

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

in the console pane of RStudio and pressing Return/Enter on your keyboard. Note you must include the quotation marks around the name of the package.

If you want to update a previously installed package to a newer version, you need to re-install it by repeating the earlier steps or you use `update.packages()`. To uninstall packages you can use `remove.packages()`.

[1] 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, see: https://cran.r-project.org.

> How to speed up the installation of packages
>
> The installation of packages can take some time. However, if your CPU has many cores, you can speed up the process a lot using the argument `Ncpus` like this `update.packages(ask = F, Ncpus = 4L)`. This option allows you to adjust the number of parallel processes R can use on your PC. So, if you have a CPU with many cores you can increase that number. A tutorial on how to set the number of cores used by R permanently can be found here.

1.9.2. Package loading

Recall that after you've installed a package, you need to *load it*. We do this by using the `library()` command. For example, to load the `ggplot2` package, run the following code in the console pane. What do we mean by "run the following code"? Either type or copy-and-paste the following code into the console pane and then hit the Enter key.

```
library("ggplot2")
```

If after running the earlier code, a blinking cursor returns next to the `>` "prompt" sign, it means you were successful and the `ggplot2` package is now loaded and ready to use. If, however, you get a red "error message" that reads

```
Error in library(ggplot2) : there is no package called 'ggplot2'
```

It means that you didn't successfully install it. If you get this error message, go back to section Section 1.9.1 on R package installation and make sure to install the `ggplot2` package before proceeding.

One very common mistake new R users make when wanting to use particular packages is they forget to *load* them first by using the `library()` command we just saw. Remember: *you have to load each package you want to use every time you start RStudio.* If you don't first *load* a package, but attempt to use one of its features, you'll see an error message similar to:

```
Error: could not find function
```

R is informing you that you are attempting to use a function from a package that has not yet been loaded. Forgetting to load packages is a common mistake made by new users, and it can be a bit frustrating to get used to at first. However, with practice, it will become second nature for you. Unloading packages can be done with `detach(package:ggplot2, unload=TRUE)`.

1.9.3. Simplified package management with `p_load`

There is a convenient way to handle both package installation and loading without the need for manual checks. The `p_load()` function does this seamlessly by verifying if a package is installed; if not, it installs the package.

For instance, instead of the traditional approach:

```
install.packages(
  c("arrow", "babynames", "curl", "duckdb", "gapminder",
    "ggrepel", "ggridges", "ggthemes", "hexbin", "janitor", "Lahman",
    "leaflet", "maps", "nycflights13", "openxlsx", "palmerpenguins",
    "repurrrsive", "tidymodels", "writexl")
  )
library(
  c("arrow", "babynames", "curl", "duckdb", "gapminder",
    "ggrepel", "ggridges", "ggthemes", "hexbin", "janitor", "Lahman",
    "leaflet", "maps", "nycflights13", "openxlsx", "palmerpenguins",
    "repurrrsive", "tidymodels", "writexl")
  )
```

You can streamline the process as follows:

```
# Install and load the required packages using p_load
if (!require(pacman)) install.packages("pacman")

pacman::p_load(
  arrow, babynames, curl, duckdb, gapminder,
  ggrepel, ggridges, ggthemes, hexbin, janitor, Lahman,
  leaflet, maps, nycflights13, openxlsx, palmerpenguins,
  repurrrsive, tidymodels, writexl
)
```

The line

```
if (!require(pacman)) install.packages("pacman")
```

ensures the installation of the `pacman` package, which is necessary for using the `p_load` function.

Before you load packages in a script, I recommend to *unload* all other packages with

```
pacman::p_unload(all)
```

to avoid conflicts of functions (see Section 3.5).

1.10. Base R and the tidyverse universe

Upon successfully installing R, you gain access to functions that are part of *Base R*. This includes standard packages automatically installed and loaded with each R session, such as `stats`, `utils`, and `graphics`, providing a broad spectrum of functionalities for statistical analysis and graphical capabilities [see Venables et al., 2022]. However, the syntax in *Base R* can become complex and less intuitive for users. Consequently, many individuals, including Hadley Wickham, the Chief Data Scientist at *Posit* (formerly RStudio), and his team, have developed an alternative suite of packages known as the `tidyverse`. These packages share a common philosophy and syntax, emphasizing readability and ease of use. We will heavily utilize the `tidyverse` in the following sections.

Figure 1.5.: The tidyverse universe

The R package `tidyverse` (see Figure 1.5) is a comprehensive collection of R packages including popular packages such are `ggplot2`, `dplyr`, `tidyr`, `readr`, `purrr`, `tibble`, `stringr`, and `forcats`, which together offer extensive capabilities for data modeling, transformation, and visualization.

How to do data science with tidyverse is the subject of multiple books and tutorials. In particular, the popular book *R for Data Science* by Wickham and Grolemund [2023] is all about the tidyverse universe. Thus, I highly recommend reading sections Workflow: basics), Data transformation, and Data tidying. Additionally, explore www.tidyverse.org for more resources, and consider completing the *tidyverse* module in my `swirl` package, *swirl-it*, as detailed in section Chapter 2.

To install and load `tidyverse` run the following lines of code:

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

1.11. Are there some guidelines for working with R?

To avoid running into issues with R, it's important to be aware of the some conventions, rules, and best practices that apply to the language. While it can be tedious to go through all of the do's and don'ts in detail, following them can make your life with R much easier. Trust me, the benefits of adhering to these guidelines will become clear over time. Here is a non-exhaustive list:

1. Do remember that R programming language is case sensitive.
2. Do start names of objects such as vectors, numbers, variables, and data frames with a letter, not a number.
3. Do avoid using dots in names of objects.
4. Do avoid using certain keywords in naming objects, such as if, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, NaN, and NA.
5. Do use front slash / instead of backslash \ for navigating the file system (see Appendix A).
6. Do not use whitespace and indentation for naming files, directories, or objects.
7. Do not ignore warnings or errors unless you know what they mean.
8. Do define objects to represent hard-coded values instead of using them directly in code.
9. Do remember to (install and) load packages that contain functions you want to use.

10. Do remember to set your working directory. (Tip: Use R Studio projects, see Section A.4)
11. Do remember to comment your code.
12. Do use `<-` instead of `=` for assignment.
13. Do clear the environment at the beginning of a script: `rm(list = ls())`

An exhaustive style guide on how to write code can be found here.

2. An interactive introduction using swirl

This section is designed to kickstart your journey into data science with R through the R package `swirl` that offers an interactive learning platform. `swirl` teaches you R programming and data science interactively, at your own pace, and right in the R console! You get immediate feedback on your progress. If you are new to R, have no fear. `swirl` will walk you through each of the steps required to employ Rstudio and R for your purpose.

For those seeking additional or alternative resources beyond swirl, exploring other introductory textbooks and resources on R is highly recommended. Please consider the resources I discuss in Section 1.3. One notable example is Irizarry [2022] who provides a comprehensive and conservative approach to understanding R.

2.1. Set up `swirl`

To install `swirl` and my learning modules, please follow my instructions precisely!

Open Rstudio and type in the console the following:

```
install.packages("swirl")
library("swirl")
install_course_github("hubchev", "swirl-it")
swirl()
```

The above four lines of code do the following:

- Install the `swirl` package, ensuring it's available for use in R.
- Load the `swirl` package, making its functions accessible.
- Install my `swirl` course that is hosted on GitHub, making its functions accessible.
- By entering swirl into the Console (located at the bottom-left in RStudio) and pressing the Enter key, you initiate swirl. This begins your interactive learning experience with the package.

> Tip 1: If the course has failed to install,
>
> you can try to download the file `swirl-it.swc` from github.com/hubchev/swirl-it and install the course with loading the `swirl` package and typing `install_course()` into the console.

After initiating the `swirl` environment, follow the instructions displayed in the Console. Specifically, select the *swirl-it* course and the *huber-intro-1* learning module to begin. You can exit `swirl` at any moment by typing `bye()` into the Console or pressing the *Esc* key on your keyboard.

2.2. swirl-it: huber-intro-1

Click to see the full content of the module

Welcome to this swirl course. If you find any errors or if you have suggestions for improvement, please let me know via stephan.huber@hs-fresenius.de.
The RStudio interface consists of several windows. You can change the size of the windows by dragging the grey bars between the windows. We'll go through the most important windows now.
Bottom left is the Console window (also called command window/line). Here you can type commands after the `>` prompt and R will then execute your command. This is the most important window, because this is where R actually does stuff.
Top left is the Editor window (also called script window). Here collections of commands (scripts) can be edited and saved. When you do not get this window, you can open it with 'File' > 'New' > 'R script'.
Just typing a command in the editor window is not enough, it has to be send to the Console before R executes the command. If you want to run a line from the script window (or the whole script), you can click 'Run' or press 'CTRL+ENTER' to send it to the command window.
The shortcut to send the current line to the console and run it there is _____.

a) CTRL+SHIFT
b) CTRL+ENTER
c) CTRL+SPACE
d) SHIFT+ENTER

Hint: You find all shortcuts in the menu at Tools > Keyboard Shortcuts Help or click ALT+SHIFT+K. If you are a Mac user, your shortcut is 'Cmd+Return' instead of 'SHIFT+ENTER'. To move on type skip().

Solution

answer: b

Top right is the environment window (a.k.a workspace). Here you can see which data R has in its memory. You can view and edit the values by clicking on them.
Bottom right is the plots / packages / help window. Here you can view plots, install and load packages or use the help function.
The first thing you should do whenever you start Rstudio is to check if you are happy with your working directory. That directory is the folder on your computer in which you are currently working. That means, when you ask R to open a certain file, it will look in the working directory for this file, and when you tell R to save a data file or figure, it will save it in the working directory.
You can check your working directory with the function `getwd()`. So let's do that. Type in the command window `getwd()` .

```
getwd()
```

```
[1] "/home/sthu/Dropbox/hsf/courses/dsr"
```

Are you happy with that place? if not, you should set your working directory to where all your data and script files are (or will be). Within RStudio you can go to 'Session' > 'Set working directory' > 'Choose directory'. Please do this now.

Instead of clicking, you can use the function `setwd("/YOURPATH")`. For example, `setwd("/Users/MYNAME/MYFOLDER")` or `setwd("C:/Users/jenny/myrstuff")`. Make sure that the slashes are forward slashes and that you do not forget the apostrophes. R is case sensitive, so make sure you write capitals where necessary.
Whenever you want R to do something you need to use a function. It is like a command. All functions of R are organized in so-called packages or libraries. With the standard installation many packages are already installed. However, many more exist and some of them are really cool. For example, with `installed.packages()` all installed packages are listed. Or, with `swirl()`, you started swirl.
Of course, you can also go to the Packages window at the bottom right. If the box in front of the package name is ticked, the package is loaded (activated) and can be used. To see via Console which packages are loaded type in the console `(.packages())`

```
(.packages())
```

```
[1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
[7] "base"
```

There are many more packages available on the R website. If you want to install and use a package (for example, the package called `geometry`) you should first install the package. Type `install.packages("geometry")` in the console. Don't be afraid about the many messages. Depending on your PC and your internet connection this may take some time.

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

After having installed a package, you need to load the package. That is a bit annoying but essential. Type in `library("geometry")` in the Console. You also did this for the swirl package (otherwise you couldn't have been doing these exercises).

```
library("geometry")
```

Check if the package is loaded typing `(.packages())`

```
(.packages())
```

Now, let's get started with the real programming.
R can be used as a calculator. You can just type your equation in the command window after the `>`. Type `10^2 + 36`.

```
10^2 + 36
```

```
[1] 136
```

And R gave the answer directly. By the way, spaces do not matter.
If you use brackets and forget to add the closing bracket, the `>` on the command line changes into a +. The + can also mean that R is still busy with some heavy computation. If you want R to quit what it was doing and give back the `>`, press ESC.
You can also give numbers a name. By doing so, they become so-called variables which can be used later. For example, you can type in the command window `A <- 4`.

```
A <- 4
```

The `<-` is the so-called assignment operator. It allows you to assign data to a named object in order to store the data.
Don't be confussed about the term object. All sorts of data are stored in so-called objects in R. All objects of a session are shown in the Environment window. In the second part of this course, I will introduce different data types.
You can see that `A` appeared in the environment window in the top right corner, which means that R now remembers what `A` is.
You can also ask R what `A` is. Just type `A` in the command window.

```
A
```

```
[1] 4
```

You can also do calculations with A. Type `A * 5` .

```
A*5
```

```
[1] 20
```

If you specify A again, it will forget what value it had before. You can also assign a new value to A using the old one. Type `A <- A + 10` .

```
A <- A + 10
```

You can see that the value in the environment window changed.
To remove all variables from R's memory, type `rm(list=ls())` .

```
rm(list=ls())
```

You see that the environment window is now empty. You can also click the broom icon (`clear all`) in the environment window. You can see that RStudio then empties the environment window. If you only want to remove the variable `A`, you can type `rm(A)`.
Like in many other programs, R organizes numbers in scalars (a single number, 0-dimensional), vectors (a row of numbers, also called arrays, 1-dimensional) and matrices (like a table, 2-dimensional).
The A you defined before was a scalar. To define a vector with the numbers 3, 4 and 5, you need the function `c()`, which is short for concatenate (paste together). Type `B=c(3,4,5)`.

```
B=c(3,4,5)
```

If you would like to compute the mean of all the elements in the vector `B` from the example above, you could type `(3+4+5)/3`. Try this

```
(3+4+5)/3
```

```
[1] 4
```

But when the vector is very long, this is very boring and time-consuming work. This is why things you do often are automated in so-called functions. For example, type `mean(x=B)` and guess what this function `mean()` can do for you.

```
mean(x=B)
```

```
[1] 4
```

Within the brackets you specify the arguments. Arguments give extra information to the function. In this case, the argument `x` says of which set of numbers (vector) the mean should be computed (namely of `B`). Sometimes, the name of the argument is not necessary; `mean(B)` works as well. Try it.

```
mean(B)
```

```
[1] 4
```

Compute the sum of 4, 5, 8 and 11 by first combining them into a vector and then using the function sum. Use the function `c` inside the function `sum`.

```
sum(c(4,5,8,11))
```

```
[1] 28
```

The function rnorm, as another example, is a standard R function which creates random samples from a normal distribution. Type `rnorm(10)` and you will see 10 random numbers

```
rnorm(10)
```

```
 [1]  0.72634301  0.33433982 -0.27355458  0.08855597  1.01658716 -0.72194123
 [7] -0.41776736  0.21526842  0.10250695  1.52416965
```

Here rnorm is the function and the 10 is an argument specifying how many random numbers you want - in this case 10 numbers (typing `n=10` instead of just 10 would also work). The result is 10 random numbers organised in a vector with length 10.
If you want 10 random numbers out of normal distribution with mean 1.2 and standard deviation 3.4 you can type `rnorm(10, mean=1.2, sd=3.4)`. Try this.

```
rnorm(10, mean=1.2, sd=3.4)
```

```
 [1] -5.6926203  2.4580820 -1.5211195 -4.7333845  2.4626019  0.6828775
 [7]  0.6281611  1.9308906  2.4799612  5.1667933
```

This shows that the same function (`rnorm()`) may have different interfaces and that R has so called named arguments (in this case `mean` and `sd`).
Comparing this example to the previous one also shows that for the function rnorm only the first argument (the number 10) is compulsory, and that R gives default values to the other so-called optional arguments. Use the help function to see which values are used as default by typing `?rnorm`.

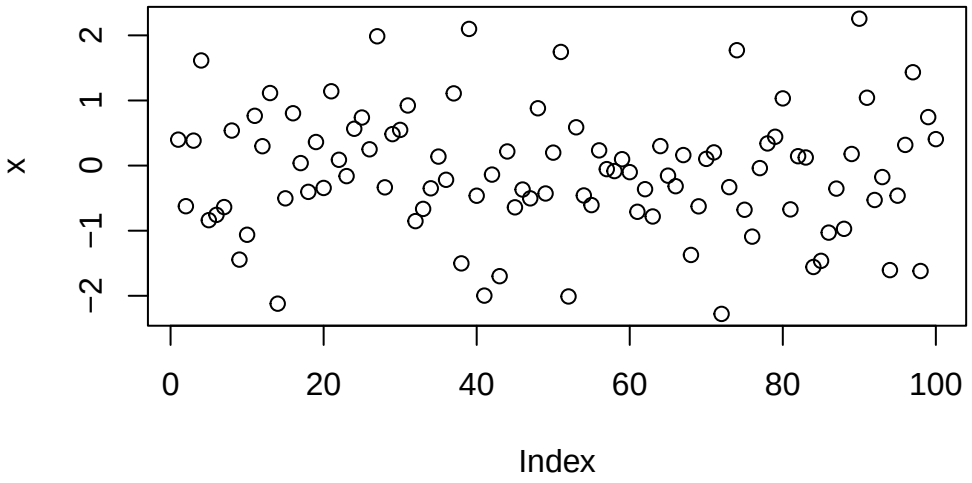
```
?rnorm
```

You see the help page for this function in the help window on the right. RStudio has a nice features such as autocompletion and snapshots of the R documentation. For example, when you type rnorm(in the command window and press TAB, RStudio will show the possible arguments.
You can also store the output of the function in a variable. Type `x=rnorm(100)`.

```
x=rnorm(100)
```

Now 100 random numbers are assigned to the variable x, which becomes a vector by this operation. You can see it appears in the Environment window.
R can also make graphs. Type `plot(x)` for a very simple example.

```
plot(x)
```



The 100 random numbers are now plotted in the plots window on the right.
You now are more familiar to RStudio and you know some basic R stuff. In particular, you know…
…that everything in R is said with functions,
…that functions can but don't have to have arguments,
…that you can install packages which contain functions,
…that you must load the installed packages every time you start a session in RStudio, and
…that this is just the beginning. Thus, please continue with the second module of this introduction.

After you have successfully finished learning module *huber-intro-1* please go ahead with the learning module *huber-intro-2* that is also part of my swirl course *swirl-it*.

2.3. swirl-it: huber-intro-2

Click to see the full content of the module

Welcome to the second module. Again, if you find any errors or if you have suggestions for improvement, please let me know via stephan.huber@hs-fresenius.de .
Before you start working, you should set your working directory to where all your data and script files are or should be stored. Within RStudio you can go to 'Session'> 'Set working directory', or you can type in `setwd(YOURPATH)`. Please do this now.

```
setwd(getwd())
```

Hint: Instead of clicking, you can also type setwd("path"), where you replace "path" with the location of your folder, for example setwd("D:/R/swirl").
R is an interpreter that uses a command line based environment. This means that you

have to type commands, rather than use the mouse and menus. This has many advantages. Foremost, it is easy to get a full transcript of everything you did and you can replicate your work easy.
As already mentioned, all commands in R are functions where arguments come (or do not come) in round brackets after the function name.
You can store your workflow in files, the so-called scripts. These scripts have typically file names with the extension, e.g., foo.R .
You can open an editor window to edit these files by clicking 'File' and 'New'. Try this. Under 'File' you also find the options 'Open file...', 'Save' and 'Save as'. Alternatively, just type CTRL+SHIFT+N.
You can run (send to the Console window) part of the code by selecting lines and pressing CTRL+ENTER or click 'Run' in the editor window. If you do not select anything, R will run the line your cursor is on.
You can always run the whole script with the console command source, so e.g. for the script in the file foo.R you type source('foo.R'). You can also click 'Run all' in the editor window or type CTRL+SHIFT+S to run the whole script at once.
Make a script called firstscript.R. Therefore, open the editor window with 'File' > 'New'. Type `plot(rnorm(100))` in the script, save it as `firstscript.R` in the working directory. Then type `source("firstscript.R")` on the command line.

```
source("firstscript.R")
```

Run your script again with `source("firstscript.R")`. The plot will change because new numbers are generated.

```
source("firstscript.R")
```

Hint: Type source("firstscript.R") again or type skip() if you are not interested.
Vectors were already introduced, but they can do more. Make a vector with numbers 1, 4, 6, 8, 10 and call it `vec1`.
Hint: Type vec1 <- c(1,4,6,8,10).

```
vec1 <- c(1,4,6,8,10)
```

Elements in vectors can be addressed by standard [i] indexing. Select the 5th element of this vector by typing `vec1[5]`.

```
vec1[5]
```

Replace the 3rd element with a new number by typing `vec1[3]=12`.

```
vec1[3] <- 12
```

Ask R what the new version is of `vec1`.

```
vec1
```

You can also see the numbers of `vec1` in the environment window. Make a new vector `vec2` using the `seq()` (sequence) function by typing `seq(from=0, to=1, by=0.25)` and check its values in the environment window.
Hint: Type vec2 <- seq(from=0, to=1, by=0.25).

```
vec2 <-seq(from=0, to=1, by=0.25)
```

Type `sum(vec1)`.

```
sum(vec1)
```

The function sum sums up the elements within a vector, leading to one number (a scalar). Now use + to add the two vectors.
Hint: Type vec1 + vec2.

```
vec1+vec2
```

If you add two vectors of the same length, the first elements of both vectors are summed, and the second elements, etc., leading to a new vector of length 5 (just like in regular vector calculus).
Matrices are nothing more than 2-dimensional vectors. To define a matrix, use the function matrix. Make a matrix with `matrix(data=c(9,2,3,4,5,6),ncol=3)` and call it `mat`.
Hint: Type `mat <- matrix(data=c(9,2,3,4,5,6),ncol=3)` or type `skip()` if you are not interested.

```
mat<-matrix(data=c(9,2,3,4,5,6),ncol=3)
```

The third type of data structure treated here is the data frame. Time series are often ordered in data frames. A data frame is a matrix with names above the columns. This is nice, because you can call and use one of the columns without knowing in which position it is. Make a data frame with `t = data.frame(x = c(11,12,14), y = c(19,20,21), z = c(10,9,7))`.

```
t <- data.frame(x = c(11,12,14), y = c(19,20,21), z = c(10,9,7))
```

Ask R what t is.
Hint: Type t or skip() if you are not interested.

```
t
```

The data frame is called `t` and the columns have the names `x`, `y` and `z`. You can select one column by typing `t$z`. Try this.

```
t$z
```

Another option is to type `t[["z"]]`. Try this as well.

```
t[["z"]]
```

Compute the mean of column `z` in data frame `t`.
Hint: Use function mean or type skip() if you are not interested.

```
mean(t$z)
```

In the following question you will be asked to modify a script that will appear as soon as you move on from this question. When you have finished modifying the script, save your

changes to the script and type `submit()` and the script will be evaluated. There will be some comments in the script that opens up. Be sure to read them!
Make a script file which constructs three random normal vectors of length 100. Call these vectors `x1`, `x2` and `x3`. Make a data frame called `t` with three columns (called `a`, `b` and `c`) containing respectively `x1`, `x1+x2` and `x1+x2+x3`. Call `plot(t)` for this data frame. Then, save it and type `submit()` on the command line.
Hint: Type plot(rnorm(100)) in the script, save it and type submit() on the command line.

```
# Text behind the #-sign is not evaluated as code by R.
# This is useful, because it allows you to add comments explaining what the script does.

# In this script, replace the ... with the appropriate commands.


x1 = ...
x2 = ...
x3 = ...
t = ...
plot(...)
```

Result

```
# Text behind the #-sign is not evaluated as code by R.
# This is useful, because it allows you to add comments explaining what the script does.

# In this script, replace the ... with the appropriate commands.

x1 = rnorm(100)
x2 = rnorm(100)
x3 = rnorm(100)
t = data.frame(a=x1, b=x1+x2, c=x1+x2+x3)
plot(t)
```

Do you understand the results?
Another basic structure in R is a list. The main advantage of lists is that the `columns` (they are not really ordered in columns any more, but are more a collection of vectors) don't have to be of the same length, unlike matrices and data frames. Make this list `L <- list(one=1, two=c(1,2), five=seq(0, 1, length=5))`.

```
L <- list(one=1, two=c(1,2), five=seq(0, 1, length=5))
```

The list `L` has names and values. You can type `L` to see the contents.

```
L
```

`L` also appeared in the environment window. To find out what's in the list, type `names(L)`.

```
names(L)
```

Add 10 to the column called five.
Hint: Type `L$five + 10`

```
L$five + 10
```

Plotting is an important statistical activity. So it should not come as a surprise that R has many plotting facilities. Type `plot(rnorm(100), type="l", col="gold")`.
Hint: The symbol between quotes after the ***type=****, is the letter* ***l****, not the number 1. To see the result you can also just type* ***skip()****.*

```
plot(rnorm(100), type="l", col="gold")
```

Hundred random numbers are plotted by connecting the points by lines in a gold color. Another very simple example is the classical statistical histogram plot, generated by the simple command hist. Make a histogram of 100 random numbers.
Hint: Type ***hist(rnorm(100))***

```
hist(rnorm(100))
```

The script that opens up is the same as the script you made before, but with more plotting commands. Type `submit()` on the command line to run it (you don't have to change anything yet).
Hint: Change plotting parameters in the script, save it and type submit() on the command line.

```
# Text behind the #-sign is not evaluated as code by R.
# This is useful, because it allows you to add comments explaining what the script does.

# Make data frame
x1 = rnorm(100)
x2 = rnorm(100)
x3 = rnorm(100)
t = data.frame(a=x1, b=x1+x2, c=x1+x2+x3)

# Plot data frame
plot(t$a, type='l', ylim=range(t), lwd=3, col=rgb(1,0,0,0.3))
lines(t$b, type='s', lwd=2, col=rgb(0.3,0.4,0.3,0.9))
points(t$c, pch=20, cex=4, col=rgb(0,0,1,0.3))

# Note that with plot you get a new plot window while points and lines add to the previous
```

Try to find out by experimenting what the meaning is of `rgb`, the last argument of `rgb`, `lwd`, `pch`, `cex`. Type `play()` on the command line to experiment. Modify lines 11, 12 and 13 of the script by putting your cursor there and pressing CTRL+ENTER. When you are finished, type `nxt()` and then `?par`.
Hint: Type ?par or type skip() if you are not interested.

```
?par
```

You searched for par in the R help. This is a useful page to learn more about formatting plots. Google 'R color chart' for a pdf file with a wealth of color options.
To copy your plot to a document, go to the plots window, click the 'Export' button, choose the nicest width and height and click 'Copy' or 'Save'.

> After having almost completed the second learning module, you are getting closer to become a nerd as you know…
> …that everything in R is stored in objects (values, vectors, matrices, lists, or data frames),
> …that you should always work in scripts and send code from scripts to the Console,
> …that you can do it if you don't give up.
> Please continue choosing another `swirl` learning module.

2.4. swirl-it: Data analytical basics

In my `swirl` modules *huber-data-1*, *huber-data-2*, and *huber-data-3* I introduce some very basic statistical principles on how to analyse data.

2.5. swirl-it: The `tidyverse` package

I compiled a short `swirl` module to introduce the *tidyverse* universe. This is a powerful collection of packages which I discuss later on. The learning module is also part of my *swirl-it* course.

2.6. Other `swirl` modules

You can also install some other courses. You find a list of courses here http://swirlstats.com/scn/index.html or here https://github.com/swirldev/swirl_courses.

I recommend this one as it gives a general overview on very basic principles of R:

```
library(swirl)
install_course_github("swirldev", "R_Programming_E")
swirl()
```

3. Work with R scripts

3.1. R Scripts: Why they are useful

Typing functions into the console to run code may seem simple, but this interactive style has limitations:

- Typing commands one at a time can be cumbersome and time-consuming.
- It's hard to save your work effectively.
- Going back to the beginning when you make a mistake is annoying.
- You can't leave notes for yourself.
- Reusing and adapting analyses can be difficult.
- It's hard to do anything except the basics.
- Sharing your work with others can be challenging.

That's where having a transcript of all the code, which can be re-run and edited at any time, becomes useful. An R script is precisely that - a plain text file that contains code and comments and this comes with advantages:

- Scripts provide a record of everything you did during your data analysis.
- You can easily edit and re-run code in a script.
- Scripts allow you to leave notes for yourself.
- Scripts make it easy to reuse and adapt analyses.
- Scripts allow you to do more complex analyses.
- Scripts make it easy to share your work with others.

3.2. Generate, write, and run R scripts

To **generate a script** you can

1. Go to the *File* menu, select *New File* and then choose *R Script* or
2. Use the keyboard shortcut *Ctrl+Shift+N* (Windows) or *Cmd+Shift+N* (Mac) or
3. Type the following command in the Console:

```
file.create("hello.R")
```

In the first two ways, a new R script window will open which can be edited and should be saved either by clicking on the *File* menu and selecting *Save*, clicking the disk icon, or by using the shortcut *Ctrl+S* (Windows) or *Cmd+S* (Mac). If you go for the third way, you need to open it manually.

Regardless of your preferred way of generating a script, we can now start **writing** our first script:

```
setwd("/home/sthu/Dropbox/hsf/23-ss/ds/")
x <- "hello world"
print(x)
```

Then save the script using the menus (File > Save) as hello.R.

The above lines of code do the following:

- `setwd()` allows to set the working directory. If you are not familiar with file systems, please read section Appendix A in the appendix.
- With the assignment operator `<-` we create an object that stores the words "hello world" in an object entitled `x`. In the next section Section 3.3 the assignment operator is further explained.
- With the third input we print the content of the object `x`.

So how do we **run the script**? Assuming that the `hello.R` file has been saved to your working directory, then you can run the script using the following command:

```
source( "hello.R" )
```

Suppose you saved the script in a sub-folder called *scripts* of your working directory, then you need to run the script using the following command:

```
source("./scripts/hello.R")
```

Just note that the dot, `.`, means the current folder. Instead of using the `source` function, you can click on the `source` button in Rstudio.

With the character `#` you can write a comment in a script and R will simply ignore everything that follows in that line onwards.

3.3. The assignment operator: <-

Suppose I'm trying to calculate how much money I'm going to make from this book. I agree, it is an unrealistic example but it will help you to understand R. Let's assume I'm only going to sell 350 copies. To create a variable called `sales` and assigns a value to it, we need to use the *assignment operator* of R, which is `<-` as follows:

```
sales <- 350
```

When you hit enter, R doesn't print out any output. If you are using Rstudio, and the *environment panel* you can see that something happened there, can you? It just gives you another command prompt. However, behind the scenes R has created a variable called `sales` and given it a value of `350`. You can check that this has happened by asking R to print the variable on screen. And the simplest way to do that is to type the name of the variable and hit enter.

```
sales
```

```
[1] 350
```

Worth a mentioning is the curious features of R that there are several different ways of making assignments. In addition to the `<-` operator, we can also use `->` and `=`. If you want to use `->`, you might expect from just looking at the symbol you should write it like this:

```
350 -> sales
```

However, it is common practice to use `<-` and I recommend only to use this one because it is easier to read in scripts.

3.4. Doing calculation in scripts

R can do any kind of arithmetic calculation with the arithmetic operators given in the table below. Using the assignment operator, R functions, and the features of a R script is easy and gives an idea how R works and how you should embrace the power of the programming language.

operation	operator	example input	example output
addition	+	10+2	12
subtraction	-	9-3	6
multiplication	*	5*5	25
division	/	10/3	3
power	^	5^2	25

So please copy and past the following lines of code into a R script of yours, try to run it on your PC, and try to understand it. Of course, you have to tweak the script a bit to make it run on your PC. For example, I doubt you have the same working directory that I decided to use.

```
# Set working directory
setwd("~/Dropbox/hsf/23-ss/ds")
# Create a vector that contains the sales data
sales_by_month <- c(0, 100, 200, 50, 3, 4, 8, 0, 0, 0, 0, 0)
sales_by_month
sales_by_month[2]
sales_by_month[4]
february_sales <- sales_by_month[2]
february_sales
sales_by_month[5] <- 25 # added May sales data
sales_by_month
# Do I have 12 month?
length( x = sales_by_month )
# Assume each unit costs 7 Euro, then the revenue is
price <- 7
revenue <- sales_by_month*price
revenue
# To get statistics for daily revenue we define the number of days:
days_per_month <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
# Calculate the daily revenue
revenue_per_day <- revenue/days_per_month
```

```
revenue_per_day
# round number
round(revenue_per_day)
```

Exercise 3.1. Round numbers (Solution 3.1)
Use the "?" to search for the documentation of all functions used. In particular, do you understand how the function `round()` works? What arguments does the function contain? How can you manipulate the pre-defined arguments. For example, can you calculate the rounded revenue per day with two or four digits? Try it out!

```
?round()
```

Solution 3.1. Round numbers (Exercise 3.1)

```
round(revenue_per_day, digits = 4)
```

3.5. User-defined functions and conflicts

One of the great strengths of R is the user's ability to add functions. Sometimes there is a small task (or series of tasks) you need done and you find yourself having to repeat it multiple times. In these types of situations it can be helpful to create your own custom function. The structure of a function is given below:

```
name_of_function <- function(argument1, argument2) {
    statements or code that does something
    return(something)
}
```

First you give your function a name. Then you assign value to it, where the value is the function. When defining the function you will want to provide the list of arguments required (inputs and/or options to modify behavior of the function), and wrapped between curly brackets place the tasks that are being executed on/using those arguments. The argument(s) can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way. Finally, you can return the value of the object from the function, meaning pass the value of it into the global environment. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function. Note, a function doesn't require any arguments.

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```
square_it <- function(x) {
   square <- x * x
   return(square)
}
```

Now, we can use the function as we would any other function. We type out the name of the function, and inside the parentheses we provide a numeric value `x`:

```
square_it(5)
```

```
[1] 25
```

Let us get back to script with sales and try to calculate the monthly growth rates of revenue using a self-written function.

The formula of a growth rate is clear:

$$g = \left(\frac{y_t - y_{t-1}}{y_{t-1}}\right) \cdot 100 = \left(\frac{y_t}{y_{t-1}} - 1\right) \cdot 100$$

So the challenge is to divide the value of `revenue` with the value of the previous period, a.k.a. the lagged value. Let us assume that the function `lag()` can give you exactly that value of a vector. Lets try it out:

```
lag(revenue)
```

```
 [1]    0  700 1400  350  175   28   56    0    0    0    0    0
attr(,"tsp")
[1]  0 11  1
```

```
(revenue/lag(revenue)-1)*100
```

```
 [1] NaN   0   0   0   0   0   0 NaN NaN NaN NaN NaN
attr(,"tsp")
[1]  0 11  1
```

Unfortunately, this does not work out. The `lag()` function does not work as we think it should. Well, the reason is simply that we are using the wrong function. The current `lag()` function is part of the stats package which is part of the package `stats` which is part of R base and is loaded automatically. The `lag()` function we aim to use stems from the dplyr package which we must install and load to be able to use it. So let's do it:

```
# check if the package is installed
find.package("dplyr")
```

```
[1] "/home/sthu/R/x86_64-pc-linux-gnu-library/4.2/dplyr"
```

```
# I already installed the package so I can just load it
# install.packages("dplyr")
library("dplyr")
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

This message informs us that among other functions the `lag()` function is *masked*. That means that now the function of the newly loaded package is active. This is one example of why I highly recommend to unload all packages at the beginning of a script and then to use `p_load` to install and load the packages that should be used in the upcoming script:

```
pacman::p_unload(all)
pacman::p_load(dplyr)
```

So, let's try again:

```
lag(revenue)
```

```
 [1]   NA    0  700 1400  350  175   28   56    0    0    0    0
```

```
(revenue/lag(revenue)-1)*100
```

```
 [1]   NA  Inf  100  -75  -50  -84  100 -100  NaN  NaN  NaN  NaN
```

That looks good now. And here is a way to calculate growth rates with a self-written function:

```
growth_rate <- function(x)(x/lag(x)-1)*100
growth_rate(revenue)
```

```
 [1]   NA  Inf  100  -75  -50  -84  100 -100  NaN  NaN  NaN  NaN
```

```
sales_gr_rate <- growth_rate(revenue)
sales_gr_rate
```

```
 [1]   NA  Inf  100  -75  -50  -84  100 -100  NaN  NaN  NaN  NaN
```

In R, all functions are written by users, and it is not uncommon for two people to name their functions identically. In such cases, we must resolve the conflict by choosing which function to use. To use the lag function from the `stats` package, you can use the double colon operator `::` like this `stats::lag(arguments)`.

4. Manage data

4.1. Import data

4.1.1. Vectors and matrices

We already got known to the c() function which allows to combine multiple values into a vector or list. Here are some examples how you can use this function to create vectors and matrices:

```
# defining multiple vectors using the colon operator `:`
v_a <- c(1:3)
v_a
```

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

```
v_b <- c(10:12)
v_b
```

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

```
# creating matrix
m_ab <- matrix(c(v_a, v_b), ncol = 2)
m_cbind <- cbind(v_a, v_b)
m_rbind <- rbind(v_a, v_b)

# print matrix
print(m_ab)
```

```
     [,1] [,2]
[1,]    1   10
[2,]    2   11
[3,]    3   12
```

```
print(m_cbind)
```

```
     v_a v_b
[1,]   1  10
[2,]   2  11
[3,]   3  12
```

```
print(m_rbind)
```

```
    [,1] [,2] [,3]
v_a    1    2    3
v_b   10   11   12
```

```
# defining row names and column names
rown <- c("row_1", "row_2", "row_3")
coln <- c("col_1", "col_2")

# creating matrix
m_ab_label <- matrix(m_ab, ncol = 2, byrow = FALSE,
             dimnames = list(rown, coln))

# print matrix
print(m_ab_label)
```

```
      col_1 col_2
row_1     1    10
row_2     2    11
row_3     3    12
```

The two most common formats to store and work with data in R are dataframe and tibble. Both formats store table-like structures of data in rows and columns. We will learn more on that in section Section 4.2.

```
# convert the matrix into dataframe
df_ab=as.data.frame(m_ab_label)
tbl_ab=data.frame(m_ab_label)
```

4.1.2. Open RData files

You can save some of your objects with `save()` or all with `save.image()`. Load data that are stored in the `.RData` format can be loaded with `load()`. Please note, when you delete an object in R, you cannot recover it by clicking some *Undo button*. With `rm()` you remove objects from your workspace and with `rm(list = ls())` you clear all objects from the workspace.

4.1.3. Open datasets of packages

The datasets package contains numerous datasets that are commonly used in textbooks. To get an overview of all the datasets provided by the package, you can use the command help(package = datasets). One such dataset that we will be using further is the `mtcars` dataset:

```
library("datasets")
head(mtcars, 3)
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
?mtcars  # data dictionary
```

4.1.4. Import data using public APIs

An API which stands for *application programming interface* specifies how computers can exchange information. There are many R packages available that provide a convenient way to access data from various online sources directly within R using the API of webpages. In most cases, it's better to download and import data within R using these tools than to navigate through the website's interface. This ensures that changes can be made easily at any time and that the data is always up-to-date. For instance, `wbstats` provides access to World Bank data, `eurostat` allows users to access Eurostat databases, `fredr` makes it easy to obtain data from the *Federal Reserve Economic Data (FRED)* platform, which offers economic data for the United States, `ecb` provides an interface to the European Central Bank's Statistical Data Warehouse, and the `OECD` package facilitates the extraction of data from the *Organization for Economic Cooperation and Development (OECD)*. Here is an example using the `wbstats` package:

```
# install.packages("wbstats")
library("wbstats")
# GDP at market prices (current US$) for all available countries and regions
df_gdp <- wb(indicator = "NY.GDP.MKTP.CD")
```

```
Warning: `wb()` was deprecated in wbstats 1.0.0.
i Please use `wb_data()` instead.
```

```
head(df_gdp, 3)
```

```
  iso3c date        value    indicatorID         indicator iso2c
2   AFE 2022 1.185138e+12 NY.GDP.MKTP.CD GDP (current US$)    ZH
3   AFE 2021 1.086531e+12 NY.GDP.MKTP.CD GDP (current US$)    ZH
4   AFE 2020 9.288802e+11 NY.GDP.MKTP.CD GDP (current US$)    ZH
                     country
2 Africa Eastern and Southern
3 Africa Eastern and Southern
4 Africa Eastern and Southern
```

```
glimpse(df_gdp)
```

```
Rows: 13,198
Columns: 7
$ iso3c       <chr> "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "A~
$ date        <chr> "2022", "2021", "2020", "2019", "2018", "2017", "2016", "2~
$ value       <dbl> 1.185138e+12, 1.086531e+12, 9.288802e+11, 1.006191e+12, 1.~
$ indicatorID <chr> "NY.GDP.MKTP.CD", "NY.GDP.MKTP.CD", "NY.GDP.MKTP.CD", "NY.~
```

```
$ indicator   <chr> "GDP (current US$)", "GDP (current US$)", "GDP (current US~
$ iso2c       <chr> "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH"~
$ country     <chr> "Africa Eastern and Southern", "Africa Eastern and Souther~
```

```
summary(df_gdp)
```

```
    iso3c               date               value           indicatorID
 Length:13198       Length:13198       Min.   :8.825e+06   Length:13198
 Class :character   Class :character   1st Qu.:2.435e+09   Class :character
 Mode  :character   Mode  :character   Median :1.786e+10   Mode  :character
                                       Mean   :1.224e+12
                                       3rd Qu.:2.264e+11
                                       Max.   :1.009e+14
  indicator            iso2c              country
 Length:13198       Length:13198       Length:13198
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character
```

4.1.5. Import various file formats

Figure 4.1.: The logo of the packages readr, haven, and readxl

RStudio provides convenient data import tools that can be accessed by clicking *File > Import Dataset*. In addition, tidyverse offers packages for importing data in various formats. This cheatsheet, for example, is about the packages `readr`, `readxl` and `googlesheets4`. The first allows you to read data in various file formats, including fixed-width files like .csv and .tsv. The package `readxl` can read in Excel files, i.e., .xls and .xlsx file formats and `googlesheets4` allows to read and write data from Google Sheets directly from R.

For more information, I recommend once again the second version book *R for Data Science* by Wickham and Grolemund [2023]. In particular, check out the "Data tidying" section for importing CSV and TSV files, the "Spreadsheets" section for Excel files, the "Databases" section

for retrieving data with SQL, the "Arrow" section for working with large datasets, and the "Web scraping" section for extracting data from web pages.

For an overview on packages for reading data that are provided by the tidyverse universe, see here.

4.1.6. Examples

Flat files such as CSV (Comma-Separated Values) are among the most common and straightforward data formats to work with.

```
data_csv <- read_csv("https://github.com/hubchev/courses/raw/main/dta/classdata.csv")
```

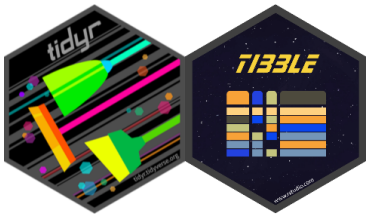
Excel files, due to their wide use in business and research, require a specific approach specifying sheets and cell ranges.

```
BWL_Zeitschriftenliste <-
  read_excel(
  "https://www.forschungsmonitoring.org/VWL_Zeitschriftenliste%202023.xlsx",
  sheet = "SJR main",
  range = "A1:D1977"
  )
```

4.2. Data

4.2.1. Data frames and tibbles

Figure 4.2.: The logos of the tidyr and tibble packages



Both *data frames* and *tibbles* are two of the most commonly used data structures in R for handling tabular data. A tibble actually is a data frame and you can use all functions that work with a data frame also with a tibble. However, a tibble has some additional features in printing and subsetting. Please note, data frames are provided by base R while tibbles are provided by the `tidyverse` package. This means that if you want to use tibbles you must load `tidyverse`. It turned out that it is helpful that a tibble has the folllowing features to simplify working with data: - Each vector is labeled by the variable name. - Variable names don't have spaces and are not put in quotes. - All variables have the same length. - Each variable is of a single type (numeric, character, logical, or a categorical).

4.2.2. Tidy data

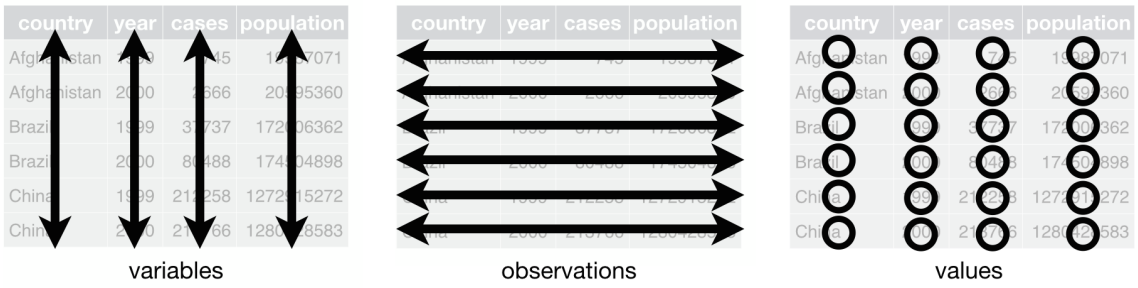
A popular quote from Hadley Wickham is that

> "tidy datasets are all alike, but every messy dataset is messy in its own way" [Hadley, 2014, p. 2].

It paraphrases the fact that it is a good idea to set rules how a dataset should structure its information to make it easier to work with the data. The tidyverse requires the data to be structured like is illustrated in Figure Figure 4.3. The rules are:

1. Each variable is a column and vice versa.
2. Each observation is a row and vice verse.
3. Each value is a cell.

Figure 4.3.: Features of a tidy dataset: variables are columns, observations are rows, and values are cells



Source: Wickham and Grolemund [2023].

Whenever data follow that consistent structure, we speak of *tidy data*. The underlying uniformity of tidy data facilitates learning and using data manipulation tools.

One difference between data frames and tibbles is that dataframes store the row names. For example, take the `mtcars` dataset which consists of 32 different cars and the names of the cars are not stored as rownames:

```
class(mtcars) # mtcars is a data frame
```

```
[1] "data.frame"
```

```
rownames(mtcars)
```

```
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
```

```
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
```

To store `mtcars` as a tibble, we can use the `as_tibble` function:

```
tbl_mtcars <- as_tibble(mtcars)
class(tbl_mtcars) # check if it is a tibble now
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
is_tibble(tbl_mtcars) # alternative check
```

```
[1] TRUE
```

```
head(tbl_mtcars, 3)
```

```
# A tibble: 3 x 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
```

When we look at the data, we've lost the names of the cars. To store the these, you need to first add a column to the dataframe containing the rownames and then you can generate the tibble:

```
tbl_mtcars <- mtcars |>
  rownames_to_column(var = "car") |>
  as_tibble()
class(tbl_mtcars)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
head(tbl_mtcars, 3)
```

```
# A tibble: 3 x 12
  car            mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Mazda RX4     21       6   160   110  3.9   2.62  16.5     0     1     4     4
2 Mazda RX4 W~  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3 Datsun 710    22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
```

4.2.3. Data types

In R, different data classes, or types of data exist:

- *numeric*: can be any real number
- *character*: strings and characters
- *integer*: any whole numbers
- *factor*: any categorical or qualitative variable with finite number of distinct outcomes
- *logical*: contain either `TRUE` or `FALSE`
- *Date*: special format that describes time

The following example should exemplify these types of data:

```
integer_var <- c(1, 2, 3, 4, 5)
numeric_var <- c(1.1, 2.2, NA, 4.4, 5.5)
character_var <- c("apple", "banana", "orange", "cherry", "grape")
factor_var <- factor(c("red", "yellow", "red", "blue", "green"))
logical_var <- c(TRUE, TRUE, TRUE, FALSE, TRUE)
date_var <- as.Date(c("2022-01-01", "2022-02-01", "2022-03-01", "2022-04-01", "2022-05-01"))

date_var[2] - date_var[5] # number of days in between these two dates
```

```
Time difference of -89 days
```

There are some special data values used in R that needs further explanation:

- `NA` stands for *not available* or *missing* and is used to represent missing or undefined values.
- `Inf` stands for *infinity* and is used to represent mathematical infinity, such as the result of dividing a non-zero number by zero. Can be positive or negative.

- `NULL` represents an empty or non-existent object. It is often used as a placeholder when a value or object is not yet available or when an object is intentionally removed.
- `NaN` stands for *not a number* and is used to represent an undefined or unrepresentable value, such as the result of taking the square root of a negative number. It can also occur as a result of certain arithmetic operations that are undefined. In contrast to `NA` it can only exist in numerical data.

4.3. The pipe operator: |>

The pipe operator, `%\>%`, comes from the *magrittr* package, which is also part of the tidyverse package. The pipe operator, `|>`, has been part of base R since version 4.1.0. For most cases, these two operators are identical. The pipe operator is designed to help you write code in a way that is easier to read and understand. As R is a functional language, code often contains a lot of parentheses, (and). Nesting these parentheses together can be complex and make your R code hard to read and understand, which is where `|>` comes to the rescue! It allows you to use the output of a function as the input of the next function. Consider the following example of code to explain the usage of the pipe operator:

```
# create some data `x`
x <- c(1, 1.002, 1.004, .99, .99)
# take the logarithm of `x`,
log_x <- log(x)
# compute the lagged and iterated differences (see `diff()`)
growth_rate_x <- diff(log_x)
growth_rate_x
```

```
[1]  0.001998003  0.001994019 -0.014042357  0.000000000
```

```
# round the result (4 digit)
growth_rate_x_round <- round(growth_rate_x, 4)
growth_rate_x_round
```

```
[1]  0.002  0.002 -0.014  0.000
```

That is rather long and we actually don't need objects `log_x`, `growth_rate_x`, and `growth_rate_x_round`. Well, then let us write that in a nested function:

```
round(diff(log(x)), 4)
```

```
[1]  0.002  0.002 -0.014  0.000
```

This is short but hard to read and understand. The solution is the "pipe":

```
# load one of these packages: `magrittr` or `tidyverse`
library(tidyverse)

# Perform the same computations on `x` as above
x |>
  log() |>
  diff() |>
  round(4)
```

```
[1]  0.002  0.002 -0.014  0.000
```

You can read the `|>` with *"and then"* because it takes the results of some function *"and then"* does something with that in the next. For example, reading out loud the following code would sound something like this:

- I take the mtcars data, *and then*
- I consider only cars with more than 4 cylinders, *and then*
- I group the cars by the number of cylinders the cars have, *and then*
- I summarize the data and show the means of miles per gallon (mpg) and horse powers (hp) by groups of cars that distinguish by their number of cylinders.

```
mtcars |>
  filter(cyl>4) |>
  group_by(cyl) |>
  summarise_at(c("mpg", "hp"), mean)
```

```
# A tibble: 2 x 3
    cyl   mpg    hp
  <dbl> <dbl> <dbl>
1     6  19.7  122.
2     8  15.1  209.
```

4.4. Tools to manipulate data

4.4.1. The %in% operator

`%in%` is used to subset a vector by comparison. Here's an example:

```
x <- c(1, 3, 5, 7)
y <- c(2, 4, 6, 8)
z <- c(1, 2, 3)

x %in% y
```

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

```
x %in% z
```

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

```
z %in% x
```

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

The %in% operator can be used in combination with other functions like `subset()` and `filter()`.

4.4.2. Extract operators

The *extract operators* are used to retrieve data from objects in R. The operator may take four forms, including `[]`, `[[]]`, and `$`.

`[]` allows to extract content from vector, lists, or data frames. For example,

```
a <- mtcars[3, ]
b <- mtcars["Datsun 710", ]
identical(a, b)
```

```
[1] TRUE
```

```
a
```

```
            mpg cyl disp hp drat   wt  qsec vs am gear carb
Datsun 710 22.8   4  108 93 3.85 2.32 18.61  1  1    4    1
```

extracts the third observation of the mtcars dataset, and

```
c <- mtcars[,"cyl"]
d <- mtcars[, 2]
identical(x, y)
```

```
[1] FALSE
```

```
c
```

```
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

extracts the variable/vector `cyl`.

The operators, `[[]]` and `$` extract a single item from an object. It is used to refer to an element in a list or a column in a data frame. For example,

```
e <- mtcars$cyl
f <- mtcars[["cyl"]]
identical(e, f)
```

```
[1] TRUE
```

```
e
```

```
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

will return the values of the variable `cyl` from the data frame `mtcars`. Thus, x$y is actually just a short form for x[[“y”]].

operation	operator	example input	answer

4.4.3. Logical operators

The extract operators can be combined with the *logical operators* (more precisely, I should call these *binary relational operators*) that are shown in Table 4.1.

Table 4.1.: Logical operators

operation	operator	example input	answer
less than	`<`	`2 < 3`	`TRUE`
less than or equal to	`<=`	`2 <= 2`	`TRUE`
greater than	`>`	`2 > 3`	`FALSE`
greater than or equal to	`>=`	`2 >= 2`	`TRUE`
equal to	`==`	`2 == 3`	`FALSE`
not equal to	`!=`	`2 != 3`	`TRUE`
not	`!`	`!(1==1)`	`FALSE`
or	`|`	`(1==1) | (2==3)`	`TRUE`
and	`&`	`(1==1) & (2==3)`	`FALSE`

Here are some examples: Select rows where the number of cylinders is greater than or equal to 6:

```
mtcars[mtcars$cyl >= 6, ]
```

```
                   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
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 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
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
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
```

Select rows where the number of cylinders is either 4 or 6:

```
mtcars[mtcars$cyl == 4 | mtcars$cyl == 6 , ]
```

```
                  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
Valiant          18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
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
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
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
Ferrari Dino     19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Volvo 142E       21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Select rows where the number of cylinders is 4 and the mpg is greater than 22:

```
mtcars[mtcars$cyl == 4 & mtcars$mpg > 22, ]
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
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
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
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
```

Select rows where the weight is less than 3.5 or the number of gears is greater than 4:

```
mtcars[mtcars$wt < 3.5 | mtcars$gear > 4, ]
```

```
                   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
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
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
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  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
```

Select rows where either mpg is greater than 25 or carb is less than 2, and the number of cylinders is either 4 or 8.

```
mtcars[(mtcars$mpg > 25 | mtcars$carb < 2) & mtcars$cyl %in% c(4,8), ]
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
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
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
```

4.4.4. If statements

In many cases, it's necessary to execute certain code only when a particular condition is met. To achieve this, there are several conditional statements that can be used in code. These include:

- The `if` statement: This is used to execute a block of code if a specified condition is true.
- The `else` statement: This is used to execute a block of code if the same condition is false.
- The `else` if statement: This is used to specify a new condition to test if the first condition is false.
- The `ifelse()` function: This is used to check a condition for every element of a vector.

The following examples should exemplify how these statements work:

```
# Example of if statement
if (mean(mtcars$mpg) > 20) {
  print("The average miles per gallon is greater than 20.")
}
```

```
[1] "The average miles per gallon is greater than 20."
```

```
# Example of if-else statement
if (mean(mtcars$mpg) > 20) {
  print("The average miles per gallon is greater than 20.")
} else {
  print("The average miles per gallon is less than or equal to 20.")
}
```

```
[1] "The average miles per gallon is greater than 20."
```

```
# Example of if-else if statement
if (mean(mtcars$mpg) > 25) {
  print("The average miles per gallon is greater than 25.")
} else if (mean(mtcars$mpg) > 20) {
  print("The average miles per gallon is between 20 and 25.")
} else {
  print("The average miles per gallon is less than or equal to 20.")
}
```

```
[1] "The average miles per gallon is between 20 and 25."
```

```
# Example of if_else function
mtcars_2 <- mtcars
mtcars_2$mpg_category <- ifelse(mtcars_2$mpg > 20, "High", "Low")
```

When you have a fixed number of cases and don't want to use a long chain of if-else statements, you can use `case_when()`;

```
mtcars_cyl <- mtcars %>%
  mutate(cyl_category = case_when(
    cyl == 4 ~ "four",
    cyl == 6 ~ "six",
    cyl == 8 ~ "eight"
  ))
```

The `mutate()` function is used to add the new variable, and `case_when()` is used to assign the values "four", "six", or "eight" to the new variable based on the number of cylinders in each car. Both functions are part of the `dplyr` package (see chapter Section 4.5).

4.5. Data manipulation with `dplyr`

The `dplyr` package is part of tidyverse and makes data manipulation easy:

- Reorder the rows (`arrange()`).
- Pick observations by their values (`filter()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).

Figure 4.4.: The logo of the `dplyr` package

- Collapse many values down to a single summary (`summarise()`).
- Rename variables (`rename()`).

These functions can all be used in conjunction with `group_by()` , which changes the scope of each function from operating on the entire dataset to operating on it group-by-group.

All functions work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame.
3. The result is a new data frame.

Here are some examples that may help to understand these functions:

```
library(tidyverse)

# load mtcars dataset
data(mtcars)

# arrange rows by mpg in descending order
mtcars |>
  arrange(desc(mpg))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
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
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	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
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2
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
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2

```
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
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
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
```

```
# filter rows where cyl = 4
mtcars |>
  filter(cyl == 4)
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
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
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
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
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
# select columns mpg, cyl, and hp
mtcars |>
  select(mpg, cyl, hp) |>
  head()
```

```
                  mpg cyl  hp
Mazda RX4        21.0   6 110
Mazda RX4 Wag    21.0   6 110
Datsun 710       22.8   4  93
Hornet 4 Drive   21.4   6 110
Hornet Sportabout 18.7   8 175
Valiant          18.1   6 105
```

```
# select columns all variables except wt and hp
mtcars |>
  select(-wt, -hp) |>
  head()
```

```
                  mpg cyl disp drat  qsec vs am gear carb
Mazda RX4         21.0   6  160 3.90 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 3.90 17.02  0  1    4    4
Datsun 710        22.8   4  108 3.85 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 3.08 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 3.15 17.02  0  0    3    2
Valiant           18.1   6  225 2.76 20.22  1  0    3    1
```

```
# select only variables starting with `c`
mtcars |>
  select(starts_with("c"))
```

```
                     cyl carb
Mazda RX4              6    4
Mazda RX4 Wag          6    4
Datsun 710             4    1
Hornet 4 Drive         6    1
Hornet Sportabout      8    2
Valiant                6    1
Duster 360             8    4
Merc 240D              4    2
Merc 230               4    2
Merc 280               6    4
Merc 280C              6    4
Merc 450SE             8    3
Merc 450SL             8    3
Merc 450SLC            8    3
Cadillac Fleetwood     8    4
Lincoln Continental    8    4
Chrysler Imperial      8    4
Fiat 128               4    1
Honda Civic            4    2
Toyota Corolla         4    1
Toyota Corona          4    1
Dodge Challenger       8    2
AMC Javelin            8    2
Camaro Z28             8    4
Pontiac Firebird       8    2
Fiat X1-9              4    1
Porsche 914-2          4    2
Lotus Europa           4    2
Ford Pantera L         8    4
Ferrari Dino           6    6
Maserati Bora          8    8
Volvo 142E             4    2
```

```
# summarize avg mpg by number of cylinders
mtcars |>
  group_by(cyl) |>
  summarize(avg_mpg = mean(mpg))
```

```
# A tibble: 3 x 2
    cyl avg_mpg
  <dbl>   <dbl>
1     4    26.7
2     6    19.7
3     8    15.1
```

```
# create new column wt_kg, which is wt in kg
mtcars |>
  select(wt) |>
  mutate(wt_kg = wt / 2.205) |>
  head()
```

```
                   wt    wt_kg
Mazda RX4         2.620 1.188209
Mazda RX4 Wag     2.875 1.303855
Datsun 710        2.320 1.052154
Hornet 4 Drive    3.215 1.458050
Hornet Sportabout 3.440 1.560091
Valiant           3.460 1.569161
```

```
# Create a new variable by calculating hp divided by wt
mtcars_new <- mtcars |>
  select(wt, hp) |>
  mutate(hp_per_t = hp/wt) |>
  head()

# Print the first few rows of the updated dataset
head(mtcars_new)
```

```
                   wt  hp hp_per_t
Mazda RX4         2.620 110 41.98473
Mazda RX4 Wag     2.875 110 38.26087
Datsun 710        2.320  93 40.08621
Hornet 4 Drive    3.215 110 34.21462
Hornet Sportabout 3.440 175 50.87209
Valiant           3.460 105 30.34682
```

```
# Rename hp to horsepower
mtcars |>
  rename(horsepower = hp) |>
  glimpse()
```

```
Rows: 32
Columns: 11
$ mpg        <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2,~
$ cyl        <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 4, 4, 4,~
$ disp       <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140~
$ horsepower <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 18~
```

```
$ drat       <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92,~
$ wt         <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.1~
$ qsec       <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.~
$ vs         <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,~
$ am         <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,~
$ gear       <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4,~
$ carb       <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1,~
```

4.6. How to explore a dataset

```
# Creating dataframe
df <- tibble(
  integer_var, numeric_var, character_var, factor_var, logical_var, date_var,
)

# Overview of the data
head(df)
```

```
# A tibble: 5 x 6
  integer_var numeric_var character_var factor_var logical_var date_var
        <dbl>       <dbl> <chr>         <fct>      <lgl>       <date>
1           1         1.1 apple         red        TRUE        2022-01-01
2           2         2.2 banana        yellow     TRUE        2022-02-01
3           3          NA orange        red        TRUE        2022-03-01
4           4         4.4 cherry        blue       FALSE       2022-04-01
5           5         5.5 grape         green      TRUE        2022-05-01
```

```
summary(df)
```

```
  integer_var  numeric_var    character_var       factor_var logical_var
 Min.   :1    Min.   :1.100   Length:5           blue  :1    Mode :logical
 1st Qu.:2    1st Qu.:1.925   Class :character   green :1    FALSE:1
 Median :3    Median :3.300   Mode  :character   red   :2    TRUE :4
 Mean   :3    Mean   :3.300                      yellow:1
 3rd Qu.:4    3rd Qu.:4.675
 Max.   :5    Max.   :5.500
              NA's   :1
    date_var
 Min.   :2022-01-01
 1st Qu.:2022-02-01
 Median :2022-03-01
 Mean   :2022-03-02
 3rd Qu.:2022-04-01
 Max.   :2022-05-01
```

```
glimpse(df)
```

```
Rows: 5
Columns: 6
$ integer_var   <dbl> 1, 2, 3, 4, 5
$ numeric_var   <dbl> 1.1, 2.2, NA, 4.4, 5.5
$ character_var <chr> "apple", "banana", "orange", "cherry", "grape"
$ factor_var    <fct> red, yellow, red, blue, green
$ logical_var   <lgl> TRUE, TRUE, TRUE, FALSE, TRUE
$ date_var      <date> 2022-01-01, 2022-02-01, 2022-03-01, 2022-04-01, 2022-05-~
```

```
# look closer at variables

# unique values
unique(df$integer_var)
```

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

```
unique(df$factor_var)
```

```
[1] red    yellow blue   green
Levels: blue green red yellow
```

```
table(df$factor_var)
```

```
  blue  green    red yellow
     1      1      2      1
```

```
length(unique(df$factor_var))
```

```
[1] 4
```

```
# distributions
df |> count(factor_var)
```

```
# A tibble: 4 x 2
  factor_var     n
  <fct>      <int>
1 blue           1
2 green          1
3 red            2
4 yellow         1
```

```
prop.table(table(df$factor_var))
```

```
  blue  green    red yellow
   0.2    0.2    0.4    0.2
```

```
df |>
  count(factor_var) |>
  mutate(prop = n / sum(n))
```

```
# A tibble: 4 x 3
  factor_var     n  prop
  <fct>      <int> <dbl>
1 blue           1   0.2
2 green          1   0.2
3 red            2   0.4
4 yellow         1   0.2
```

```
aggregate(df$numeric_var,
          by = list(fruit = df$factor_var),
          mean)
```

```
   fruit   x
1   blue 4.4
2  green 5.5
3    red  NA
4 yellow 2.2
```

```
# --> the mean of red cannot be calculated as there is a NA in it
# Solution: exclude NAs from calculation:
aggregate(df$numeric_var,
          by = list(fruit = df$factor_var),
          mean,
          na.rm = TRUE)
```

```
   fruit   x
1   blue 4.4
2  green 5.5
3    red 1.1
4 yellow 2.2
```

```
#install.packages("janitor")
require("janitor")
```

```
Loading required package: janitor
```

```
Attaching package: 'janitor'
```

```
The following objects are masked from 'package:stats':

    chisq.test, fisher.test
```

```
mtcars |>
  tabyl(cyl)
```

```
 cyl  n percent
   4 11 0.34375
   6  7 0.21875
   8 14 0.43750
```

```
mtcars |>
  tabyl(cyl, hp)
```

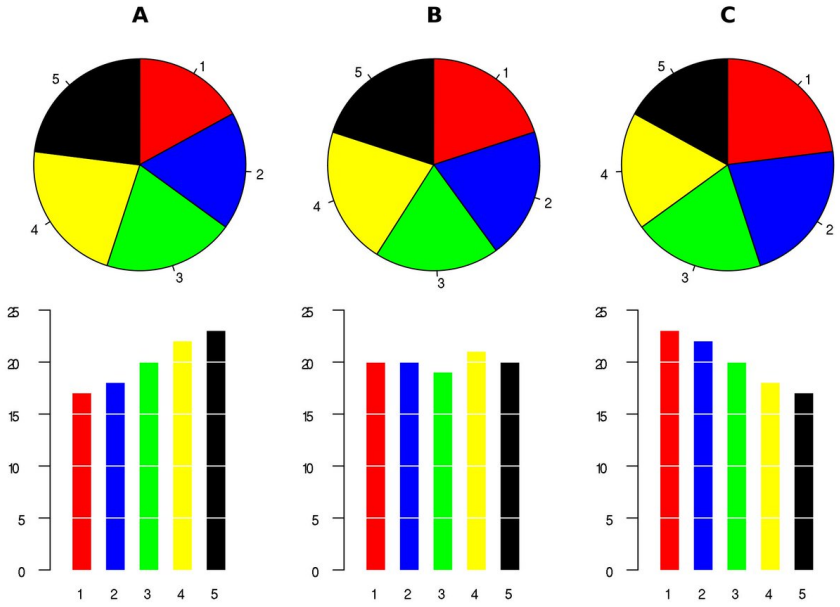
```
 cyl 52 62 65 66 91 93 95 97 105 109 110 113 123 150 175 180 205 215 230 245
   4  1  1  1  2  1  1  1  1   0   1   0   1   0   0   0   0   0   0   0   0
   6  0  0  0  0  0  0  0  0   1   0   3   0   2   0   1   0   0   0   0   0
   8  0  0  0  0  0  0  0  0   0   0   0   0   0   2   2   3   1   1   1   2
 264 335
   0   0
   0   0
   1   1
```

5. Visualizing data

Data visualization is an art. The purposes of visualizing data are manifold. You can emphasize facts, get known to data, detect anomalies, and communicate a large amount of information simply and intuitive. Whatever your goal is, thousand of appropriate ways exist to visualize data. Many decisions to take are simply a matter of taste. However, there are some conventions and guidelines that help you to make on average better decisions when designing a visualization:

- Good graphs are easy to understand and eye catching.
- Graphs can be misleading and manipulative and that is opposing to the ideas of science. Thus, be responsible and honest.
- Minimize colors and other attention-grabbing elements that are not directly related to the data of interest. Worldwide, there are approximately 300 million color blind people. In particular, red, green or blue light are problematic to color blind people. Thus, better rely on color schemes that are designed for colorblind people.
- Don't truncate an axis or change the scaling within an axis just to make you your story more appealing. Show the full scale of the graph, then zoom to show the data of interest, if necessary.
- Label and describe your chart sufficiently so that everybody can fully understand the content of the shown data set and statistics without having to study the notes of the graph for too long.
- Don't do pie charts. They may look simple, but they're tricky to get right and there are usually better alternatives. Humans are not very good at comparing the size of angles and as there's no scale in pie plots, reading accurate values is difficult. Figure Figure 5.1 may proof this.

Figure 5.1.: Pie charts are problematic



Source: https://en.wikipedia.org/wiki/Pie_chart

More tips

- Data Visualization: Chart Dos and Don'ts (by Duke University)
- Graphs and Visualising Data by Oliver Kirchkamp. In particular, I highly recommend his handout [Kirchkamp, 2018]. It discusses many pitfalls of visualizing data, instructs how to do good graphs, and he shows the corresponding R code of all graphs.

- The R Graph Gallery and R CHARTS by R CODER shows graphs and the corresponding R code to replicate the graphs
- The work of Edward Tufte and his book *The Visual Display of Quantitative Information* [Tufte, 2022] are classical readings.

A great resource to learn how to visualize data is Wickham and Grolemund [2023]. As I cannot do that any better, I refer to that source and refrain from writing section myself. It introduces the `ggplot` function which is part of the `ggplot2` package which, in turn, is part of the tidyverse package. Thus, if you've installed and loaded tidyverse, you automatically have access to `ggplot`. Creating beautiful and informative graphs is easy with `ggplot`. To proof that claim, study the chapter (Data visualization) of Wickham and Grolemund [2023]. Another good resource on modern data visualization is Kabacoff [2024].

To reap the best benefits from studying,

I recommend to copy all the code that is shown in the book into a R script and try to run it on your PC. That is the best way to learn, understand, and create your own notes that may guide you later on. Whenever you see interesting code somewhere, try to run it on your PC. Moreover, I recommend the exercises of the book, they are challenging sometimes but to really understand code you need to run code yourself.

6. Collection of exercises

6.1. Links to the R scripts with the solutions

- exe_duplicates.R
- exe_import_covid.R
- exe_genanddrop.R
- exe_base_pipe.R
- exe_subset.R
- exe_data_transformation.R
- exe_poser.R
- exe_datasauRus.R
- exe_convergence.R
- exe_un_gdp_ger_fra.R
- exe_hortacsu_figure_3.R
- exe_regress_lecture.R
- exe_calories.R
- exe_bundesliga.R
- exe_okun_solution.R
- exe_zipf_solution.R

6.2. Links to the output of the scripts

Here you find the output

EXERCISE: Generate and drop variables

Use the *mtcars* dataset. It is part of the package *datasets* and can be called with

```
mtcars
```

a) Create a new tibble called `mtcars_new` using the pipe operator |>. Generate a new dummy variable called `d_cyl_6to8` that takes the value 1 if the number of cylinders (cyl) is greater than 6, and 0 otherwise. Do all of this in a single pipe.

b) Generate a new dummy variable called `posercar` that takes a value of 1 if a car has more than 6 cylinders (cyl) and can drive less than 18 miles per gallon (mpg), and 0 otherwise. Add this variable to the tibble `mtcars_new`.

c) Remove the variable `d_cyl_6to8` from the data frame.

Please find solutions here.

EXERCISE: Import data

Table 6.1 shows COVID for three states in Germany:

Table 6.1.: Covid cases and deaths till August 2022

state	Bavaria	North Rhine-Westphalia	Baden-Württemberg
deaths (in mio)	4,92M	5,32M	3,69M
cases	24.111	25.466	16.145

Write down the code you would need to put into the R-console…

- …to store each of variables *state* and *deaths* in a vector.
- …to store both vectors in a data frame with the name `df_covid`.
- …to store both vectors in a tibble with the name `tbl_covid`.

Please find solution to the exercise here

EXERCISE: Base R or pipe

a) Using the mtcars dataset, write code to create a new dataframe that includes only the rows where the number of cylinders is either 4 or 6, and the weight (wt) is less than 3.5.

Do this in two different ways using:

1. The `%in%` operator and the pipe |> .
2. Base R without the pipe |>.

Compare the resulting dataframes using the identical() function.

b) Using the mtcars dataset, generate a logical variable that indicates with `TRUE` all cars with either 4 or 6 cylinders that wt is less than 3.5 and add this variable to a new dataset.

Please find solutions here.

EXERCISE: Subsetting

1. Check to see if you have the mtcars dataset by entering the command mtcars.
2. Save the mtcars dataset in an object named cars.
3. What class is cars?
4. How many observations (rows) and variables (columns) are in the mtcars dataset?
5. Rename mpg in cars to MPG. Use rename().
6. Convert the column names of cars to all upper case. Use rename_all, and the toupper command.
7. Convert the rownames of cars to a column called car using rownames_to_column.
8. Subset the columns from cars that end in “p” and call it pvars using ends_with().
9. Create a subset cars that only contains the columns: wt, qsec, and hp and assign this object to carsSub. (Use select().)
10. What are the dimensions of carsSub? (Use dim().)

11. Convert the column names of carsSub to all upper case. Use rename_all(), and toupper() (or colnames()).
12. Subset the rows of cars that get more than 20 miles per gallon (mpg) of fuel efficiency. How many are there? (Use filter().)
13. Subset the rows that get less than 16 miles per gallon (mpg) of fuel efficiency and have more than 100 horsepower (hp). How many are there? (Use filter() and the pipe operator.)
14. Create a subset of the cars data that only contains the columns: wt, qsec, and hp for cars with 8 cylinders (cyl) and reassign this object to carsSub. What are the dimensions of this dataset? Do not use the pipe operator.
15. Create a subset of the cars data that only contains the columns: wt, qsec, and hp for cars with 8 cylinders (cyl) and reassign this object to carsSub2. Use the pipe operator.
16. Re-order the rows of carsSub by weight (wt) in increasing order. (Use arrange().)
17. Create a new variable in carsSub called wt2, which is equal to wt^2, using mutate() and piping `%>%`.

Please find solutions here.

EXERCISE: Data transformation

Please download and open the R-script you find here and try to answer the questions therein.

Solutions to the questions are linked in the script.

EXERCISE: Load the Stata dataset "auto" using R

1. Create a scatter plot illustrating the relationship between the price and weight of a car. Provide a meaningful title for the graph and try to make it clear which car each observation corresponds to.
2. Save this graph in the formats of .png and .pdf.
3. Create a variable "'lp100km"' that indicates the fuel consumption of an average car in liters per 100 kilometers. (Note: One gallon is approximately equal to 3.8 liters, and one mile is about 1.6 kilometers.)
4. Create a dummy variable "'larger6000"' that is equal to 1 if the price of a car is above $6000.
5. Now, search for the "most unreasonable poser car" that costs no more than $6000. A "poser" car is defined as one that is expensive, has a large turning radius, consumes a lot of fuel, and is often defective (rep78 is low). For this purpose, create a metric indicator for each corresponding variable that indicates a value of 1 for the car that is the most unreasonable in that variable and 0 for the most reasonable car. All other cars should fall between 0 and 1.

Please find the solutions here.

Figure 6.1.: The logo of the DatasauRus package

Source: https://github.com/jumpingrivers/datasauRus_

EXERCISE: DatasauRus

a) Load the packages `datasauRus` and `tidyverse`. If necessary, install these packages.

b) The package`datasauRus` comes with a dataset in two different formats: `datasaurus_dozen` and `datasaurus_dozen_wide`. Store them as `ds` and `ds_wide`.

c) Open and read the R vignette of the `datasauRus` package. Also open the R documentation of the dataset `datasaurus_dozen`.

d) Explore the dataset: What are the dimensions of this dataset? Look at the descriptive statistics.

e) How many unique values does the variable `dataset` of the tibble `ds` have? Hint: The function unique() return the unique values of a variable and the function length() returns the length of a vector, such as the unique elements.

f) Compute the mean values of the `x` and `y` variables for each entry in `dataset`. Hint: Use the group_by() function to group the data by the appropriate column and then the summarise() function to calculate the mean.

g) Compute the standard deviation, the correlation, and the median in the same way. Round the numbers.

h) What can you conclude?

i) Plot all datasets of `ds`. Hide the legend. Hint: Use the `facet_wrap()` and the `theme()` function.

j) Create a loop that generates separate scatter plots for each unique datatset of the tibble `ds`. Export each graph as a png file.

k) Watch the video Animating the Datasaurus Dozen Dataset in R from The Data Digest on YouTube.

Please find the solutions here.

EXERCISE: Convergence

The dataset convergence.dta, see https://github.com/hubchev/courses/blob/main/dta/convergence.dta, contains the per capita GDP of 1960 (gdppc60) and the average growth rate of GDP per capita between 1960 and 1995 (growth) for different countries (country), as well as 3 dummy variables indicating the belonging of a country to the region Asia (asia), Western Europe (weurope) or Africa (africa).

- Some countries are not assigned to a certain country group. Name the countries which are assign to be part of Western Europe, Africa or Asia. If you find countries that are members of the EU, assign them a '1' in the variable weurope.
- Create a table that shows the average GDP per capita for all available points in time. Group by Western European, Asian, African, and the remaining countries.
- Create the growth rate of GDP per capita from 1960 to 1995 and call it gdpgrowth. (Note: The log value X minus the log value X of the previous period is approximately equal to the growth rate).
- Calculate the unconditional convergence of all countries by constructing a graph in which a scatterplot shows the GDP per capita growth rate between 1960 and 1995 (gdpgrowth) on the y-axis and the 1960 GDP per capita (gdppc60) on the x-axis. Add to the same graph the estimated linear relationship. You do not need to label the graph further, just two things: title the graph `world` and label the individual observations with the country names.
- Create three graphs describing the same relationship for the sample of Western European, African and Asian countries. Title the graph accordingly with `weurope`, `africa` and `asia`.
- Combine the four graphs into one image. Discuss how an upward or downward sloping regression line can be interpreted.
- Estimate the relationships illustrated in the 4 graphs using the least squares method. Present the 4 estimation results in a table, indicating the significance level with stars. In addition, the Akaike information criterion, and the number of observations should be displayed in the table. Interpret the four estimation results regarding their significance.
- Put the data set into the so-called long format and calculate the GDP per capita growth rates for the available time points in the countries.

Please find solutions here.

EXERCISE: Unemployment and GDP in Germany and France

The following exercise was a former exam.

Please answer all (!) questions in an R script. Normal text should be written as comments, using the '#' to comment out text. Make sure the script runs without errors before submitting it. Each task (starting with 1) is worth five points. You have a total of 120 minutes of editing time. Please do not forget to number your answers.

When you are done with your work, save the R script, export the script to pdf format and upload the pdf file.

Suppose you aim to empirically examine unemployment and GDP for Germany and France. The data set that we use in the following is 'forest.Rdata'.

(0) Write down your name, matriculation number, and date.

(1) Set your working directory.

(2) Clear your global environment.

(3) Install and load the following packages: 'tidyverse', 'sjPlot', and 'ggpubr'

(4) Download and load the data, respectively, with the following code:

```
load(url("https://github.com/hubchev/courses/raw/main/dta/forest.Rdata"))
```

If that is not working, you can also download the data from ILIAS, save it in your working directory and load it from there with:

```
# load("forest.Rdata")
```

(5) Show the **first eight** observations of the dataset 'df'.

(6) Show the **last observation** of the dataset 'df'.

(7) Which type of data do we have here (Panel, cross-section,time series, …)? Name the variable(s) that are necessary to identify the observations in the dataset.

(8) Explain what the **assignment operator** in R is and what it is good for.

(9) Write down the R code to store the number of observations and the number of variables that are in the dataset 'df'. Name the object in which you store these numbers 'observations_df'.

(10) In the dataset 'df', rename the variable 'country.x' to 'nation' and the variable 'date' to 'year'.

(11) Explain what the **pipe operator** in R is and what it is good for.

(12) For the upcoming analysis you are only interested the following **variables** that are part of the dataframe 'df': nation, year, gdp, pop, gdppc, and unemployment. Drop all other variables from the dataframe 'df'.

(13) Create a variable that indicates the GDP per capita ('gdp' divided by 'pop'). Name the variable 'gdp_pc'. (Hint: If you fail here, use the variable 'gdppc' which is already in the dataset as a replacement for 'gdp_pc' in the following tasks.)

(14) For the upcoming analysis you are only interested the following **countries** that are part of the dataframe 'df': Germany and France. Drop all other countries from the dataframe 'df'.

(15) Create a table showing the **average** unemployment rate and GDP per capita for Germany and France in the given years. Use the pipe operator. (Hint: See below for how your results should look like.)

```
# A tibble: 2 x 3
  nation  `mean(unemployment)` `mean(gdppc)`
  <chr>                  <dbl>         <dbl>
1 France                  9.75        34356.
2 Germany                 7.22        36739.
```

(16) Create a table showing the unemployment rate and GDP per capita for Germany and France in the **year 2020**. Use the pipe operator. (Hint: See below for how your results should look like.)

```
# A tibble: 2 x 3
  nation  `mean(unemployment)` `mean(gdppc)`
  <chr>                  <dbl>         <dbl>
1 France                  8.01        35786.
2 Germany                 3.81        41315.
```

(17) Create a table showing the **highest** unemployment rate and the **highest** GDP per capita for Germany and France during the given period. Use the pipe operator. (Hint: See below for how your results should look like.)

```
# A tibble: 2 x 3
  nation  `max(unemployment)` `max(gdppc)`
  <chr>                 <dbl>        <dbl>
1 France                 12.6       38912.
2 Germany                11.2       43329.
```

(18) Calculate the standard deviation of the unemployment rate and GDP per capita for Germany and France in the given years. (Hint: See below for how your result should look like.)

```
# A tibble: 2 x 3
  nation  `sd(gdppc)` `sd(unemployment)`
  <chr>         <dbl>              <dbl>
1 France        2940.               1.58
2 Germany       4015.               2.37
```

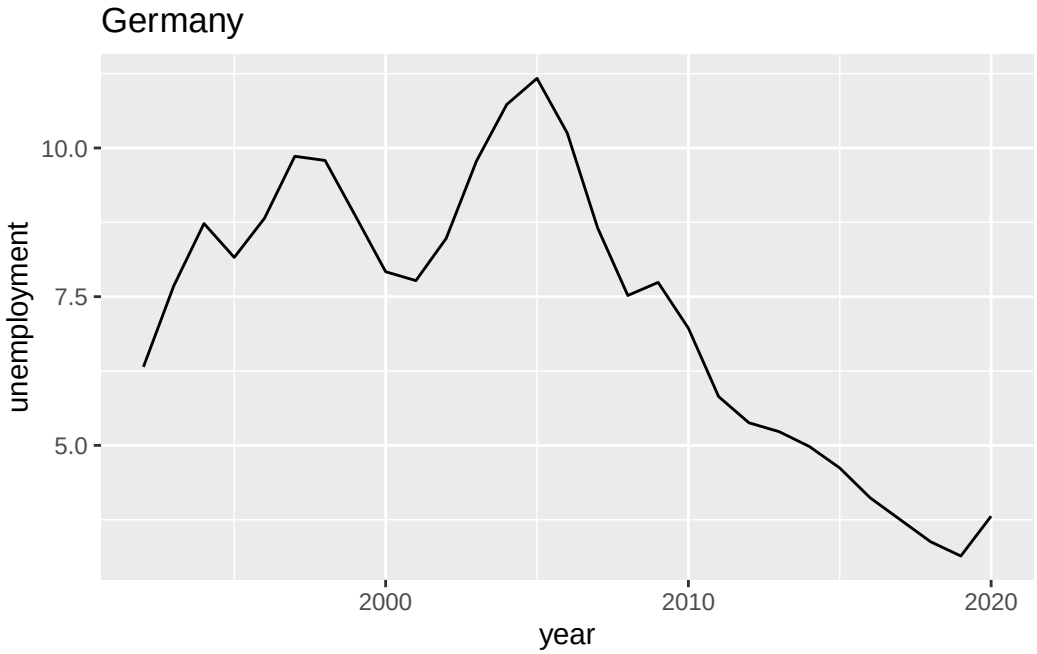
(19) In statistics, the coefficient of variation (COV) is a standardized measure of dispersion. It is defined as the ratio of the standard deviation (σ) to the mean (μ): $COV = \frac{\sigma}{\mu}$. Write down the R code to calculate the coefficient of variation (COV) for the **unemployment rate** in Germany and France. (Hint: See below for what your result should should look like.)

```
# A tibble: 2 x 4
  nation  `sd(unemployment)` `mean(unemployment)`   cov
  <chr>                <dbl>                <dbl> <dbl>
1 France                1.58                 9.75 0.162
2 Germany               2.37                 7.22 0.328
```

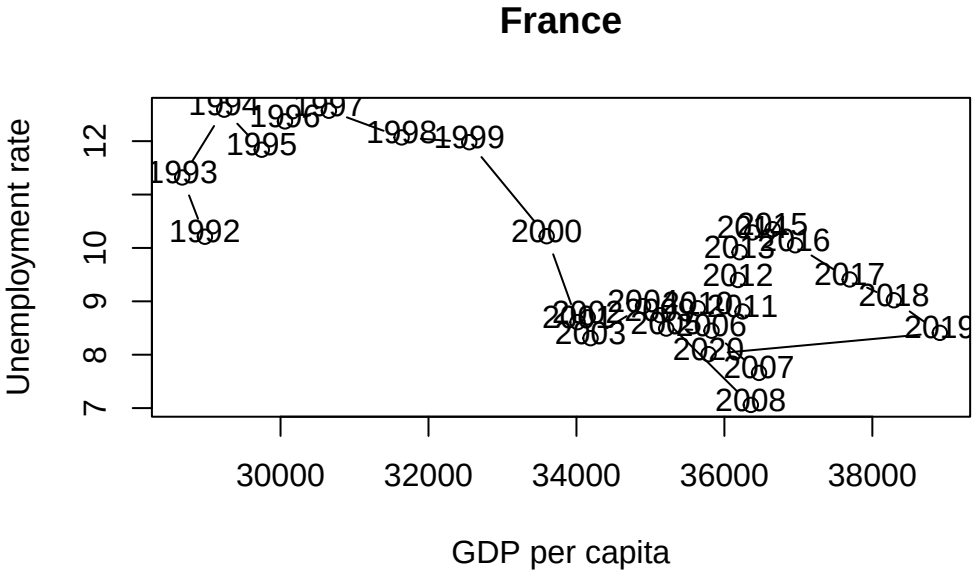
(20) Write down the R code to calculate the coefficient of variation (COV) for the **GDP per capita** in Germany and France. (Hint: See below for what your result should look like.)

```
# A tibble: 2 x 4
  nation  `sd(gdppc)` `mean(gdppc)`    cov
  <chr>         <dbl>         <dbl>  <dbl>
1 France        2940.        34356. 0.0856
2 Germany       4015.        36739. 0.109
```

(21) Create a chart (bar chart, line chart, or scatter plot) that shows the unemployment rate of **Germany** over the available years. Label the chart 'Germany' with 'ggtitle("Germany")'. Please note that you may choose any type of graphical representation. (Hint: Below you can see one of many |> of what your result may look like).



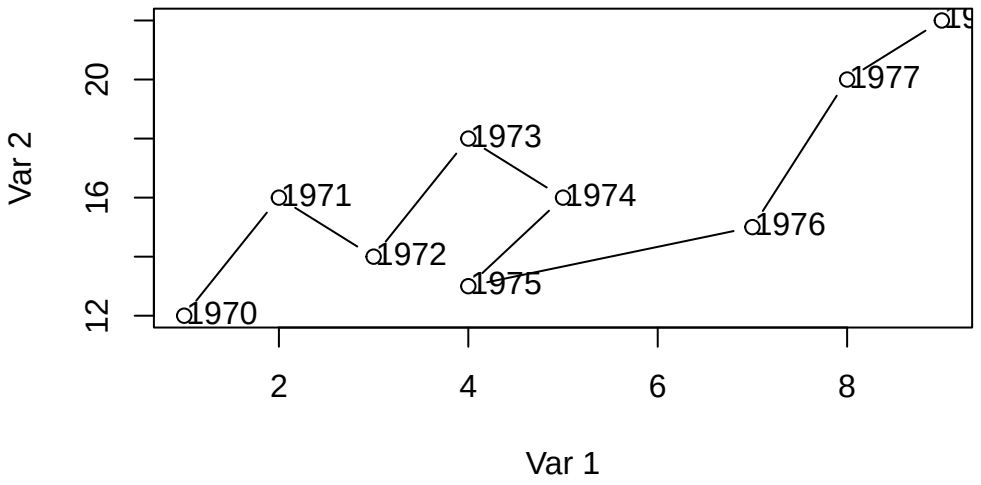
(22) and 23. *(This task is worth 10 points)* The following chart shows the simultaneous development of the unemployment rate and GDP per capita over time for **France**.



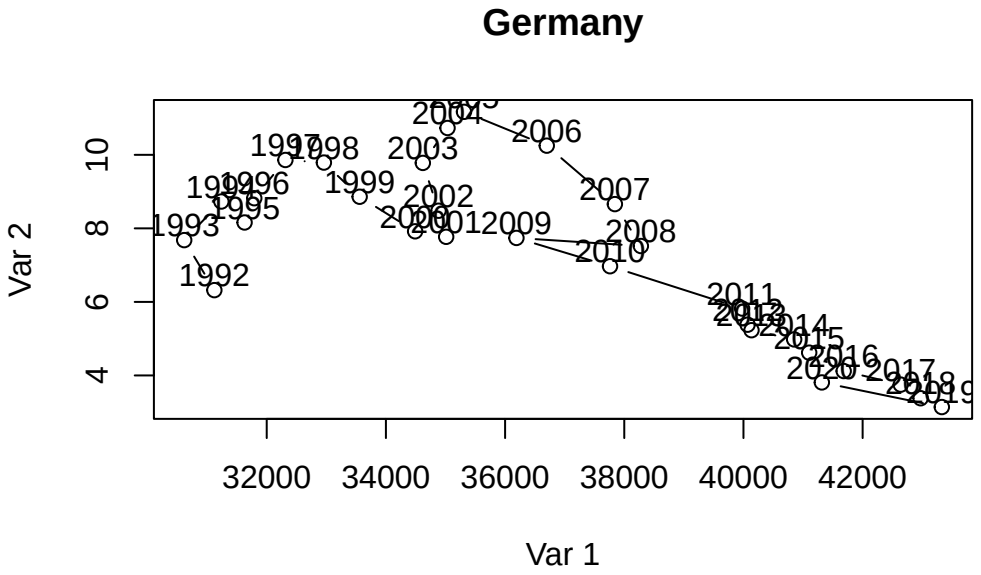
Suppose you want to visualize the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany as well.
Suppose further that you have found the following lines of code that create the kind of chart you are looking for.

```
# Data
x <- c(1, 2, 3, 4, 5, 4, 7, 8, 9)
y <- c(12, 16, 14, 18, 16, 13, 15, 20, 22)
labels <- 1970:1978
```

```
# Connected scatter plot with text
plot(x, y, type = "b", xlab = "Var 1", ylab = "Var 2"); text(x + 0.4, y + 0.1, labels)
```



Use these lines of code and customize them to create the co-movement visualization for **Germany** using the available 'df' data. The result should look something like this:



(24) Interpret the two graphs above, which show the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany and France. What are your expectations regarding the correlation between the unemployment rate and GDP per capita variables? Can you see this expectation in the figures? Discuss.

Please find solutions here.

EXERCISE: Import data and write a report

Reproduce Figure 3 of Hortaçsu and Syverson [2015, p. 99] using R. Write a clear report about your work, i.e., document everything with a R script or a R Markdown file.

Here are the required steps:

1. Go to https://www.aeaweb.org/articles?id=10.1257/jep.29.4.89 and download the *replication package* from the *OPENICPSR* page. Please note, that you can download the replication package after you have registered for the platform.
2. Unzip the replication package.
3. In the file *diffusion_curves_figure.xlsx* you find the required data. Import them to R.
4. Reproduce the plot using ggplot().

Please find solutions here.

EXERCISE: Explain the weight

In the statistic course of WS 2020, I asked 23 students about their weight, height, sex, and number of siblings. I wonder how good the height can explain the weight of students. Examine with corelations and a regression analysis the association. Load the data as follows:

```
library("haven")
classdata <- read.csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/classdata
```

A sketch of a solution is provided here.

EXERCISE: Calories and weight

a) Write down your name, your matriculation number, and the date.
b) Set your working directory.
c) Clear your global environment.
d) Load the following package: `tidyverse`

The following table stems from a survey carried out at the Campus of the German Sport University of Cologne at Opening Day (first day of the new semester) between 8:00am and 8:20am. The survey consists of 6 individuals with the following information:

id	sex	age	weight	calories	sport
1	f	21	48	1700	60
2	f	19	55	1800	120
3	f	23	50	2300	180
4	m	18	71	2000	60
5	m	20	77	2800	240
6	m	61	85	2500	30

Data Description:

- **id:** Variable with an anonymized identifier for each participant.
- **sex:** Gender, i.e., the participants replied to be either male (m) or female (f).
- **age:** The age in years of the participants at the time of the survey.
- **weight:** Number of kg the participants pretended to weight.
- **calories:** Estimate of the participants on their average daily consumption of calories.
- **sport:** Estimate of the participants on their average daily time that they spend on doing sports (measured in minutes).

e) Which type of data do we have here? (Panel data, repeated cross-sectional data, cross-sectional data, time Series data)

f) Store each of the five variables in a vector and put all five variables into a dataframe with the title `df`. If you fail here, read in the data using this line of code:

```
df <- read_csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/df-calories.csv")
```

```
Rows: 6 Columns: 5
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): sex
dbl (4): age, weight, calories, sport

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

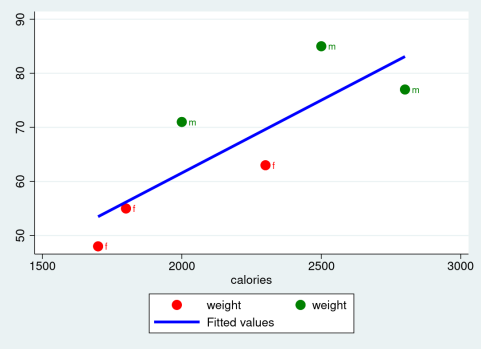
g) Show for all numerical variables the summary statistics including the mean, median, minimum, and the maximum.

h) Show for all numerical variables the summary statistics including the mean and the standard deviation, **separated by male and female**. Use therefore the pipe operator.

i) Suppose you want to analyze the general impact of average calories consumption per day on the weight. Discuss if the sample design is appropriate to draw conclusions on the population. What may cause some bias in the data? Discuss possibilities to improve the sampling and the survey, respectively.

j) The following plot visualizes the two variables weight and calories. Discuss what can be improved in the graphical visualization.

Figure 6.2.: Weight vs. Calories



k) Make a scatterplot matrix containing all numerical variables.

l) Calculate the Pearson Correlation Coefficient of the two variables

1. `calories` and `sport`
2. `weight` and `calories`

m) Make a scatterplot with `weight` in the y-axis and `calories` on the x-axis. Additionally, the plot should contain a linear fit and the points should be labeled with the `sex` just like in the figure shown above.

n) Estimate the following regression specification using the OLS method: [weight_i= _0+ _1 calories_i+ _i]

Show a summary of the estimates that look like the following:

```
Call:
lm(formula = weight ~ calories, data = df)

Residuals:
     1      2      3      4      5      6
-5.490 -1.182 -6.640  9.435 -6.099  9.976

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  7.730275  20.197867   0.383   0.7214
calories     0.026917   0.009107   2.956   0.0417 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.68 on 4 degrees of freedom
Multiple R-squared:  0.6859,    Adjusted R-squared:  0.6074
F-statistic: 8.735 on 1 and 4 DF,  p-value: 0.04174
```

o) Interpret the results. In particular, interpret how many kg the estimated weight increases—on average and ceteris paribus—if calories increase by 100 calories. Additionally, discuss the statistical properties of the estimated coefficient $\hat{\beta}_1$ and the meaning of the **Adjusted R-squared**.

p) OLS estimates can suffer from omitted variable bias. State the two conditions that need to be fulfilled for omitted bias to occur.

q) Discuss potential confounding variables that may cause omitted variable bias. Given the dataset above how can some of the confounding variables be *controlled for*?

Solutions are provided here.

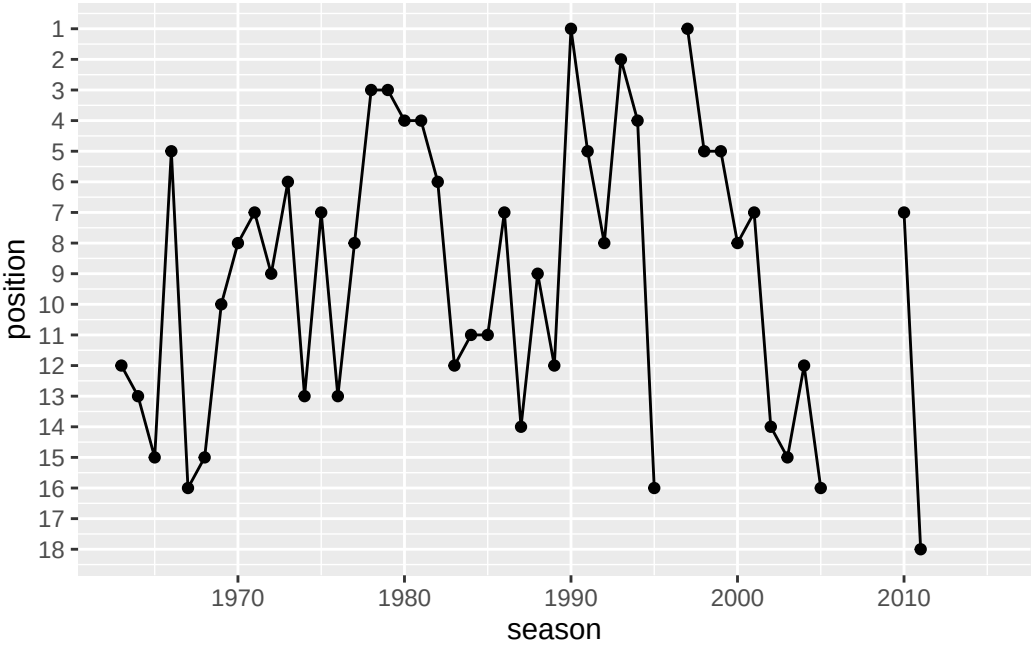
EXERCISE: Bundesliga

Open the script that you find here and work on the following tasks:

1. Set your working directory.
2. Clear th environment.
3. Install and load the `bundesligR` and `tidyverse`.
4. Read in the data `bundesligR` as a tibble.
5. Replace "Bor. Moenchengladbach" with "Borussia Moenchengladbach."
6. Check for the data class.
7. View the data.

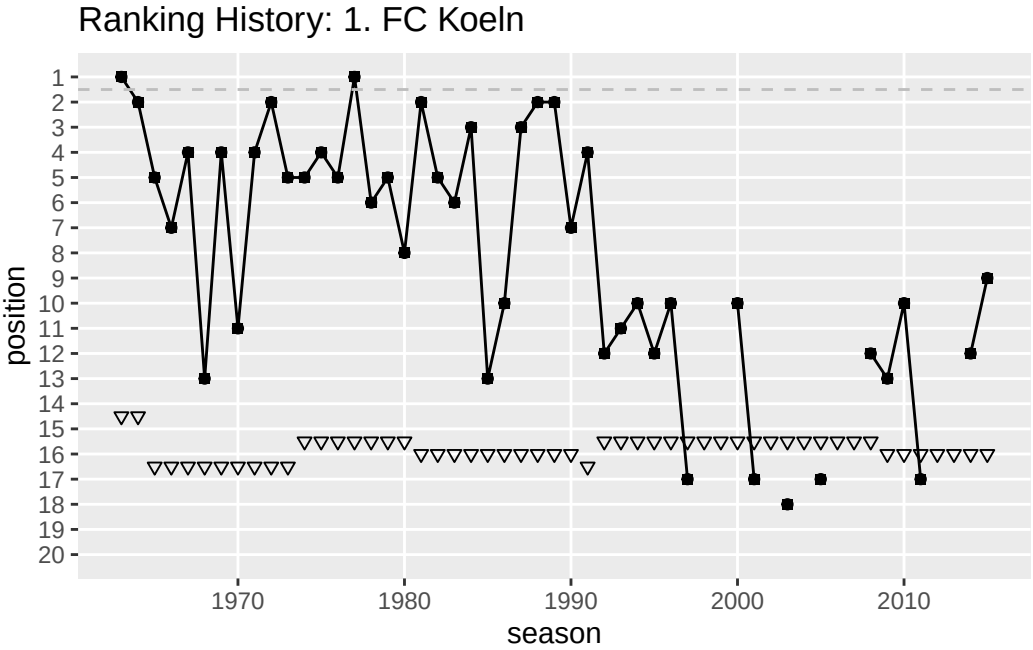
8. Glimpse on the data.
9. Show the first and last observations.
10. Show summary statistics to all variables.
11. How many teams have played in the league over the years?
12. Which teams have played Bundesliga so far?
13. How many teams have played Bundesliga?
14. How often has each team played in the Bundesliga?
15. Show summary statistics of variable `Season` only.
16. Show summary statistics of all numeric variables (`Team` is a character).
17. What is the highest number of points ever received by a team? Show only the name of the club with the highest number of points ever received.
18. Create a new tibble using `liga` removing the variable `Pts_pre_95` from the data.
19. Create a new tibble using `liga` renaming W, D, and L to Win, Draw, and Loss. Additionally rename GF, GA, GD to Goals_shot, Goals_received, Goal_difference.
20. Create a new tibble using `liga` without the variable `Pts_pre_95` and only observations before the year 1996.
21. Remove the three tibbles just created from the environment.
22. Rename all variables of `liga` to lowercase and store it as `dfb`.
23. Show the winner and the runner up after the year 2010. Additionally show the points received.
24. Create a variable that counts how often a team was ranked first.
25. How often has each team played in the Bundesliga?
26. Make a ranking that shows which team has played the Bundesliga most often.
27. Add a variable to `dfb` that contains the number of appearances of a team in the league.
28. Create a number that indicates how often a team has played Bundesliga in a given year.
29. Make a ranking with the number of titles of all teams that ever won the league.
30. Create a numeric identifying variable for each team.
31. When a team is in the league, what is the probability that it wins the league?
32. Make a scatterplot with points on the y-axis and position on the x-axis.
33. Make a scatterplot with points on the y-axis and position on the x-axis. Additionally, only consider seasons with 18 teams and add lines that make clear how many points you needed to be placed in between rank 2 and 15.
34. Remove all objects from the environment except `dfb` and `liga`.
35. In Figure @ref(fig:fckpic), the ranking history of *1. FC Kaiserslautern* is shown. Replicate that plot.

Figure 6.3.: Ranking history: 1. FC Kaiserslautern



34. In Figure @ref(fig:efzplot), I made the graph a bit nicer. Can you spot all differences and can you guess what the dashed line and the triangles mean? How could the visualization be improved further? Replicate the plot.

Figure 6.4.: Ranking history: 1. FC Köln



35. Try to make the ranking history for each club ever played the league and export the graph as a `png` file.

Solutions are provided here.

EXERCISE: Okun's Law

Suppose you aim to empirically examine unemployment and GDP for Germany and France. The data set that we use in the following is 'forest.Rdata' and should already been known to you from the lecture.

(0) Write down your name, matriculation number, and date.

(1) Set your working directory.

(2) Clear your global environment.

(3) Install and load the following packages: 'tidyverse', 'sjPlot', and 'ggpubr'

(4) Download and load the data, respectively, with the following code:

```
load(url("https://github.com/hubchev/courses/raw/main/dta/forest.Rdata"))
```

If that is not working, you can also download the data from ILIAS, save it in your working directory and load it from there with:

```
load("forest.Rdata")
```

(5) Show the **first eight** observations of the dataset 'df'.

(6) Show the **last observation** of the dataset 'df'.

(7) Which type of data do we have here (Panel, cross-section,time series, ...)? Name the variable(s) that are necessary to identify the observations in the dataset.

(8) Explain what the **assignment operator** in R is and what it is good for.

(9) Write down the R code to store the number of observations and the number of variables that are in the dataset 'df'. Name the object in which you store these numbers 'observations_df'.

(10) In the dataset 'df', rename the variable 'country.x' to 'nation' and the variable 'date' to 'year'.

(11) Explain what the **pipe operator** in R is and what it is good for.

(12) For the upcoming analysis you are only interested the following **variables** that are part of the dataframe 'df': nation, year, gdp, pop, gdppc, and unemployment. Drop all other variables from the dataframe 'df'.

(13) Create a variable that indicates the GDP per capita ('gdp' divided by 'pop'). Name the variable 'gdp_pc'. (Hint: If you fail here, use the variable 'gdppc' which is already in the dataset as a replacement for 'gdp_pc' in the following tasks.)

(14) For the upcoming analysis you are only interested the following **countries** that are part of the dataframe 'df': Germany and France. Drop all other countries from the dataframe 'df'.

(15) Create a table showing the **average** unemployment rate and GDP per capita for Germany and France in the given years. Use the pipe operator. (Hint: See below for how your results should look like.)

```
# A tibble: 2 x 3
  nation  `mean(unemployment)` `mean(gdppc)`
  <chr>                  <dbl>         <dbl>
1 France                  9.75        34356.
2 Germany                 7.22        36739.
```

(16) Create a table showing the unemployment rate and GDP per capita for Germany and France in the **year 2020**. Use the pipe operator. (Hint: See below for how your results should look like.)

```
# A tibble: 2 x 3
  nation  `mean(unemployment)` `mean(gdppc)`
  <chr>                  <dbl>         <dbl>
1 France                  8.01        35786.
2 Germany                 3.81        41315.
```

(17) Create a table showing the **highest** unemployment rate and the **highest** GDP per capita for Germany and France during the given period. Use the pipe operator. (Hint: See below for how your results should look like.)

```
# A tibble: 2 x 3
  nation  `max(unemployment)` `max(gdppc)`
  <chr>                 <dbl>        <dbl>
1 France                 12.6       38912.
2 Germany                11.2       43329.
```

(18) Calculate the standard deviation of the unemployment rate and GDP per capita for Germany and France in the given years. (Hint: See below for how your result should look like.)

```
# A tibble: 2 x 3
  nation  `sd(gdppc)` `sd(unemployment)`
  <chr>         <dbl>              <dbl>
1 France        2940.               1.58
2 Germany       4015.               2.37
```

(19) In statistics, the coefficient of variation (COV) is a standardized measure of dispersion. It is defined as the ratio of the standard deviation (σ) to the mean (μ): $COV = \frac{\sigma}{\mu}$. Write down the R code to calculate the coefficient of variation (COV) for the **unemployment rate** in Germany and France. (Hint: See below for what your result should should look like.)

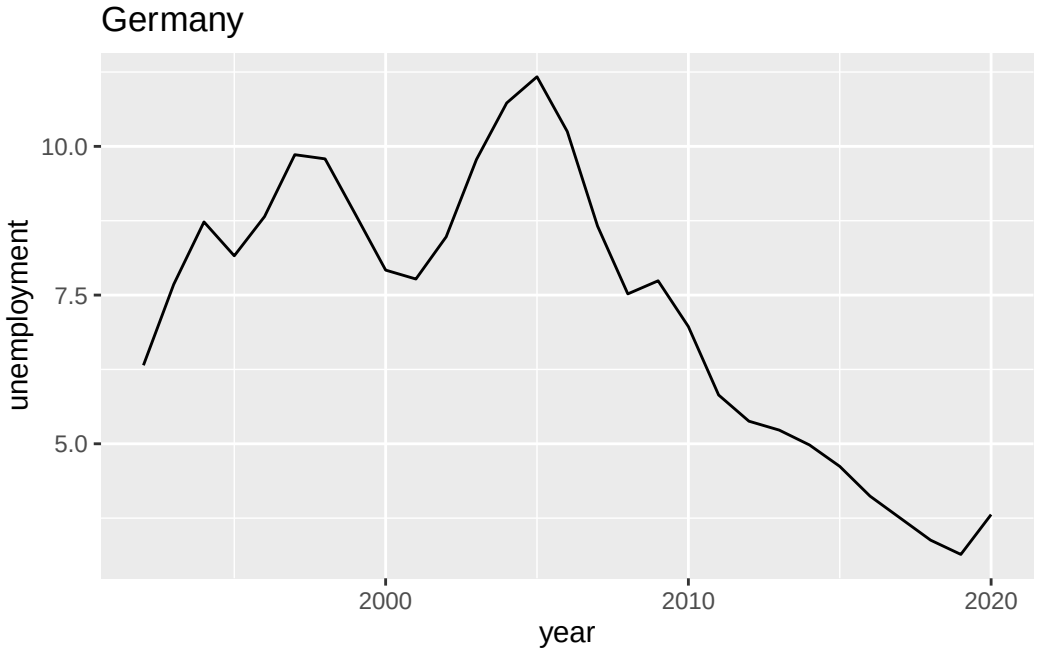
```
# A tibble: 2 x 4
  nation  `sd(unemployment)` `mean(unemployment)`   cov
  <chr>                <dbl>                <dbl> <dbl>
1 France                1.58                 9.75 0.162
2 Germany               2.37                 7.22 0.328
```

(20) Write down the R code to calculate the coefficient of variation (COV) for the **GDP per capita** in Germany and France. (Hint: See below for what your result should should look like.)

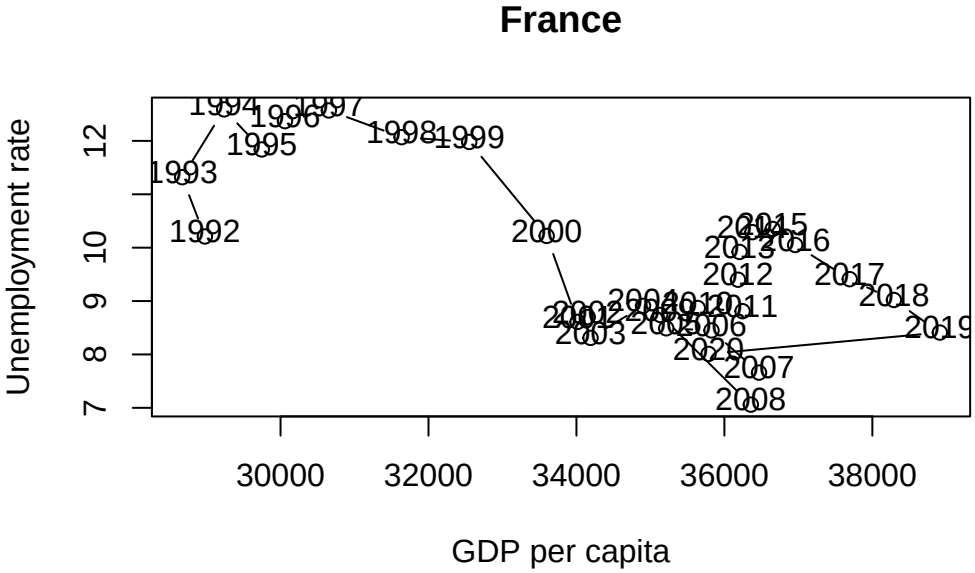
look like.)

```
# A tibble: 2 x 4
  nation  `sd(gdppc)` `mean(gdppc)`    cov
  <chr>         <dbl>         <dbl>  <dbl>
1 France        2940.        34356. 0.0856
2 Germany       4015.        36739. 0.109
```

(21) Create a chart (bar chart, line chart, or scatter plot) that shows the unemployment rate of **Germany** over the available years. Label the chart 'Germany' with 'ggtitle("Germany")'. Please note that you may choose any type of graphical representation. (Hint: Below you can see one of many possible examples of what your result may look like).



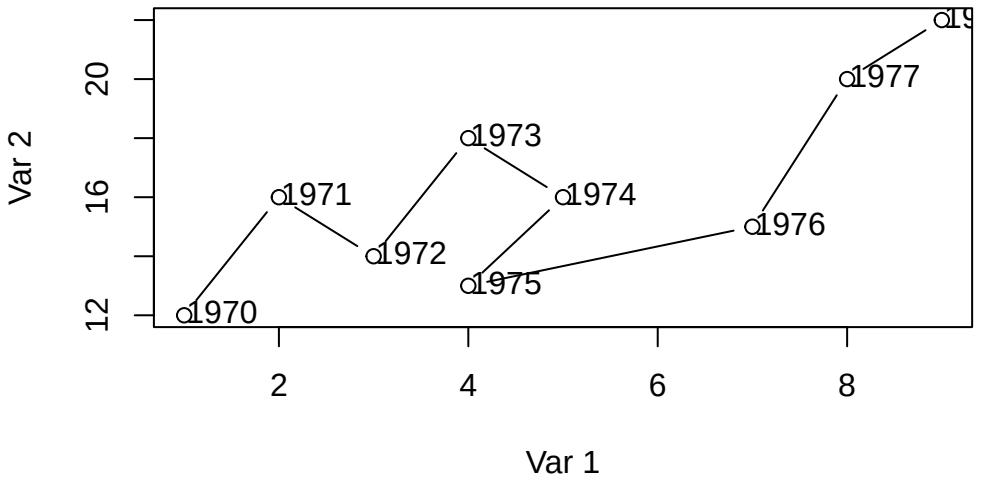
(22) and 23. *(This task is worth 10 points)* The following chart shows the simultaneous development of the unemployment rate and GDP per capita over time for **France**.



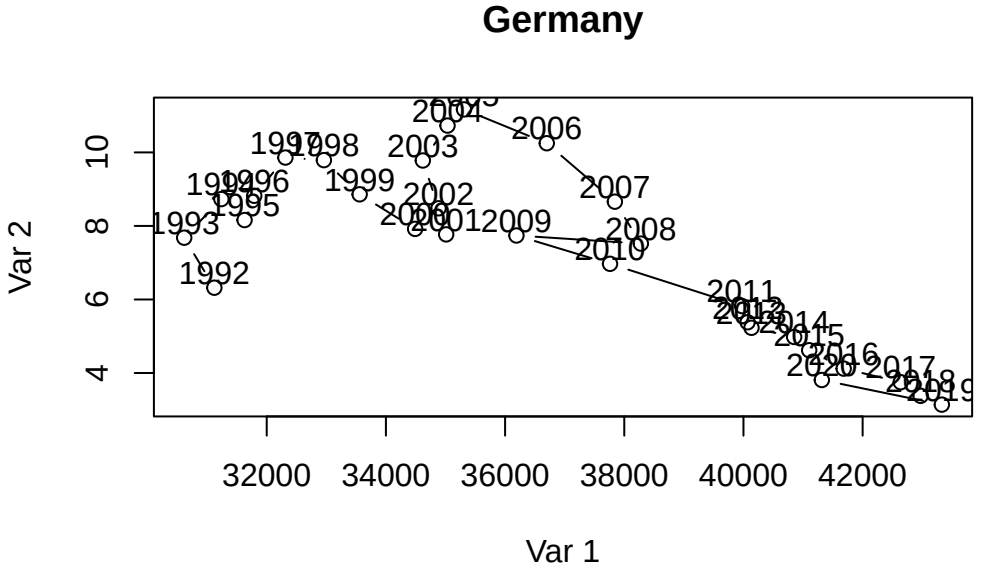
Suppose you want to visualize the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany as well.
Suppose further that you have found the following lines of code that create the kind of chart you are looking for.

```
# Data
x <- c(1, 2, 3, 4, 5, 4, 7, 8, 9)
y <- c(12, 16, 14, 18, 16, 13, 15, 20, 22)
labels <- 1970:1978

# Connected scatter plot with text
plot(x, y, type = "b", xlab = "Var 1", ylab = "Var 2"); text(x + 0.4, y + 0.1, labels)
```



Use these lines of code and customize them to create the co-movement visualization for **Germany** using the available 'df' data. The result should look something like this:



(24) Interpret the two graphs above, which show the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany and France. What are your expectations regarding the correlation between the unemployment rate and GDP per capita variables? Can you see this expectation in the figures? Discuss.

Solutions are provided here.

EXERCISE: Names and duplicates

1. Load the required packages (`pacman`, `tidyverse`, `janitor`, `babynames`, `stringr`).
2. Load the dataset from the URL: https://github.com/hubchev/courses/raw/main/dta/df_names.RData. Make yourself familiar with the data.
3. After loading the dataset, remove all objects except `df_2022` and `df_2022_error`.
4. Reorder the data using the `relocate` function so that `surname`, `name`, and `age` appear first. Save the changed data in a tibble called `df`.
5. Sort the data according to `surname`, `name`, and `age`.
6. Make a variable named `born` that contains the year of birth. How is the `born` variable calculated?
7. Create a new variable named `id` that identifies each person by `surname`, `name`, and their birth year (`born`). Why is this identifier useful?
8. Investigate how the data is identified. Are there any duplicates? If so, can you think of strategies to identify and how to deal with these duplicates.
9. Unload the packages used in the script. Why is unloading packages considered good practice?

Solutions are provided here.

EXERCISE: Zipf's law

The data under investigation includes population information for various German cities, identified by the variable `stadt`, spanning the years 1970, 1987, and 2010. The variable status provides details about the legislative status of the cities, and the variable `state` (Bundesland) indicates the state in which each respective city is situated.

Preamble

(1) Set your working directory.

```
Loading required package: pacman
```

(2) Clear your global environment.

(3) Install and load the following packages: 'tidyverse', 'haven', and 'janitor'.

Read in, inspect, and clean the data

(4) Download and load the data, respectively, with the following code:

```
df <- read_dta(
  "https://github.com/hubchev/courses/raw/main/dta/city.dta",
  encoding="latin1") |>
  as_tibble()
```

If that is not working, you can also download the data from ILIAS, save it in your working directory and load it from there with:

```
load("city.RData")
```

(5) Show the first six and the last six observations of the dataset 'df'.

(6) How many observations (rows) and variables (columns) are in the dataset?

(7) Show for all numerical variables the summary statistics including the mean, median, minimum, and the maximum.

(8) Rename the variable `stadt` to `city`.

(9) Remove the variables `pop1970` and `pop1987`.

(10) Replicate the following table which contains some summary statistics.

```
# A tibble: 17 x 3
   state                  `mean(pop2011)` `sum(pop2011)`
   <chr>                            <dbl>          <dbl>
 1 Baden-Wrttemberg                 7580            7580
 2 Baden-Württemberg               23680.        7837917
 3 Bayern                          23996.        7558677
 4 Berlin                        3292365         3292365
 5 Brandenburg                     18472.        1865632
 6 Bremen                         325432.         650863
 7 Hamburg                       1706696         1706696
 8 Hessen                          22996.        5036121
 9 Mecklenburg-Vorpommern          27034.         811005
10 Niedersachsen                   24107.        6219515
11 Nordrhein-Westfalen             47465.       18036727
12 Rheinland-Pfalz                 25644.        1871995
13 Saarland                           NA              NA
14 Sachsen                         27788.        2973351
15 Sachsen-Anhalt                  21212.        1993915
16 Schleswig-Holstein              24157.        1739269
17 Th_ringen                       29192.        1167692
```

(11) The states "Baden-Wrttemberg" and "Th_ringen" are falsely pronounced. Correct the names and regenerate the summary statistics table presented above. Your result should look like this:

```
# A tibble: 16 x 3
   state                  `mean(pop2011)` `sum(pop2011)`
   <chr>                            <dbl>          <dbl>
 1 Baden-Württemberg               23631.        7845497
 2 Bayern                          23996.        7558677
 3 Berlin                        3292365         3292365
 4 Brandenburg                     18472.        1865632
 5 Bremen                         325432.         650863
 6 Hamburg                       1706696         1706696
 7 Hessen                          22996.        5036121
 8 Mecklenburg-Vorpommern          27034.         811005
 9 Niedersachsen                   24107.        6219515
10 Nordrhein-Westfalen             47465.       18036727
```

```
11 Rheinland-Pfalz                25644.        1871995
12 Saarland                         NA             NA
13 Sachsen                        27788.        2973351
14 Sachsen-Anhalt                 21212.        1993915
15 Schleswig-Holstein             24157.        1739269
16 Thüringen                      29192.        1167692
```

(12) To investigate the reason for observing only `NAs` for Saarland, examine all cities within Saarland. Therefore, please display all observations for cities in Saarland in the Console, as illustrated below.

```
# A tibble: 47 x 5
   city                status  state    pop2011 rankX
   <chr>               <chr>   <chr>      <dbl> <dbl>
 1 Perl                Commune Saarland    7775  2003
 2 Freisen             Commune Saarland    8270  1894
 3 Großrosseln         Commune Saarland    8403  1868
 4 Nonnweiler          Commune Saarland    8844  1775
 5 Nalbach             Commune Saarland    9302  1678
 6 Wallerfangen        Commune Saarland    9542  1642
 7 Kirkel              Commune Saarland   10058  1541
 8 Merchweiler         Commune Saarland   10219  1515
 9 Nohfelden           Commune Saarland   10247  1511
10 Friedrichsthal      City    Saarland   10409  1489
11 Marpingen           Commune Saarland   10590  1461
12 Mandelbachtal       Commune Saarland   11107  1390
13 Kleinblittersdorf   Commune Saarland   11396  1354
14 Überherrn           Commune Saarland   11655  1317
15 Mettlach            Commune Saarland   12180  1241
16 Tholey              Commune Saarland   12385  1217
17 Saarwellingen       Commune Saarland   13348  1104
18 Quierschied         Commune Saarland   13506  1088
19 Spiesen-Elversberg  Commune Saarland   13509  1086
20 Rehlingen-Siersburg Commune Saarland   14526   996
21 Riegelsberg         Commune Saarland   14763   982
22 Ottweiler           City    Saarland   14934   969
23 Beckingen           Commune Saarland   15355   931
24 Losheim am See      Commune Saarland   15906   887
25 Schiffweiler        Commune Saarland   15993   882
26 Wadern              City    Saarland   16181   874
27 Schmelz             Commune Saarland   16435   857
28 Sulzbach/Saar       City    Saarland   16591   849
29 Illingen            Commune Saarland   16978   827
30 Schwalbach          Commune Saarland   17320   812
31 Eppelborn           Commune Saarland   17726   793
32 Wadgassen           Commune Saarland   17885   785
33 Bexbach             City    Saarland   18038   777
34 Heusweiler          Commune Saarland   18201   762
35 Püttlingen          City    Saarland   19134   718
36 Lebach              City    Saarland   19484   701
37 Dillingen/Saar      City    Saarland   20253   654
```

```
38 Blieskastel         City    Saarland   21255   601
39 St. Wendel          City    Saarland   26220   460
40 Merzig              City    Saarland   29727   392
41 Saarlouis           City    Saarland   34479   323
42 St. Ingbert         City    Saarland   36645   299
43 Völklingen          City    Saarland   38809   279
44 Homburg             City    Saarland   41502   247
45 Neunkirchen         City    Saarland   46172   206
46 Saarbrücken         City    Saarland  175853    43
47 Perl                Commune Saarland      NA    NA
```

(13) With reference to the table above, we have identified an entry for the city of Perl that solely consists of `NAs`. This city is duplicated in the dataset, appearing at positions 1 and 47. The latter duplicate contains only `NAs` and can be safely removed without the loss of valuable information. Please eliminate this duplification and regenerate the list of all cities in the Saarland.

(14) Calculate the total population and average size of cities in Saarland.

(15) Check if any other city is recorded more than once in the dataset. To do so, reproduce the table below.

```
# A tibble: 23 x 5
# Groups:   city [11]
   city        status                  state               pop2011 unique_count
   <chr>       <chr>                   <chr>                 <dbl>        <int>
 1 Bonn        City with County Rights Nordrhein-Westfalen  305765            3
 2 Bonn        City with County Rights Nordrhein-Westfalen  305765            3
 3 Bonn        City with County Rights Nordrhein-Westfalen  305765            3
 4 Brühl       Commune                 Baden-Württemberg    13805            2
 5 Brühl       City                    Nordrhein-Westfalen  43568            2
 6 Erbach      City                    Baden-Württemberg    13024            2
 7 Erbach      City                    Hessen               13245            2
 8 Fürth       City with County Rights Bayern              115613            2
 9 Fürth       Commune                 Hessen               10481            2
10 Lichtenau   City                    Nordrhein-Westfalen  10473            2
11 Lichtenau   Commune                 Sachsen               7544            2
12 Münster     Commune                 Hessen               14071            2
13 Münster     City with County Rights Nordrhein-Westfalen  289576            2
14 Neunkirchen Commune                 Nordrhein-Westfalen  13930            2
15 Neunkirchen City                    Saarland             46172            2
16 Neuried     Commune                 Baden-Württemberg     9383            2
17 Neuried     Commune                 Bayern                8277            2
18 Petersberg  Commune                 Hessen               14766            2
19 Petersberg  Commune                 Sachsen-Anhalt       10097            2
20 Senden      City                    Bayern               21560            2
21 Senden      Commune                 Nordrhein-Westfalen  19976            2
22 Staufenberg City                    Hessen                8114            2
23 Staufenberg Commune                 Niedersachsen         7983            2
```

(16) The table indicates that the city of Bonn appears three times in the dataset, and all three observations contain identical information. Thus, remove two of these observations to

ensure that Bonn is uniquely represented in the dataset. All other cities that occur more than once in the data are situated in different states. That means, these are distinct cities that coincidentally share the same name.

Data analysis (Zipf's Law)

Note: *If you have failed to solve the data cleaning tasks above, you can download the cleaned data from ILIAS, save it in your working directory and load it from there with:* `load("city_clean.RData")`

In the following, you aim to examine the validity of Zipf's Law for Germany. Zipf's Law postulates how the size of cities is distributed. The "law" states that there is a special relationship between the size of a city and the rank it occupies in a series sorted by city size. In the estimation equation

$$\log(M_j) = c - q\log(R_j),$$

the law postulates a coefficient of $(q = 1)$. c is a constant; M_j is the size of city j; R_j is the rank that city j occupies in a series sorted by city size.

(17)

Create a variable named `rank` that includes a ranking of cities based on the population size in the year 2011. Therefore, Berlin should have a rank of 1, Hamburg a rank of 2, Munich a rank of 3, and so on.

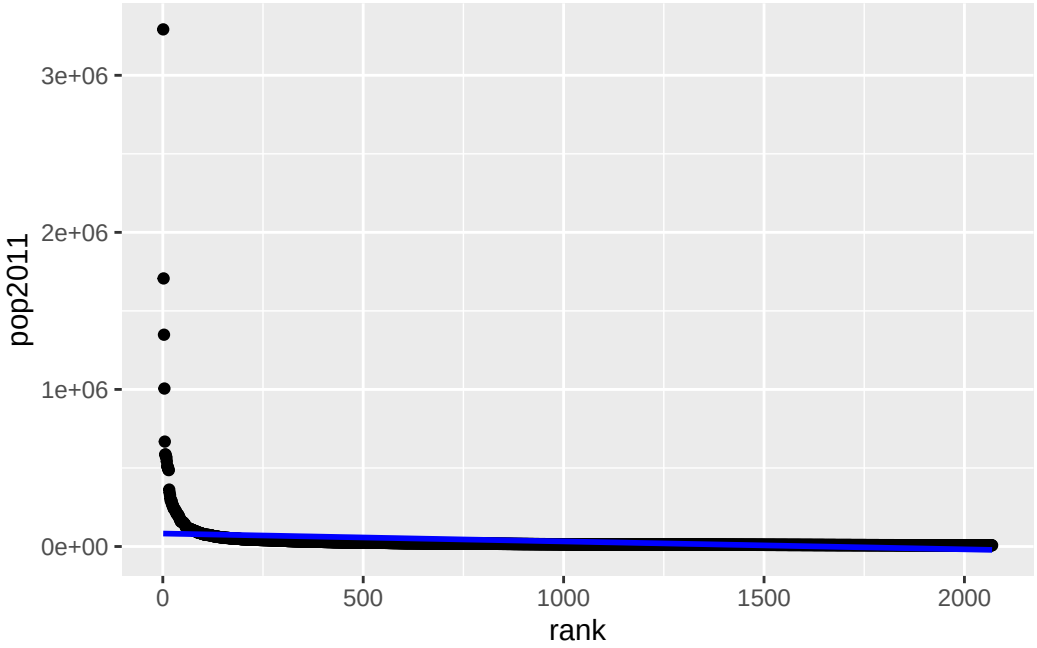
Note: If you cannot solve this task, use the variable `rankX` as a substitute for the variable* `rank` that was not generated.

```
# A tibble: 6 x 3
  city                    pop2011  rank
  <chr>                     <dbl> <int>
1 Berlin                  3292365     1
2 Hamburg                 1706696     2
3 München [Munich]        1348335     3
4 Köln [Cologne]          1005775     4
5 Frankfurt am Main        667925     5
6 Düsseldorf [Dusseldorf]  586291     6
```

(18) Calculate the Pearson Correlation Coefficient of the two variables `pop2011` and `rank`. The result should be:

```
[1] -0.2948903
```

(19) Create a scatter plot. On the x-axis, plot the variable `rank`, and on the y-axis, plot `pop2011`. Add a regression line representing the observed relationship to the same scatter plot.



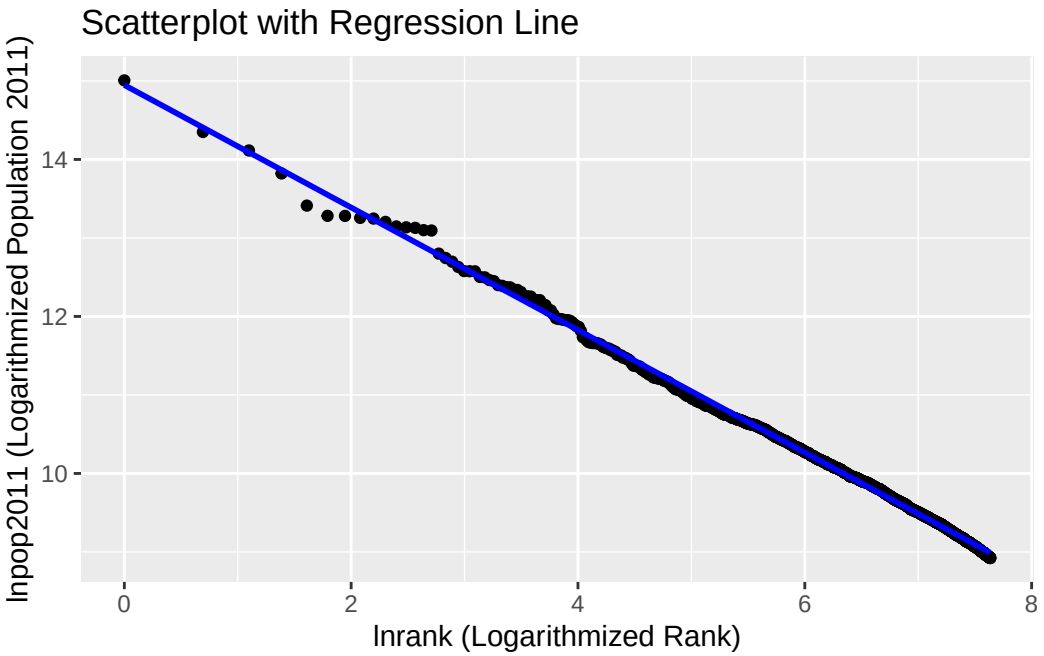
(20) Logarithmize the variables `rank` and `pop2011`. Title the new variables as `lnrank` and `lnpop2011`, respectively. Here is a snapshot of the resulting variables:

```
# A tibble: 6 x 5
  city                      rank lnrank pop2011 lnpop2011
  <chr>                    <int>  <dbl>   <dbl>     <dbl>
1 Berlin                       1  0     3292365      15.0
2 Hamburg                      2  0.693 1706696      14.4
3 München [Munich]             3  1.10  1348335      14.1
4 Köln [Cologne]               4  1.39  1005775      13.8
5 Frankfurt am Main            5  1.61   667925      13.4
6 Düsseldorf [Dusseldorf]      6  1.79   586291      13.3
```

(21) Calculate the Pearson Correlation Coefficient of the two variables `lnpop2011` and `lnrank`. The result should be:

```
[1] -0.9990053
```

(22) Create a scatter plot. On the x-axis, plot the variable `lnrank`, and on the y-axis, plot `lnpop2011`. Add a regression line representing the observed relationship to the same scatter plot. Additionally, add a title and label the axes like is shown here:



(23) Now, test the relationship postulated in Zipf's Law. Regress the logarithmized city size in the year 2011 on the logarithmized rank of a city in a series sorted by city size. Briefly interpret the results, addressing the coefficient of determination. Show the regression results. Here is one way to present the results of the regression *(Note: The way how you present your regression results do not matter)*:

```
Call:
lm(formula = lnpop2011 ~ lnrank, data = df)

Residuals:
     Min       1Q   Median       3Q      Max
-0.28015 -0.01879  0.01083  0.02005  0.25973

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 14.947859   0.005141    2908   <2e-16 ***
lnrank      -0.780259   0.000766   -1019   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03454 on 2067 degrees of freedom
Multiple R-squared:  0.998, Adjusted R-squared:  0.998
F-statistic: 1.038e+06 on 1 and 2067 DF,  p-value: < 2.2e-16
```

(24) Explain the following lines of code.

```
df <- df |>
  mutate(prediction = predict(zipf, newdata = df)) |>
  mutate(pred_pop = exp(prediction))
df |>
  select(city, pop2011, pred_pop) |>
  filter(city == "Regensburg")
```

```
# A tibble: 1 x 3
  city       pop2011 pred_pop
  <chr>        <dbl>    <dbl>
1 Regensburg  135403  134194.
```

Solutions are provided here.

References

John M. Chambers. *Extending R*. CRC Press, 2017.

Thomas H Davenport and DJ Patil. Data scientist: The sexiest job of the 21st century. *Harvard Business Review*, 90(5):70–76, 2012.

Wickham Hadley. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014.

Ali Hortaçsu and Chad Syverson. The ongoing evolution of US retail: A format tug-of-war. *Journal of Economic Perspectives*, 29(4):89–112, 2015.

Rafael A. Irizarry. *Introduction to Data Science: Data Analysis and Prediction Algorithms With R*. CRC Press, 2022. URL https://rafalab.github.io/dsbook/.

Chester Ismay and Albert Y. Kim. *Statistical inference via data science: A ModernDive into R and the tidyverse*. CRC Press, 2022. URL https://moderndive.com/.

Robert Kabacoff. *Modern Data Visualization with R*. Chapman and Hall/CRC, 2024. URL https://rkabacoff.github.io/datavis/.

John D Kelleher and Brendan Tierney. *Data Science*. MIT Press, 2018.

Oliver Kirchkamp. Using graphs and visualising data. Technical report, 2018. https://www.kirchkamp.de/oekonometrie/pdf/gra-p.pdf (retrieved on 2022/05/20).

John Muschelli and Andrew Jaffe. Introduction to R for public health researchers. Technical report, GitHub, 2022. URL https://github.com/muschellij2/intro_to_r.

Danielle Navarro. *Learning Statistics With R*. Version 0.6 edition, 2020. URL https://learningstatisticswithr.com.

Hansjörg Neth. *ds4psy: Data Science for Psychologists*. Social Psychology and Decision Sciences, University of Konstanz, Konstanz, Germany, 2023. URL https://CRAN.R-project.org/package=ds4psy. R package (version 0.9.0, October 20, 2022); Textbook at <https://bookdown.org/hneth/ds4psy/>.

Måns Thulin. *Modern Statistics With R: From Wrangling and Exploring Data to Inference and Predictive Modelling*. Eos Chasma Press, 2021. URL https://www.modernstatisticswithr.com/.

Tiffany Timbers, Trevor Campbell, and Melissa Lee. *Data Science: A First Introduction*. CRC Press, 2022. URL https://datasciencebook.ca/.

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2 edition, 2022.

William N. Venables, David M. Smith, and R Core Team. *An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics*. Version 4.3.2 (2023-10-31) edition, 2022. URL http://cran.r-project.org/doc/manuals/R-intro.pdf. http://cran.r-project.org/doc/manuals/R-intro.pdf (retrieved on 2022/04/06).

Hadley Wickham and Garrett Grolemund. R for data science (2e), 2023. URL https://r4ds.hadley.nz/.

Jeffrey M. Wooldridge. *Introductory Econometrics: A Modern Approach*. South-Western, 2nd edition, 2002.

A. Navigating the file system

It is essential to know how R interacts with the file system on your computer. Modern operating systems are incredibly user-friendly and try to hide boring and annoying stuff from the customer. In the following, I will try to give a brief introduction on how to navigate around a computer using a DOS or UNIX shell. If you familiar with that, you can skip this part of the notes.

A.1. The file system

In this section, I describe the basic idea behind file locations and file paths. Regardless of whether you are using Windows, macOS, or Linux, every file on the computer is assigned a human-readable address, and every address has the same basic structure: it describes a path that starts from a root location, through a series of folders (or directories), and finally ends up at the file.

On a Windows computer, the root is the storage device on which the file is stored, and for many home computers, the name of the storage device that stores all your files is `C:`. After that comes the folders, and on Windows, the folder names are separated by a backslash symbol `\`. So, the complete path to this book on my Windows computer might be something like this:

```
C:\Users\huber\Rbook\rcourse-book.pdf
```

On Linux, Unix, and macOS systems, the addresses look a little different, but they are more or less identical in spirit. Instead of using the backslash, folders are separated using a forward slash, and unlike Windows, they do not treat the storage device as being the root of the file system. So, the path on a Mac might be something like this:

/Users/huber/Rbook/rcourse-book.pdf

That is what we mean by the *path* to a file. The next concept to grasp is the idea of a working directory and how to change it. For those of you who have used command-line interfaces previously, this should be obvious already. But if not, here is what I mean. The working directory is just whatever folder I am currently looking at. Suppose that I am currently looking for files in Explorer (if you are using Windows) or using Finder (on a Mac). The folder I currently have open is my user directory (i.e., `C:\Users\huber` or `/Users/huber`). That is my current working directory.

A.2. Working directory

The next concept to grasp is the idea of a *working directory* and how to change it. For those of you who have used command line interfaces previously, this should be obvious already. But if not, here's what I mean. The working directory is just "whatever folder I'm currently looking at". Suppose that I'm currently looking for files in Explorer (if you're using Windows) or using

Finder (on a Mac). The folder I currently have open is my user directory (i.e., `C:\Users\huber` or `/Users/huber`). That's my current working directory.

The fact that we can imagine that the program is "in" a particular directory means that we can talk about moving *from* our current location *to* a new one. What that means is that we might want to specify a new location in relation to our current location. To do so, we need to introduce two new conventions. Regardless of what operating system you're using, we use `.` to refer to the current working directory, and `..` to refer to the directory above it. This allows us to specify a path to a new location in relation to our current location, as the following examples illustrate. Let's assume that I'm using my Windows computer, and my working directory is `C:\Users\huber\Rbook`. The table below shows several addresses in relation to my current one:

Absolute path	Relative path
`C:\Users\huber`	`..`
`C:\Users`	`..\..`
`C:\Users\huber\Rbook\source`	`.\source`
`C:\Users\huber\nerdstuff`	`..\nerdstuff`

It is quite common on computers that have multiple users to define `~` to be the user's *home directory*. The home directory on a Mac for the 'huber'' user is/Users/huber/. And so, not surprisingly, it is possible to define other directories in terms of their relationship to the home directory. For example, an alternative way to describe the location of thercourse-book.pdf' file on a Mac would be

```
~\Rbook\rcourse-book.pdf
```

You can find out your home directory with the `path.expand()` function:

```
path.expand("~")
```

```
[1] "/home/sthu"
```

Thus, on my machine ~ is an abbreviation for the path /home/sthu.

```
getwd()
```

```
[1] "/home/sthu/Dropbox/hsf/courses/dsr"
```

A.3. Navigating the file system using the R console

When you want to load or save a file in R it's important to know what the working directory is. You can find out by using the `getwd()` command. For the moment, let's assume that I'm using Mac OS or Linux, since things are different on Windows, see section @ref(sec:winbackslash). Let's check the current active working directory:

```
getwd()
```

```
[1] "/home/sthu/Dropbox/hsf/courses/dsr"
```

The function `setwd()` allows to change the working directory:

```
setwd("/Users/huber/Rbook/data")
setwd("./Rbook/data")
```

The function `list.files()` lists all the files in that directory:

```
list.files()
```

A.4. R Studio projects

Setting the working directory repeatedly can be a cumbersome task. Fortunately, R Studio projects can automate this process for you. When you open an R Studio project, the working directory is automatically set to the project directory.

Creating a new project in R Studio is simple. Just click on *File > New Project...*. This will create a directory on your computer with a _*.Rproj_ file that can be used to open the saved project at a later date. The newly created directory contains your R code, data files, and other project-related files. By working within projects, all of your files and data are organized in one place, making it easier to share your work with others, reproduce your analyses, and keep track of changes over time.

A.5. Why do the Windows paths use the back-slash?

Let's suppose I'm using a computer with Windows. As before, I can find out what my current working directory is like this:

```
getwd()
[1] "C:/Users/huber/
```

R is displaying a Windows path using the wrong type of slash, the back-slash. The answer has to do with the fact that R treats the \ character as *special*. If you're deeply wedded to the idea of specifying a path using the Windows style slashes, then what you need to use two back-slashes \\ whenever you mean \. In other words, if you want to specify the working directory on a Windows computer, you need to use one of the following commands:

```
setwd( "C:/Users/huber" )
setwd( "C:\\Users\\huber" )
```

B. Troubleshooting

Troubleshooting is perhaps the most important skill for a data scientist. To tackle problems effectively, you need to understand them, replicate them, and then work to solve them. To help you with this process, here are some guidelines:

When seeking help, be sure to provide information about your machine, including the operating system, the version of R, and the packages you have loaded. You can use the sessionInfo() function to gather this information. Here's an example from my machine:

```
sessionInfo()
```

```
R version 4.2.2 Patched (2022-11-10 r83330)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Debian GNU/Linux 12 (bookworm)

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.21.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.2.2    fastmap_1.1.1     cli_3.6.2         tools_4.2.2
 [5] htmltools_0.5.7   rstudioapi_0.15.0 rmarkdown_2.25    knitr_1.45
 [9] jsonlite_1.8.8    xfun_0.41         digest_0.6.33     rlang_1.1.2
[13] evaluate_0.23
```

C. Operators

C.1. Assignment:

- `<-` (assignment operator)

C.2. Arithmetic:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `^` or `**` (exponentiation)
- `%%` (modulo, remainder)
- `%/%` (integer division)

C.3. Relational:

- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)
- `==` (equal to)
- `!=` or `<>` (not equal to)

C.4. Logical:

- `&` (element-wise AND)
- `|` (element-wise OR)
- `!` (logical NOT)
- `&&` (scalar AND)
- `||` (scalar OR)

C.5. Others:

- `%*%` (matrix multiplication)
- `%in%` (checks if an element is in a vector)
- `%>%` or `|>` (pipe operator from the magrittr package)
- `[]`: Extract content from vectors, lists, or data frames.
- `[[ ]]` and `$`: Extract a single item from an object.

D. Popular functions

D.1. Help

- ?: Search R documentation for a specific term.
- ?? Search R help files for a word or phrase.
- RSiteSearch: Search search.r-project.org
- findFn: Search search.r-project.org for functions (Hint: requires the "sos" library loaded!)
- help.start: Access to html manuals and documentations implemented in R
- apropos: Returns a character vector giving the names of objects in the search list matching (as a regular expression) what.
- find: Returns where objects of a given name can be found.
- vignette: View a specified package vignette, i.e., supporting material such as introductions.

D.2. Package management

- `install.packages`: Installs packages from CRAN.
- `pacman::p_load`: Installs and loads specified R packages.
- `library`: (Install and) loads specified R packages.

D.3. General

- `setwd`: Sets the working directory to the specified path.
- `rm`: Removes objects (variables) from the workspace.
- `sessionInfo`: Information about the R environment.
- `source`: Executes R code from a file.

D.4. Tools

- `else`: Execute a block of code if the preceding condition is false.
- `else if`: Specify a new condition to test if the first condition is false.
- `if`: Execute a block of code if a specified condition is true.
- `ifelse`: Check a condition for every element of a vector.

D.5. Data import

- `c`: Combine values into a vector or list.
- `read.csv`: Reads a CSV file into a data frame.
- `read_dta`: Read Stata dataset.
- `load`: Loads an RData file.

D.6. Inspect data

- `dim`: Returns the dimensions (number of rows and columns) of a data frame.
- `glimpse`: Provide a concise summary.
- `head`: Returns the first elements.
- `length`: Returns the number of elements in an object.
- `print`: Prints the specified object.
- `names`: Returns the variable names in a data frame.
- `n() or nrow()`: Counts the number of observations in a data frame or group of observations.
- `ncol`: Returns the number of columns in a data frame.
- `summary`: Summary statistics.
- `table`: Create a table of counts or cross-tabulation.
- `tail`: Returns the first n elements.
- `unique`: Extracts unique elements from a vector.
- `view`: Opens a viewer for data frames.

D.7. Graphics

- `abline`: Adds lines to a plot.
- `aes`: Aesthetic mapping in ggplot.
- `facet_wrap`: Creates a grid of facetted plots.
- `geom_hline`: Adds horizontal lines to a ggplot.
- `geom_line`: Adds lines to a ggplot.
- `geom_point`: Adds points to a ggplot.
- `geom_smooth`: Adds a smoothed line to a ggplot.
- `geom_text`: Adds text to a ggplot.
- `geom_vline`: Adds vertical lines to a ggplot.
- `ggsave`: Saves a ggplot to a file.
- `labs`: Adds or modifies plot labels.
- `plot`: Creates a scatter plot.
- `scale_y_reverse`: Reverses the y-axis in a ggplot.
- `stat_smooth`: Adds a smoothed line to a ggplot.
- `theme_classic`: Applies a classic theme to a ggplot.
- `theme_minimal`: Applies a minimal theme to a ggplot.

D.8. Data management

- `arrange`: Reorder the rows of a data frame.
- `clean_names`: Cleans names of an object (usually a data.frame).
- `complete`: Completes a data frame with all combinations of specified columns.
- `data.frame`: Creates a data frame.
- `distinct`: Removes duplicate rows from a data frame.
- `identical`: Check if two objects are identical.
- `is(na)`: Identify and flag a missing or undefined value (NA).
- `is_tibble`: Check if an object is a tibble.
- `rm`: Removes objects (variables) from the workspace.
- `relocate`: Reorders columns in a dataframe.

- `round`: Rounds a numeric vector to the nearest integer.
- `rownames`: Get or set the row names of a matrix-like object.
- `tibble`: Creates a tibble, a modern and tidy data frame.

D.9. `dplyr` functions

- `arrange`: Reorder the rows of a data frame.
- `complete`: Completes a data frame with all combinations of specified columns.
- `ends_with`: matches to a specified suffix
- `filter`: Pick observations by their values.
- `first`: Returns the first element.
- `group_by`: Group data by one or more variables.
- `last`: Returns the last element.
- `mutate`: Add new variables or modify existing variables in a data frame.
- `nth`: Returns the nth element.
- `n_distinct`: Returns the number of distinct elements.
- `rename`: Rename variables in a data frame.
- `rename_all`: Renames all variables in a data frame.
- `row_number`: Adds a column with row numbers.
- `rowwise`: Perform operations row by row.
- `select`: Pick variables by their names.
- `select_all`: Selects all columns in a data frame.
- `slice_head`: Selects the top N rows from each group.
- `starts_with`: Select variables whose names start with a certain string.
- `summarise`: Reduce data to a single summary value.

D.10. Data analysis

- `aggregate`: Apply a function to the data by levels of one or more factors.
- `anti_join`: Return rows from the first data frame that do not have a match in the second data frame.
- `cor`: Computes correlation coefficients.
- `cov`: Computes covariance.
- `diff`: Calculates differences between consecutive elements.
- `get_dupes`: Identify duplicate rows in a data frame (from the janitor package).
- `paste0`: Concatenate vectors after converting to character.
- `predict`: Predict method for model fits.
- `prop.table`: Create a table of proportions.

D.11. Statistical functions

- `cor()`: Computes correlation coefficients.
- `cov()`: Computes the covariance.
- `exp()`: Exponential function.
- `IQR()`: Computes the interquartile range.
- `kurtosis()`: Computes the kurtosis.
- `log()`: Natural logarithm.

- `mad()`: Computes the mean absolute deviation.
- `max()`: Returns the maximum value.
- `mean()`: Calculates the mean.
- `median()`: Computes the median.
- `min()`: Returns the minimum value.
- `quantile()`: Computes sample quantiles.
- `sd()`: Calculates the standard deviation.
- `skewness()`: Calculates the skewness.
- `var()`: Calculates the variance.

E. Helpful shortcuts

Table E.1.: Table 1: Different OS, different keys

Key in Windows/Linux	Key in Mac
CTRL	Command Key
Alt	Option Key

Table E.2.: Table 2: Helpful shortcuts

Action	Shortcut Keys	Description
Run code	Ctrl + Enter	Runs the current line and jumps to the next one, or runs the selected part without jumping further.
	Alt + Enter	Allows running code without moving the cursor to the next line if you want to run one line of code multiple times without selecting it.
	Ctrl + Alt + R	Runs the entire script.
	Ctrl + Alt + B/E	Run the script from the Beginning to the current line and from the current line to the End.
Write code	Alt + (-)	Inserts the assignment operator (<-) with spaces surrounding it.
	Ctrl + Shift + M	Inserts the magrittr/pipe operator (%>%) with spaces surrounding it.
	Ctrl + Shift + C	Comments out code by putting a # in front of each line of marked code of a script.
	Ctrl + Shift + R	Creates a foldable comment section in your code.
Navigating in RStudio	Ctrl + 1	Move focus to editor.
	Ctrl + 2	Move focus to console.
	Ctrl+Tab and Ctrl+Shift+Tab	to switch between tabs.
	Ctrl + Shift + N	Open a new R script.
	Ctrl + w	Close a tab.