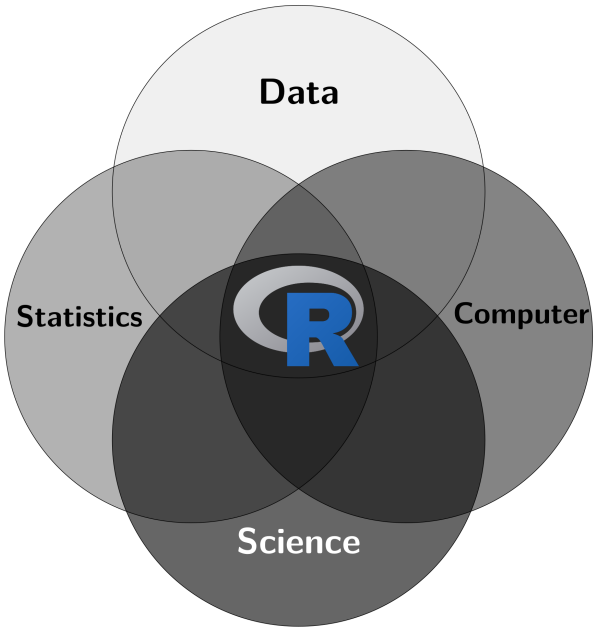


How to Use R for Data Science

Lecture Notes

© Prof. Dr. Stephan Huber (Stephan.Huber@hs-fresenius.de)

Last compiled on 02 June, 2023

Contents

Preface

- These notes aims to support my lecture at the HS Fresenius but are incomplete and no substitute for taking actively part in class. - A pdf version of these notes is available here
- I host the notes in a GitHub repo.
- I appreciate you reading it, and I appreciate any comments.
- This is work in progress so please check for updates regularly.
-

About the author

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 and the University of Perugia. 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 www.t1p.de/stephanhuber.

Contact

Hochschule Fresenius für Wirtschaft & Medien GmbH
Im MediaPark 4c
50670 Cologne

Office: 4b OG-1 Bü01 (Office hour: Thursday 1-2 p.m.)
Telefon: +49 221 973199-523
Mail: stephan.huber@hs-fresenius.de
Private homepage: www.t1p.de/stephanhuber
Github: https://github.com/hubchev

Chapter 1

Getting Started with R

Before we can start exploring data in R, there are some key concepts to understand first:

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. 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:
  - https://www.r-graph-gallery.com/

  - http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html
  - https://www.r-bloggers.com/2020/05/7-useful-interactive-charts-in-r/

- **R is an employment insurance.** If you are good in R programming or if you are good in writing programming code in general, you have plenty of opportunities to earn a decent salary.
- **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 excel at mindless repetitive tasks. For most people, the only reason to ever do statistical calculations with pencil and paper is for learning purposes.
- **Excel is bad!** Doing statistics in a spreadsheet (e.g., Microsoft Excel) is often a bad idea. Although many people are likely feel more familiar with them, spreadsheets are very limited in terms of what analyses they allow you to do. You can easily lose the overview and it is hard to keep track of what you have done and in comparison with command line driven programs. In particular, the ability to make your analysis *replicable* is limited.
- **R is good, proprietary software is bad!** Avoiding proprietary software is a very good idea because it costly, support is exclusively provided by the owner of the software (if they stop supporting your version you are lost), security issues cannot be checked as the source code is not available, and possibilities for customization are limited.
- **R is big!** Something that you might not appreciate now, but will love later on if you do anything involving data analysis, is the fact that R is highly extensible. When you download and install R, you get all the basic packages, and those are very powerful on their own. However, because R is so open and so widely used, it's become something of a standard tool in statistics, and so lots of people write their own packages that extend the system. And these are freely available too. One of the consequences of this, I've noticed, is that if you open up an advanced textbook (a recent one, that is) rather than introductory textbooks, is that a *lot* of them use R. In other words, if you learn how to do your basic statistics in R, then you're a lot closer to being able to use the state of the art methods than you would be if you'd started out with a "simpler" system: so if you want to become a genuine expert in data analysis, learning R is a very good use of your time.
- **R is the future!** Programming is a core skill in research, economics, and business. 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.

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 the learning goals. However, I offer these notes and if you are in one of my classes, you can ask at any time for help.

The notes should walk you through many of the things that are important when working in R, and it should help you dig deeper and learn more if you want to. For beginners, I recommend starting with my swirl courses, see section 2. However, there are thousands of other resources for learning R: textbooks, online courses, videos, guided tutorials, etc.. I give some recommendations about learning resources in section 1.3.

Below, I'll give you a list of resources that are worth a look. You might find what you're looking for there. If not, just keep reading this book. Above all, those who have personally taken one of my courses are welcome to contact me if they think I can help them.

Warning: R is not without its weaknesses: It's not easy to learn, it has some very annoying quirks that we all have to deal with, it's slower than other languages (Phyton, MATLAB), and R's algorithms and sources are spread across many packages (since there's no big company behind it that wants you to buy it). This sometimes makes it very difficult for beginners to find what they are looking for. In simple words: you can get lost!

Tips on learning to code: Learning to code/program is quite similar to learning a foreign language. It can be daunting and frustrating at first. Such frustrations are common and it is normal to feel discouraged as you learn. However, just as with learning a foreign language, if you put in the effort and are not afraid to make mistakes, anybody can learn and improve.

Here are a few useful tips to keep in mind as you learn to program:

- **Remember that computers are not actually that smart**: You may think your computer or smartphone is "smart," but really people spent a lot of time and energy designing them to appear "smart." In reality, you have to tell a computer everything it needs to do. Furthermore, the instructions you give your computer can't have any mistakes in them, nor can they be ambiguous in any way.
- **Take the "copy, paste, and tweak" approach**: While learning code from scratch is sometimes essential, I prefer to take existing code that I know works and modify it to suit my ends. I call this the *"copy, paste, and tweak"* approach. It is a good way to learn code and get a job done quick. After you start feeling more confident, you can slowly move away from this approach and write code from scratch.

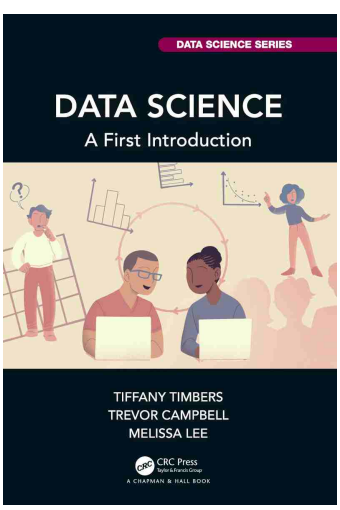
- **Have a purpose when coding**: Rather than learning to code for its own sake, it goes much smoother when you have a goal in mind or when you are working on a particular project, like analyzing data that you are interested in, replicating a paper to understand it, or do something with R that is important to you.
- **Practice is key**: Just as the only method to improve your foreign language skills is through lots of practice and speaking, the only method to improving your coding skills is through lots of practice. Don't worry, however, we'll give you plenty of opportunities to do so!

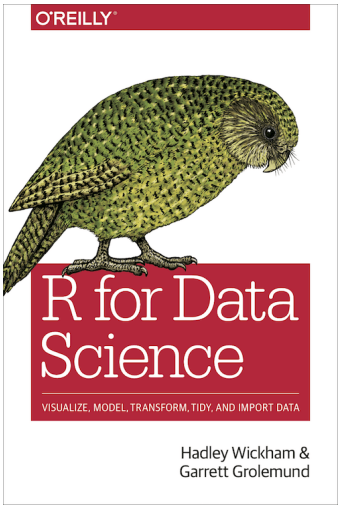
1.3 Learning resources

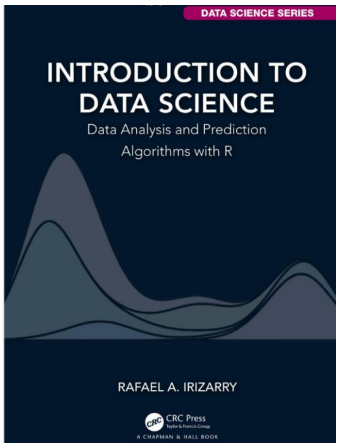
Thousand of freely available books and resources exist. On bookdown.org and in the Big Book of R is a big collection of links to R books that verifies my claim. Another nice collection of learning ressources can be found here: AWESOME R Learning-Resources

In Rstudio you find in the left panel 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

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







1. Timbers, Campbell, and Lee (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, Smith, and R Core Team (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.

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, we 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 1.1.

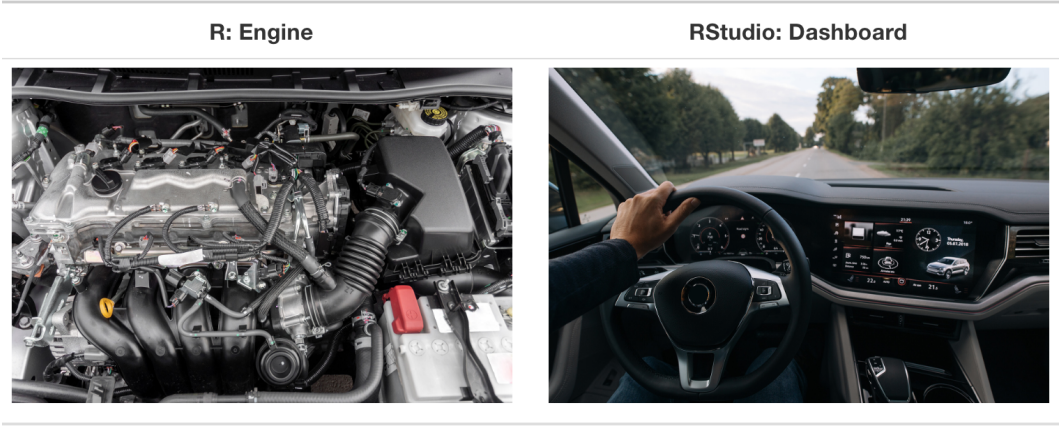


Figure 1.1: Analogy of difference between R and RStudio.

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 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 1.2 shows what icon you should be clicking on your computer.

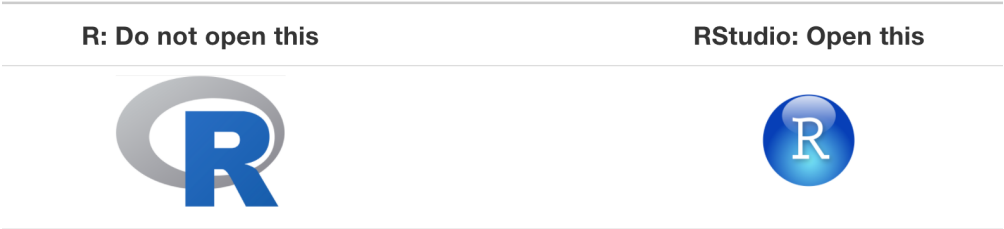


Figure 1.2: Icons of R versus RStudio on your computer.

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

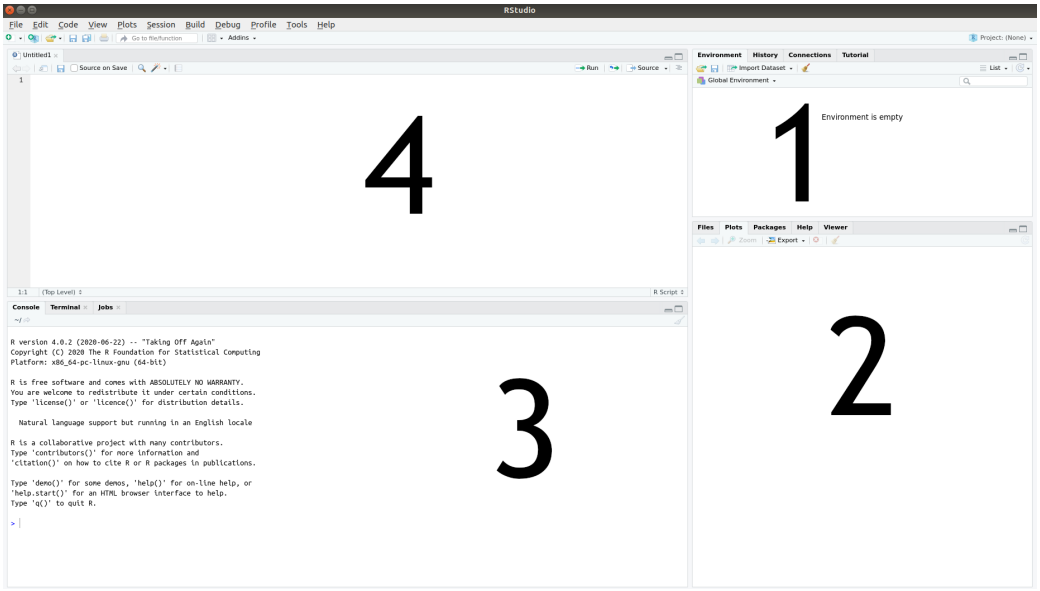


Figure 1.3: RStudio interface to R.

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:

> R version 4.1.2 (2021-11-01) – "Bird Hippie" Copyright (C) 2021 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.

When you exit RStudio, you will be asked if you wish to *save your workspace*, meaning that the data that you've worked with will be stored so that it is available the next time you run R. That might sound like a good idea, but in general, I recommend that you don't save your workspace, as that often turns out to cause problems down the line. It is almost invariably a much better idea to simply rerun the code you worked with in your next R session.

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, 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 do, share, teach and learn data science online. It is free for individuals with some restrictions and limited capacities.

1.6 Installing R and RStudio

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 November 25, 2019 was R-3.6.1.
- 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 are R packages?

A package is basically just a big collection of functions, data sets and other R objects that are all grouped together under a common name. Some packages are already installed when you put R on your computer, but the vast majority of them of R packages are out there on the internet, waiting for you to download, install and use them. R packages are collections of functions and data sets developed by the community. They increase the power of R by improving existing base R functionalities, or by adding new ones. For example, if you are usually working with data frames, probably you will have heard about *dplyr* or *data.table*, two of the most popular R packages. More than 10,000 packages are available at the official repository (CRAN) and many more are publicly available through the internet.

In this section, I'll describe how to work with packages using the Rstudio tools. Along the way, you'll see that whenever you get Rstudio to do something (e.g., install a package), you'll actually see the R commands that get created.

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. Basically what it boils down to is this:

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 the library every time we start a new R environment. You can think of this as installing a bulb versus turning on the light.

Figure 1.4: Installing packages

The two step process might seem a little odd at first, but the designers of R had very good reasons to do it this way. That is, there are more than 10.000 packages, and probably about 8000 authors of packages, and no-one really knows what all of them do. Keeping the installation separate from the loading minimizes the chances that two packages will interact with each other in a nasty way. Moreover having installed all available packages would probably blow your hard disk.

Another good analogy for R packages is they are like apps you can download onto a mobile phone:

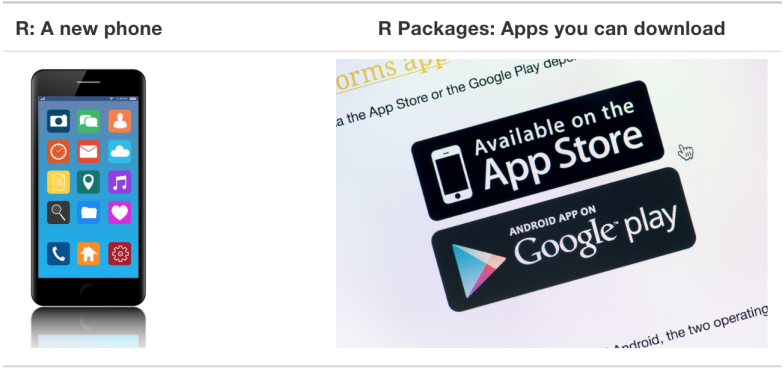


Figure 1.5: Analogy of R versus R packages.

So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn't have everything. R packages are like the apps you can download onto your phone from Apple's App Store or Android's Google Play.

1.7.1 Package installation

There are two ways to install an R package: an easy way and a very easy way. Let's install the `ggplot2` package the easy way first as shown in Figure 1.6. In the Files pane of RStudio:

a) Click on the "Packages" tab.
b) Click on "Install" next to Update.
c) Type the name of the package under "Packages (separate multiple with space or comma):" In this case, type `ggplot2`.
d) Click "Install."

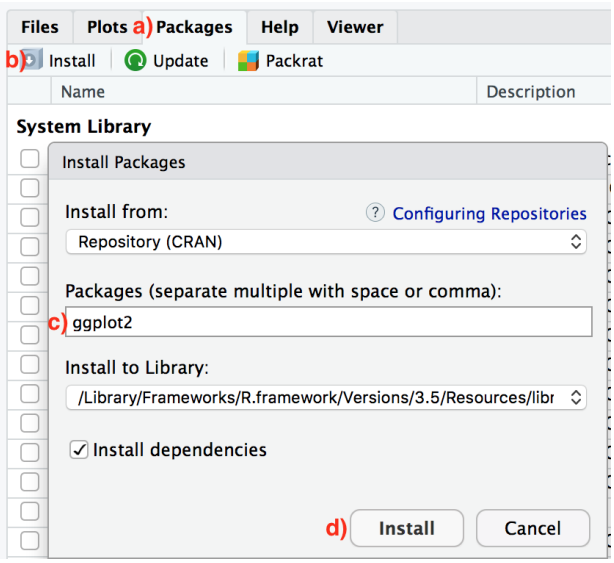


Figure 1.6: Installing packages in R the easy way.

An alternative way to install a package is by typing

```
install.packages("ggplot2")
```

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.

Much like an app on your phone, you only have to install a package once. However, 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()`.

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.

1.7.2 Package loading

Recall that after you've installed a package, you need to *load it*. In other words, you need to *open 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 1.7.1 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(, unload=TRUE)`.

1.8 What is a function in R?

One of the keys to understand R is a *functional programming language*. Or, to put in in the words of Chambers (2017, p. 4): "Everything that happens is a function call." When you like to exit R, for example, you do it with the function `q()`:

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

If you don't want to be asked what you want to do with your workspace (that is the place where you store all your objects, see section 1.9) you can do it with an argument that is part of the function:

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

1.9 What are objects in R?

Everything you do is made with a function and "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 created using a variety of data types, including vectors, matrices, data frames, lists, functions. 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.

1.10 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.
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 7.2.4)
11. Do remember to comment your code.
12. Do use <- instead of = for assignment.

An exhaustive style guide on how to write code can be found here: https://style.tidyverse.org

Chapter 2

Learn interactively with swirl

The R package *swirl* makes it fun and easy to learn R programming and data science. *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. To start it, 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 lines of code do the following:

- Install the *swirl* package.
- Load the *swirl* package.
- Install my *swirl* course that is hosted on GitHub.
- With *swirl* you start *swirl* and your learning experience.

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

Please choose the course *swirl-it* and the learning module *huber-intro-1*. You can exit *swirl* at any time by typing `bye()` or by clicking the *Esc* on your keyboard.

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*.

swirl modules on 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.

swirl module on 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.

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()
```

Chapter 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)
```

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

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 7.2 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 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)
```

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()
```

3.5 User-defined functions

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/4.1/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. 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)`.

Exercise 3.1. All roads lead to R(ome)

If you ask ten programmers to solve a particular problem, you will probably receive ten different soltuions that are all valid. R is no exception here. This can be very confussing when just started to learn R.

Below you find more ways to calculate the a growth rate. Do you understand them?

```
c(NA, diff(revenue)/head(revenue, -1))*100
```

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

```
(revenue/c(NA,revenue[-length(revenue)])-1)*100
```

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

An often-used method for approximating growth rates is to calculate the differences in the logarithm of values over time. Section 5.2 illustrates how to perform this calculation with R.

Chapter 4

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 infirmation simply and intuitive. Whatever your goal is, thousand of appropiate 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 4.1 may proof this.
- For more tips, see:

  - 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 offers 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.

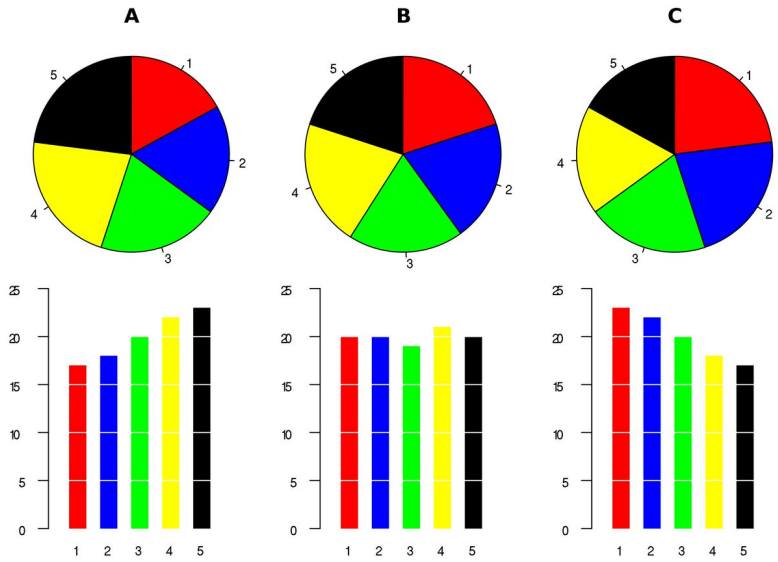


Figure 4.1: Pie charts are problematic[1]

When it comes to data visualization, there's no better resource than the book by Wickham and Grolemund (2023). 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 section 2 (Data visualization) of Wickham and Grolemund (2023). 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.

[1] Picture is taken from `https://en.wikipedia.org/wiki/Pie_chart`

Chapter 5

Manage data

5.1 The tidyverse universe

Figure 5.1: The tidyverse universe

The R package *tidyverse* is a collection of R packages. All packages share an underlying design philosophy, grammar, and data structures. The most popular packages are ggplot2, dplyr, tidyr, readr, purrr, tibble, stringr, and forcats. They provide functionality to model, transform, and visualize data. Tidyverse is extremely popular and many individual packages of tidyverse are regularly in the top 10 most downloaded R packages on CRAN[1]. 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

[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.

3 (Workflow: basics) and 4 (Data transformation) of Wickham and Grolemund (2023). Addidionally, visit www.tidyverse.org and, if you still haven‘t done so, do the tidyverse module of my swirl package, called *swirl-it*, see section 2.

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

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

5.2 The pipe operator

The pipe operator, %>%, comes from the magrittr package, which is also part of the tidyverse package. The base 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.
```

5.3 Import data

5.3.1 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 5.4

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

5.3.2 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.

5.3.3 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
```

5.3.4 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")
head(df_gdp, 3)
```

```
##   iso3c date        value    indicatorID         indicator iso2c
## 2   AFE 2021 1.089454e+12 NY.GDP.MKTP.CD GDP (current US$)    ZH Africa Easter
## 3   AFE 2020 9.341791e+11 NY.GDP.MKTP.CD GDP (current US$)    ZH Africa Easter
## 4   AFE 2019 1.009052e+12 NY.GDP.MKTP.CD GDP (current US$)    ZH Africa Easter
```

```
glimpse(df_gdp)
```

```
## Rows: 13,156
## Columns: 7
## $ iso3c       <chr> "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "A
## $ date        <chr> "2021", "2020", "2019", "2018", "2017", "2016", "2015", "2
## $ value       <dbl> 1.089454e+12, 9.341791e+11, 1.009052e+12, 1.016697e+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:13156       Length:13156       Min.   :8.824e+06   Length:13156
##  Class :character   Class :character   1st Qu.:2.442e+09   Class :character
##  Mode  :character   Mode  :character   Median :1.784e+10   Mode  :character
##                                        Mean   :1.161e+12
##                                        3rd Qu.:2.156e+11
##                                        Max.   :9.653e+13
```

5.3.5 Import various file formats

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"

Figure 5.2: The logo of the packages readr, haven, and readxl

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 the packages that are provided by the tidyverse universe, see .

5.4 Data

5.4.1 Data frames and tibbles

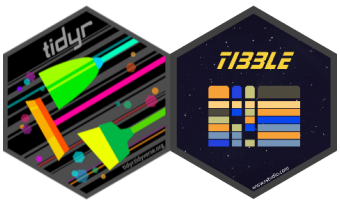


Figure 5.3: The logos of the tidyr and tibble packages

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).

5.4.2 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 5.4. 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.

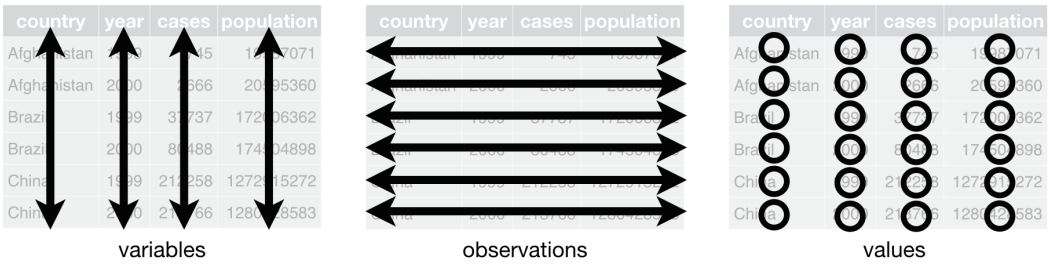


Figure 5.4: Features of a tidy dataset: variables are columns, observations are rows, and values are cells[3]

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

[3]The figure stems from Wickham and Grolemund (2023)

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"          "Hornet
##  [8] "Merc 240D"           "Merc 230"            "Merc 280"            "Merc 2
## [15] "Cadillac Fleetwood"  "Lincoln Continental" "Chrysler Imperial"   "Fiat 1
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"          "Pontia
## [29] "Ford Pantera L"      "Ferrari Dino"        "Maserati Bora"       "Volvo
```

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  c
##   <chr>          <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <d
## 1 Mazda RX4       21       6   160   110  3.9   2.62  16.5     0     1     4
## 2 Mazda RX4 Wag   21       6   160   110  3.9   2.88  17.0     0     1     4
## 3 Datsun 710      22.8     4   108    93  3.85  2.32  18.6     1     1     4
```

5.4.3 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", "20

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.

5.5 Tools to manipulate data

5.5.1 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()`.

5.5.2 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"]].

5.5.3 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 the table below.

Table 5.1: 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
```

5.5.4 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.

It evaluates one of several expressions based on the value of a given control expression. It takes two arguments: the control expression and a list of named expressions, each of which is evaluated if its name matches the value of the control expression. If no match is found, an optional default expression is evaluated. The switch() function is particularly useful

5.6 Data manipulation with dplyr

Figure 5.5: The logo of the dplyr package

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()`).
- 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, 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,
```

Exercise 5.1. Generate and drop variables

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.

5.7 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   M
```

```
##  1st Qu.:2    1st Qu.:1.925   Class :character   green :1    FALSE:1         1
##  Median :3    Median :3.300   Mode  :character   red   :2    TRUE :4         M
##  Mean   :3    Mean   :3.300                      yellow:1                    M
##  3rd Qu.:4    3rd Qu.:4.675                                                  3
##  Max.   :5    Max.   :5.500                                                  M
##               NA's   :1
```

```
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")
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 2
##    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
```

Exercise 5.2. 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 5.3. 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 5.4. 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 5.5. DatasauRus

Figure 5.6: The logo of the DatasauRus package[4]

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.

[4]Figure is taken from https://github.com/jumpingrivers/datasauRus

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 5.6. 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 5.7. 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
```

A sketch of a solutions is provided here.

Exercise 5.8. 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
```

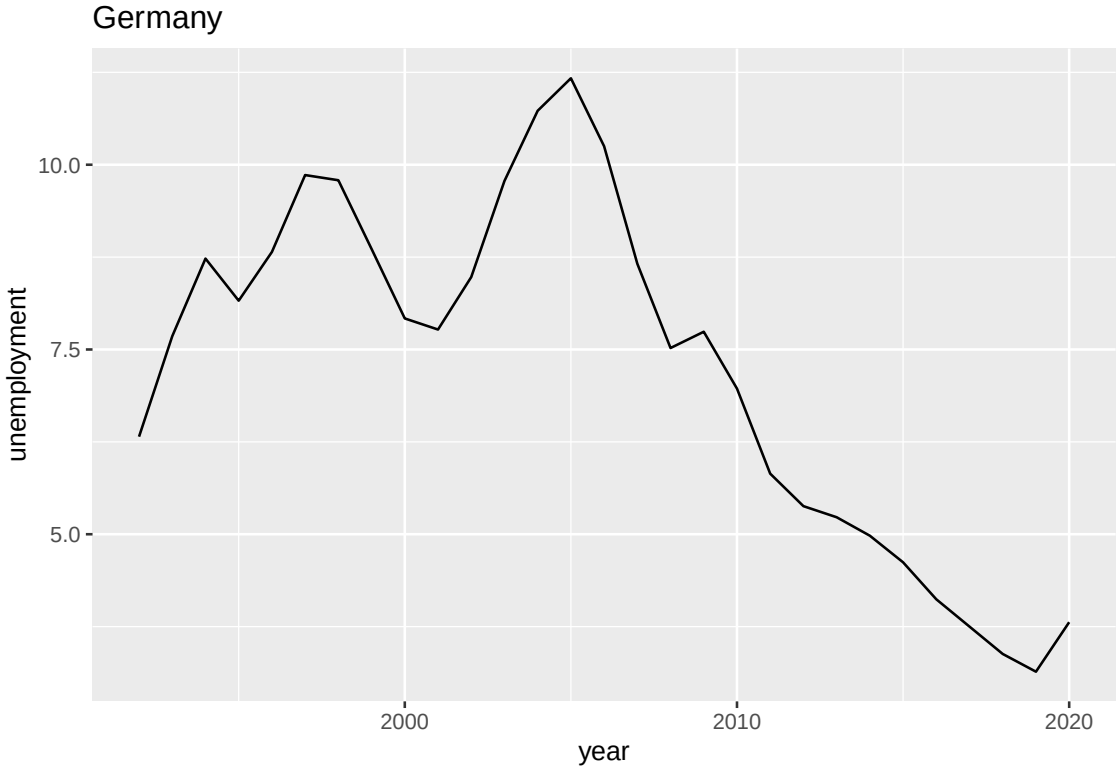
(19) In statistics, the coefficient of variation (COV) is a standardized measure of dispersion. It is defined as the ratio of the standard deviation (σ) to the mean (μ): $COV = \frac{\sigma}{\mu}$. Write down the R code to calculate the coefficient of variation (COV) for the **unemployment rate** in 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
```

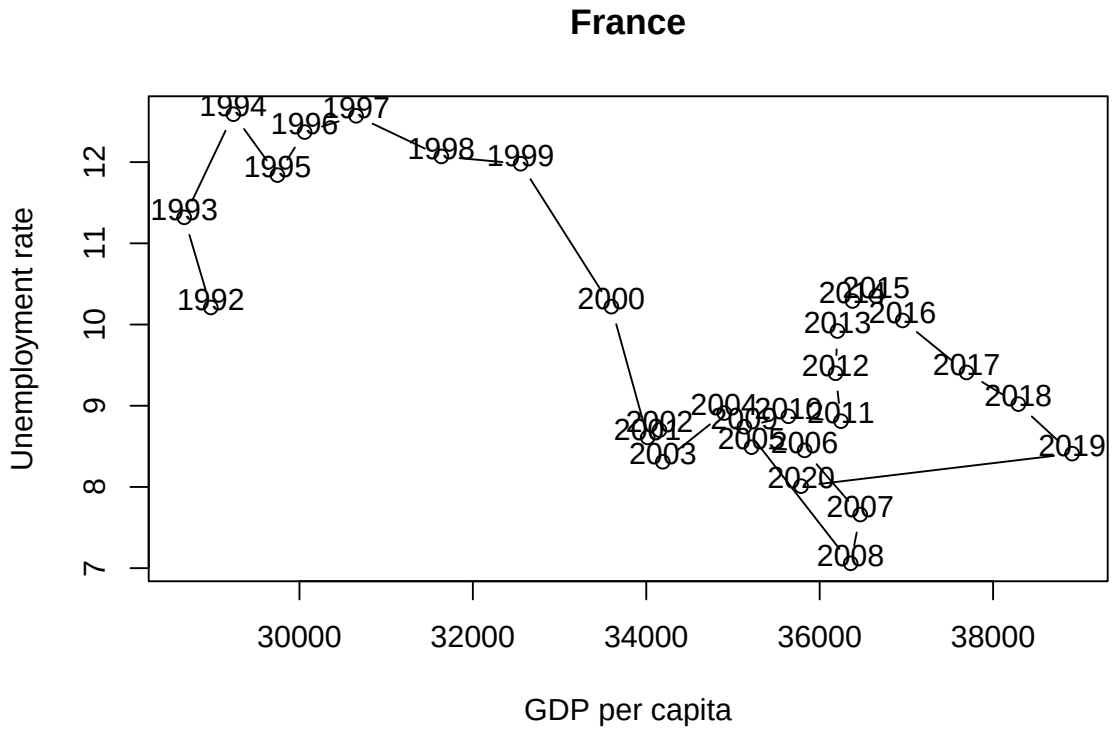
(20) Write down the R code to calculate the coefficient of variation (COV) for the **GDP per capita** in Germany and France. (Hint: See below for what your result should look like.)

```
## # 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 possible examples of what your result may look like).



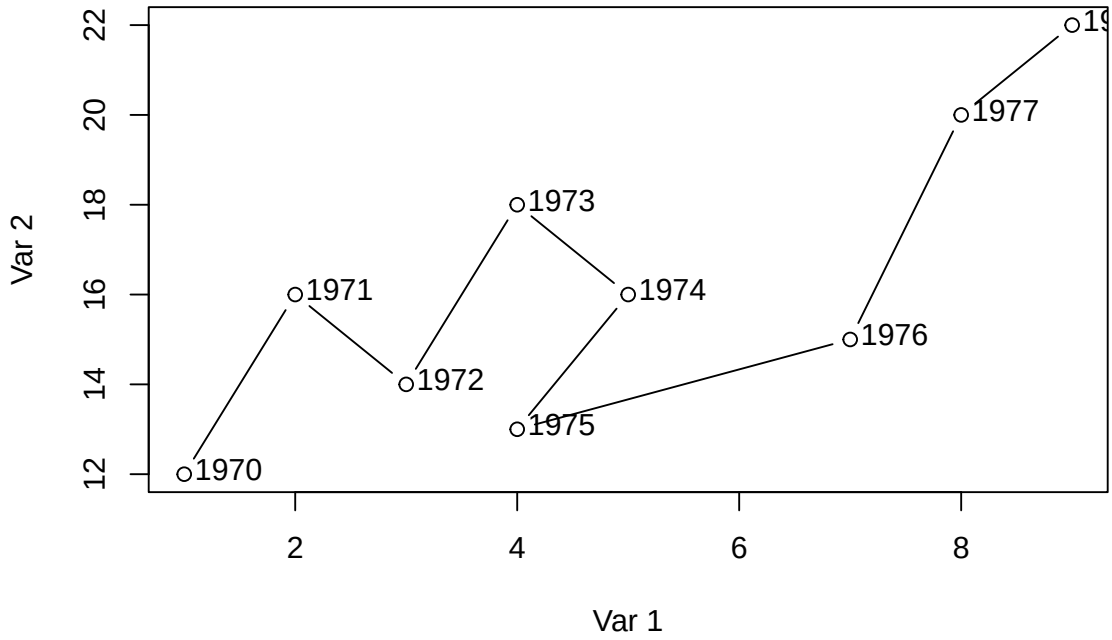
(22) and 23. *(This task is worth 10 points)* The following chart shows the simultaneous development of the unemployment rate and GDP per capita over time for **France**.



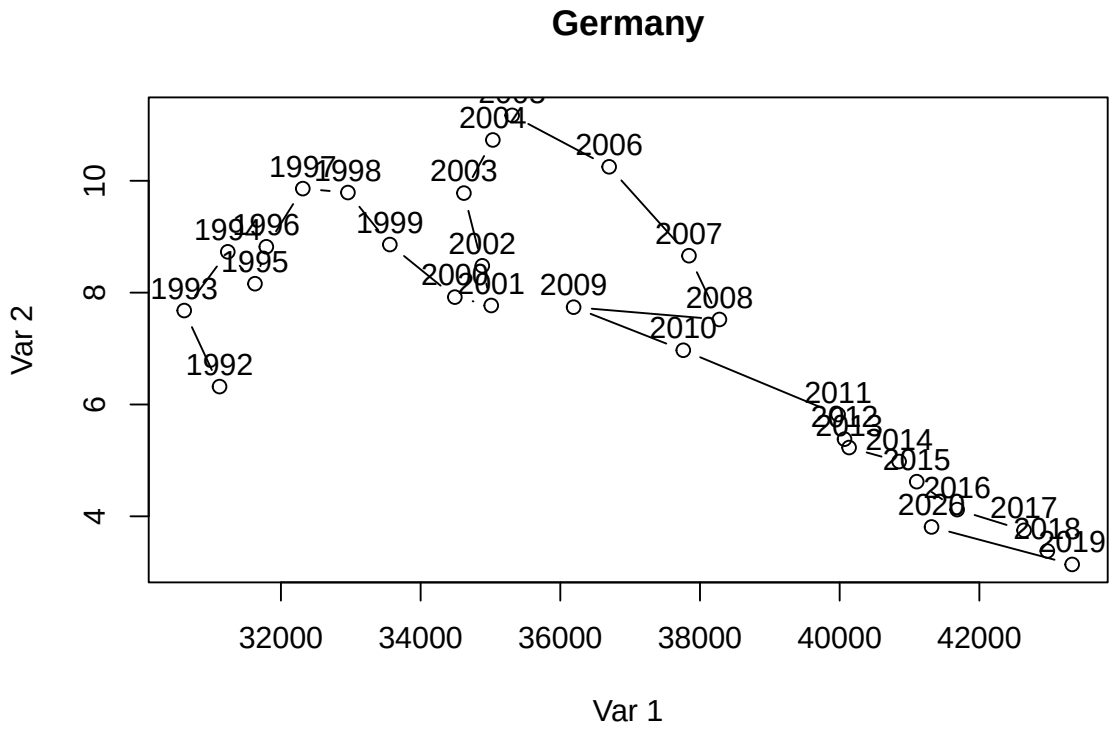
Suppose you want to visualize the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany as well.
Suppose further that you have found the following lines of code that create the kind of chart you are looking for.

```
# 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, la
```



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 un-

employment 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.

Chapter 6

Write with R Markdown

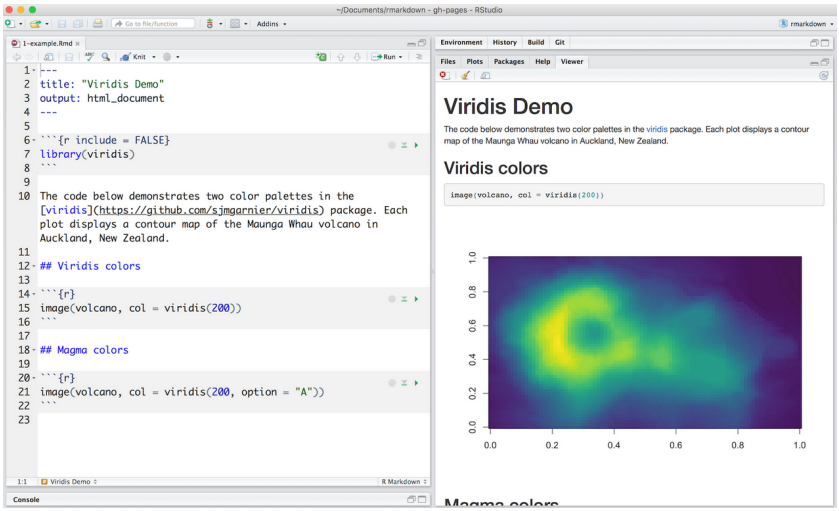


Figure 6.1: Example of an R Markdown file

R Markdown provides an authoring framework for data science. You can use a single R Markdown file to transcript your work, run code, and generate high quality reports, books, websites, articles, theses, blogs, and many more.

In contrast to Quarto (see: https://quarto.org and https://quarto.org/docs/get-started/hello/rstudio.html), which is the more recent format, R Markdown is around for some time and hence there are uncountable resources to learn it. For example:

- The R Markdown Cheatsheet from posit offers an overview on the most important features of R Markdown.

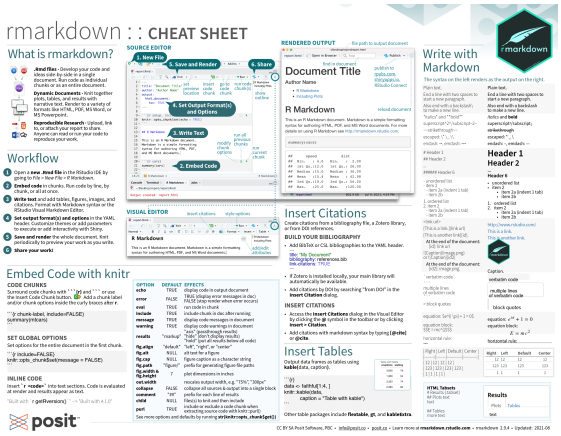


Figure 6.2: R Markdown Cheatsheet from posit

- The book *R Markdown Cookbook* by Xie, Dervieux, and Riederer (2020) offers an introduction. The online version of the book is regularly updated and free of costs.

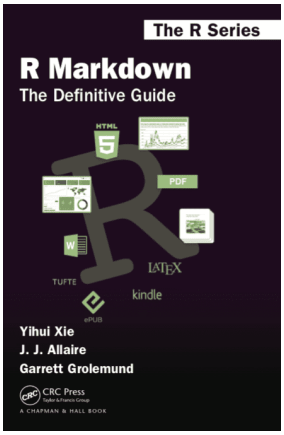


Figure 6.3: Xie et al. (2020): R Markdown Cookbook

- The book *R Markdown: The Definitive Guide* by Xie, Allaire, and Grolemund (2018) offers a comprehensive introduction. The online version of the book is regularly updated and free of costs.

Please watch the video What is R Markdown? and then study the R Markdown tutorial from RStudio.

Exercise 6.1. Start Markdown and R Markdown

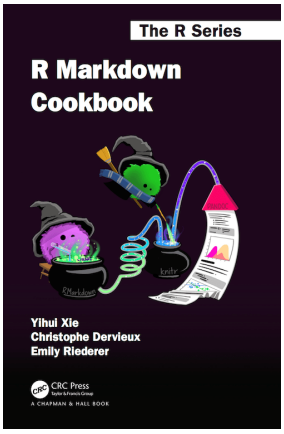


Figure 6.4: Xie et al. (2018): R Markdown: The Definitive Guide

a) You can learn Markdown (not R Markdown!) in 10 minutes. Just go to https://www.markdowntutorial.com and work throught the interactive lessons.
b) Now create your first R Markdown file in 3 minutes by doing the following:

- click in RStudio on *File* > *New File* > *R Markdown*
- click *OK*
- look for a button entitled *Knit* and click it
- save your file (it will be saved with .Rmd file extension)

c) Play around with the file. For example, change the output format can you create a word file or a presentation. Play around with the code chunks. Add a picture that you find somewhere online.
d) Set your working directory to the folder where you have saved your first Rmd-file. Can you come up with a way to generate different output format with just one function.

Exercise 6.2. R Markdown template

Please follow the instructions below to access the file "23-04_ds-project-desc.Rmd" from my GitHub account:

1. Download the file from my GitHub account by clicking on the link provided here.
2. Save the file in your working directory.
3. Use the knit function to run the file, but be aware that it may not work properly at first. If you encounter any issues, troubleshooting may be required. Don't worry, error messages will usually provide guidance to help you resolve the issue. Please note that the YAML header is sensitive to spacing, so be careful when setting it up to avoid breaking the code.

4. In the project template, I have used BibTeX to cite literature. This method is excellent for automating tedious tasks such as citing papers and generating reference lists based on citation styles, saving time and reducing the likelihood of citation errors. The literature cited is in a separate file, which can be found on one of my GitHub repositories.

Chapter 7

Appendix

7.1 Helpful shortcuts

Table 7.1: Table 1: Different OS, different keys

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

Table 7.2: 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.

Action	Shortcut Keys	Description
	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.

7.2 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.

7.2.1 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.

7.2.2 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.pdffile on a Mac would be ``` ~\Rbook\rcourse-book.pdf ``` You can find out your home directory with thepath.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/23-ss/ds"
```

7.2.3 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 7.2.5. Let's check the current active working directory:

```
getwd()
```

```
## [1] "/home/sthu/Dropbox/hsf/23-ss/ds"
```

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()
```

7.2.4 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.

7.2.5 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" )
```

7.3 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 4.1.2 (2021-11-01)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 11 (bullseye)
##
## 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-r0.3.13.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] dplyr_1.1.2      sjPlot_2.8.14    ggpubr_0.6.0     haven_2.5.2      datas
## [11] purrr_1.0.1      readr_2.1.4      tidyr_1.3.0      tibble_3.2.1     ggplo
##
## loaded via a namespace (and not attached):
##  [1] nlme_3.1-162       insight_0.19.1     httr_1.4.5         tools_4.1.2
## [10] colorspace_2.1-0   withr_2.5.0        tidyselect_1.2.0   emmeans_1.8.5
## [19] labeling_0.4.2     bookdown_0.33.3    bayestestR_0.13.1  sass_0.4.6
## [28] pkgconfig_2.0.3    htmltools_0.5.5    lme4_1.1-33        fastmap_1.1.1
## [37] generics_0.1.3     jsonlite_1.8.4     car_3.1-2          magrittr_2.0.3
## [46] lifecycle_1.0.3    stringi_1.7.12     yaml_2.3.7         snakecase_0.11.0
## [55] lattice_0.21-8     ggeffects_1.2.2    cowplot_1.1.1      splines_4.1.2
## [64] estimability_1.4.1 ggsignif_0.6.4     glue_1.6.2         evaluate_0.21
## [73] cachem_1.0.8       xfun_0.39          xtable_1.8-4       broom_1.0.4
```

References

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

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

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/.

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. https://github.com/muschellij2/intro_to_r.

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

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*. This manual is for r, version 4.1.3 (2022-03-10) edition, 2022. 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). Accessed January 30, 2023, 2023. URL https://r4ds.hadley.nz/.

Yihui Xie, Joseph J. Allaire, and Garrett Grolemund. *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, 2018.

Yihui Xie, Christophe Dervieux, and Emily Riederer. *R Markdown Cookbook*.

Chapman and Hall/CRC, 2020. available at https://bookdown.org/yihui/rmarkdown-cookbook.