How to Use R for Data Science

Lecture Notes

Prof. Dr. Stephan Huber

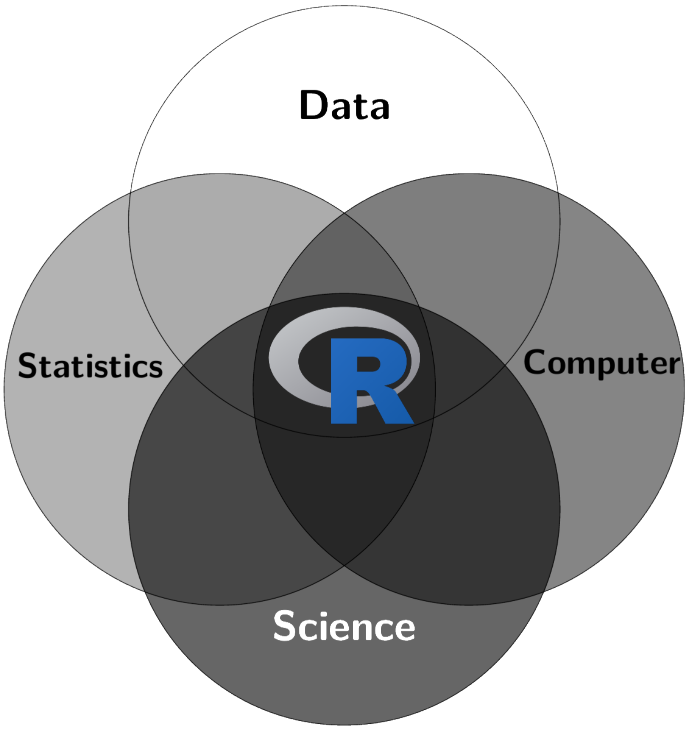
June 18, 2024

Table of contents

List of figures

List of tables

Preface



About R

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

About the cover of the notes

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

About the notes

> A PDF version of these notes is available here.
>
> Please note that while the PDF contains the same content, it has not been optimized for PDF format. Therefore, some parts may not appear as intended.

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

> 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.

Structure of these notes

Chapter	Explanations
…R	Learn the basics everyone should know about R and RStudio, including how to install them.
…writing code	Learn the basics of writing code.
writing R scripts	Learn how to use R scripts and their benefits.
Interactive introduction using swirl	A hands-on tutorial on how to use the swirl package. This section is optional.
Kickstart	A quick start guide for beginners on how to dive into R, showcasing some of its capabilities.
Pitfalls	Discover common mistakes beginners often make and how to avoid them to save time on troubleshooting.
Manage data	Learn how to manipulate data in R.
Visualize data	A quick guide on where to find resources to learn about creating graphical visualizations in R.
Collection of exercises	A set of exercises to practice R programming skills.

Chapter	Explanations
Appendix	A set of useful stuff that will help you to navigate through your file system, find the right operator and function, or to learn some useful shortcuts.

About the author

Contact:

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

Figure 1.: Prof. Dr. Stephan Huber

I am a Professor of *International Economics and Data Science* at HS Fresenius, holding a Diploma in Economics from the University of Regensburg and a Doctoral Degree (summa cum laude) from the University of Trier. I completed postgraduate studies at the Interdisciplinary Graduate Center of Excellence at the Institute for Labor Law and Industrial Relations in the European Union (IAAEU) in Trier. Prior to my current position, I worked as a research assistant to Prof. Dr. Dr. h.c. Joachim Möller at the University of Regensburg, a post-doc at the Leibniz Institute for East and Southeast European Studies (IOS) in Regensburg, and a freelancer at Charles University in Prague.

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

I was always fascinated by data and statistics. For example, in 1992 I could name all soccer players in Germany's first division including how many goals they scored. Later, in 2003 I joined the introductory statistics course of Daniel Rösch. I learned among others that probabilities often play a role when analyzing data. I continued my data science journey with Harry Haupt's *Introductory Econometrics* course, where I studied the infamous Jeffrey M. Wooldridge [2002] textbook. It got me hooked and so I took all the courses Rolf Tschernig offered at his chair of

Econometrics, where I became a tutor at the University of Regensburg and a research assistant of Joachim Möller. Despite everything we did had to do with how to make sense out of data, we never actually used the term *data science* which is also absent in the more 850 pages long textbook by Wooldridge [2002]. The book also remains silent about *machine learning* or *artificial intelligence*. These terms became popular only after I graduated. The *Harvard Business Review* article by Davenport and Patil [2012] who claimed that data scientist is "The Sexiest Job of the 21st Century" may have boosted the popularity.

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

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

Part I.

Getting started with...

1. ...R

Learning Objectives

- Understand the reasons for choosing R as a programming language.
- Learn effective strategies and resources for mastering R programming.
- Learn the basic priciples of R.
- Distinguish between R and RStudio.
- Demonstrate how to write and execute code in R and RStudio.
- Learn how to install R, RStudio, and R packages.
- Explore options to use R and RStudio without installing them on your machine, such as through cloud-based platforms.

Content

1.1. Why R?

R is an open-source programming language that allows to analyse and manipulate data, create state-of-the-art graphics, and many more. It supports larger data sets, reads any type of data, and runs on multiple platforms (Windows, Mac, Linux) and CPU architectures (x86_64, arm64). R makes it easier to automate tasks, organize projects, ensure reproducibility, and find and fix errors, and anyone can contribute packages to improve its functionality. Moreover, the following points are worth to emphasize:

- **R is an artist!** Check out:
  - The R Graph Gallery
  - R CHARTS by R CODER
- **R is an employment insurance!** Programming is a core skill in research, economics, and business. If you can write code, you have plenty of opportunities to earn a decent salary. R is one of the most widely used programming languages in the world today. It is used in almost every industry such as finance, banking, medicine or manufacturing. R is

used for portfolio management, risk analytics in finance and banking industries. Even if you need to learn a new programming language later, knowing R makes it much easier to pick up another one.

- **R uses the computer and computers are great!** Doing statistics on a computer is faster, easier and more powerful than doing it by hand. Computers are an extension to your brain and can do repetitive tasks better and faster without making logical errors. The only reason to do statistical calculations with pencil and paper is for learning purposes.
- **Low-code and no-code applications such as Excel are limited!** Using spreadsheets software like Microsoft Excel for research can be problematic. It's easy to lose track of operations, making the process difficult to oversee and document. Command-line programs are maybe not as easy to learn but offer a more straightforward approach that allows the results to be replicated easily.
- **R is open source!** Proprietary software expansive, support can only be provided by the copyright owner which means the software expires and you can't do anything against it. Moreover, security issues cannot be checked as the source code is not available, and possibilities for customization are limited. R is yours and everybody can contribute to its success.
- **R is big!** When you download and install R, you get some basic packages, that contain functions that allow you to do already a lot of things. Beyond that, you can write your own packages or install user-written packages that extend your possibilities. With over 20,684 packages on the CRAN repository and many more available on GitHub and other platforms, R's extensive library supports a wide variety of data science tasks. Its widespread use and open-source availability have cemented R as a standard tool in data science and ensured that there are multiple approaches to most data handling processes. These can be easily adopted.

R has weaknesses

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

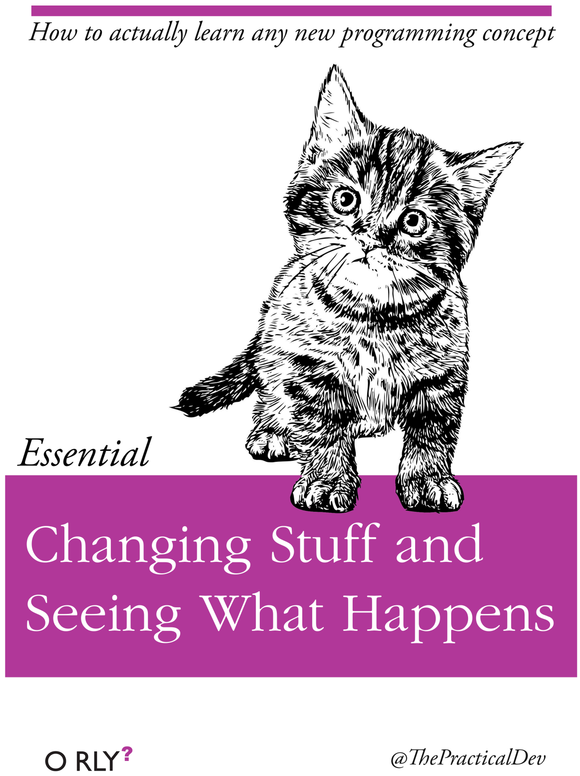
1.2. How to learn R

There are many different approaches to learning R. It pretty much depends on your preferences, needs, goals, prerequisites and limitations. It is up to you to search and find a suitable way to achieve your learning goals. While I hope you find my notes helpful, I additionally provide in section Section 1.3 a list of other resources that are worth considering. To start with, I recommend my swirl courses that provide an interactive learning environment, see Chapter 4.

Make your hands dirty!

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

Figure 1.1.: Play around with code



Source: *DEV Community on GitHub*

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

- **Computers need clear and precise instructions to work:** They can't handle mistakes or unclear directions. They are actually sort of stupid as they do not have an intuition. They just take you literally. Even small errors like a missing comma or an unclosed bracket can cause your code to not work. Computers do exactly what you tell them, no more and no less.

Computers take you literally

Let me illustrate what I mean: Suppose you send your grandfather the following message:

> "Let's eat grandpa."

He will probably understand that you're inviting him to dinner. However, if you sent the same message to a computer, it would interpret the sentence literally due to the missing comma:

> "Let's eat, grandpa."

> The comma makes all the difference in clarifying that you're speaking to your grandpa, not about eating him! Similarly, in programming, an incorrectly placed comma can break your code or change the meaning of your code.

- **Copy, paste, and tweak**: While learning code from scratch is sometimes essential, you can speed up your work by modifying code that already exists. I call this the *"copy, paste, and tweak"* approach. While this is not the only way to learn code, it gets a job done quick, and it is fun, see Figure 1.1.
- **Have a purpose when coding**: Rather than learning to code for its own sake, it is more fun and you'll probably learn faster when you have a goal in mind. Try to analyze data that you are interested in. Another good exercise is replicating a research paper.
- **Practice is key**: The best method to improving your coding skills is through lots of practice. Consequently, these notes give you plenty of exercises.
- **Use ChatGPT**: The usage of supporting tools is not forbidden. ChatGPT can help you to understand code and brainstorm solutions. However, it's important to know that ChatGPT might suggest complex methods when there are shorter and more elegant solutions available Absolute beginners might find ChatGPT's solutions overwhelming and have difficulties to tweak the proposed sketch of a solution. So, use it thoughtfully.

1.3. Learning resources

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

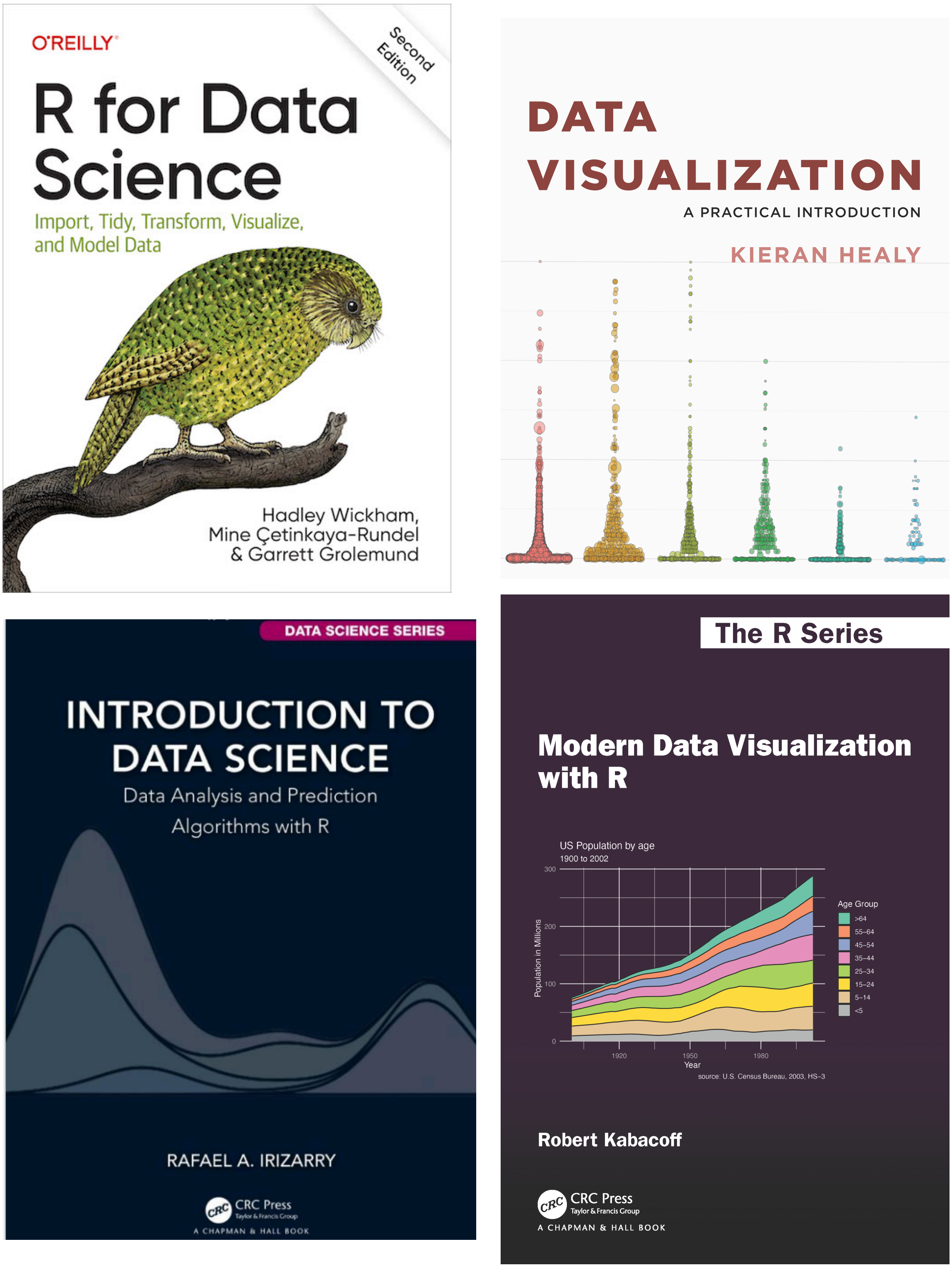
In RStudio you find in the right side at the bottom a panel that is called *Help*. There you find a lot of links, manuals, and references that offer you tons of resources to learn R for free including: education.rstudio.com and Links for Getting Help with R. At the top right of RStudio you find a panel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

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

1. 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.
2. Healy [2018]: Data Visualization: A Practical Introduction is a hands-on introduction to the principles and practice of looking at and presenting data using R and `ggplot`.
3. Irizarry [2022]: Introduction to Data Science: Data Analysis and Prediction Algorithms With R is a complete, up to date, and applied introduction.
4. Venables et al. [2022] An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics is a manual from the R Core Development Team that shows how to use R without having to install and load additional packages.
5. Neth [2023]: Data Science for Psychologists is a comprehensive introduction to R and data science for non experts of both programming and data science. It uses a variety of data types and includes many examples and exercises.
6. Kabacoff [2024]: Modern Data Visualization with R teaches how to create graphs from scratch providing a lot of examples that you can copy, paste and tweak.

Some other sources that are worth mentioning are these:

Figure 1.2.: A collection of textbooks



- 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 is a function in R?

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

> "Everything that happens is a function call."

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

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

If you want to specify what exactly you want R to do for you, you need to refer to the arguments of a function. For example, if you don't want to be asked interactively what you want to do with your workspace (this is the place where you store all your objects, see section Section 1.5), you can do this with an argument that is part of the `q()` function:

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

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

```
?q()
```

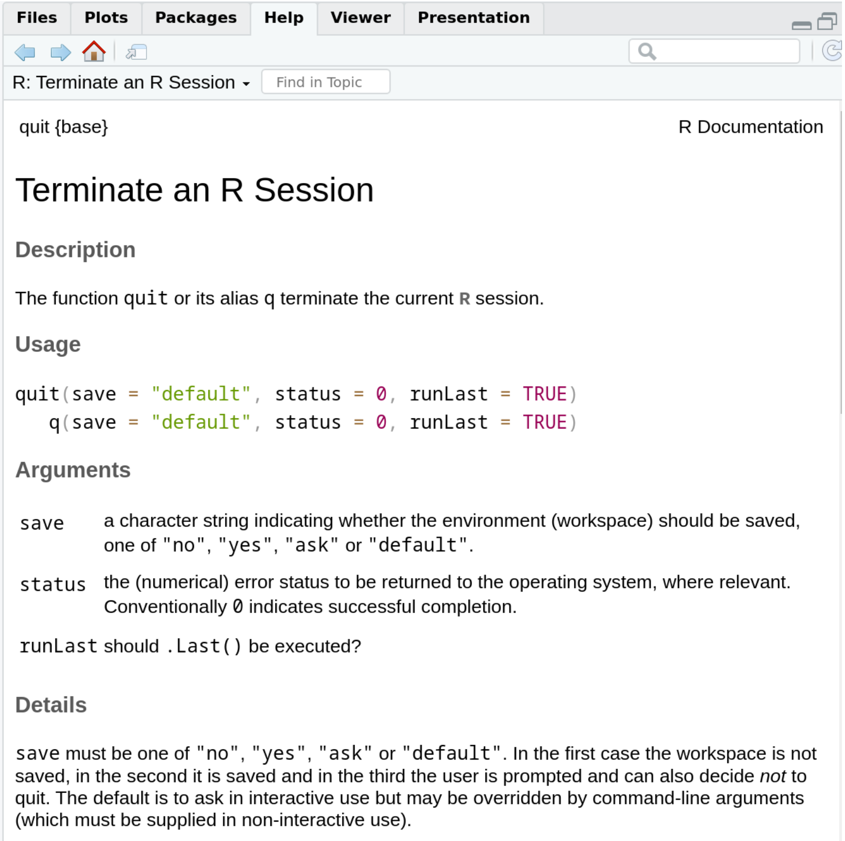
Unfortunately, the documentation can sometimes be a bit confusing for beginners in applied contexts. However, the documentation for all functions is structured similarly, typically featuring several key sections:

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

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

An excerpt of the *R Documentation* for the function `q()` is shown in Figure 1.3. Here, we observe that the function has three arguments that you can manipulate. If you do not specify any of these arguments explicitly, we see that *by default*, R sets the three arguments as shown.

Figure 1.3.: The R Documentation of `q()`



1.5. What are objects in R?

R is an object oriented programming language. That means,

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

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

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

1.6. What are R and RStudio?

While R has a command line interface, there are multiple third-party graphical user interfaces available that improve the user experience a lot. The most successful graphical user interface or integrated development environment (IDE) is RStudio. Throughout this book, I will assume that you are using R via RStudio. First time users often confuse the two. At its simplest, R is like a car’s engine while RStudio is like a car’s dashboard as illustrated in Figure Figure 1.4.

More precisely, R is a functional 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

Figure 1.4.: Analogy of difference between R and RStudio

R: Engine	RStudio: Dashboard

mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

Much as we don't drive a car by interacting directly with the engine but rather by interacting with elements on the car's dashboard, we won't be using R directly but rather we will use RStudio's interface. After you install R and RStudio on your computer, you'll have two new *programs* (also called *applications*) you can open. We'll always work in RStudio and not in the R application. Figure Figure 1.5 shows what icon you should be clicking on your computer.

Figure 1.5.: Icons of R versus RStudio on your computer

R: Do not open this	RStudio: Open this

After you open RStudio, you should see something similar to Figure Figure 1.6 where three or four panels dividing the screen.

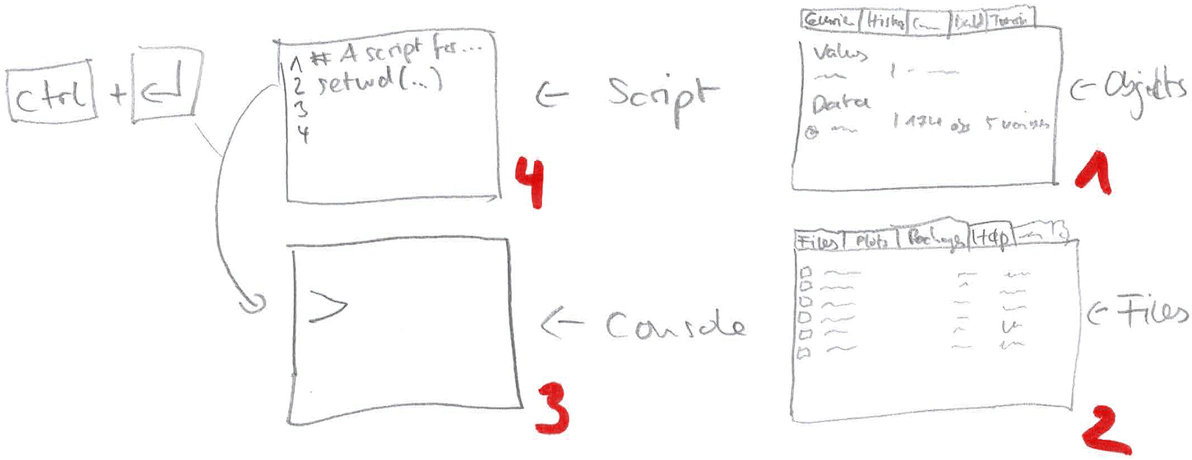
1. The *Environment* panel, where a list of all objects is shown.
2. The *Files*, *Plots* and *Help* panel, allow you to manage files, preview plots, and find help for different functions of R.
3. The *Console* panel, used for running code.
4. The *Script* panel, used for writing code.

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).

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

Figure 1.6.: A sketch of RStudio interface to R




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

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

 Natural language support but running in an English locale

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

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

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

1.7. How to write and run code in R and RStudio

In the Console you can type in code and push Enter to run the line of code. For example, you can calculate:

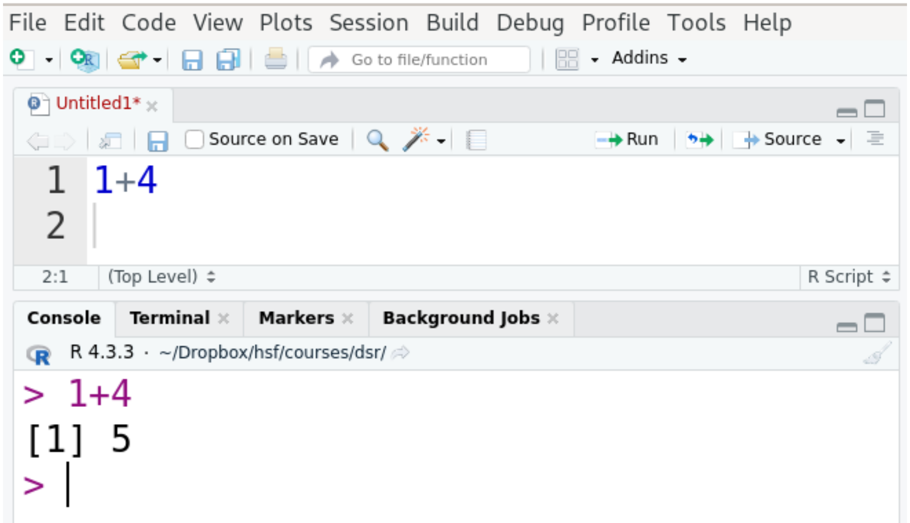
```
1+4
```

```
[1] 5
```

While working in the Console is possible, we usually work in RStudio using so-called *scripts*. These scripts are plain text files with the file extension ".R". Scripts are discussed in detail in Chapter 3. To create a script, go to the *File* menu, select *New File* and then choose *R Script*.

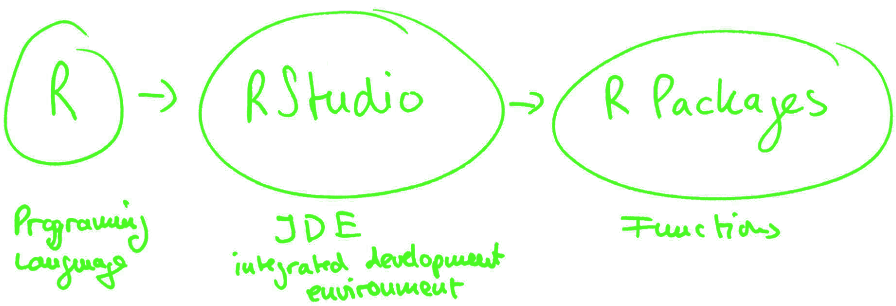
With the key shortcut `Ctrl+Enter` for Windows and Linux user or by `Cmd+Enter` for MacOs users (or by clicking `Run`) you can run a line of a script, that means you send one line of code to the Console. See Figure 1.7 how this looks like in RStudio.

Figure 1.7.: One plus four in a R script



1.8. How to install R, RStudio, and R packages

Figure 1.8.: Set up R in three steps



As shown in Figure 1.8, setting up R on your personal computer (Windows, Mac, Linux) is a three step process: You will first need to download and install R. After that has been successful you can download and install RStudio. Please note that it is important that you install R first and then install RStudio. As a third but optional step you can install R packages.

1. **Do this firstly:** Download and install R here.
    - If you are a Windows user: Click on "Download R for Windows", then click on "base", then click on the Download link.
    - If you are macOS user: Click on "Download R for (Mac) OS X", then under "Latest release:" click on R-X.X.X.pkg, where R-X.X.X is the version number. For example, the latest version of R as of March 29, 2024 was R-4.3.3.
    - If you are a Linux user: Click on "Download R for Linux" and choose your distribution for more information on installing R for your setup.
2. **Do this secondly:** Download and install RStudio here.

- 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.

3. **Do this thirdly:** Install R packages. This step is optionally as you can install R packages at any time. However, it may be a good idea to install frequently used packages in one take because the installation of some packages can be time consuming. Therefore, I recommend to read Section 1.9 and follow the instructions therein.

How to use R and RStudio without installation

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

1.9. What are R packages?

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

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.

Package management

1. A package must be installed before it can be loaded.
2. A package must be loaded before it can be used.

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

1.9.1. Package installation

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

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

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

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

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

1.9.2. Package loading

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

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

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

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

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

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

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

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

1.9.3. Simplified package management with `p_load`

I recommend to install and load packages using the `p_load()` function of the `pacman` package. It is superior because

- it only installs a package if it is has not been installed yet,
- it loads the package, and
- does not require quotes nor the `c()`function.

For example, instead of the traditional approach:

```
install.packages(
  c("tidyverse", "janitor", "haven", "readxl")
  )
library(
  c("tidyverse", "janitor", "haven", "readxl")
  )
```

You can streamline the process as follows:

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, janitor, haven, readxl)
```

The line `if (!require(pacman)) install.packages("pacman")` ensures the installation of the `pacman` package, which is necessary for using the `p_load` function.

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

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

to avoid conflicts of functions (see Section 7.5).

Tip 1: Install everything now

Throughout the lecture notes and in the exercises, I will use different packages. The installation can be time consuming and hence I recommend to install all packages by running the following lines of code in the Console. This takes some minutes depending on your PC and your internet connection. However, after installing all these packages you have all packages that are used in my exercises, my lecture notes *How to Use R for Data Science*, and the book *R for Data Science (2e)* by Wickham and Grolemund [2023].

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load(
      arrow, babynames, car, curl, devtools, dplyr, duckdb, devtools,
      expss, gapminder, ggplot2, ggrepel, ggridges, ggpubr,
      ggstats, ggthemes, haven, HH, janitor, kableExtra, knitr,
      Lahman, labelled, likert, magick, maps, MASS, nycflights13,
      openxlsx, palmerpenguins, papaja, plm, psych,
      remotes, rempsyc, repurrrsive, rstatix, skimr, sjlabelled,
      sjmisc, sjPlot, stargazer, texreg, tidymodels, tidyr,
      tidyverse, tinylabels, usethis, WDI, wbstats, writexl
)
```

In addition to these packages, I recommend to install a package that I created to offer you some tutorials and functions. I host this package on my GitHub account and you can install it as follows:

```
devtools::install_github("hubchev/hubchev")
```

1.10. Base R and the tidyverse universe

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

Figure 1.9.: The tidyverse universe

The R package `tidyverse` (see Figure 1.9) is a comprehensive collection of R packages including popular packages such are `ggplot2`, `dplyr`, `tidyr`, `readr`, `purrr`, `tibble`, `stringr`, and `forcats`, which together offer extensive capabilities for data modeling, transformation, and visualization.

How to do data science with tidyverse is the subject of multiple books and tutorials. In particular, the popular book *R for Data Science* by Wickham and Grolemund [2023] is all about the tidyverse universe. Thus, I highly recommend reading sections Workflow: basics), Data transformation, and Data tidying. Additionally, explore www.tidyverse.org for more resources, and consider completing the *tidyverse* module in my `swirl` package, *swirl-it*, as detailed in section Chapter 4.

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

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse)
```

Exercise 1.1. Set up R, RStudio, and R packages
Open this interactive tutorial and work through it.

2. ...writing code

When you talk to a computer, you use a programming language. Computers can handle certain tasks for you, like doing math or creating graphs. They don't complain and work fast. They just need clear instructions. Giving clear instructions to a machine, however, isn't easy. Unlike humans, computers lack intuition and cannot adapt to the context of your commands; they interpret instructions literally. Check out this video. It'll show you that good communication among human beings isn't about explaining things perfectly, but about using your gut feeling and common sense effectively.

If you want a computer to handle computationally intensive and repetitive tasks, you must learn to "speak" in a way that the computer understands. Never assume the computer knows what you mean. The following sections will emphasize this: Be precise and be specific.

2.1. Bake a cake

As a teacher, I often contemplate how I can teach students to communicate with a computer. In this section, I will attempt to translate a cake baking recipe into a programming language, using R. I hope you find this both amusing and insightful. Let's start by examining a simplified cake recipe:

Instructions to bake a cake:

1. Buy an oven.
2. Buy ingredients (flour, sugar, butter, eggs, soda).
3. Buy tools.
4. Clean everything..

5. Heat up the oven.
6. Prepare the tools (springform, bowl, mixer).
7. Weight all ingredients.
8. Take a bow and all the weighted ingredients and put everything in a bowl.
9. Take the mixer and mix all ingredients in the bowl for 3 minutes.
10. Put all the mixed ingredients in a springform pan.
11. Take the springform pan with the mixed ingredients and put it in the oven.
12. Bake for 30 minutes, take the springform pan it out of the oven, and turn off the oven.
13. Clean the kitchen and the tools.

Although this recipe is simplified, it illustrates a process you might be familiar with. Now, let's assume a computer is tasked with baking a cake. How would we explain the necessary steps to the computer using the R programming language?

In R, a functional programming language, we understand that everything that happens is a function call, and every entity is an object. Therefore, we must translate all actions into functions and all items into objects.

Here's what the translated recipe could look like in R:

```
buy(oven, springform, bowl, mixer, flour, sugar, butter, eggs, soda)
clean(oven, springform, bowl, mixer)
turn_on(oven)
prepare(springform, bowl, mixer)
weigh(flour, sugar, butter, eggs, soda)
dough <- bowl |>
  put(flour, sugar, butter, eggs, soda) |>
  action(tool = mixer, time = 3)

dough_springform <- springform |>
  put(dough) |>

dough_oven <- oven |>
  put(dough_springform) |>
  action(tool = oven, time = 30) |>
  pull()

turn_of(oven)
clean(oven, springform, bowl, mixer)
```

Understanding the translation of a recipe into code becomes clearer when we familiarize ourselves with two key programming operators:

1. The "`<-`" is known as the *assignment operator*. It saves or stores data into a new object. It might be helpful to think of it as saying, "I create the object *`<name of object>`* and store therein"
2. The "`|>`" is known as the *pipe operator*. It passes the output of one action to serve as the input for the next. Think of it as saying "and then."

For example, the following lines:

```
dough <- bowl |>
  put(flour, sugar, butter, eggs, soda) |>
  action(tool = mixer, time = 3)
```

can be interpreted as:

```
I create the object `dough` and I store therein the bowl, and then
  I put flour, sugar, butter, eggs, and soda to it, and then
  I take action with the mixer for 3 minutes
```

In the preceding functions, you'll notice objects separated by commas and parameters like `tool = mixer, time = 3`. These parameters define the behavior of the function. When there's nothing within the brackets, as in `pull()`, the input is merely the output of the preceding pipe operator.

Even though R is no good as a cook and the recipe is missing some steps, this analogy helps to illustrate how programming languages work: they allow us to instruct the computer in a sequential way. Next, I will showcase why coding is appealing.

2.2. Elegant code

Let's make our code more *elegant*, that is, easy to read, understand, and modify. For example, while it is equivalent to write everything in one line

```
dough <- bowl |> put(flour, sugar, butter, eggs, soda) |> action(tool = mixer, time = 3)
```

or spread out over three lines,

```
dough <- bowl |>
  put(flour, sugar, butter, eggs, soda) |>
  action(tool = mixer, time = 3)
```

it is easier for the human eye to read the text in spread out form.

> Style in Writing Code
>
> Writing code involves certain conventions, often referred to as a *coding style*. Although not strictly necessary, a consistent style can significantly enhance clarity and prevent common pitfalls. Numerous style guides aim to standardize coding practices. For example, you might find *The tidyverse style guide* by Hadley Wickham particularly helpful in adopting a harmonious coding style in R.

By using the assignment operator `<-`, we can create two objects: `ingredients` and `tools`. These objects are used multiple times throughout the process.

Here is an improved version of the script:

```
ingredients <- c(flour, sugar, butter, eggs, soda)
tools <- c(oven, springform, bowl, mixer)

buy(tools, ingredients)

clean(tools)
turn_on(oven)
prepare(tools)
weight(ingredients)
dough <- bowl |>
  put(ingredients) |>
  action(tool = mixer, time = 3)

dough_springform <- springform |>
  put(dough)

dough_oven <- oven |>
  put(dough_springform) |>
```

```
  action(tool = oven, time = 30) |>
  pull()

turn_of(oven)
clean(, tools)
```

This version refines the process, making the code more streamlined and easier to follow.

2.3. Bake a cheese cake

Now, let's assume you want to bake another cake, this time with chocolate and banana, but without eggs. Moreover, you need to bake it for 45 minutes. We can easily adapt the code snippet from above to accommodate the ingredients for this new recipe:

```
ingredients <- c(flour, sugar, butter, soda, banana, chocolate)
tools <- c(oven, springform, bowl, mixer)

buy(tools, ingredients)

clean(tools)
turn_on(oven)
prepare(tools)
weight(ingredients)
dough <- bowl |>
  put(ingredients) |>
  action(tool = mixer, time = 3)

dough_springform <- springform |>
  put(dough)

dough_oven <- oven |>
  put(dough_springform) |>
  action(tool = oven, time = 45) |>
  pull()

turn_of(oven)
clean(kitchen, tools)
```

2.4. Comment what you do

Sometimes code can be difficult to understand for humans. It is therefore helpful to add comments to clarify what the individual code sections are supposed to do. In R, comments can be added with a leading hashtag, `#`.

```
# Decide on tools and ingredients
ingredients <- c(flour, sugar, butter, soda, banana, chocolate)
tools <- c(oven, springform, bowl, mixer)
```

```
# Go shopping
buy(tools, ingredients)

# Prepare the kitchen, tools, and ingredients
clean(tools)
turn_on(oven)
prepare(tools)
weight(ingredients)

# Make the dough
dough <- bowl |>
  put(ingredients) |>
  action(tool = mixer, time = 3)
dough_springform <- springform |>
  put(dough) |>

# bake the cake
dough_oven <- oven |>
  put(dough_springform) |>
  action(tool = oven, time = 45) |>
  pull()

# Clean up
turn_of(oven)
clean(kitchen, tools)
```

2.5. Bake 10 cakes

As a computer can reproduce a cake within seconds (I mean, not really, just in my little fun exercise here), we now have the opportunity to experiment with several versions of the cake by varying the baking time from 35 to 45 minutes. Here's how the corresponding code might look:

```
ingredients <- c(flour, sugar, butter, soda, banana, chocolate)
tools <- c(springform, bowl, mixer)

buy(tools, ingredients)

clean(tools)
turn_on(oven)
prepare(tools)
weight(ingredients)
dough <- bowl |>
  put(ingredients) |>
  action(tool = mixer, time = 3)

dough_springform <- springform |>
  put(dough)
```

```
for (timing in 35:44) {
  dough_oven <- oven |>
    put(dough_springform) |>
    action(tool = oven, time = timing) |>
    pull()
  assign(paste("dough_oven_min_", timing, sep = ""), dough_oven)
}

turn_of(oven)
clean(tools)
```

You can see a loop with some new and tweaked lines:

```
for (timing in 35:44) {
  dough_oven <- oven |>
    put(dough_springform) |>
    action(tool = oven, time = timing) |>
    pull()
  assign(paste("dough_oven_min_", timing, sep = ""), dough_oven)
}
```

These lines sequentially execute the following actions:

```
Let the object timing be 35, make a cake, and save it in the object `dough_oven_min_35` the
let the object timing be 36, make a cake, and save it in the object `dough_oven_min_36` the
let the object timing be 37, make a cake, and save it in the object `dough_oven_min_37` the
let the object timing be 38, make a cake, and save it in the object `dough_oven_min_38` the
let the object timing be 39, make a cake, and save it in the object `dough_oven_min_39` the
let the object timing be 40, make a cake, and save it in the object `dough_oven_min_40` the
let the object timing be 41, make a cake, and save it in the object `dough_oven_min_41` the
let the object timing be 42, make a cake, and save it in the object `dough_oven_min_42` the
let the object timing be 43, make a cake, and save it in the object `dough_oven_min_43` the
let the object timing be 44, make a cake, and save it in the object `dough_oven_min_44` the
```

After all, we have ten cakes. This shows how we can harness the processing power of a computer. Computers are excellent at performing everyday, repetitive tasks so that we can automate processes and perform procedures effortlessly over and over again.

2.6. Writing real code

Of course, computers can't *bake a cake*. The R programming language can do none of the above. Nevertheless, there are analogies to the programming language R. Let me present a few lines of code and explain these lines of code to you, and you will see that the similarities are striking.

Copy that code chunk, paste it into a R script and run it.

```r
# This script demonstrates a typical data analysis workflow in R
# ---------------------------------------------------------------

# Install and load required libraries
if (!require(pacman)) install.packages("pacman")
pacman::p_unload(all)
pacman::p_load(tidyverse,haven, janitor)

# Set the working directory to a project-specific folder
setwd("~/Documents")

# Clear the current environment of any objects
rm(list = ls())

# Load data from a Stata file available online
auto <- read_dta("http://www.stata-press.com/data/r18/auto.dta")

# Display basic information about the dataset
ncol(auto) # Number of columns
nrow(auto) # Number of rows
dim(auto) # Dimensions of the dataset
names(auto) # Names of variables
head(auto) # First few rows
tail(auto) # Last few rows
summary(auto) # Summary statistics for each column
glimpse(auto) # Compact display of the structure of the dataset
print(auto, n = Inf) # Print all rows of the dataset

# Check for duplicate entries based on the 'make' variable
auto |>
  get_dupes(make)

# Create and display a scatter plot of car price versus weight
plot_weight_price <- ggplot(auto, aes(x = weight, y = price)) +
  geom_point()
plot_weight_price

# Save the plot to a file
ggsave("plot_weight_price.png", plot = plot_weight_price, dpi = 300)
```

3. ...writing R scripts

3.1. The limitations of no-code applications

No-Code Applications (NCA) such as Microsoft Excel, RapidMiner, KNIME, DataRobot, Tableau, Microsoft Power BI, and Google AutoML are popular for good reasons. They enable the application of advanced empirical methods with no or minimal programming effort. Their intuitive graphical user interfaces comes with pre-built templates and drag-and-drop functionality which helps to get things done quick, without having to study the programm documentation for hours. Despite their apparent ease of use and efficiency, these platforms come with several disadvantages compared to traditional ways of working with computer by scripting and coding. Understanding these weaknesses helps to see why professional researchers, especially those actively publishing in academic journals, tend to rely on scripting languages like R and Python. These programming languages offer full control, are customizeable and extendable, offer extensive opportunities for automation and reproducibility, and are often better suited for demanding data science tasks.

> **Disadvantages of No-Code Applications (NCA)**
>
> - NCA lack flexibility and are limited in their adaptability.
> - NCA often have problems scaling with increasing data volumes or user requirements.
> - NCA are often closed systems that make it difficult to integrate other systems or applications.
> - NCA often bind a company to a specific ecosystem. This can lead to dependencies and vendor lock-in.
> - NCA can obscure the underlying logic of how applications work, which can make troubleshooting more difficult.
> - NCA abstracts the coding process and prevents users from understanding fundamental concepts that could be beneficial to their professional growth and ability to tackle more complex problems.

In summary, while low-code and no-code platforms offer quick deployment and ease of use, they can lack the depth, flexibility, and control provided by traditional scripting. Researchers and businesses must consider these trade-offs, especially when planning for long-term scalability, complex customizations, or in-depth integrations.

> **A hypothetical but realistic example with Excel**
>
> Suppose you work with a spreadsheet software like Excel. You import a CSV file using the implemented import tool, you save the converted file. You notice that Excel has messed up the dates during the import, so you spend a few minutes cleanig that manually. Then, you visualize the data and you save the visualisations in various tabs. Maybe, you can spot some outliers and you document in footnote that these outliers are the result of some issues with the raw data. As these errors cannot be solved, you delete the observations and variables that contain a significant amount of errors. Finally, you use filters to calculate

some summary statistics. All your results are written in a new tab. You're convinced that you've done a great job. However, you send the file to your supervisor and you ask him for her opinion.

- She probably asks you what you have done to the data. How can you efficiently and completely communicate that?
- She comes back to you some time later and asks you to do the same analysis with an updated version of the data. How can you exactly redo the analysis and how long do you think it will take you to complete the job?
- She finds an error in your work or she has an idea to improve your analysis by making a few adjustments. Can you implement the adjustments easily, or do you have to redo everything from scratch?
- She sends you back the file with the comment that she has worked out some things in the file and now everything should be fine. How do you know what she has done to the data?

To say it in the words of Stephenson [2023, sec. 5.1]:

> "*Spreadsheets are a nightmare for quality control and reproducibility, and you should always think twice before using one. Spreadsheets will always be a handy way to manipulate tabular datasets, and you'll probably find them useful for data collection and quick back-of-the-envelope calculations, but they're often more trouble than they're worth.*"

3.2. R Scripts: Why they are useful

I have already discussed in Section 1.7 that you can run code either directly by typing your code into the console of R or by writing a script and then sending the code with `Ctrl+Enter` or with the `Run` button to the console of R. 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 just plain text that is interpreted as code or as a comment if the text follows a hastag `#`. A script comes with important advantages.

Scripts…

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

3.3. Create, write, and run R scripts

3.3.1. Create

ℹ 3 equivalent ways to create a script

1. Use the menu: `File > New File > R Script`
2. Use the keyboard shortcut: `Ctrl+Shift+N` (Windows/Linux) or `Cmd+Shift+N` (Mac) or
3. Type the following 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/Linux) or `Cmd+S` (Mac). If you go for the third way, you need to open it manually.

3.3.2. Write

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

```
x <- "hello world"
print(x)
```

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

The above lines of code do the following:

- With the assignment operator `<-` we create an object that stores the words "hello world" in an object entitled `x`. In Section 7.1.1 the assignment operator is further explained.
- With the third input we print the content of the object `x`.

3.3.3. Run

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.

Exercise 3.1. Run a script and round numbers
Please copy and paste the following lines of code into an R script, run it on your computer, and try to understand how it works.

```
# 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()
```

Solution

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

3.4. What to do at the header of each script

At the beginning of each script, ensure that all required packages are loaded correctly. Use the `pacman` package, which provides the `p_load()` function to load and, if necessary, install packages, and the `p_unload(all)` function to unload all packages. Additionally, set your working directory with `setwd()` and clear all objects from the environment with `rm(list = ls())`. This ensures that everything in the environment after sourcing the script originates from the script itself. Below is the code that I use at the beginning of all my scripts. I recommend you do the same.

Tip 5: Start your script with

```
if (!require(pacman)) install.packages("pacman")
pacman::p_unload(all)
pacman::p_load(tidyverse, janitor)
setwd("~/your-directory/")
rm(list = ls())
```

Part II.

Basics of coding

4. Interactive introduction with swirl

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

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

4.1. Set up `swirl`

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

Open Rstudio and type in the console the following:

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

The above four lines of code do the following:

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

> Tip 3: If the course has failed to install,
>
> you can try to download the file `swirl-it.swc` from github.com/hubchev/swirl-it and install the course with loading the `swirl` package and typing `install_course()` into the console.

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

4.2. swirl-it: huber-intro-1

i Click to see the full content of the module

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

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

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

i Solution

answer: b

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

```
getwd()
```

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

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

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

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

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

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

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

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

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

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

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

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

```
10^2 + 36
```

```
[1] 136
```

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

```
A <- 4
```

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

```
A
```

```
[1] 4
```

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

```
A * 5
```

```
[1] 20
```

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

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

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

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

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

```
B <- c(3, 4, 5)
```

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

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

```
[1] 4
```

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

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

```
[1] 4
```

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

```r
mean(B)
```

```
[1] 4
```

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

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

```
[1] 28
```

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

```r
rnorm(10)
```

```
 [1] -0.8462885 -0.4935327  0.9338698  1.8779053 -0.5766011  1.0768528
 [7] -0.4758123  1.3445777 -0.7367863  0.1692006
```

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

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

```
 [1]  7.1268596 -3.3637419 -7.5968553  3.5095991 -0.6394695  7.2545035
 [7] -1.0008906  5.7231421 -2.7122617 -0.8327368
```

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

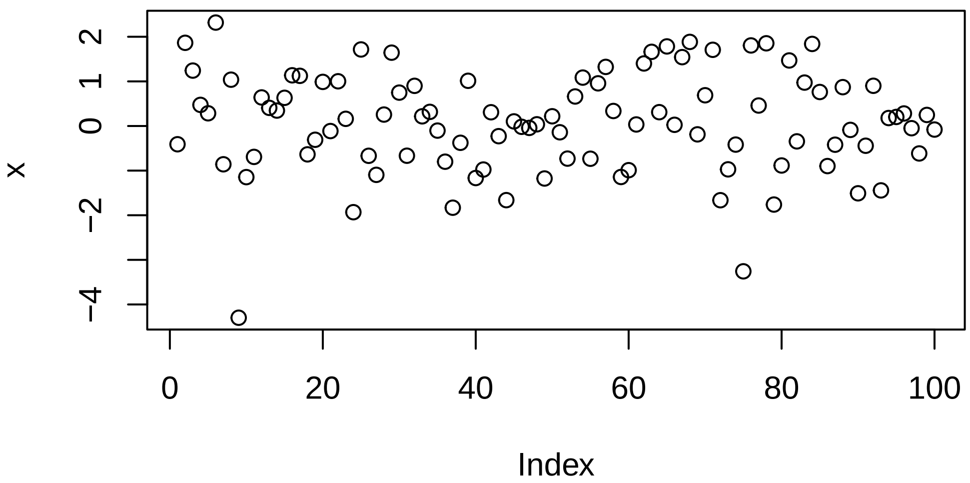
```r
?rnorm
```

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

```
x <- rnorm(100)
```

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

```
plot(x)
```



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

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

4.3. swirl-it: huber-intro-2

Click to see the full content of the module

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

```
setwd("/home/sthu/Documents/mydir")
```

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

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

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

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

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

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

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

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

```
vec1[5]
```

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

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

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

```
vec1
```

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

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

Type `sum(vec1)`.

```
sum(vec1)
```

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

```
vec1 + vec2
```

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

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

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

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

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

```
t
```

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

```
t$z
```

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

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

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

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

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

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

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

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


x1 <- ...
x2 <- ...
x3 <- ...
t <- ...
plot(...)
```

Result

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

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

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

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

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

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

```
L
```

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

```
names(L)
```

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

```
L$five + 10
```

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

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

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

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

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

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

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

# Plot data frame
plot(t$a, type = "l", ylim = range(t), lwd = 3, col = rgb(1, 0, 0, 0.3))
lines(t$b, type = "s", lwd = 2, col = rgb(0.3, 0.4, 0.3, 0.9))
points(t$c, pch = 20, cex = 4, col = rgb(0, 0, 1, 0.3))

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

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

```
?par
```

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

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

4.4. swirl-it: Data analytical basics

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

4.5. swirl-it: The `tidyverse` package

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

4.6. Other `swirl` modules

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

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

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

5. Kickstart

Ever got a kick that actually moved you forward? Well, let's kickstart your R adventure by walking you through a typical data analysis workflow in R, covering everything from setting up your environment to performing data analysis and visualization. Along the way, we'll also tackle some common troubleshooting to smooth out any bumps in the road. Please don't worry if you don't understand some lines of code. You will learn that later on, however, you hopefully will get a sense of what it is like to work with a command line based program.

5.1. Analysing the association of weight and the price of cars

Before we start, we need to ensure that all necessary libraries are installed and loaded. We use the pacman package for convenient package management.

```
# Install and load required libraries
# Installs 'pacman' if not already available, which is used for package management
if (!require(pacman)) install.packages("pacman")

# Unload all previously loaded packages to start fresh
suppressMessages(pacman::p_unload(all))

# Load necessary packages for data manipulation, cleaning, and visualization
pacman::p_load(
  tidyverse, # A suite of packages designed for data science that includes tools for data ma
  haven,     # Used for importing and exporting data with SPSS, Stata, and SAS formats.
  janitor,   # Provides functions for examining and cleaning data, such as `clean_names()` a
  WDI,       # Facilitates downloading data from the World Bank's World Development Indicato
  wbstats    # Provides an interface to the World Bank's APIs for a comprehensive range of d
)

# Set the working directory to a project-specific folder
setwd("~/Dropbox/hsf/courses/dsr")

# Clear the current environment of any objects
rm(list = ls())
```

Now, let us load the dataset from a Stata file (`auto.dta`) and explore its basic properties.

```
# Load data from a Stata file available online
auto <- read_dta("http://www.stata-press.com/data/r18/auto.dta")
# 'auto': Dataset contains information about different car models

# Display basic information about the dataset
ncol(auto)  # Number of columns
```

```
[1] 12
```

```r
nrow(auto)  # Number of rows
```

```
[1] 74
```

```r
dim(auto)   # Dimensions of the dataset
```

```
[1] 74 12
```

```r
names(auto) # Names of variables
```

```
 [1] "make"         "price"        "mpg"          "rep78"        "headroom"
 [6] "trunk"        "weight"       "length"       "turn"         "displacement"
[11] "gear_ratio"   "foreign"
```

```r
head(auto)  # First few rows
```

```
# A tibble: 6 x 12
  make          price   mpg rep78 headroom trunk weight length  turn displacement
  <chr>         <dbl> <dbl> <dbl>    <dbl> <dbl>  <dbl>  <dbl> <dbl>        <dbl>
1 AMC Concord    4099    22     3      2.5    11   2930    186    40          121
2 AMC Pacer      4749    17     3      3      11   3350    173    40          258
3 AMC Spirit     3799    22    NA      3      12   2640    168    35          121
4 Buick Centu~   4816    20     3      4.5    16   3250    196    40          196
5 Buick Elect~   7827    15     4      4      20   4080    222    43          350
6 Buick LeSab~   5788    18     3      4      21   3670    218    43          231
# i 2 more variables: gear_ratio <dbl>, foreign <dbl+lbl>
```

```r
tail(auto)  # Last few rows
```

```
# A tibble: 6 x 12
  make          price   mpg rep78 headroom trunk weight length  turn displacement
  <chr>         <dbl> <dbl> <dbl>    <dbl> <dbl>  <dbl>  <dbl> <dbl>        <dbl>
1 Toyota Coro~   5719    18     5      2      11   2670    175    36          134
2 VW Dasher      7140    23     4      2.5    12   2160    172    36           97
3 VW Diesel      5397    41     5      3      15   2040    155    35           90
4 VW Rabbit      4697    25     4      3      15   1930    155    35           89
5 VW Scirocco    6850    25     4      2      16   1990    156    36           97
6 Volvo 260     11995    17     5      2.5    14   3170    193    37          163
# i 2 more variables: gear_ratio <dbl>, foreign <dbl+lbl>
```

```r
summary(auto) # Summary statistics for each column
```

```
     make               price            mpg            rep78      
 Length:74          Min.   : 3291   Min.   :12.00   Min.   :1.000  
 Class :character   1st Qu.: 4220   1st Qu.:18.00   1st Qu.:3.000  
 Mode  :character   Median : 5006   Median :20.00   Median :3.000  
                    Mean   : 6165   Mean   :21.30   Mean   :3.406  
                    3rd Qu.: 6332   3rd Qu.:24.75   3rd Qu.:4.000  
                    Max.   :15906   Max.   :41.00   Max.   :5.000  
                                                    NA's   :5      
    headroom         trunk           weight         length           turn      
 Min.   :1.500   Min.   : 5.00   Min.   :1760   Min.   :142.0   Min.   :31.00  
 1st Qu.:2.500   1st Qu.:10.25   1st Qu.:2250   1st Qu.:170.0   1st Qu.:36.00  
 Median :3.000   Median :14.00   Median :3190   Median :192.5   Median :40.00  
 Mean   :2.993   Mean   :13.76   Mean   :3019   Mean   :187.9   Mean   :39.65  
 3rd Qu.:3.500   3rd Qu.:16.75   3rd Qu.:3600   3rd Qu.:203.8   3rd Qu.:43.00  
 Max.   :5.000   Max.   :23.00   Max.   :4840   Max.   :233.0   Max.   :51.00  

  displacement     gear_ratio       foreign      
 Min.   : 79.0   Min.   :2.190   Min.   :0.0000  
 1st Qu.:119.0   1st Qu.:2.730   1st Qu.:0.0000  
 Median :196.0   Median :2.955   Median :0.0000  
 Mean   :197.3   Mean   :3.015   Mean   :0.2973  
 3rd Qu.:245.2   3rd Qu.:3.353   3rd Qu.:1.0000  
 Max.   :425.0   Max.   :3.890   Max.   :1.0000  
```

```r
glimpse(auto) # Compact display of the structure of the dataset
```

```
Rows: 74
Columns: 12
$ make         <chr> "AMC Concord", "AMC Pacer", "AMC Spirit", "Buick Century"~
$ price        <dbl> 4099, 4749, 3799, 4816, 7827, 5788, 4453, 5189, 10372, 40~
$ mpg          <dbl> 22, 17, 22, 20, 15, 18, 26, 20, 16, 19, 14, 14, 21, 29, 1~
$ rep78        <dbl> 3, 3, NA, 3, 4, 3, NA, 3, 3, 3, 3, 2, 3, 3, 4, 3, 2, 2, 3~
$ headroom     <dbl> 2.5, 3.0, 3.0, 4.5, 4.0, 4.0, 3.0, 2.0, 3.5, 3.5, 4.0, 3.~
$ trunk        <dbl> 11, 11, 12, 16, 20, 21, 10, 16, 17, 13, 20, 16, 13, 9, 20~
$ weight       <dbl> 2930, 3350, 2640, 3250, 4080, 3670, 2230, 3280, 3880, 340~
$ length       <dbl> 186, 173, 168, 196, 222, 218, 170, 200, 207, 200, 221, 20~
$ turn         <dbl> 40, 40, 35, 40, 43, 43, 34, 42, 43, 42, 44, 43, 45, 34, 4~
$ displacement <dbl> 121, 258, 121, 196, 350, 231, 304, 196, 231, 231, 425, 35~
$ gear_ratio   <dbl> 3.58, 2.53, 3.08, 2.93, 2.41, 2.73, 2.87, 2.93, 2.93, 3.0~
$ foreign      <dbl+lbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
```

```r
print(auto, n = Inf) # Print all rows of the dataset
```

```
# A tibble: 74 x 12
   make          price   mpg rep78 headroom trunk weight length  turn displacement
   <chr>         <dbl> <dbl> <dbl>    <dbl> <dbl>  <dbl>  <dbl> <dbl>        <dbl>
 1 AMC Concord    4099    22     3      2.5    11   2930    186    40          121
 2 AMC Pacer      4749    17     3      3      11   3350    173    40          258
 3 AMC Spirit     3799    22    NA      3      12   2640    168    35          121
```

4	Buick Cent~	4816	20	3	4.5	16	3250	196	40	196
5	Buick Elec~	7827	15	4	4	20	4080	222	43	350
6	Buick LeSa~	5788	18	3	4	21	3670	218	43	231
7	Buick Opel	4453	26	NA	3	10	2230	170	34	304
8	Buick Regal	5189	20	3	2	16	3280	200	42	196
9	Buick Rivi~	10372	16	3	3.5	17	3880	207	43	231
10	Buick Skyl~	4082	19	3	3.5	13	3400	200	42	231
11	Cad. Devil~	11385	14	3	4	20	4330	221	44	425
12	Cad. Eldor~	14500	14	2	3.5	16	3900	204	43	350
13	Cad. Sevil~	15906	21	3	3	13	4290	204	45	350
14	Chev. Chev~	3299	29	3	2.5	9	2110	163	34	231
15	Chev. Impa~	5705	16	4	4	20	3690	212	43	250
16	Chev. Mali~	4504	22	3	3.5	17	3180	193	31	200
17	Chev. Mont~	5104	22	2	2	16	3220	200	41	200
18	Chev. Monza	3667	24	2	2	7	2750	179	40	151
19	Chev. Nova	3955	19	3	3.5	13	3430	197	43	250
20	Dodge Colt	3984	30	5	2	8	2120	163	35	98
21	Dodge Dipl~	4010	18	2	4	17	3600	206	46	318
22	Dodge Magn~	5886	16	2	4	17	3600	206	46	318
23	Dodge St. ~	6342	17	2	4.5	21	3740	220	46	225
24	Ford Fiesta	4389	28	4	1.5	9	1800	147	33	98
25	Ford Musta~	4187	21	3	2	10	2650	179	43	140
26	Linc. Cont~	11497	12	3	3.5	22	4840	233	51	400
27	Linc. Mark~	13594	12	3	2.5	18	4720	230	48	400
28	Linc. Vers~	13466	14	3	3.5	15	3830	201	41	302
29	Merc. Bobc~	3829	22	4	3	9	2580	169	39	140
30	Merc. Coug~	5379	14	4	3.5	16	4060	221	48	302
31	Merc. Marq~	6165	15	3	3.5	23	3720	212	44	302
32	Merc. Mona~	4516	18	3	3	15	3370	198	41	250
33	Merc. XR-7	6303	14	4	3	16	4130	217	45	302
34	Merc. Zeph~	3291	20	3	3.5	17	2830	195	43	140
35	Olds 98	8814	21	4	4	20	4060	220	43	350
36	Olds Cutl ~	5172	19	3	2	16	3310	198	42	231
37	Olds Cutla~	4733	19	3	4.5	16	3300	198	42	231
38	Olds Delta~	4890	18	4	4	20	3690	218	42	231
39	Olds Omega	4181	19	3	4.5	14	3370	200	43	231
40	Olds Starf~	4195	24	1	2	10	2730	180	40	151
41	Olds Toron~	10371	16	3	3.5	17	4030	206	43	350
42	Plym. Arrow	4647	28	3	2	11	3260	170	37	156
43	Plym. Champ	4425	34	5	2.5	11	1800	157	37	86
44	Plym. Hori~	4482	25	3	4	17	2200	165	36	105
45	Plym. Sapp~	6486	26	NA	1.5	8	2520	182	38	119
46	Plym. Vola~	4060	18	2	5	16	3330	201	44	225
47	Pont. Cata~	5798	18	4	4	20	3700	214	42	231
48	Pont. Fire~	4934	18	1	1.5	7	3470	198	42	231
49	Pont. Gran~	5222	19	3	2	16	3210	201	45	231
50	Pont. Le M~	4723	19	3	3.5	17	3200	199	40	231
51	Pont. Phoe~	4424	19	NA	3.5	13	3420	203	43	231
52	Pont. Sunb~	4172	24	2	2	7	2690	179	41	151
53	Audi 5000	9690	17	5	3	15	2830	189	37	131
54	Audi Fox	6295	23	3	2.5	11	2070	174	36	97

```
55 BMW 320i      9735    25     4       2.5     12   2650     177    34              121
56 Datsun 200    6229    23     4       1.5      6   2370     170    35              119
57 Datsun 210    4589    35     5       2        8   2020     165    32               85
58 Datsun 510    5079    24     4       2.5      8   2280     170    34              119
59 Datsun 810    8129    21     4       2.5      8   2750     184    38              146
60 Fiat Strada   4296    21     3       2.5     16   2130     161    36              105
61 Honda Acco~   5799    25     5       3       10   2240     172    36              107
62 Honda Civic   4499    28     4       2.5      5   1760     149    34               91
63 Mazda GLC     3995    30     4       3.5     11   1980     154    33               86
64 Peugeot 604 12990    14    NA       3.5     14   3420     192    38              163
65 Renault Le~   3895    26     3       3       10   1830     142    34               79
66 Subaru        3798    35     5       2.5     11   2050     164    36               97
67 Toyota Cel~   5899    18     5       2.5     14   2410     174    36              134
68 Toyota Cor~   3748    31     5       3        9   2200     165    35               97
69 Toyota Cor~   5719    18     5       2       11   2670     175    36              134
70 VW Dasher     7140    23     4       2.5     12   2160     172    36               97
71 VW Diesel     5397    41     5       3       15   2040     155    35               90
72 VW Rabbit     4697    25     4       3       15   1930     155    35               89
73 VW Scirocco   6850    25     4       2       16   1990     156    36               97
74 Volvo 260    11995    17     5       2.5     14   3170     193    37              163
# i 2 more variables: gear_ratio <dbl>, foreign <dbl+lbl>
```

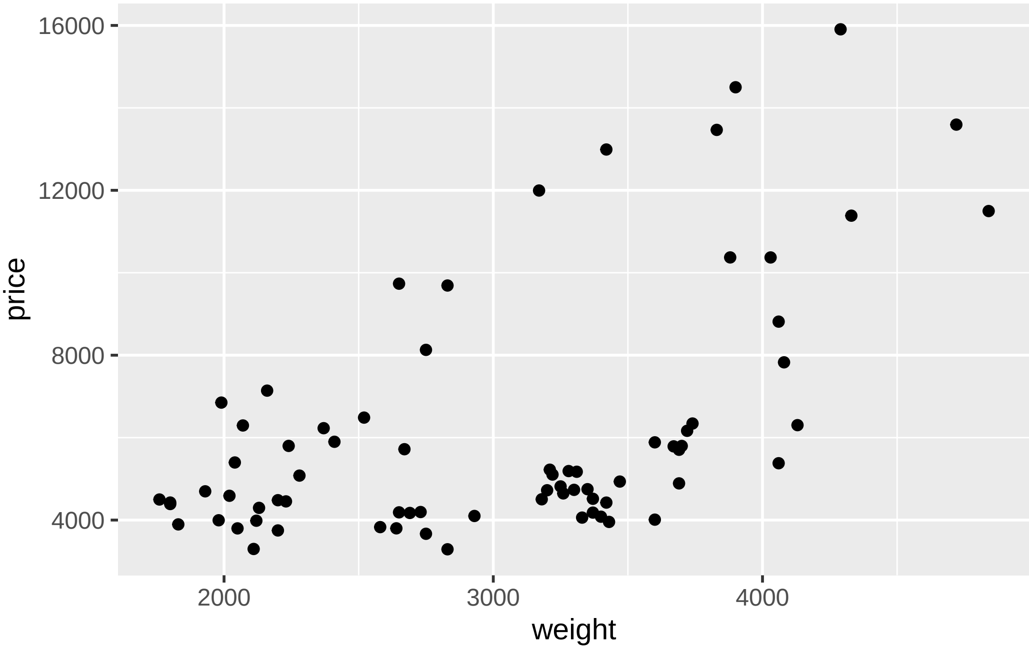
The data seems to be a cross-section of cars. Let us check if the variable `make` identifies each line uniquely:

```r
# Check for duplicate entries based on the 'make' variable
auto |>
  get_dupes(make)
```

```
No duplicate combinations found of: make

# A tibble: 0 x 13
# i 13 variables: make <chr>, dupe_count <int>, price <dbl>, mpg <dbl>,
#   rep78 <dbl>, headroom <dbl>, trunk <dbl>, weight <dbl>, length <dbl>,
#   turn <dbl>, displacement <dbl>, gear_ratio <dbl>, foreign <dbl+lbl>
```

Indeed, the variable `make` has no duplicates. Now, let's make and save some graphical visualizations:

```r
# Create and display a scatter plot of car price versus weight
plot_weight_price <- ggplot(auto, aes(x = weight, y = price)) +
  geom_point()
plot_weight_price
```

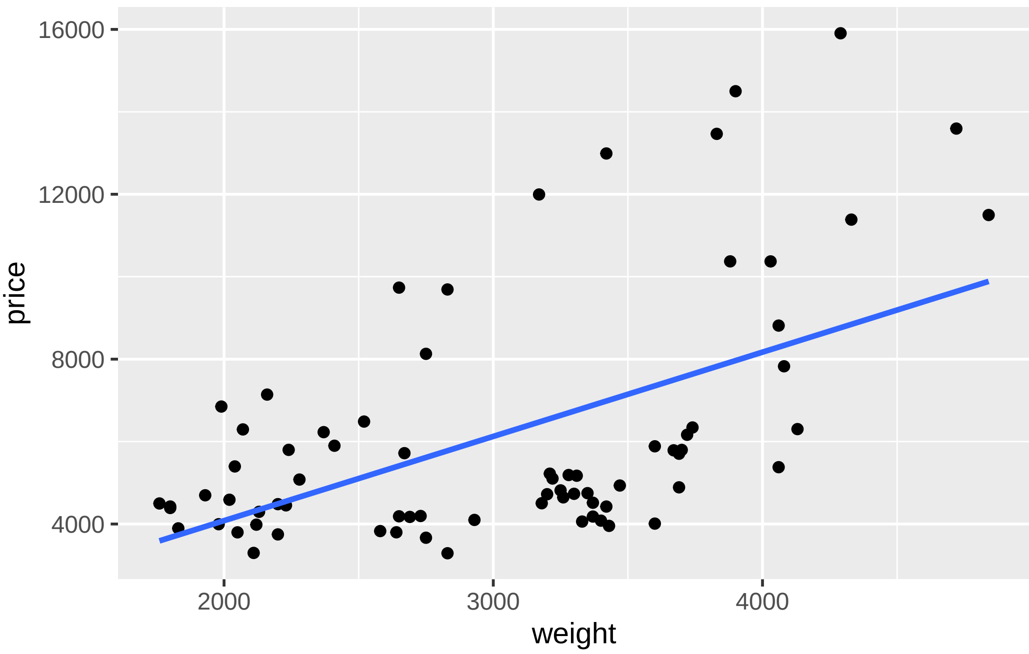



```
# Save the plot to a file
ggsave("fig/plot_weight_price.png", plot = plot_weight_price, dpi = 300)
```

```
Saving 5.5 x 3.5 in image
```

```
# Create a scatter plot with a linear regression line of price vs weight
plot_weight_price_fit <- ggplot(auto, aes(x = weight, y = price)) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE)  # 'lm' denotes linear model, 'se' is standard error
plot_weight_price_fit
```

```
`geom_smooth()` using formula = 'y ~ x'
```



```r
# Save the plot to a file
ggsave("fig/plot_weight_price_fit.png", plot = plot_weight_price_fit, dpi = 300)
```

```
Saving 5.5 x 3.5 in image
`geom_smooth()` using formula = 'y ~ x'
```

Let us perform a linear regression to quantify the impact of `weight` on `price`:

```r
# Perform a linear regression to analyze the relationship between weight and price
reg_result <- lm(price ~ weight , data = auto)
summary(reg_result) # Display the regression results
```

```
Call:
lm(formula = price ~ weight, data = auto)

Residuals:
    Min      1Q  Median      3Q     Max 
-3341.9 -1828.3  -624.1  1232.1  7143.7 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)   -6.7074  1174.4296  -0.006    0.995    
weight         2.0441     0.3768   5.424 7.42e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2502 on 72 degrees of freedom
Multiple R-squared:  0.2901,	Adjusted R-squared:  0.2802 
F-statistic: 29.42 on 1 and 72 DF,  p-value: 7.416e-07
```

5.2. Accessing World Bank's *World Development Indicators*

The World Wide Web is a treasure trove of data, and most major databases offer researchers direct download options. Numerous user-supplied packages are available for seamless access to such data. In this section, I present two popular packages that facilitate the downloading of data from the World Bank's *World Development Indicators*:

WDI (World Development Indicators) Package - Official CRAN package documentation: WDI on CRAN - Source code on GitHub: WDI GitHub Repository

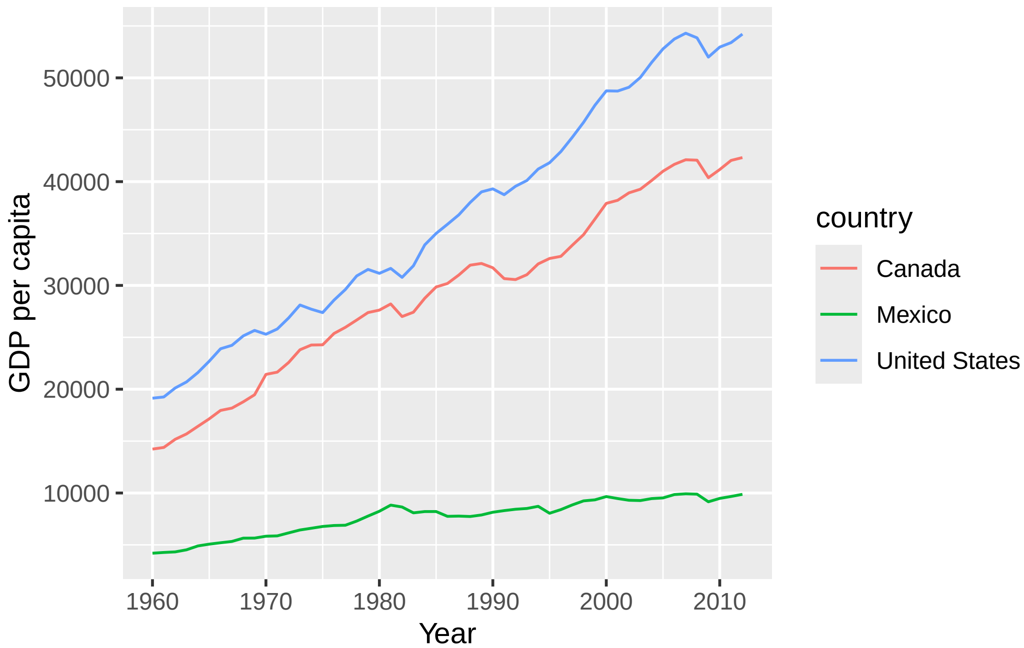
wbstats (World Bank Statistics) Package - Official CRAN package documentation: wbstats on CRAN - Source code on GitHub: wbstats GitHub Repository

Now, let's download some GDP data and explore how to manipulate it. This exercise will demonstrate practical applications of the tools.

```r
# Search for GDP indicators and display the first 10
WDIsearch("gdp")[1:10, ]
```

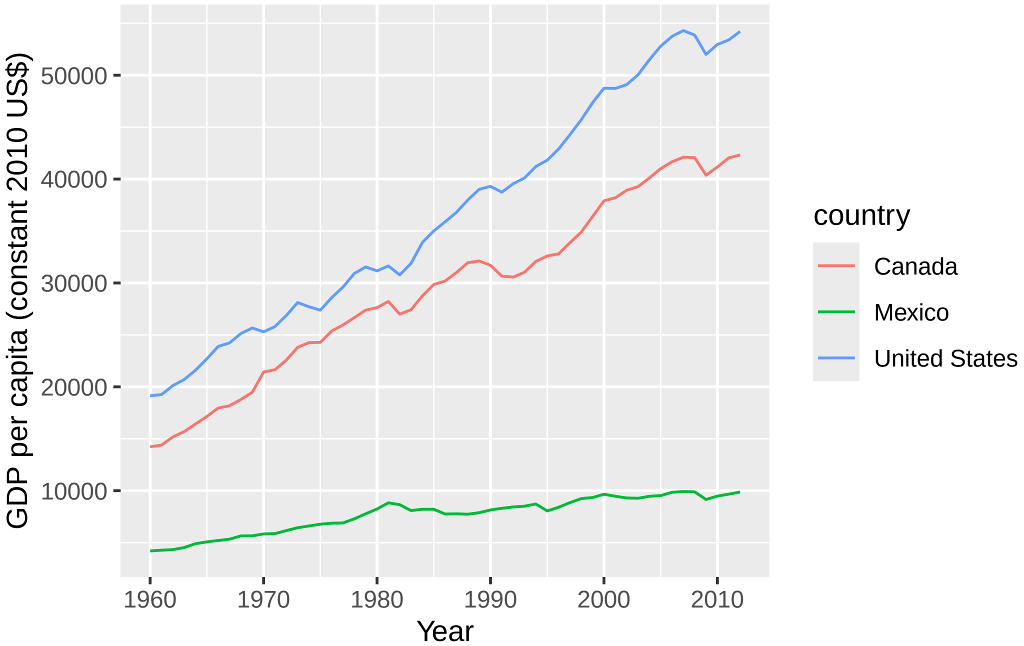
```
                 indicator                                                  name
712          5.51.01.10.gdp                                 Per capita GDP growth
714         6.0.GDP_current                                     GDP (current $)
715          6.0.GDP_growth                                 GDP growth (annual %)
716             6.0.GDP_usd                                 GDP (constant 2005 $)
717     6.0.GDPpc_constant GDP per capita, PPP (constant 2011 international $)
1557     BG.GSR.NFSV.GD.ZS                           Trade in services (% of GDP)
1558 BG.KAC.FNEI.GD.PP.ZS               Gross private capital flows (% of GDP, PPP)
1559     BG.KAC.FNEI.GD.ZS                    Gross private capital flows (% of GDP)
1560 BG.KLT.DINV.GD.PP.ZS          Gross foreign direct investment (% of GDP, PPP)
1561     BG.KLT.DINV.GD.ZS                 Gross foreign direct investment (% of GDP)
```

```r
# Retrieve GDP per capita data for specified countries and years
df_WDI <- WDI(
  indicator = "NY.GDP.PCAP.KD",
  country = c("MX", "CA", "US"),
  start = 1960,
  end = 2012
)

# Plot GDP per capita over time for the specified countries
ggplot(df_WDI, aes(year, NY.GDP.PCAP.KD, color = country)) +
  geom_line() +
  xlab("Year") +
  ylab("GDP per capita")
```




```r
# Retrieve GDP per capita data for specified countries and years using the wbstats package
df_wb <- wb_data(
  indicator = "NY.GDP.PCAP.KD",
  country = c("MX", "CA", "US"),
  start = 1960,
  end = 2012,
```

```r
  return_wide = TRUE
)

# Plot GDP per capita over time for the specified countries
ggplot(df_wb, aes(date, NY.GDP.PCAP.KD, color = country)) +
  geom_line() +
  xlab("Year") +
  ylab("GDP per capita (constant 2010 US$)")
```



The latter graph appears empty. Why? Let's take a closer look at the data to identify any discrepancies that might explain this issue:

```r
# Look at the data types year and date are different:
glimpse(df_WDI)
```

```
Rows: 159
Columns: 5
$ country        <chr> "Canada", "Canada", "Canada", "Canada", "Canada", "Cana~
$ iso2c          <chr> "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "~
$ iso3c          <chr> "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN",~
$ year           <int> 2012, 2011, 2010, 2009, 2008, 2007, 2006, 2005, 2004, 2~
$ NY.GDP.PCAP.KD <dbl> 42319.63, 42043.51, 41165.57, 40377.82, 42067.80, 42106~
```

```r
glimpse(df_wb)
```

```
Rows: 159
Columns: 9
$ iso2c          <chr> "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "~
$ iso3c          <chr> "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN",~
$ country        <chr> "Canada", "Canada", "Canada", "Canada", "Canada", "Cana~
$ date           <dbl> 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1~
```

```
$ NY.GDP.PCAP.KD <dbl> 14229.83, 14389.40, 15173.02, 15689.70, 16419.39, 17143~
$ unit           <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ obs_status     <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ footnote       <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ last_updated   <date> 2024-05-30, 2024-05-30, 2024-05-30, 2024-05-30, 2024-0~
```

The answer is: A lineplot with a character variable (`date` is `<chr>`) on the x-axis does not work!

Now, let us manipulate the `df_wb` data so that the two dataset are equal:

```r
df_wb_cln <- df_wb |>
  # Convert 'date' in df_wb from character to integer
  mutate(year = as.integer(date)) |>
  # Since 'year' has been created, remove the original 'date' column
  select(-date) |>
  # Relocate columns to organize the data frame
  relocate(country, iso2c, iso3c, year, NY.GDP.PCAP.KD)

glimpse(df_WDI)
```

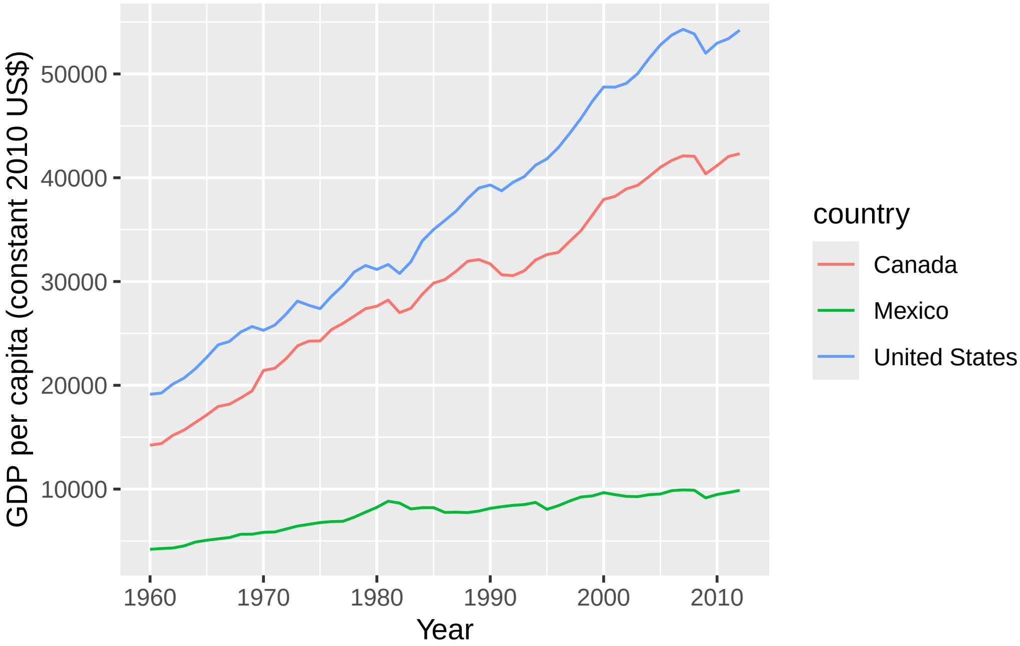
```
Rows: 159
Columns: 5
$ country        <chr> "Canada", "Canada", "Canada", "Canada", "Canada", "Cana~
$ iso2c          <chr> "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "~
$ iso3c          <chr> "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN",~
$ year           <int> 2012, 2011, 2010, 2009, 2008, 2007, 2006, 2005, 2004, 2~
$ NY.GDP.PCAP.KD <dbl> 42319.63, 42043.51, 41165.57, 40377.82, 42067.80, 42106~
```

```r
glimpse(df_wb_cln)
```

```
Rows: 159
Columns: 9
$ country        <chr> "Canada", "Canada", "Canada", "Canada", "Canada", "Cana~
$ iso2c          <chr> "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "~
$ iso3c          <chr> "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN", "CAN",~
$ year           <int> 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1~
$ NY.GDP.PCAP.KD <dbl> 14229.83, 14389.40, 15173.02, 15689.70, 16419.39, 17143~
$ unit           <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ obs_status     <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ footnote       <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ last_updated   <date> 2024-05-30, 2024-05-30, 2024-05-30, 2024-05-30, 2024-0~
```

Now it works:

```r
# Plot GDP per capita over time for the specified countries
ggplot(df_wb_cln, aes(year, NY.GDP.PCAP.KD, color = country)) +
  geom_line() +
  xlab("Year") +
  ylab("GDP per capita (constant 2010 US$)")
```



Solution

The script uses the following functions: `aes`, `as.integer`, `c`, `dim`, `geom_line`, `geom_point`, `geom_smooth`, `get_dupes`, `ggplot`, `ggsave`, `glimpse`, `head`, `lm`, `mutate`, `names`, `ncol`, `nrow`, `print`, `read_dta`, `relocate`, `select`, `setwd`, `summary`, `tail`, `wb`, `WDI`, `WDIsearch`, `xlab`, `ylab`.

R script

```
# This script demonstrates a typical data analysis workflow in R
# ---------------------------------------------------------------

# Install and load required libraries
# Installs 'pacman' if not already available, which is used for package management
if (!require(pacman)) install.packages("pacman")

# Unload all previously loaded packages to start fresh
suppressMessages(pacman::p_unload(all))

# Load necessary packages for data manipulation, cleaning, and visualization
pacman::p_load(
  tidyverse, # A suite of packages designed for data science that includes tools for dat
  haven, # Used for importing and exporting data with SPSS, Stata, and SAS formats.
  janitor, # Provides functions for examining and cleaning data, such as `clean_names()`
  WDI, # Facilitates downloading data from the World Bank's World Development Indicators
  wbstats # Provides an interface to the World Bank's APIs for a comprehensive range of
)

# Set the working directory to a project-specific folder
setwd("~/Dropbox/hsf/courses/dsr")

# Clear the current environment of any objects
rm(list = ls())

# ---------------------------------------------------------------

# Load data from a Stata file available online
auto <- read_dta("http://www.stata-press.com/data/r8/auto.dta")
# 'auto': Dataset contains information about different car models

# Display basic information about the dataset
ncol(auto) # Number of columns
nrow(auto) # Number of rows
dim(auto) # Dimensions of the dataset
names(auto) # Names of variables
head(auto) # First few rows
tail(auto) # Last few rows
summary(auto) # Summary statistics for each column
glimpse(auto) # Compact display of the structure of the dataset
print(auto, n = Inf) # Print all rows of the dataset

# Check for duplicate entries based on the 'make' variable
auto |>
  get_dupes(make)

# Create and display a scatter plot of car price versus weight
plot_weight_price <- ggplot(auto, aes(x = weight, y = price)) +
  geom_point()
plot_weight_price

# Save the plot to a file
ggsave("fig/plot_weight_price.png", plot = plot_weight_price, dpi = 300)
```

6. Pitfalls

R newbies often make the same small mistakes that can lead to major confusion, frustration and inefficiency. Some of them can be easily avoided. In this section, I will outline common pitfalls that I have repeatedly observed as an R instructor and offer practical solutions to avoid them.

6.1. No clue about the "working directory"

Problem: Students start their R sessions unaware of their current working directory. This can lead to difficulties when reading and writing files.

Solution: At the beginning of your R script set a working directory using `setwd()`. Consider using R Studio projects, see Section A.4. For more information, see Appendix A: *Navigating the file system* and *Workflow: scripts and projects* of Wickham and Grolemund [2023].

6.2. No consistent directory structure

Problem: Students save files in different directories without a clear scheme. This disorganization often leads to problems: Scripts and data gets lost and code breaks.

Solution: Organize your project into a clear directory structure from the beginning. Here is my suggestion for a directory structure but feel free to come up with your own:

Table 6.1.: Typical folder structure

Sub-Directory	What to save here
`doc/`	documentation
`dta/`	processed data
`fig/`	figures
`lit/`	literature and pdfs
`ori/`	original raw data that you should never change
`qmd/`	reports
`scr/`	R scripts
`tab/`	tables
`tmp/`	temporary files

This structure will save time and headaches when navigating projects.

Tip 4: Do not save processed data unless necessary

It may seem reasonable to save data after editing, but this often isn't necessary if you're using scripts to create your data. These scripts can be rerun whenever needed, regenerating the dataset each time. To avoid wasting disk space and maintain an organized project folder, it's advisable to save processed datasets only when the preprocessing steps are time-consuming. This way, you can keep your project folder more organized and ensure that your data analyses are always reproducible with the latest updates to your code.

6.3. Working manually outside R

Problem: Students want to get their work done quickly. This sometimes leads to them relying on manual processes that they have already mastered for their data work. This approach can lead to serious problems when it comes to the reproducibility of their data work.

Consider a typical three-step process for loading data: (1) downloading the data, (2) unpacking the data, and finally (3) importing the data into R. Many students often take a manual approach by using their Internet browser to download the data, then using their operating system's unpacking application, and finally importing the data into an R script. While this method is not inherently wrong, there is a risk that students will forget to unpack the downloaded data, resulting in them accidentally working with outdated data. In the kickstart example provided by Section 5.2, I show that all three steps can be performed seamlessly in R. This way you ensure that you are always working with the most up-to-date data.

Solution: Do as much as possible in the script. Invest some time to find out how to download and manipulate the data within R. If it is not possible or if alternatives are superior, describe what you do outside of R explicitly and write a warning note at the top of your script.

6.4. No active R Packages management

Problem: Students often forget to install and/or load the packages correctly at the beginning of a script. Some unnecessarily install packages repeatedly when running a script. All this can lead to errors and interruptions.

Solution: At the beginning of each script, make sure that all required packages are loaded correctly. Use the `pacman` package, which provides the `p_load()` function to load and, if necessary, install packages and the `p_unload(all)` function to unload all packages.

Tip 5: Start your script with

```
if (!require(pacman)) install.packages("pacman")
pacman::p_unload(all)
pacman::p_load(tidyverse, janitor)
setwd("~/your-directory/")
rm(list = ls())
```

6.5. Confusion between console and script

Problem: Alternating between running code in the console and from the script without a systematic approach can lead to untracked changes and confusion about the current state of objects in the workspace. Additionally, students often borrow code snippets from others and run only the sections that seem immediately relevant. This practice can lead to errors or unexpected results, as such code often relies on previous commands or setups.

Solution: Develop the habit of testing small blocks of code in the console but run the complete script regularly to ensure everything works in sequence. Use shortcuts like `Ctrl + Alt + R` to source the entire script or `Ctrl + Alt + B/E` to execute it up to a specific point.

6.6. Misunderstanding data types and formats

Problem: Misusing or misunderstanding R's data types and structures can lead to errors in data manipulation and analysis. Many functions require certain types of data. For example, the `tidyverse` packages required data to be "tidy". Moreover, data often comes with errors and/or missings (`NA`). Beginners overlook data cleaning and considering missings.

Solution: Familiarize yourself with basic data types and structures like vectors, lists, data frames/tibbles, and factors. For more information, see Section 7.2 and Data tidying of Wickham and Grolemund [2023]. Moreover, spend adequate time on data cleaning and preprocessing. Techniques such as handling missing values, normalizing data, and correcting data types are critical. For more information, see Missing values of Wickham and Grolemund [2023].

6.7. Lack of knowledge about data identification

Problem: Students often handle data without understanding which variables uniquely identify the information contained in other variables. It is crucial to recognize these identifying variables and verify their uniqueness to ensure data integrity.

Solution: Perform checks for uniqueness at the beginning of their data analysis process. See exercise *Names and duplicates* in Section 9.16 and the `get_dupes` fuction introduced in Section 7.4.3.2.

> Tip 6: Always check your data with get_dupes
>
> For example, you expect that your dataframe `df` is a panel dataset. With
>
> ```
> get_dupes(df, country, year)
> ```
>
> you can check whether the two variables `country` and `year` indeed identify each row uniquely.

6.8. Losing track of data due to excessive overwriting

Problem: Students often manipulate their data by repeatedly overwriting the same object. This can lead to confusion about the data's current state and the transformations applied.

Solution: Minimize the number of assignments to a single object. Instead, create a new object with a descriptive and concise name each time you alter the data. This practice helps maintain clarity about each stage of data manipulation.

For example, if you're working with data `df`, you might store the cleanded data as `df_cln`, then after filtering for specific criteria, you could use `df_cln_flt`, and finally, if you aggregate the data, name it `df_cln_flt_agg`. Having some clear naming convention makes it clear what each dataset represents and the transformations it has undergone.

Tip 7: Do clear the environment at the beginning of a script with

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

6.9. No documentation

Problem: Students do not comment code. This makes it hard to remember the purpose of various lines of code and difficult for other people to read and understand the code.

Solution: Regularly comment your code, explaining why something is done, not just what is done. Use clear, concise comments to improve readability and maintainability.

6.10. Ignoring error messages and warnings

Problem: Students see that their code doesn't work but do not read the error message which often contains hints for solving the problem.

Solution: Read and follow error messages. Do not ignore warnings or errors unless you know what they mean. Study what the error message might mean. Use online resources such as Google and ChatGPT, see Figure 6.1. Finally, have the confidence to implement the suggested solution. Don't be frustrated if the first attempt does not work: Try again and play around.

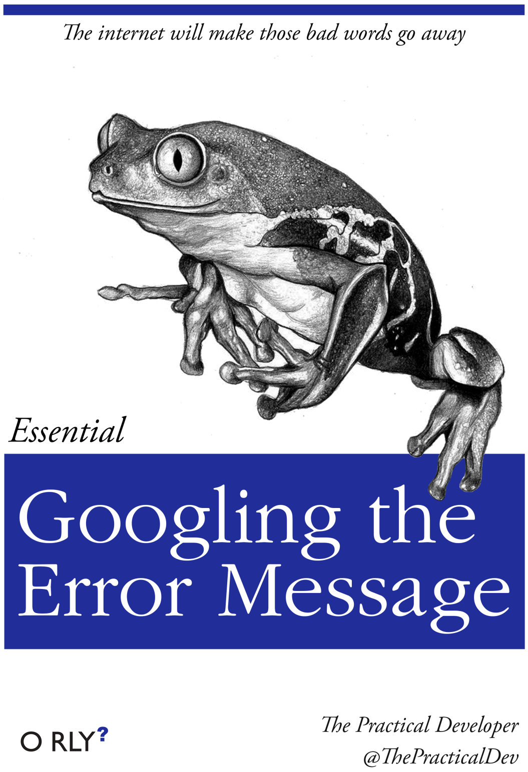
Tip 8: Common error messages

Students often come to me with error messages suggesting that they install the `tinytex` package or the `RTools` compiler for Windows. Since they are not familiar with these R and software packages, they wonder if it is safe to proceed with the installation. My answer here is: yes, it is recommendable to install it.
Based the report of Noam Ross who examine roughly 10,000 R error messages, the most frequently encountered error messages of R are shown in Table 6.2 with some ideas of mine what to do.

Table 6.2.: Most frequent errors

Error Type	Some suggestions on what to do
`Could not find function`	Check spelling of the function and whether the respective packages are loaded properly.
`Error in if`	This suggests an issue with non-logical or missing values in a conditional statement. Check syntax and spelling. Maybe use ChatGPT to debug the code.
`Error in eval`	Points to references to non-existent objects.
`Cannot open`	Check if the files exist at the place you try to call them.
`No applicable method`	Check your data type and whether it fits to the requirements of the functions you're trying to use.

Figure 6.1.: Googling the error message



Source: DEV Community on GitHub

`Package errors`	This can stem from issues with installing, compiling, or loading a package. Maybe you try to re-install the package and their dependencies, or update the packages.

6.11. No attempt to identify the problem and troubleshoot

Problem: Students often do not fully understand the problems they encounter, which can lead to difficulties in seeking solutions. It's common for students to feel overwhelmed and seek help without attempting to find the source of the problem and without attempting to work out possible solutions first.

Solution: When you encounter an issue in your R code, it's crucial to methodically dissect the problem. Here's how you can effectively *troubleshoot*, that is, a problem-solving skill that is essential for becoming proficient in programming:

- **Identify the problem**: Attempt to identify the issue to better understand its nature. Once you know which line of code is causing some trouble, you are often close to a solution. Commenting out parts of your script or going back until there is no error can help here.
- **Active solution search:** Once you've identified the problem, actively look for solutions. This can include consulting the R documentation, searching for similar issues online, or asking others for help.
- **Trial and error process:** Don't hesitate to experiment with different solutions to see what works best.

- **Seek Help:** If you're stuck, ask for help from more experienced R users or communities.
- **Minimal Reproducible Example (MRE):** When asking for help, explain your problem precisely and provide a MRE. That is the simplest version of the code that still produces the error, including only essential data and code. This practice not only aids in self-troubleshooting but also makes it easier for others to help by providing a clear, concise context.
- **Additional information**: Sometime the interplay of your the packages loaded, your operating system, the version of R, and/or the RStudio version may play a role in your problem. Thus, 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:

```r
sessionInfo()
```

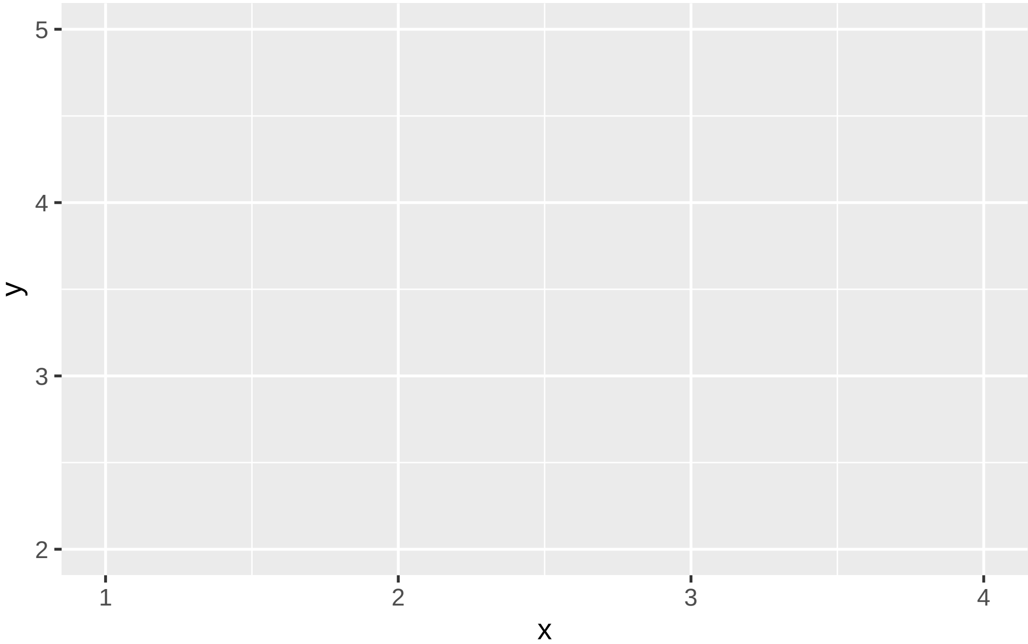
```
R version 4.4.1 (2024-06-14)
Platform: x86_64-pc-linux-gnu
Running under: Debian GNU/Linux 12 (bookworm)

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

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

time zone: Europe/Berlin
tzcode source: system (glibc)

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

loaded via a namespace (and not attached):
 [1] compiler_4.4.1    fastmap_1.1.1     cli_3.6.2         tools_4.4.1
 [5] htmltools_0.5.8.1 rstudioapi_0.16.0 rmarkdown_2.26    knitr_1.46
 [9] jsonlite_1.8.8    xfun_0.43         digest_0.6.35     rlang_1.1.3
[13] evaluate_0.23
```

For more information, see Workflow: getting help of Wickham and Grolemund [2023].

Example of a minimal reproducible example

For example, the following script is a MRE:

```r
library(ggplot2)
data <- data.frame(x = 1:4, y = c(2, 3, 5, 3.4))
ggplot(data, aes(x, y))
```

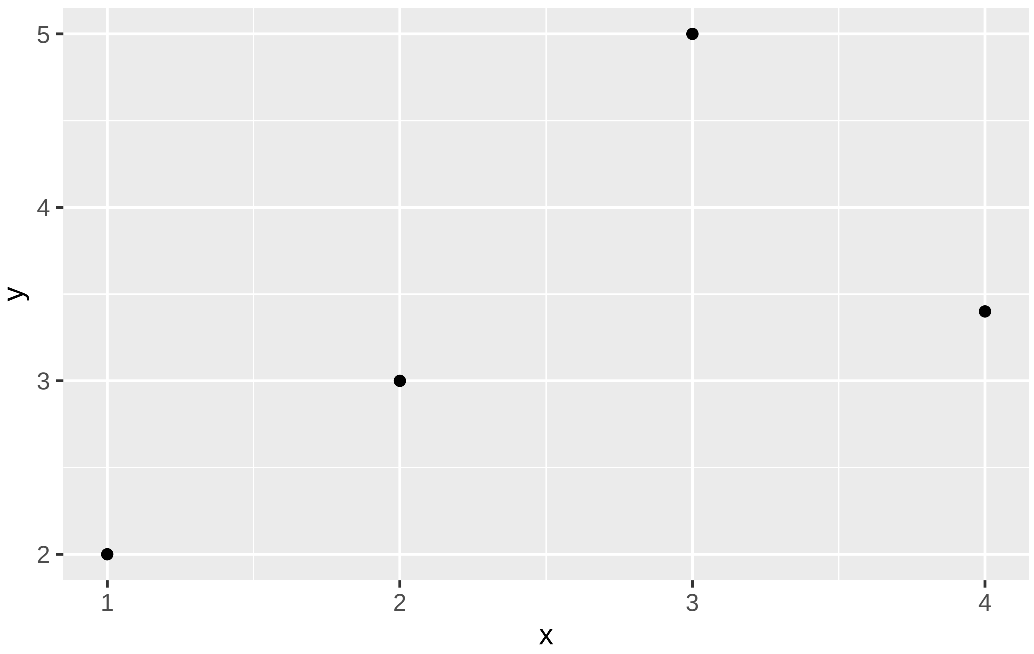



```r
+geom_point()
```

```
Error:
! Cannot use `+` with a single argument.
i Did you accidentally put `+` on a new line?
```

Everybody who copies these few lines of code can reproduce the shown error message and hence can work on a solution. Obviously, the `+` was set falsly. It must be placed in the line of `ggplot`:

```r
library(ggplot2)
data <- data.frame(x = 1:4, y = c(2, 3, 5, 3.4))
ggplot(data, aes(x, y)) +
  geom_point()
```



6.12. Unstylish code

To avoid issues while programming in R, it's essential to understand and adhere to various conventions, rules, and best practices specific to the language. Following these conventions makes your code more readable and simplifies your own experience with R. Below, you will find a non-exhaustive list of these guidelines.

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

> **Tip 9**
>
> There are two packages, `styler` and `lintr`, that support you writing code according to the *The tidyverse style guide* of Wickham [2024].

Part III.

Do stuff

7. Manage data

7.1. Import and generate data

7.1.1. Assigning data to an object using the assignment operator <-

Suppose I'm trying to calculate how much money I'm going to make from selling an item. Let's assume you sell 350 units. To create a variable called `sales` and assigns a value to it, we need to use the assignment operator of R, that is, `<-`:

```
sales <- 350
```

When you send that line of code to the console, it doesn't print out any output but it creates the object `sales`. In Rstudio, you can see the object in the *environment panel* at the top right. Alternatively, you can call the object in the console:

```
sales
```

```
[1] 350
```

R also allows to use `->` and `=` for the assigment. For example, the following ways of assigning data are equivalent:

```
350 -> sales
sales = 350
sales <- 350
```

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

7.1.2. Vectors and matrices

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

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

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

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

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

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

# print matrix
print(m_ab)
```

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

```
print(m_cbind)
```

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

```
print(m_rbind)
```

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

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

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

# print matrix
print(m_ab_label)
```

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

The two most common formats to store and work with data in R are dataframe and tibble. Both formats store table-like structures of data in rows and columns. We will learn more on that in section Section 7.2.

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

Exercise

See exercise in Section 9.1: *Import data with* `c()`.

7.1.3. 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.

7.1.4. 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:

```r
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
```

```r
?mtcars # data dictionary
```

7.1.5. 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 example, `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:

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

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

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

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

```r
glimpse(df_gdp)
```

```
Rows: 13,178
Columns: 7
$ iso3c       <chr> "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "AFE", "A~
$ date        <chr> "2022", "2021", "2020", "2019", "2018", "2017", "2016", "2~
$ value       <dbl> 1.185138e+12, 1.086531e+12, 9.288802e+11, 1.006191e+12, 1.~
$ indicatorID <chr> "NY.GDP.MKTP.CD", "NY.GDP.MKTP.CD", "NY.GDP.MKTP.CD", "NY.~
$ indicator   <chr> "GDP (current US$)", "GDP (current US$)", "GDP (current US~
$ iso2c       <chr> "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH", "ZH"~
$ country     <chr> "Africa Eastern and Southern", "Africa Eastern and Souther~
```

```r
summary(df_gdp)
```

```
    iso3c               date               value            indicatorID
 Length:13178       Length:13178       Min.   :8.825e+06   Length:13178
 Class :character   Class :character   1st Qu.:2.429e+09   Class :character
 Mode  :character   Mode  :character   Median :1.779e+10   Mode  :character
                                       Mean   :1.225e+12
                                       3rd Qu.:2.281e+11
                                       Max.   :1.009e+14
  indicator            iso2c             country
 Length:13178       Length:13178       Length:13178
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character
```

Figure 7.1.: The logo of the packages readr, haven, and readxl

7.1.6. 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" section for importing CSV and TSV files, the "Spreadsheets" section for Excel files, the "Databases" section for retrieving data with SQL, the "Arrow" section for working with large datasets, and the "Web scraping" section for extracting data from web pages.

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

7.1.7. Examples

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

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

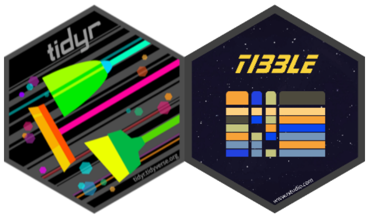
Excel files, due to their wide use in business and research, require a specific approach specifying sheets and cell ranges.

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

7.2. Data

7.2.1. Data frames and tibbles

Figure 7.2.: The logos of the tidyr and tibble packages



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

7.2.2. Tidy data

A popular quote from Hadley Wickham is that

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

It paraphrases the fact that it is a good idea to set rules how a dataset should structure its information to make it easier to work with the data. The tidyverse requires the data to be structured like is illustrated in Figure Figure 7.3. The rules are:

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

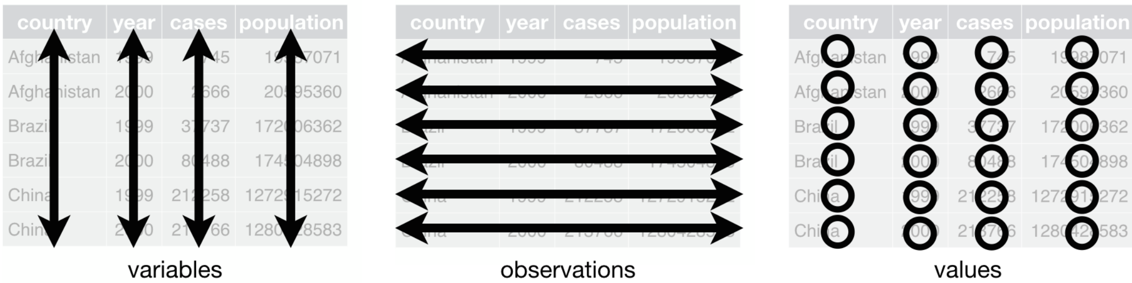
Whenever data follow that consistent structure, we speak of *tidy data*. The underlying uniformity of tidy data facilitates learning and using data manipulation tools.

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

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

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

Figure 7.3.: Features of a tidy dataset: variables are columns, observations are rows, and values are cells



Source: *Wickham and Grolemund [2023].*

```r
rownames(mtcars)
```

```
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
```

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

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

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

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

```
[1] TRUE
```

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

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

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

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

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

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

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

7.2.3. Data types

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

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

The following example should exemplify these types of data:

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

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

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

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

- `NA` stands for *not available* or *missing* and is used to represent missing or undefined values.

- `Inf` stands for *infinity* and is used to represent mathematical infinity, such as the result of dividing a non-zero number by zero. Can be positive or negative.

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

7.3. Operators

An overview of the most important operators of R is proided in Appendix B.

7.3.1. Algebraic operators

R can perform any kind of arithmetic calculation using the operators listed in Table 7.1.

Table 7.1.: Basic algebraic operators

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

7.3.2. The pipe operator: `|>`

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

> Tip 10: Set the native pipe in RStudio
>
> With the keyboard shortcut `Ctrl+Shift+M`, RStudio inserts `%>%`. To change that behavior, simply check the box labeled “Use native pipe operator, |>” in the Global Options, see: `Tools` > `Global Options` > `Code` > `Editing`.

Consider the following example of code to explain the usage of the pipe operator:

```r
# 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
```

```r
# 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:

```r
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":

```r
# 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.
```

Exercise

See exercise in Section 9.3: *Base R,`%in%` operator, and the pipe `|>`*.

7.3.3. 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()`.

Exercise

See exercise in Section 9.3: *Base R,`%in%` operator, and the pipe `|>`*.

7.3.4. 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"]]`.

7.3.5. Logical operators

The extract operators can be combined with the *logical operators* (more precisely, I should call these *binary relational operators*) that are shown in Table 7.2.

Table 7.2.: 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:

```r
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:

```r
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:

```r
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:

```r
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
```

7.4. Data manipulation

7.4.1. `dplyr`: A human readable grammar of data manipulation

Figure 7.4.: The logo of the `dplyr` package

The `dplyr` package is part of tidyverse and makes data manipulation easy as it works well with the pipe operator `|>`. The most important function are the following:

- Reorder the rows with `arrange()`.
- Pick observations by their values with `filter()`.
- Pick variables by their names with `select()`.
- Create new variables with functions of existing variables with `mutate()`.
- Collapse many values down to a single summary with `summarise()`.
- Rename variables with `rename()`.
- Change the position of variables with `relocate()`.

These functions can be used in conjunction with `group_by()` and/or `rowwise()`, which changes the scope of each function from operating on the entire dataset to operating on it group-by-group or by rows. Moreover, you can check for conditions and take action with, for example, `if_else()` and `case_when()`.

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.

Read the vignette of `dplyr` that you find here or with:

```
vignette("dplyr")
```

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

```
library(tidyverse)

# load mtcars dataset
data(mtcars)

# filter only cars with four gears
mtcars_gear_4 <- mtcars |>
  filter(gear == 4)

# arrange rows by mpg in descending order
mtcars_gear_4 |>
  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
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
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
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
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
```

```
# Change the order of the variables
glimpse(mtcars_gear_4)
```

```
Rows: 12
Columns: 11
$ mpg  <dbl> 21.0, 21.0, 22.8, 24.4, 22.8, 19.2, 17.8, 32.4, 30.4, 33.9, 27.3,~
$ cyl  <dbl> 6, 6, 4, 4, 4, 6, 6, 4, 4, 4, 4, 4
$ disp <dbl> 160.0, 160.0, 108.0, 146.7, 140.8, 167.6, 167.6, 78.7, 75.7, 71.1~
$ hp   <dbl> 110, 110, 93, 62, 95, 123, 123, 66, 52, 65, 66, 109
$ drat <dbl> 3.90, 3.90, 3.85, 3.69, 3.92, 3.92, 3.92, 4.08, 4.93, 4.22, 4.08,~
$ wt   <dbl> 2.620, 2.875, 2.320, 3.190, 3.150, 3.440, 3.440, 2.200, 1.615, 1.~
$ qsec <dbl> 16.46, 17.02, 18.61, 20.00, 22.90, 18.30, 18.90, 19.47, 18.52, 19~
$ vs   <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1
$ gear <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
$ carb <dbl> 4, 4, 1, 2, 2, 4, 4, 1, 2, 1, 1, 2
```

```
mtcars_gear_4 |>
  relocate(cyl, disp, carb) |>
  glimpse()
```

```
Rows: 12
Columns: 11
$ cyl  <dbl> 6, 6, 4, 4, 4, 6, 6, 4, 4, 4, 4, 4
$ disp <dbl> 160.0, 160.0, 108.0, 146.7, 140.8, 167.6, 167.6, 78.7, 75.7, 71.1~
$ carb <dbl> 4, 4, 1, 2, 2, 4, 4, 1, 2, 1, 1, 2
$ mpg  <dbl> 21.0, 21.0, 22.8, 24.4, 22.8, 19.2, 17.8, 32.4, 30.4, 33.9, 27.3,~
$ hp   <dbl> 110, 110, 93, 62, 95, 123, 123, 66, 52, 65, 66, 109
$ drat <dbl> 3.90, 3.90, 3.85, 3.69, 3.92, 3.92, 3.92, 4.08, 4.93, 4.22, 4.08,~
$ wt   <dbl> 2.620, 2.875, 2.320, 3.190, 3.150, 3.440, 3.440, 2.200, 1.615, 1.~
$ qsec <dbl> 16.46, 17.02, 18.61, 20.00, 22.90, 18.30, 18.90, 19.47, 18.52, 19~
$ vs   <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1
$ gear <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
```

```
mtcars_gear_4 |>
  relocate(sort(names(mtcars_gear_4))) |>
  glimpse()
```

```
Rows: 12
Columns: 11
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1
$ carb <dbl> 4, 4, 1, 2, 2, 4, 4, 1, 2, 1, 1, 2
$ cyl  <dbl> 6, 6, 4, 4, 4, 6, 6, 4, 4, 4, 4, 4
$ disp <dbl> 160.0, 160.0, 108.0, 146.7, 140.8, 167.6, 167.6, 78.7, 75.7, 71.1~
$ drat <dbl> 3.90, 3.90, 3.85, 3.69, 3.92, 3.92, 3.92, 4.08, 4.93, 4.22, 4.08,~
$ gear <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
$ hp   <dbl> 110, 110, 93, 62, 95, 123, 123, 66, 52, 65, 66, 109
$ mpg  <dbl> 21.0, 21.0, 22.8, 24.4, 22.8, 19.2, 17.8, 32.4, 30.4, 33.9, 27.3,~
```

```
$ qsec <dbl> 16.46, 17.02, 18.61, 20.00, 22.90, 18.30, 18.90, 19.47, 18.52, 19~
$ vs   <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
$ wt   <dbl> 2.620, 2.875, 2.320, 3.190, 3.150, 3.440, 3.440, 2.200, 1.615, 1.~
```

```
# filter rows where cyl = 4
mtcars_gear_4 |>
  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
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
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_gear_4 |>
  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
Merc 240D     24.4   4  62
Merc 230      22.8   4  95
Merc 280      19.2   6 123
```

```
# select columns all variables except wt and hp
mtcars_gear_4 |>
  select(-wt, -hp) |>
  head()
```

```
               mpg cyl  disp drat  qsec vs am gear carb
Mazda RX4     21.0   6 160.0 3.90 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6 160.0 3.90 17.02  0  1    4    4
Datsun 710    22.8   4 108.0 3.85 18.61  1  1    4    1
Merc 240D     24.4   4 146.7 3.69 20.00  1  0    4    2
Merc 230      22.8   4 140.8 3.92 22.90  1  0    4    2
Merc 280      19.2   6 167.6 3.92 18.30  1  0    4    4
```

```
# select only variables starting with `c`
mtcars_gear_4 |>
  select(starts_with("c"))
```

```
                cyl carb
Mazda RX4         6    4
Mazda RX4 Wag     6    4
Datsun 710        4    1
Merc 240D         4    2
Merc 230          4    2
Merc 280          6    4
Merc 280C         6    4
Fiat 128          4    1
Honda Civic       4    2
Toyota Corolla    4    1
Fiat X1-9         4    1
Volvo 142E        4    2
```

```
# summarize avg mpg by number of cylinders
mtcars_gear_4 |>
  group_by(cyl) |>
  summarize(avg_mpg = mean(mpg))
```

```
# A tibble: 2 x 2
    cyl avg_mpg
  <dbl>   <dbl>
1     4    26.9
2     6    19.8
```

```
# create new column wt_kg, which is wt in kg
mtcars_gear_4 |>
  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
Merc 240D     3.190 1.446712
Merc 230      3.150 1.428571
Merc 280      3.440 1.560091
```

```
# 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_gear_4 |>
  rename(horsepower = hp) |>
  glimpse()
```

```
Rows: 12
Columns: 11
$ mpg        <dbl> 21.0, 21.0, 22.8, 24.4, 22.8, 19.2, 17.8, 32.4, 30.4, 33.9,~
$ cyl        <dbl> 6, 6, 4, 4, 4, 6, 6, 4, 4, 4, 4, 4
$ disp       <dbl> 160.0, 160.0, 108.0, 146.7, 140.8, 167.6, 167.6, 78.7, 75.7~
$ horsepower <dbl> 110, 110, 93, 62, 95, 123, 123, 66, 52, 65, 66, 109
$ drat       <dbl> 3.90, 3.90, 3.85, 3.69, 3.92, 3.92, 3.92, 4.08, 4.93, 4.22,~
$ wt         <dbl> 2.620, 2.875, 2.320, 3.190, 3.150, 3.440, 3.440, 2.200, 1.6~
$ qsec       <dbl> 16.46, 17.02, 18.61, 20.00, 22.90, 18.30, 18.90, 19.47, 18.~
$ vs         <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
$ am         <dbl> 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1
$ gear       <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
$ carb       <dbl> 4, 4, 1, 2, 2, 4, 4, 1, 2, 1, 1, 2
```

Exercise

See exercise:

- Section 9.4: *Generate and drop variables*
- Section 9.5: *Subsetting*

7.4.2. 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 `if_else()` 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$mpg_category <- if_else(mtcars_2$mpg > 20, "High", "Low")
```

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

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

The `mutate()` function is used to add the new variable, and `case_when()` is used to assign the values "four", "six", or "eight" to the new variable based on the number of cylinders in each car. Both functions are part of the `dplyr` package (see chapter Section 7.4.1).

7.4.3. Examining and cleaning data with the `janitor` package

The `janitor` package follows the principles of the tidyverse and works well with the pipe operator `|>`. The janitor functions has many usefull functions for the initial data exploration and cleaning that are essential when you load any new data set.

First, make sure the `janitor` package is installed and loaded:

7.4.3.1. Clean data.frame names with `clean_names()`

I call this function frequently when I read in new data. It handles problematic variable names, especially those that are so well-preserved by `readxl::read_excel()` and `readr::read_csv()`. For example, it does the following:

- Parses letter cases and separators to a consistent format.
- Handles special characters and spaces, including transliterating characters like `œ` to `oe`.
- Appends numbers to duplicated names
- Converts "%" to "percent" and "#" to "number" to retain meaning
- Spacing (or lack thereof) around numbers is preserved

To exemplify what it does, let's create some data with akward names and then clean them:

```
df_test <- as.data.frame(matrix(ncol = 6))
names(df_test) <- c(
  "firstName", "ábc@!*", "% successful (2009)",
  "REPEAT VALUE", "REPEAT VALUE", ""
)
df_cln <- df_test |>
  clean_names()
names(df_test)
```

```
[1] "firstName"           "ábc@!*"              "% successful (2009)"
[4] "REPEAT VALUE"        "REPEAT VALUE"        ""
```

```
names(df_cln)
```

```
[1] "first_name"               "abc"
[3] "percent_successful_2009" "repeat_value"
[5] "repeat_value_2"           "x"
```

7.4.3.2. Find duplicated values for specific combinations of variables with `get_dupes()`

`get_dupes` allows you to check for the indentifying variable. In other words, it shows you duplicates for specific combinations of variables.

For example, consider the following tibble:

```
df_panel <- tibble(
  country = c(rep("a", 3), rep("b", 3), rep("c", 3)),
  year = rep(1:3, 3),
  GDP = c(1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000)
)
df_panel
```

```
# A tibble: 9 x 3
  country  year   GDP
  <chr>   <int> <dbl>
1 a           1  1000
```

```
2 a             2  2000
3 a             3  3000
4 b             1  4000
5 b             2  5000
6 b             3  6000
7 c             1  7000
8 c             2  8000
9 c             3  9000
```

with

```
get_dupes(df_panel, country, year)
```

```
No duplicate combinations found of: country, year
```

```
# A tibble: 0 x 4
# i 4 variables: country <chr>, year <int>, dupe_count <int>, GDP <dbl>
```

we see that this is a panel dataset identified by a combination of `country` and `year`. Now let us introduce a duplicate and check again:

```
new_obs <- tibble(country = "b", year = 2, GDP = 5000)
df_panel_dup <- bind_rows(df_panel, new_obs)
get_dupes(df_panel_dup, country, year)
```

```
# A tibble: 2 x 4
  country  year dupe_count   GDP
  <chr>   <dbl>      <int> <dbl>
1 b           2          2  5000
2 b           2          2  5000
```

Tip 11: The `plm` package

Speaking of panel datasets, it's worth mentioning the `plm` package, which is excellent for managing such data. For example, you can use `is.pbalanced` to verify whether a panel is balanced, meaning it has the same years for all countries.

```
pacman::p_load(plm)
is.pbalanced(df_panel)
```

```
[1] TRUE
```

If your panel is unbalanced, you can use `make.pbalanced` to rectify it:

```
df_unbal <- df_panel |>
  filter(row_number() != 7)
df_unbal
```

```
# A tibble: 8 x 3
```

```
  country  year   GDP
  <chr>   <int> <dbl>
1 a           1  1000
2 a           2  2000
3 a           3  3000
4 b           1  4000
5 b           2  5000
6 b           3  6000
7 c           2  8000
8 c           3  9000
```

```r
is.pbalanced(df_unbal)
```

```
[1] FALSE
```

```r
df_unbal_balanced <- make.pbalanced(df_unbal)
df_unbal_balanced
```

```
  country year  GDP
1       a    1 1000
2       a    2 2000
3       a    3 3000
4       b    1 4000
5       b    2 5000
6       b    3 6000
7       c    1   NA
8       c    2 8000
9       c    3 9000
```

7.4.3.3. `remove_empty()` rows and columns

For cleaning Excel files that contain empty rows and columns after being read into R, `remove_empty` can be very helpful:

```r
q <- data.frame(
  v1 = c(1, NA, 3),
  v2 = c(NA, NA, NA),
  v3 = c("a", NA, "b")
)
q |>
  remove_empty(c("rows", "cols"))
```

```
  v1 v3
1  1  a
3  3  b
```

7.4.3.4. `remove_constant()` columns

Removes variables from data that contain only a single constant value (with an `na.rm` option to control whether `NA`s should be considered as different values from the constant).

```
a <- data.frame(good = 1:3, boring = "the same")
a
```

```
  good   boring
1    1 the same
2    2 the same
3    3 the same
```

```
a |>
  remove_constant()
```

```
  good
1    1
2    2
3    3
```

7.4.4. `tabyl()` - a better version of `table()`

`tabyl()` is a tidyverse-oriented replacement for `table()`. It counts combinations of one, two, or three variables, and then can be formatted with a suite of `adorn_*` functions to look just how you want. For example:

```
mtcars |>
  tabyl(gear, cyl) |>
  adorn_totals("col") |>
  adorn_percentages("row") |>
  adorn_pct_formatting(digits = 2) |>
  adorn_ns() |>
  adorn_title()
```

```
             cyl
 gear          4           6            8        Total
    3  6.67% (1) 13.33% (2) 80.00% (12) 100.00                        (15)
    4 66.67% (8) 33.33% (4)  0.00%  (0) 100.00                        (12)
    5 40.00% (2) 20.00% (1) 40.00%  (2) 100.00%  (5)
```

Learn more in the tabyls vignette.

7.5. User-defined functions and conflicts

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

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

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

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

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

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

```
square_it(5)
```

```
[1] 25
```

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

The formula of a growth rate is clear:

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

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

```r
lag(revenue)
```

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

```r
(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:

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

```
[1] "/usr/local/lib/R/site-library/dplyr"
```

```r
# 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:plm':

    between, lag, lead

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

    filter, lag

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

    intersect, setdiff, setequal, union
```

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

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

So, let's try again:

```
lag(revenue)
```

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

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

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

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

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

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

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

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

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

7.6. Example: How to explore a dataset

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

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

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

```
summary(df)
```

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

```
glimpse(df)
```

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

```
# look closer at variables

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

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

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

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

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

```

  blue  green    red yellow
     1      1      2      1
```

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

```
[1] 4
```

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

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

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

```

  blue  green    red yellow
   0.2    0.2    0.4    0.2
```

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

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

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

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

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

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

```
# install.packages("janitor")
require("janitor")
mtcars |>
  tabyl(cyl)
```

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

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

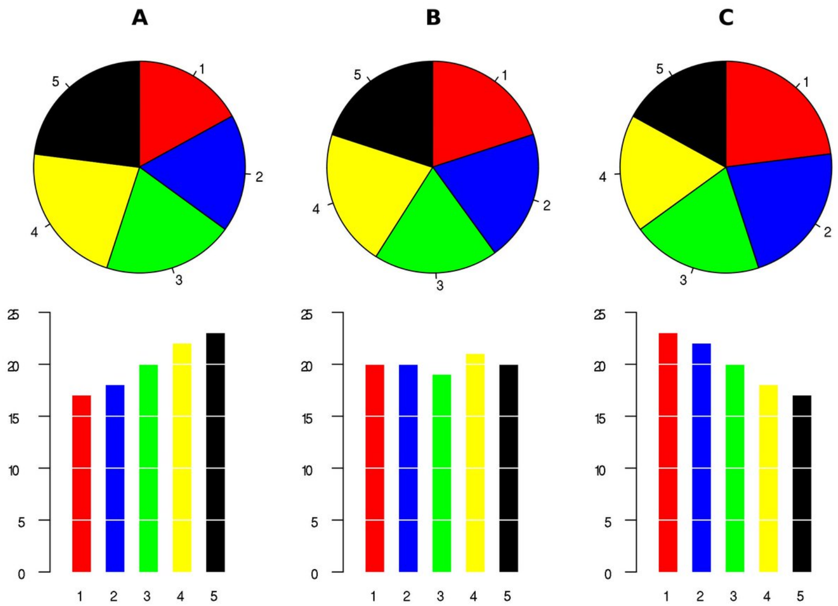
```
 cyl 52 62 65 66 91 93 95 97 105 109 110 113 123 150 175 180 205 215 230 245
   4  1  1  1  2  1  1  1  1   0   1   0   1   0   0   0   0   0   0   0   0
   6  0  0  0  0  0  0  0  0   1   0   3   0   2   0   1   0   0   0   0   0
   8  0  0  0  0  0  0  0  0   0   0   0   0   0   2   2   3   1 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

8. Visualize data

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

- Good graphs are easy to understand and eye catching.
- Graphs can be misleading and manipulative and that is opposing to the ideas of science. Thus, be responsible and honest.
- Minimize colors and other attention-grabbing elements that are not directly related to the data of interest. Worldwide, there are approximately 300 million color blind people. In particular, red, green or blue light are problematic to color blind people. Thus, better rely on color schemes that are designed for colorblind people.
- Don't truncate an axis or change the scaling within an axis just to make you your story more appealing. Show the full scale of the graph, then zoom to show the data of interest, if necessary.
- Label and describe your chart sufficiently so that everybody can fully understand the content of the shown data set and statistics without having to study the notes of the graph for too long.
- Don't do pie charts. They may look simple, but they're tricky to get right and there are usually better alternatives. Humans are not very good at comparing the size of angles and as there's no scale in pie plots, reading accurate values is difficult. Figure Figure 8.1 may proof this.

Figure 8.1.: Pie charts are problematic



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

More tips

- Data Visualization: Chart Dos and Don'ts (by Duke University)
- Graphs and Visualising Data by Oliver Kirchkamp. In particular, I highly recommend his handout [Kirchkamp, 2018]. It discusses many pitfalls of visualizing data, instructs how to do good graphs, and he shows the corresponding R code of all graphs.
- The *From Data to Viz* website leads you to the most appropriate graph for your data. It links to the code to build it and lists common caveats you should avoid.
- The R Graph Gallery and R CHARTS by R CODER shows graphs and the corresponding R code to replicate the graphs
- The work of Edward Tufte and his book *The Visual Display of Quantitative Information* [Tufte, 2022] are classical readings.

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

9. Collection of exercises

9.1. Import data with `c()`

Table 9.1 shows COVID for three states in Germany:

Table 9.1.: Covid cases and deaths till August 2022

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

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

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

Solution

The script uses the following functions: `c`, `data.frame`, `tibble`.

R script

```r
# Solution to excercise "Import data":

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tibble)

state <- c("BY", "NRW", "BW")
deaths <- c(4.92, 5.32, 3.69)
cases <- c(24111, 25466, 16145)
df_covid <- data.frame(state, deaths)
tbl_covid <- tibble(state, deaths)

tbl_covid

suppressMessages(pacman::p_unload(tibble))
```

Output of the R script

```r
# Solution to excercise "Import data":

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tibble)

state <- c("BY", "NRW", "BW")
deaths <- c(4.92, 5.32, 3.69)
cases <- c(24111, 25466, 16145)
df_covid <- data.frame(state, deaths)
tbl_covid <- tibble(state, deaths)

tbl_covid
```

```
# A tibble: 3 x 2
  state deaths
  <chr>  <dbl>
1 BY      4.92
2 NRW     5.32
3 BW      3.69
```

```r
suppressMessages(pacman::p_unload(tibble))
```

9.2. Filter and select observations

Set up R, RStudio, and R packages

Open this interactive tutorial and work through it.

The script uses among others the following functions: `filter`, `is.na`, `select`.

9.3. Base R,`%in%` operator, and the 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.

Solution

The script uses the following functions: `c`, `filter`, `identical`, `if_else`, `mutate`, `subset`, `transform`, `with`.

R script

```
# Base R or pipe
# exe_base_pipe.R
# Stephan Huber; 2023-05-08

# setwd("/home/sthu/Dropbox/hsf/test")
rm(list=ls())

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(datasets, tidyverse)

# a)
# Using the pipe |>
# Select rows where cyl is 4 or 6 and wt is less than 3.5
df1 <- mtcars |>
  filter(cyl %in% c(4, 6) & wt < 3.5)
df1

# Without the pipe |>
# Select rows where cyl is 4 or 6 and wt is less than 3.5
df2 <- subset(mtcars, cyl %in% c(4, 6) & wt < 3.5)
df2

# Check if the resulting dataframe is identical to the expected output
identical(df1, df2)


# b)
# Using the pipe |> and tidyverse (mutate)
df3 <- mtcars |>
  mutate(cyl_4_or_6 =
           if_else(cyl %in% c(4, 6) & wt < 3.5, TRUE, FALSE))
df3

# without pipe and with base R (transform)
df4 <- mtcars
df4$cyl_4_or_6 <- with(mtcars, cyl %in% c(4, 6) & wt < 3.5)

# Alternatively in one line:
df5 <- transform(mtcars, cyl_4_or_6 = cyl %in% c(4,6) & wt < 3.5)

# Check if the resulting dataframe is identical to the expected output
identical(df3, df4)
identical(df3, df5)

# unload packages
suppressMessages(pacman::p_unload(datasets, tidyverse))
```

9.4. Generate and drop variables

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

```
mtcars
```

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

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

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

Solution

The script uses the following functions: `as_tibble`, `if_else`, `mutate`, `rownames_to_column`, `select`.

R script

```r
# Generate and drop variables
# exe_genanddrop.R
# Stephan Huber; 2023-05-09

# setwd("/home/sthu/Dropbox/hsf/test")
rm(list = ls())

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(datasets, tidyverse)

# a)
mtcars_new <- mtcars |>
  rownames_to_column(var = "car") |>
  as_tibble() |>
  mutate(d_cyl_6to8 = if_else(cyl > 6, 1, 0))
mtcars_new

# b)
mtcars_new <- mtcars_new |>
  mutate(posercar = if_else(cyl > 6 & mpg < 18, 1, 0))
mtcars_new

# c)
mtcars_new <- mtcars_new |>
  select(-d_cyl_6to8)
mtcars_new

# unload packages
suppressMessages(pacman::p_unload(datasets, tidyverse))
```

Output of the R script

```r
# Generate and drop variables
# exe_genanddrop.R
# Stephan Huber; 2023-05-09

# setwd("/home/sthu/Dropbox/hsf/test")
rm(list = ls())

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(datasets, tidyverse)

# a)
mtcars_new <- mtcars |>
  rownames_to_column(var = "car") |>
  as_tibble() |>
  mutate(d_cyl_6to8 = if_else(cyl > 6, 1, 0))
mtcars_new
```

```
# A tibble: 32 x 13
   car            mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Mazda RX4     21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2 Mazda RX4 ~   21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3 Datsun 710    22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4 Hornet 4 D~   21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5 Hornet Spo~   18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6 Valiant       18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7 Duster 360    14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8 Merc 240D     24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9 Merc 230      22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10 Merc 280      19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# i 22 more rows
# i 1 more variable: d_cyl_6to8 <dbl>
```

```r
# b)
mtcars_new <- mtcars_new |>
  mutate(posercar = if_else(cyl > 6 & mpg < 18, 1, 0))
mtcars_new
```

```
# A tibble: 32 x 14
   car            mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Mazda RX4     21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2 Mazda RX4 ~   21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3 Datsun 710    22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4 Hornet 4 D~   21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5 Hornet Spo~   18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6 Valiant       18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7 Duster 360    14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8 Merc 240D     24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9 Merc 230      22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10 Merc 280      19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# i 22 more rows
```

9.5. Subsetting

1. Check to see if you have the `mtcars` dataset by entering `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` function.
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 `|>`.

Solution

The script uses the following functions: `arrange`, `class`, `dim`, `ends_with`, `filter`, `mutate`, `ncol`, `nrow`, `rename`, `rename_all`, `rownames_to_column`, `select`.

R script

```
# Subsetting with \R
# exe_subset.R
# Stephan Huber; 2022-06-07

# setwd("/home/sthu/Dropbox/hsf/22-ss/dsda/work/")
rm(list = ls())

# 0
# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, dplyr, tibble)

# 1
mtcars

# 2
cars <- mtcars

# 3
class(cars)

# 4
dim(cars)

# Alternative
ncol(cars)
nrow(cars)

# 5
cars <- rename(cars, MPG = mpg)

# 6
cars <- rename_all(cars, toupper)
# if you like lower cases:
# cars <- rename_all(cars, tolower)

# 7
cars <- rownames_to_column(mtcars, var = "car")

# 8
pvars <- select(cars, car, ends_with("p"))

# 9
carsSub <- select(cars, car, wt, qsec, hp)

# 10
dim(carsSub)

# 11
carsSub <- rename_all(carsSub, toupper)

# 12
cars_mpg <- filter(cars, mpg > 20)
dim(cars_mpg)
```

Output of the R script

```
# Subsetting with \R
# exe_subset.R
# Stephan Huber; 2022-06-07

# setwd("/home/sthu/Dropbox/hsf/22-ss/dsda/work/")
rm(list = ls())

# 0
# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, dplyr, tibble)

# 1
mtcars
```

9.6. Consumer prices over time

1. Read in the following data:

```
https://raw.githubusercontent.com/hubchev/courses/main/dta/PCEPI.csv
```

The data stem from the Federal Reserve Bank of St. Louis (FRED).

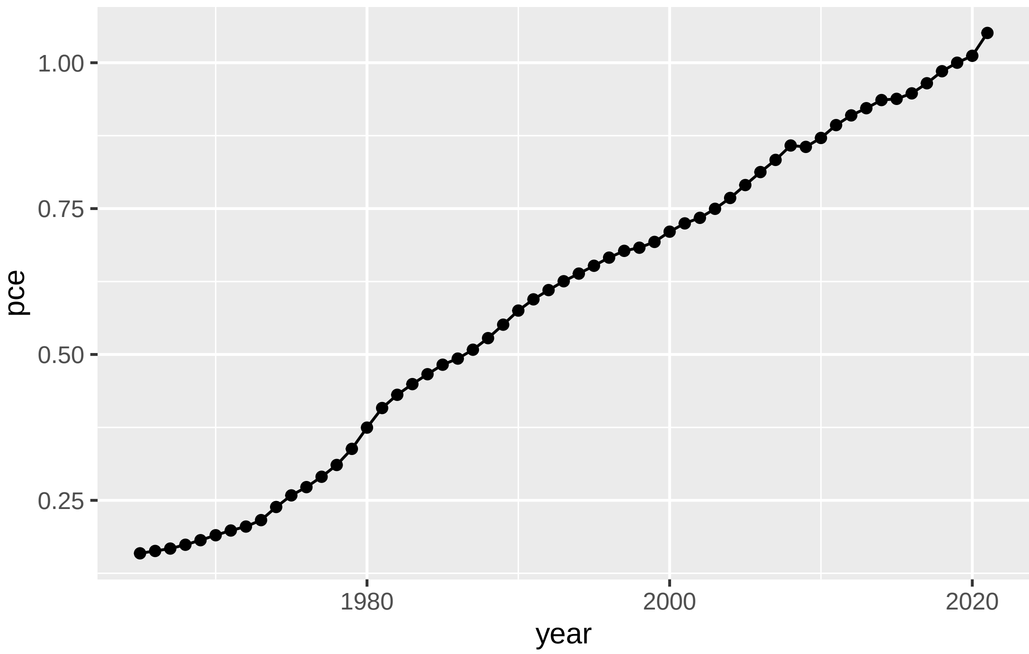
> **Tip**
>
> You should always try to use most recent data and you should not work outside of R. Thus, it would be optimal to download the data directly from FRED and using a R package. If this would be serious research, I would not recommend to use the data that I have downloaded from FRED, I'd recommend to use the `fredr` package which allows to fetch the observation directly from FRED database using the function `fredr`.

Here is an excerpt of the data:

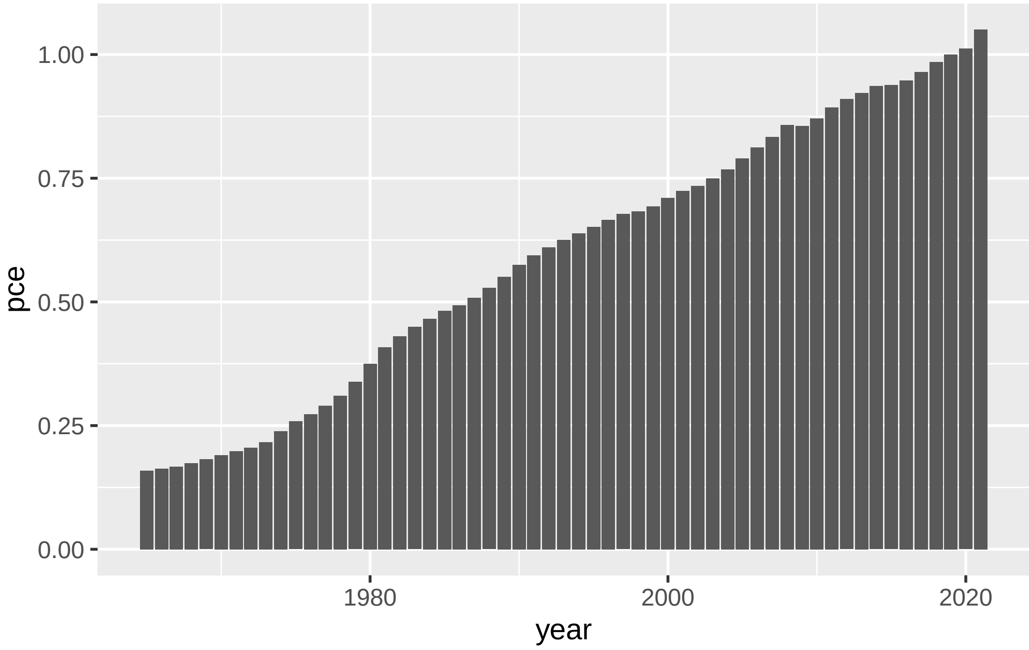
```
        DATE PCEPI_NBD20190101
1 1965-01-01          15.92229
2 1966-01-01          16.32477
3 1967-01-01          16.73491
4 1968-01-01          17.38977
5 1969-01-01          18.17313
6 1970-01-01          19.02267
```

2. Divide the variable `PCEPI_NBD20190101` by 100 and name the resulting variable `pce`. Additionally, generate a new variable `year` that contains the respective year. Save the modified dataframe as `pce_cl`.

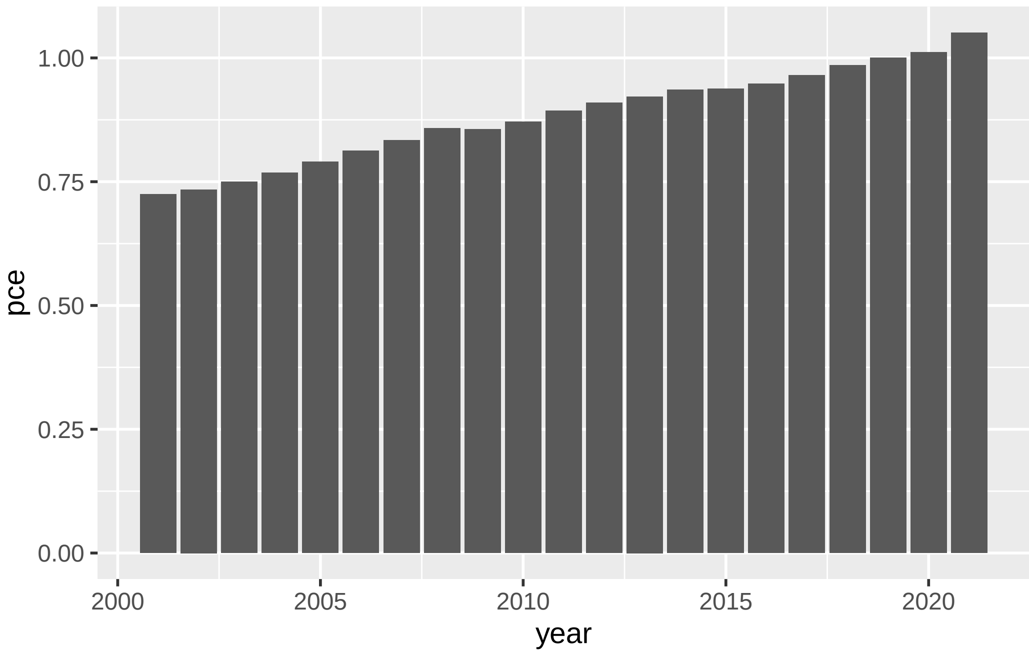
3. Make the following plot:



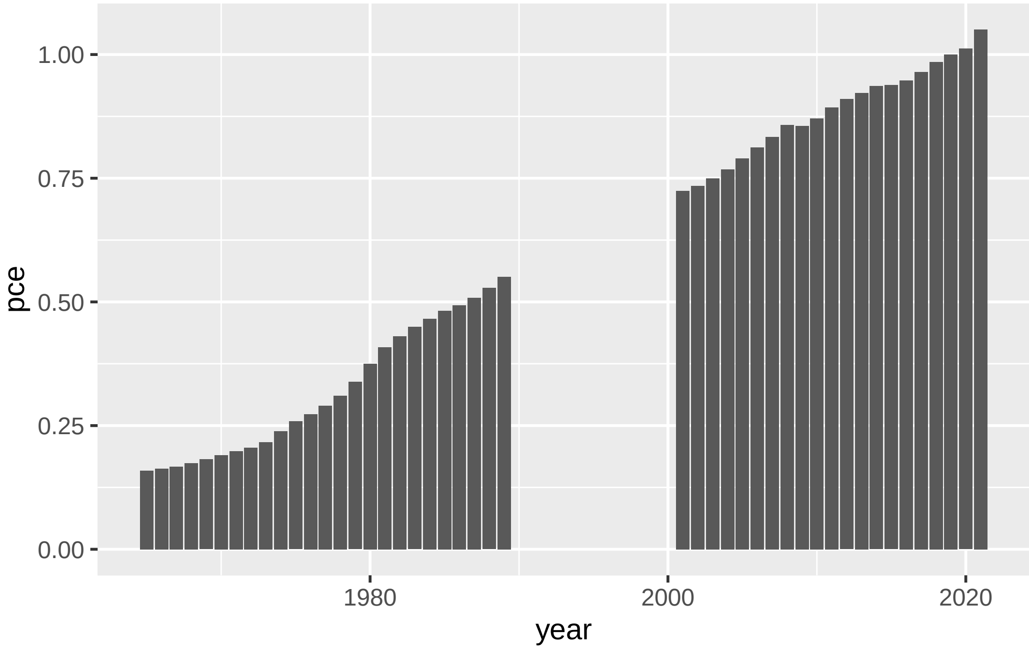
3. Make the following plot:



4. Make the following plot:



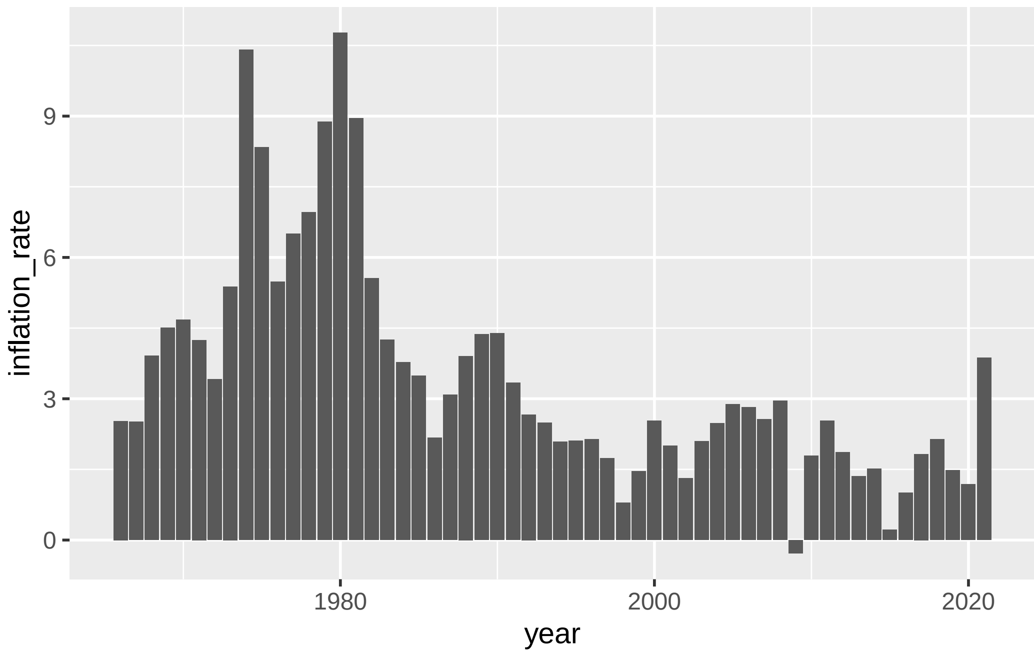
5. Make a plot of inflation for all years except the 90s:



6. Calculate the yearly inflation. Here is an excpert of how the data should look like:

```
  year       pce inflation_rate
1 1965 0.1592229             NA
2 1966 0.1632477       2.527777
3 1967 0.1673491       2.512378
4 1968 0.1738977       3.913137
5 1969 0.1817313       4.504717
6 1970 0.1902267       4.674704
```

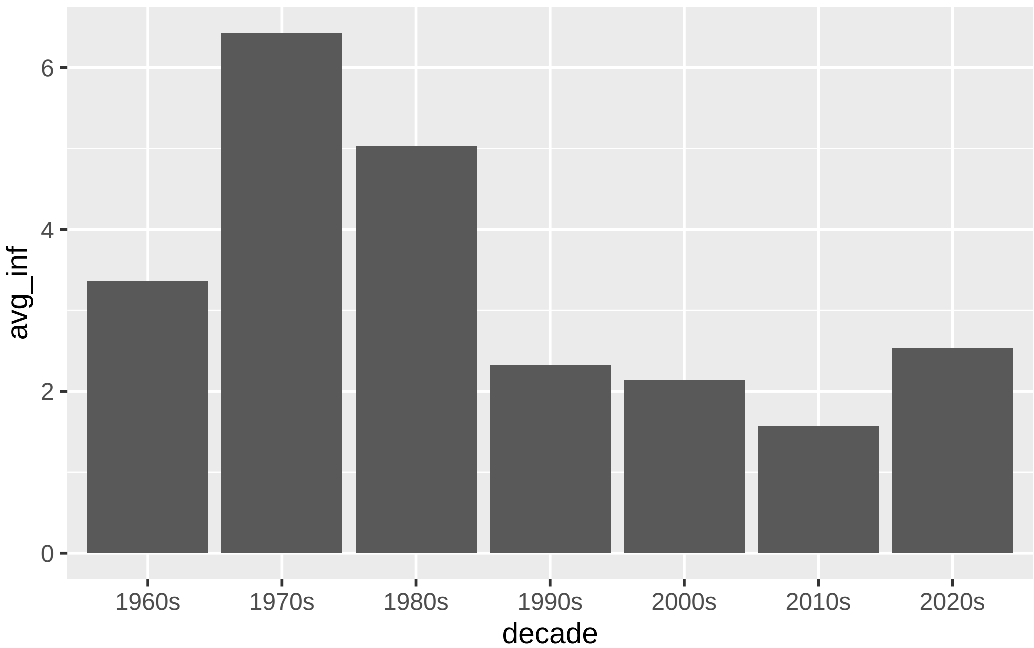
7. Plot the yearly inflation rate:



8. Calculate the average inflation rate over all years.
9. Calculate the average inflation rate for each decade:

```
# A tibble: 7 x 2
  decade avg_inf
  <chr>    <dbl>
1 1960s     3.36
2 1970s     6.43
3 1980s     5.03
4 1990s     2.32
5 2000s     2.14
6 2010s     1.57
7 2020s     2.53
```

10. Make the following plot:



Solution

The script uses the following functions: `aes`, `as.integer`, `case_when`, `filter`, `geom_bar`, `geom_line`, `geom_point`, `ggplot`, `group_by`, `lag`, `mean`, `mutate`, `read.csv`, `select`, `str_sub`, `summarize`.

R script

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, janitor, expss)
# setwd("~/yourdirectory-of-choice")
rm(list = ls())


# read in raw data
# PCEPI <- read.csv("PCEPI.csv")
PCEPI <- read.csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/PCEPI.csv"

# clean data
pce_cl <- PCEPI |>
  mutate(pce = PCEPI_NBD20190101/100) |>
  mutate(year = str_sub(DATE, 1, 4)) |>
  mutate(year = as.integer(year)) |>
  select(pce, year)

# make a plot
pce_cl |>
  ggplot( aes(x=year, y=pce))+
  geom_line() +
  geom_point()

# make a barplot
pce_cl |>
  ggplot( aes(x=year, y=pce))+
  geom_bar(stat="identity")

# make a plot for all years from 2000 onwards
pce_cl |>
  filter(year > 2000 ) |>
  ggplot( aes(x=year, y=pce)) +
  geom_bar(stat="identity")

# make a plot for all years except the 90s
pce_cl |>
  filter(year > 2000 | year <1990) |>
  ggplot( aes(x=year, y=pce)) +
  geom_bar(stat="identity")

# calculate yearly inflation
pce_cl <- pce_cl |>
  mutate(inflation_rate = (pce/lag(pce)-1)*100 )

# plot the inflation rate
pce_cl |>
  ggplot( aes(x=year, y=inflation_rate))+
  geom_bar(stat="identity")

# what is the avergage inflation rate
pce_cl |>
  summarize(avg_mpg = mean(inflation_rate, na.rm = TRUE))

# make a variable that indicates the decades 1 for 60s, 2 for 70s, etc.
```

9.7. Load the Stata dataset "auto" using R

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

Solution

The script uses the following functions: `aes`, `arrange`, `desc`, `dir.create`, `dir.exists`, `filter`, `geom_point`, `geom_text_repel`, `ggplot`, `head`, `ifelse`, `max`, `min`, `min_max_norm`, `mutate`, `na.omit`, `read_dta`, `select`, `tail`, `theme_minimal`, `xlab`, `ylab`.

R script

```
# Load the required libraries
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, haven, ggrepel)

# setwd("~/Dropbox/hsf/23-ws/R_begin")

rm(list = ls())

# Read the Stata dataset
auto <- read_dta("http://www.stata-press.com/data/r8/auto.dta")


# Create a scatter plot of price vs. weight
scatter_plot <- ggplot(auto, aes(x = mpg, y = price, label = make)) +
  geom_point() +
  geom_text_repel() +
  xlab("Miles per Gallon") +
  ylab("Price in Dollar") +
  theme_minimal()

scatter_plot

# Create "fig" directory if it doesn't already exist
if (!dir.exists("fig")) {
  dir.create("fig")
}

# Save the scatter plot in different formats
# ggsave("fig/scatter_plot.png", plot = scatter_plot, device = "png")
# ggsave("fig/scatter_plot.pdf", plot = scatter_plot, device = "pdf")

# Create 'lp100km' variable for fuel consumption
n_auto <- auto |>
  mutate(lp100km = (1 / (mpg * 1.6 / 3.8)) * 100)

# Create 'larger6000' dummy variable
n_auto <- n_auto |>
  mutate(larger6000 = ifelse(price > 6000, 1, 0))



# Normalize variables

## Do it slowly
n_auto <- n_auto |>
  mutate(sprice = (price - min(auto$price)) / (max(auto$price) - min(auto$price)))

n_auto <- n_auto |>
  filter(larger6000 == 0)

## Do it with a self-written function
min_max_norm <- function(x) {
  (x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
}
```

9.8. DatasauRus

Figure 9.1.: The logo of the DatasauRus package

Source: https://github.com/jumpingrivers/datasauRus _

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

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

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

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

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

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

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

h) What can you conclude?

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

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

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

Solution

The script uses the following functions: `aes`, `cor`, `dim`, `dir.create`, `dir.exists`, `facet_wrap`, `filter`, `geom_point`, `ggplot`, `ggsave`, `glimpse`, `group_by`, `head`, `labs`, `length`, `mean`, `median`, `paste`, `paste0`, `round`, `sd`, `select`, `summarise`, `summary`, `theme`, `theme_bw`, `unique`, `view`.

R script

```
# setwd("/home/sthu/Dropbox/hsf/23-ws/ds_mim/")
rm(list = ls())

# Load the packages datasauRus and tidyverse. If necessary, install these packages.

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(datasauRus, tidyverse)

# The packagedatasauRus comes with a dataset in two different formats:
#   datasaurus_dozen and datasaurus_dozen_wide. Store them as ds and ds_wide.

ds <- datasaurus_dozen
ds_wide <- datasaurus_dozen_wide

# Open and read the R vignette of the datasauRus package.
#   Also open the R documentation of the dataset datasaurus_dozen.

??datasaurus

# Explore the dataset: What are the dimensions of this dataset? Look at the descriptive

ds
dim(ds)
head(ds)
glimpse(ds)
view(ds)
summary(ds)

# 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.

unique(ds$dataset)

unique(ds$dataset) |>
  length()

# 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.

ds |>
  group_by(dataset) |>
  summarise(
    mean_x = mean(x),
    mean_y = mean(y)
  )

# Compute the standard deviation, the correlation, and the median in the same way. Round

ds |>
  group_by(dataset) |>
  summarise(
```

9.9. 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.

Solution

The script uses the following functions: `aes`, `as.numeric`, `c`, `cor`, `describe`, `diff`, `filter`, `gather`, `geom_point`, `geom_text`, `ggarrange`, `ggplot`, `ggtitle`, `group_by`, `head`, `ifelse`, `lag`, `list`, `lm`, `log`, `mean`, `mutate`, `names`, `read_dta`, `select`, `set_label`, `starts_with`, `stat_smooth`, `stat.desc`, `str`, `subset`, `substr`, `summarise`, `summarise_all`, `summarise_at`, `summary`, `tab_model`, `tail`, `tbl_summary`, `vars`, `view`.

R script

```
# Convergence

# set working directory
# setwd("/home/sthu/Dropbox/hsf/github/courses/")

# clear the environment
rm(list = ls())

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(
  haven, tidyverse, vtable, gtsummary, pastecs, Hmisc,
  sjlabelled, tis, ggpubr, sjPlot, psych
)

# import data
data <- read_dta("https://github.com/hubchev/courses/raw/main/dta/convergence.dta")

# inspect data
names(data)
str(data)
data
head(data)
tail(data)
summary(data)
view(data)

# library(vtable)
# vtable(data, missing=TRUE)

# library(pastecs)
stat.desc(data)

# library(Hmisc)
describe(data)

# library(gtsummary)
tbl_summary(data)

# check the assignments of countries to continents
data |>
  select(country, africa, asia, weurope) |>
  view()

data <- mutate(data, x_1 = africa + asia + weurope)

data |>
  filter(x_1 == 0) |>
  select(africa, asia, weurope, country) |>
  view()

# correct the assignment manually
data$weurope[data$country == "Austria"] <- 1
data$weurope[data$country == "Greece"] <- 1
```

9.10. Unemployment and GDP in Germany and France

The following exercise was a former exam.

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

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

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

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

(1) Set your working directory.

(2) Clear your global environment.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

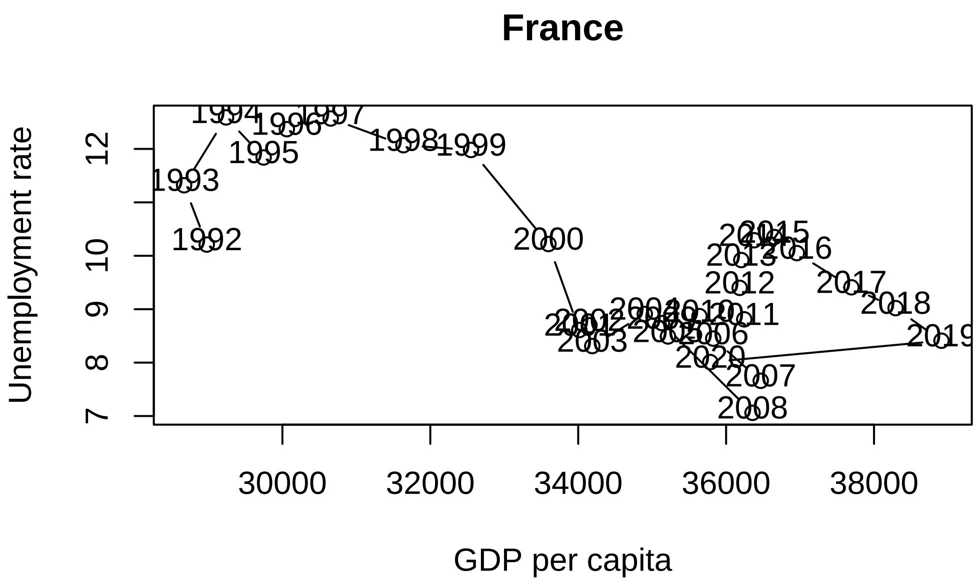
```
# A tibble: 2 x 4
  nation  `sd(unemployment)` `mean(unemployment)`   cov
  <chr>                <dbl>                <dbl> <dbl>
1 France                1.58                 9.75 0.162
2 Germany               2.37                 7.22 0.328
```

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

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

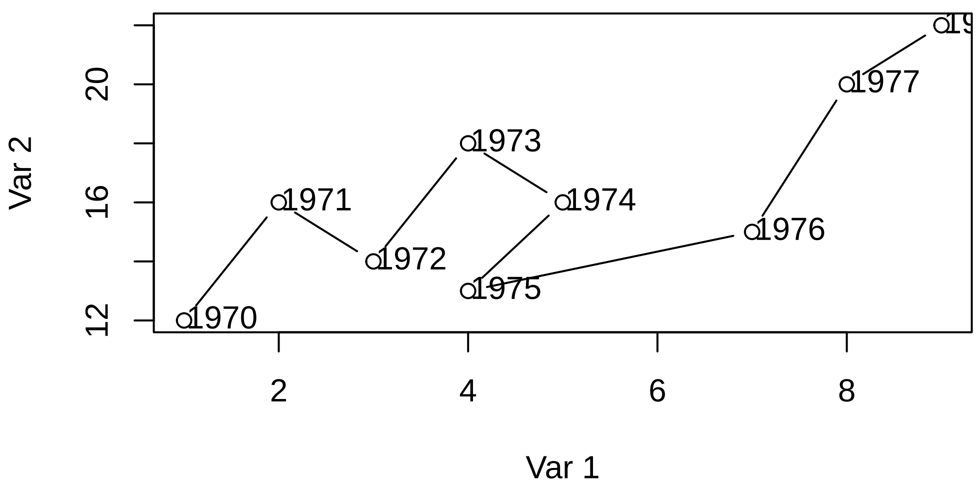
(21) Create a chart (bar chart, line chart, or scatter plot) that shows the unemployment rate of **Germany** over the available years. Label the chart 'Germany' with 'ggtitle("Germany")'. Please note that you may choose any type of graphical representation. (Hint: Below you can see one of many `|>` of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

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

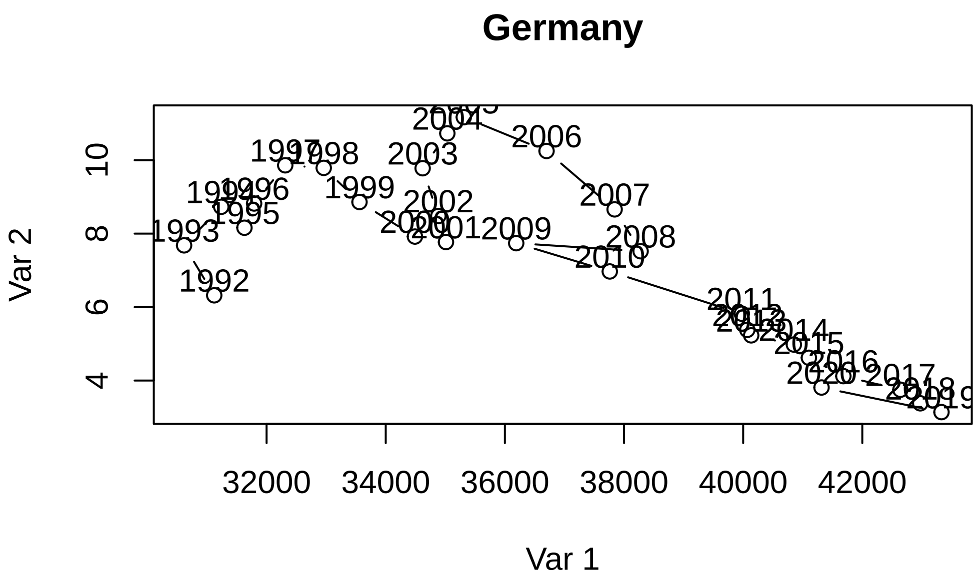


Suppose you want to visualize the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany as well.
Suppose further that you have found the following lines of code that create the kind of chart you are looking for.

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

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



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



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

Solution

The script uses the following functions: `aes`, `c`, `dim`, `filter`, `geom_line`, `ggplot`, `ggtitle`, `group_by`, `head`, `load`, `max`, `mean`, `mutate`, `plot`, `rename`, `sd`, `select`, `summarise`, `tail`, `text`, `title`, `url`.

R script

```
# setwd("/home/sthu/Dropbox/hsf/exams/22-11/scr/")

rm(list = ls())

if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, ggpubr, sjPlot)

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

head(df, 8)

tail(df, 1)

# panel data set
# date and country.x

observations_df <- dim(df)

df <- rename(df, nation = country.x)
df <- rename(df, year = date)

df <- df |>
  select(nation, year, gdp, pop, gdppc, unemployment)

df <- df |>
  mutate(gdp_pc = gdp / pop)

df <- df |> filter(nation == "Germany" | nation == "France")

df |>
  group_by(nation) |>
  summarise(mean(unemployment), mean(gdppc))

df |>
  filter(year == 2020) |>
  group_by(nation) |>
  summarise(mean(unemployment), mean(gdppc))

df |>
  group_by(nation) |>
  summarise(max(unemployment), max(gdppc))

df |>
  group_by(nation) |>
  summarise(sd(gdppc), sd(unemployment))

df |>
  group_by(nation) |>
  summarise(sd(unemployment), mean(unemployment), cov = sd(unemployment) / mean(unemploy

df |>
  group_by(nation) |>
  summarise(sd(gdppc), mean(gdppc), cov = sd(gdppc) / mean(gdppc))
```

9.11. Import data and write a report

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

Here are the required steps:

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

Solution

The script uses the following functions: `aes`, `download.file`, `geom_line`, `ggplot`, `pivot_longer`, `read_excel`, `unzip`.

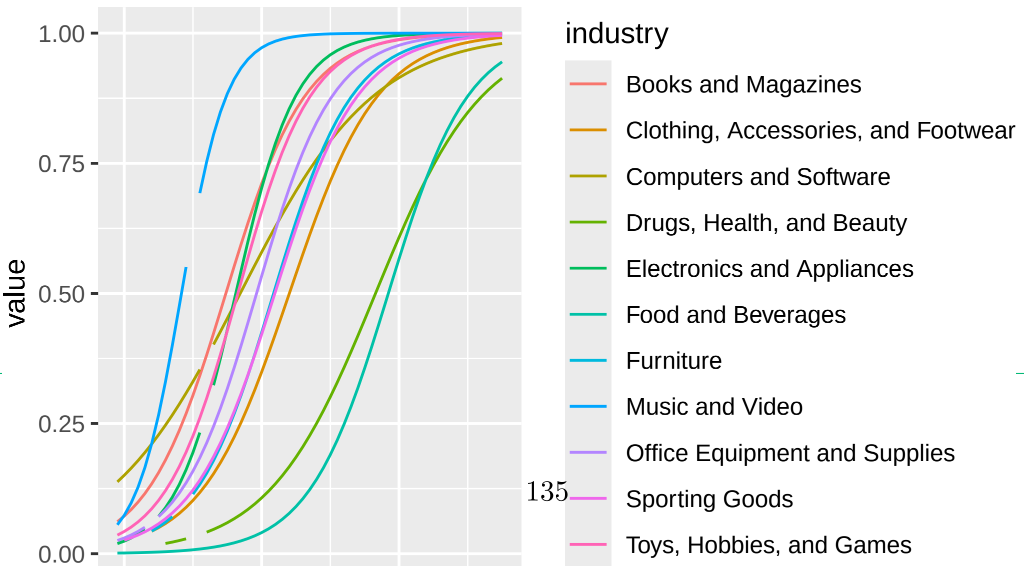
R script

```
# setwd("~/Dropbox/hsf/courses/Rlang/hortacsu")

rm(list = ls())


# install and load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, readxl)


# Define the URL of the ZIP file
zip_f <- "https://github.com/hubchev/courses/raw/main/dta/113962-V1.zip"

# Download the ZIP file
download.file(zip_f, destfile = "113962-V1.zip")

# Unzip the contents
unzip("113962-V1.zip")

df_curves <- read_excel("Hortacsu_Syverson_JEP_Retail/diffusion_curves_figure.xlsx",
  sheet = "Data and Predictions", range = "N3:Y60"
)

df <- df_curves |>
  pivot_longer(
    cols = "Music and Video":"Food and Beverages",
    names_to = "industry",
    values_to = "value"
  )

# Plot
df |>
  ggplot(aes(x = Year, y = value, group = industry, color = industry)) +
  geom_line()


#  unload packages
suppressMessages(pacman::p_unload(tidyverse, readxl))
```

Output of the R script

```r
# setwd("~/Dropbox/hsf/courses/Rlang/hortacsu")

rm(list = ls())


# install and load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, readxl)


# Define the URL of the ZIP file
zip_f <- "https://github.com/hubchev/courses/raw/main/dta/113962-V1.zip"

# Download the ZIP file
download.file(zip_f, destfile = "113962-V1.zip")

# Unzip the contents
unzip("113962-V1.zip")

df_curves <- read_excel("Hortacsu_Syverson_JEP_Retail/diffusion_curves_figure.xlsx",
  sheet = "Data and Predictions", range = "N3:Y60"
)

df <- df_curves |>
  pivot_longer(
    cols = "Music and Video":"Food and Beverages",
    names_to = "industry",
    values_to = "value"
  )

# Plot
df |>
  ggplot(aes(x = Year, y = value, group = industry, color = industry)) +
  geom_line()
```

```
Warning: Removed 18 rows containing missing values or values outside the scale range
(`geom_line()`).
```



9.12. Explain the weight of students

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")
```

```
Attaching package: 'haven'


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

    is.labelled, read_spss
```

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

Solution

The script uses the following functions: `aes`, `c`, `coef`, `fitted`, `geom_abline`, `geom_point`, `ggplot`, `head`, `library`, `lm`, `plot`, `read.csv`, `residuals`, `show`, `stat_smooth`, `subset`, `summary`, `tab_model`.

R script

```
## ---- echo = TRUE-------------------------------------------------------
# install and load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, haven)

classdata <- read.csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/classd

head(classdata)

## ---- echo = TRUE-------------------------------------------------------

summary(classdata)

## ----pressure, echo=TRUE------------------------------------------------
library("ggplot2")
ggplot(classdata, aes(x = height, y = weight)) +
  geom_point()


## ---- echo=TRUE---------------------------------------------------------
ggplot(classdata, aes(x = height, y = weight)) +
  geom_point() +
  stat_smooth(formula = y ~ x, method = "lm", se = FALSE, colour = "red", linetype = 1)


## ---- echo=TRUE---------------------------------------------------------
## baseline regression  model
model <- lm(weight ~ height + sex, data = classdata)
show(model)
interm <- model$coefficients[1]
slope <- model$coefficients[2]
interw <- model$coefficients[1] + model$coefficients[3]

## ---- echo=TRUE---------------------------------------------------------
summary(model)

## ---- echo=TRUE---------------------------------------------------------
ggplot(classdata, aes(x = height, y = weight, shape = sex)) +
  geom_point() +
  geom_abline(slope = slope, intercept = interw, linetype = 2, size = 1.5) +
  geom_abline(slope = slope, intercept = interm, linetype = 2, size = 1.5) +
  geom_abline(slope = coef(model)[[2]], intercept = coef(model)[[1]])


## ---- echo=TRUE---------------------------------------------------------

ggplot(classdata, aes(x = height, y = weight, shape = sex)) +
  geom_point(aes(size = 2)) +
  stat_smooth(
    formula = y ~ x, method = "lm",
    se = FALSE, colour = "red", linetype = 1
  )
```

9.13. Calories and weight

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

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

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

Data Description:

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

e) Which type of data do we have here? (Panel data, repeated cross-sectional data, cross-sectional data, time Series data)
f) Store each of the five variables in a vector and put all five variables into a dataframe with the title `df`. If you fail here, read in the data using this line of code:

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

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

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

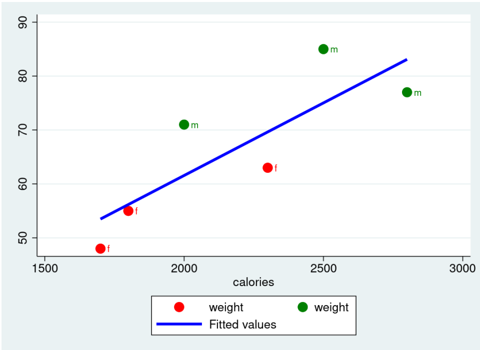
g) Show for all numerical variables the summary statistics including the mean, median, minimum, and the maximum.

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

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

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

Figure 9.2.: Weight vs. Calories



k) Make a scatterplot matrix containing all numerical variables.

l) Calculate the Pearson Correlation Coefficient of the two variables

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

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

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

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

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

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

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

Residual standard error: 8.68 on 4 degrees of freedom
Multiple R-squared:  0.6859,    Adjusted R-squared:  0.6074
```

```
F-statistic: 8.735 on 1 and 4 DF,  p-value: 0.04174
```

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

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

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

Solution

The script uses the following functions: `aes`, `c`, `cor`, `data.frame`, `geom_point`, `geom_text`, `ggplot`, `group_by`, `lm`, `mean`, `plot`, `read_csv`, `sd`, `stat_smooth`, `summarise`, `summary`.

R script

```
# 1
# Stephan Huber, 000, 2020-May-30

# 2
# setwd("/home/sthu/Dropbox/hsf/22-ss/dsb_bac/work/")

# 3
rm(list = ls())

# 4
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, haven)

# 5
# cross-section

# 6
sex <- c("f", "f", "f", "m", "m", "m")
age <- c(21, 19, 23, 18, 20, 61)
weight <- c(48, 55, 63, 71, 77, 85)
calories <- c(1700, 1800, 2300, 2000, 2800, 2500)
sport <- c(60, 120, 180, 60, 240, 30)
df <- data.frame(sex, age, weight, calories, sport)

# write_csv(df, file = "/home/sthu/Dropbox/hsf/exams/21-04/stuff/df.csv")
# write_csv(df, file = "/home/sthu/Dropbox/hsf/github/courses/dta/df-calories.csv")
df <- read_csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/df-calories.c

# 7
summary(df)

# 8
df |>
  group_by(sex) |>
  summarise(
    mcal = mean(calories),
    sdcal = sd(calories),
    mweight = mean(weight),
    sdweight = sd(weight)
  )

# 9
# discussed in class

# 10
# Many things can be mentioned here such as the use of colors
# (red/blue is not a good choice for color blind people),
# the legend makes no sense as red and green both refer to \textit{sport},
# the label of `f' and `m' is not explained in the legend,
# rotating the labels of the y-axis would increase readability, and
# both axes do not start at zero which is hard to see.
# Also, it is a common to draw the variable you want to explain
# (here: calories) on the y-axis.
```

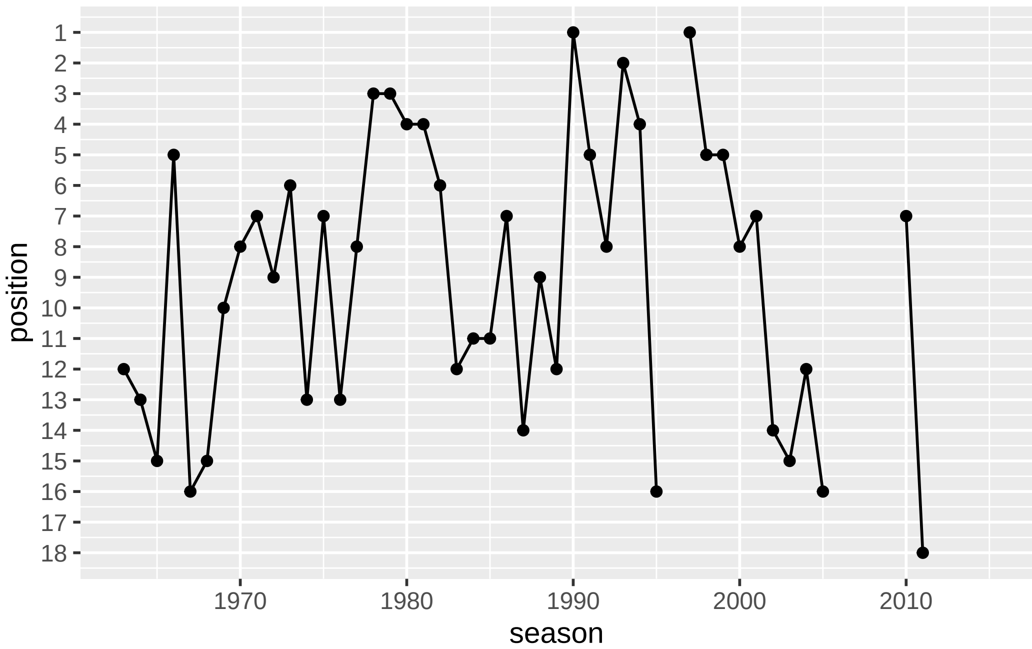
9.14. Bundesliga

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

1. Set your working directory.
2. Clear th environment.
3. Install and load the `bundesligR` and `tidyverse`.
4. Read in the data `bundesligR` as a tibble.
5. Replace "Bor. Moenchengladbach" with "Borussia Moenchengladbach."
6. Check for the data class.
7. View the data.
8. Glimpse on the data.
9. Show the first and last observations.
10. Show summary statistics to all variables.
11. How many teams have played in the league over the years?
12. Which teams have played Bundesliga so far?
13. How many teams have played Bundesliga?
14. How often has each team played in the Bundesliga?
15. Show summary statistics of variable `Season` only.
16. Show summary statistics of all numeric variables (`Team` is a character).
17. What is the highest number of points ever received by a team? Show only the name of the club with the highest number of points ever received.
18. Create a new tibble using `liga` removing the variable `Pts_pre_95` from the data.
19. Create a new tibble using `liga` renaming W, D, and L to Win, Draw, and Loss. Additionally rename GF, GA, GD to Goals_shot, Goals_received, Goal_difference.
20. Create a new tibble using `liga` without the variable `Pts_pre_95` and only observations before the year 1996.
21. Remove the three tibbles just created from the environment.
22. Rename all variables of `liga` to lowercase and store it as `dfb`.
23. Show the winner and the runner up after the year 2010. Additionally show the points received.
24. Create a variable that counts how often a team was ranked first.
25. How often has each team played in the Bundesliga?
26. Make a ranking that shows which team has played the Bundesliga most often.
27. Add a variable to `dfb` that contains the number of appearances of a team in the league.
28. Create a number that indicates how often a team has played Bundesliga in a given year.
29. Make a ranking with the number of titles of all teams that ever won the league.

30. Create a numeric identifying variable for each team.
31. When a team is in the league, what is the probability that it wins the league?
32. Make a scatterplot with points on the y-axis and position on the x-axis.
33. Make a scatterplot with points on the y-axis and position on the x-axis. Additionally, only consider seasons with 18 teams and add lines that make clear how many points you needed to be placed in between rank 2 and 15.
34. Remove all objects from the environment except `dfb` and `liga`.
35. In Figure Figure 9.3, the ranking history of *1. FC Kaiserslautern* is shown. Replicate that plot.

Figure 9.3.: Ranking history: 1. FC Kaiserslautern



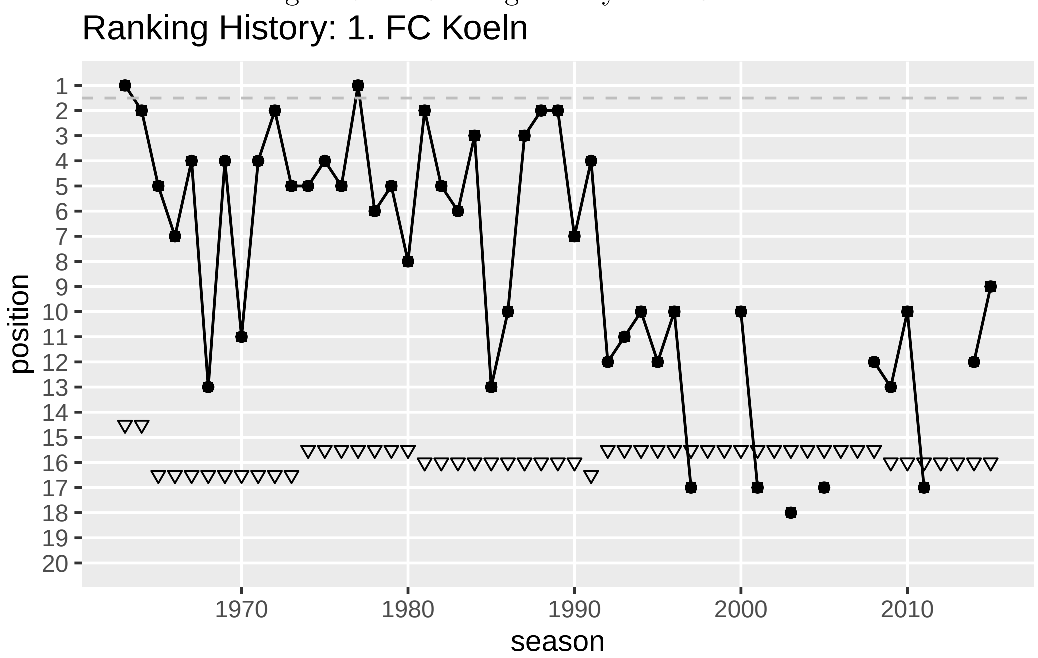
34. In Figure Figure 9.4, I made the graph a bit nicer. Can you spot all differences and can you guess what the dashed line and the triangles mean? How could the visualization be improved further? Replicate the plot.

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

Solution

The script uses the following functions: `aes`, `arrange`, `as_tibble`, `as.numeric`, `between`, `c`, `case_when`, `class`, `complete`, `desc`, `dir.create`, `dir.exists`, `element_blank`, `facet_wrap`, `factor`, `filter`, `geom_hline`, `geom_line`, `geom_point`, `geom_vline`, `ggplot`, `ggsave`, `glimpse`, `group_by`, `head`, `ifelse`, `is.na`, `labs`, `list`, `max`, `mutate`, `n_distinct`, `paste`, `print`, `rename`, `rename_all`, `row_number`, `scale_x_continuous`, `scale_y_continuous`, `scale_y_reverse`, `select`, `seq`, `setdiff`, `slice_head`, `subset`, `sum`, `summarise`, `summary`, `table`, `tail`, `theme`, `theme_classic`, `theme_minimal`, `unique`, `unlink`, `view`, `xlim`.

Figure 9.4.: Ranking history: 1. FC Köln



R script

```
# In dfb.R I analyze German soccer results

# set working directory
# setwd("~/Dropbox/hsf/23-ws/dsda/scripts")

# clear environment
rm(list = ls())

# (Install and) load packagages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(
  bundesligR,
  tidyverse
)

# Read in the data as tibble
liga <- as_tibble(bundesligR)

# ----------------------------------------
# !!! ERRORS / ISSUES:
# "Borussia Moenchengladbach" is also entitled "Bor. Moenchengladbach"!
# Leverkusen is falsly entitled "SV Bayer 04 Leverkusen"
# Uerdingen has changed its name several times
# Stuttgarter Kickers are named differently

# How often is "Bor. Moenchengladbach" in the data?
sum(liga$Team == "Bor. Moenchengladbach")

# show the entries
liga |>
  filter(Team == "Bor. Moenchengladbach")

# Replace "Bor. Moenchengladbach" with "Borussia Moenchengladbach"
```

9.15. Okun's Law

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

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

(1) Set your working directory.

(2) Clear your global environment.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

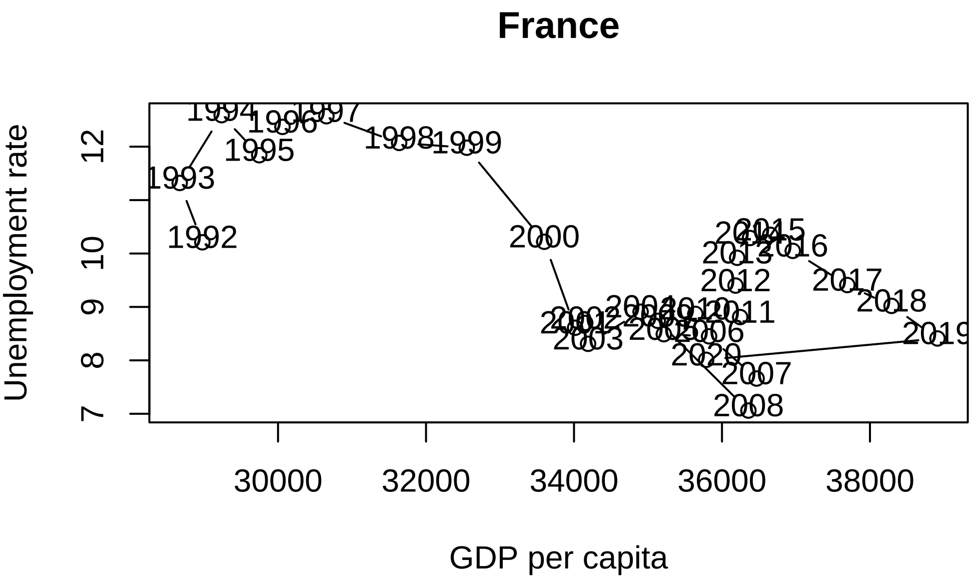
(20) Write down the R code to calculate the coefficient of variation (COV) for the **GDP per capita** in Germany and France. (Hint: See below for what your result should should look like.)

look like.)

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

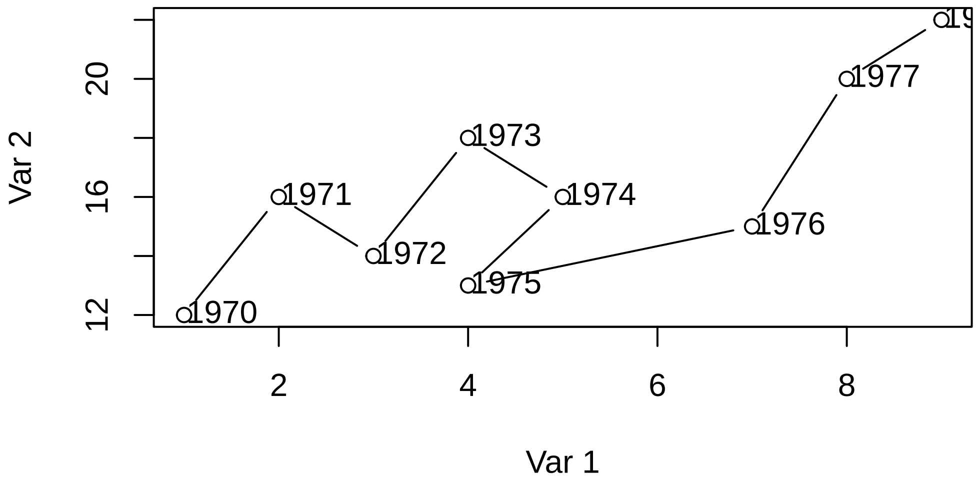
(21) Create a chart (bar chart, line chart, or scatter plot) that shows the unemployment rate of **Germany** over the available years. Label the chart 'Germany' with `ggtitle("Germany")`. Please note that you may choose any type of graphical representation. (Hint: Below you can see one of many possible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

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

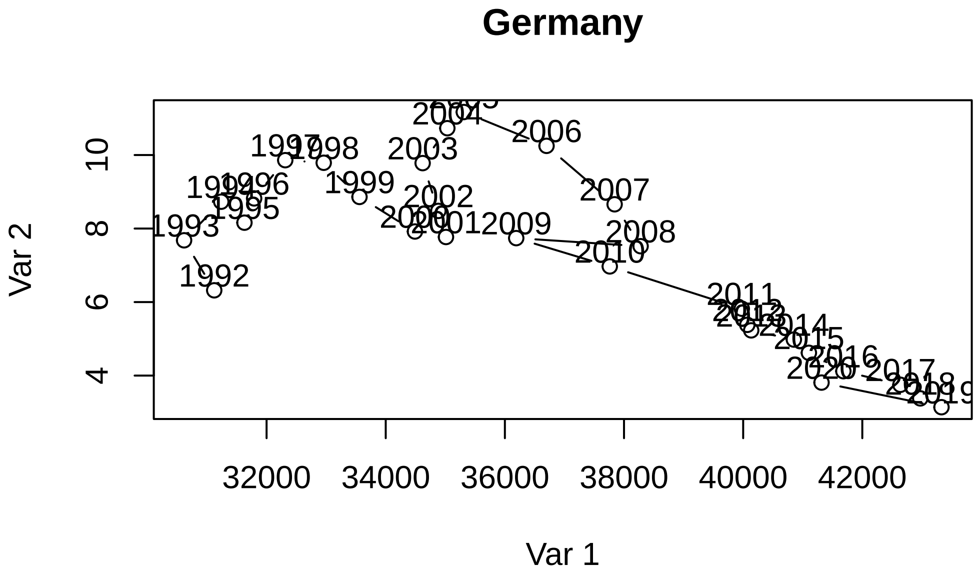


Suppose you want to visualize the simultaneous evolution of the unemployment rate and GDP per capita over time for Germany as well.
Suppose further that you have found the following lines of code that create the kind of chart you are looking for.

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

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



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



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

Solution

The script uses the following functions: `aes`, `c`, `dim`, `filter`, `geom_line`, `ggplot`, `ggtitle`, `group_by`, `head`, `load`, `max`, `mean`, `mutate`, `plot`, `rename`, `sd`, `select`, `summarise`, `tail`, `text`, `title`, `url`.

R script

```
# setwd("/home/sthu/Dropbox/hsf/exams/22-11/scr/")

rm(list = ls())

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, ggpubr, sjPlot)

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

head(df, 8)

tail(df, 1)

# panel data set
# date and country.x

observations_df <- dim(df)

df <- rename(df, nation = country.x)
df <- rename(df, year = date)

df <- df |>
  select(nation, year, gdp, pop, gdppc, unemployment)

df <- df |>
  mutate(gdp_pc = gdp / pop)

df <- df |> filter(nation == "Germany" | nation == "France")

df |>
  group_by(nation) |>
  summarise(mean(unemployment), mean(gdppc))

df |>
  filter(year == 2020) |>
  group_by(nation) |>
  summarise(mean(unemployment), mean(gdppc))

df |>
  group_by(nation) |>
  summarise(max(unemployment), max(gdppc))

df |>
  group_by(nation) |>
  summarise(sd(gdppc), sd(unemployment))

df |>
  group_by(nation) |>
  summarise(sd(unemployment), mean(unemployment), cov = sd(unemployment) / mean(unemploy

df |>
  group_by(nation) |>
  summarise(sd(gdppc), mean(gdppc), cov = sd(gdppc) / mean(gdppc))
```

9.16. Names and duplicates

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

Solution

The script uses the following functions: `anti_join`, `arrange`, `c`, `cur_group_id`, `desc`, `dim`, `distinct`, `filter`, `get_dupes`, `glimpse`, `group_by`, `head`, `load`, `max`, `mutate`, `n`, `paste`, `relocate`, `row_number`, `setdiff`, `summary`, `tail`, `ungroup`, `url`.

R script

```r
# Find duplicates

# set working directory
# setwd("~/Dropbox/hsf/test/initial_script")

# clear environment
rm(list = ls())

# load packages
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, janitor, babynames, stringr)

load(url("https://github.com/hubchev/courses/raw/main/dta/df_names.RData"))

# Remove all objects except df_2022 and df_2022_error
rm(list = setdiff(ls(), c("df_2022_error", "df_2022")))

# Re-order the data so that surname, name, and age appears first.
# Save the changed data in a tibble called `df`.
df <- df_2022 |>
  relocate(surname, name, age)

# Sort the data according to surname, name, and age.
df <- df |>
  arrange(surname, name, age)

# Inspect df_2022 and df_2022_error
df
dim(df)
head(df)
tail(df)
glimpse(df)
summary(df)

df_2022_error

# Make a variable that contains the year of birth. Name the variable `born`
# and new dataframe `df`.
df <- df_2022 |>
  mutate(born = time - age)

# Make a new variable that identifies each person by surname, name,
# and their birth born. Name the variable `id`.
df <- df |>
  mutate(id = paste(surname, name, born, sep = "_"))

# How many different groups do exist?
df <- df |>
  group_by(id) |>
  mutate(id_num = cur_group_id()) |>
  ungroup()

max(df$id_num)
```

9.17. Zipf's law

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

Preamble

(1) Set your working directory.

(2) Clear your global environment.

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

Read in, inspect, and clean the data

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

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

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

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

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

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

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

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

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

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

```
# A tibble: 17 x 3
   state                   `mean(pop2011)` `sum(pop2011)`
   <chr>                             <dbl>          <dbl>
 1 Baden-Wrttemberg                   7580           7580
 2 Baden-Württemberg                23680.        7837917
 3 Bayern                           23996.        7558677
 4 Berlin                          3292365        3292365
 5 Brandenburg                      18472.        1865632
 6 Bremen                          325432.         650863
 7 Hamburg                         1706696        1706696
 8 Hessen                           22996.        5036121
 9 Mecklenburg-Vorpommern           27034.         811005
```

```
10 Niedersachsen                    24107.        6219515
11 Nordrhein-Westfalen              47465.       18036727
12 Rheinland-Pfalz                  25644.        1871995
13 Saarland                            NA              NA
14 Sachsen                          27788.        2973351
15 Sachsen-Anhalt                   21212.        1993915
16 Schleswig-Holstein               24157.        1739269
17 Th_ringen                        29192.        1167692
```

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

```
# A tibble: 16 x 3
   state                  `mean(pop2011)` `sum(pop2011)`
   <chr>                            <dbl>          <dbl>
 1 Baden-Württemberg               23631.        7845497
 2 Bayern                          23996.        7558677
 3 Berlin                        3292365         3292365
 4 Brandenburg                     18472.        1865632
 5 Bremen                         325432.         650863
 6 Hamburg                       1706696         1706696
 7 Hessen                          22996.        5036121
 8 Mecklenburg-Vorpommern          27034.         811005
 9 Niedersachsen                   24107.        6219515
10 Nordrhein-Westfalen             47465.       18036727
11 Rheinland-Pfalz                 25644.        1871995
12 Saarland                           NA              NA
13 Sachsen                         27788.        2973351
14 Sachsen-Anhalt                  21212.        1993915
15 Schleswig-Holstein              24157.        1739269
16 Thüringen                       29192.        1167692
```

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

```
# A tibble: 47 x 5
   city               status  state    pop2011 rankX
   <chr>              <chr>   <chr>      <dbl> <dbl>
 1 Perl               Commune Saarland    7775  2003
 2 Freisen            Commune Saarland    8270  1894
 3 Großrosseln        Commune Saarland    8403  1868
 4 Nonnweiler         Commune Saarland    8844  1775
 5 Nalbach            Commune Saarland    9302  1678
 6 Wallerfangen       Commune Saarland    9542  1642
 7 Kirkel             Commune Saarland   10058  1541
 8 Merchweiler        Commune Saarland   10219  1515
 9 Nohfelden          Commune Saarland   10247  1511
10 Friedrichsthal     City    Saarland   10409  1489
11 Marpingen          Commune Saarland   10590  1461
```

```
12 Mandelbachtal       Commune Saarland   11107  1390
13 Kleinblittersdorf   Commune Saarland   11396  1354
14 Überherrn           Commune Saarland   11655  1317
15 Mettlach            Commune Saarland   12180  1241
16 Tholey              Commune Saarland   12385  1217
17 Saarwellingen       Commune Saarland   13348  1104
18 Quierschied         Commune Saarland   13506  1088
19 Spiesen-Elversberg  Commune Saarland   13509  1086
20 Rehlingen-Siersburg Commune Saarland   14526   996
21 Riegelsberg         Commune Saarland   14763   982
22 Ottweiler           City    Saarland   14934   969
23 Beckingen           Commune Saarland   15355   931
24 Losheim am See      Commune Saarland   15906   887
25 Schiffweiler        Commune Saarland   15993   882
26 Wadern              City    Saarland   16181   874
27 Schmelz             Commune Saarland   16435   857
28 Sulzbach/Saar       City    Saarland   16591   849
29 Illingen            Commune Saarland   16978   827
30 Schwalbach          Commune Saarland   17320   812
31 Eppelborn           Commune Saarland   17726   793
32 Wadgassen           Commune Saarland   17885   785
33 Bexbach             City    Saarland   18038   777
34 Heusweiler          Commune Saarland   18201   762
35 Püttlingen          City    Saarland   19134   718
36 Lebach              City    Saarland   19484   701
37 Dillingen/Saar      City    Saarland   20253   654
38 Blieskastel         City    Saarland   21255   601
39 St. Wendel          City    Saarland   26220   460
40 Merzig              City    Saarland   29727   392
41 Saarlouis           City    Saarland   34479   323
42 St. Ingbert         City    Saarland   36645   299
43 Völklingen          City    Saarland   38809   279
44 Homburg             City    Saarland   41502   247
45 Neunkirchen         City    Saarland   46172   206
46 Saarbrücken         City    Saarland  175853    43
47 Perl                Commune Saarland      NA    NA
```

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

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

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

```
# A tibble: 23 x 5
# Groups:   city [11]
   city        status                   state                pop2011 unique_count
   <chr>       <chr>                    <chr>                  <dbl>        <int>
```

1	Bonn	City with County Rights	Nordrhein-Westfalen	305765	3
2	Bonn	City with County Rights	Nordrhein-Westfalen	305765	3
3	Bonn	City with County Rights	Nordrhein-Westfalen	305765	3
4	Brühl	Commune	Baden-Württemberg	13805	2
5	Brühl	City	Nordrhein-Westfalen	43568	2
6	Erbach	City	Baden-Württemberg	13024	2
7	Erbach	City	Hessen	13245	2
8	Fürth	City with County Rights	Bayern	115613	2
9	Fürth	Commune	Hessen	10481	2
10	Lichtenau	City	Nordrhein-Westfalen	10473	2
11	Lichtenau	Commune	Sachsen	7544	2
12	Münster	Commune	Hessen	14071	2
13	Münster	City with County Rights	Nordrhein-Westfalen	289576	2
14	Neunkirchen	Commune	Nordrhein-Westfalen	13930	2
15	Neunkirchen	City	Saarland	46172	2
16	Neuried	Commune	Baden-Württemberg	9383	2
17	Neuried	Commune	Bayern	8277	2
18	Petersberg	Commune	Hessen	14766	2
19	Petersberg	Commune	Sachsen-Anhalt	10097	2
20	Senden	City	Bayern	21560	2
21	Senden	Commune	Nordrhein-Westfalen	19976	2
22	Staufenberg	City	Hessen	8114	2
23	Staufenberg	Commune	Niedersachsen	7983	2

(16) The table indicates that the city of Bonn appears three times in the dataset, and all three observations contain identical information. Thus, remove two of these observations to ensure that Bonn is uniquely represented in the dataset. All other cities that occur more than once in the data are situated in different states. That means, these are distinct cities that coincidentally share the same name.

Data analysis (Zipf's Law)

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

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

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

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

(17)

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

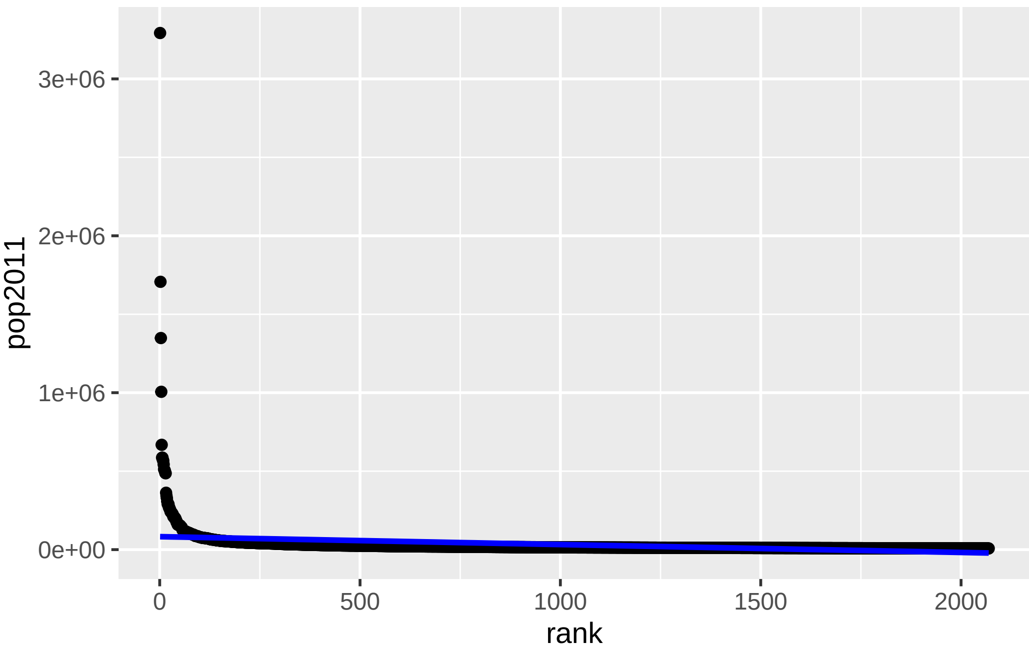
Note: If you cannot solve this task, use the variable `rankX` as a substitute for the variable `rank` that was not generated.

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

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

```
[1] -0.2948903
```

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



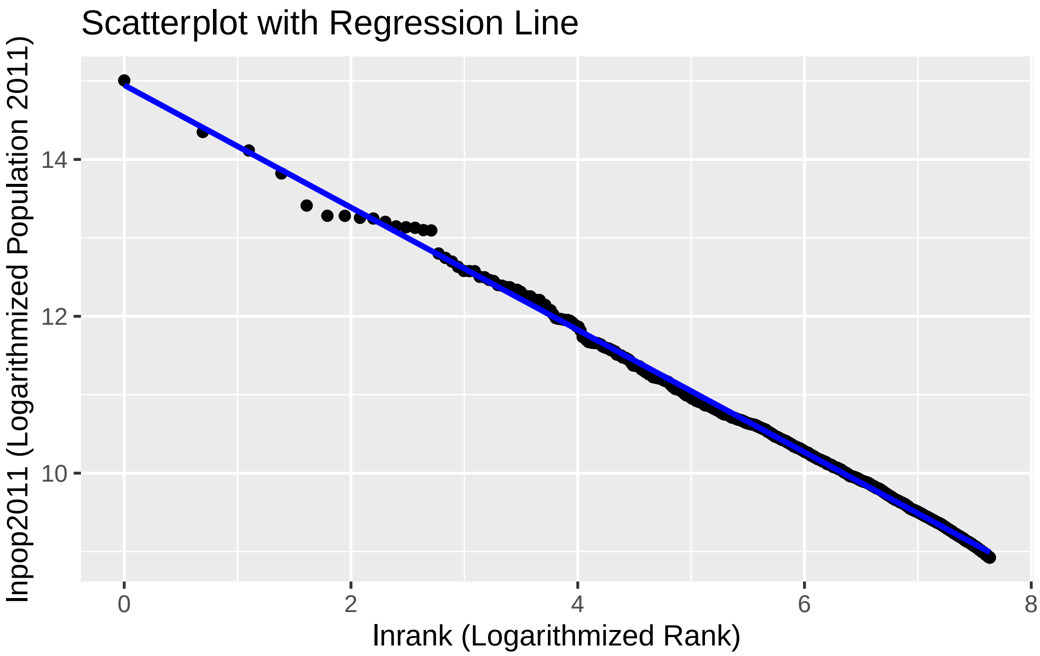
(20) Logarithmize the variables `rank` and `pop2011`. Title the new variables as `lnrank` and `lnpop2011`, respectively. Here is a snapshot of the resulting variables:

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

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

```
[1] -0.9990053
```

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



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

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

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

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

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

(24) Explain the following lines of code.

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

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

Solution

The script uses the following functions: `aes`, `arrange`, `as_tibble`, `c`, `case_when`, `cor`, `desc`, `dim`, `exp`, `filter`, `geom_point`, `geom_smooth`, `ggplot`, `group_by`, `head`, `is.na`, `labs`, `lm`, `log`, `mean`, `mutate`, `n`, `predict`, `print`, `read_dta`, `rename`, `row_number`, `save`, `select`, `starts_with`, `sum`, `summarise`, `summary`, `tail`, `ungroup`.

R script

```
# load packages
if (!require(pacman)) install.packages("pacman")
suppressMessages(pacman::p_unload(all))
# setwd("~/Dropbox/hsf/exams/24-01/Rmd")

rm(list = ls())

pacman::p_load(tidyverse, haven, janitor, jtools)

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

head(df)
tail(df)

dim(df)

summary(df)

df <- df |>
  rename(city = stadt)

df <- df |>
  select(-pop1970, -pop1987)

df |>
  group_by(state) |>
  summarise(
    mean(pop2011),
    sum(pop2011)
  )

df <- df |>
  mutate(state = case_when(
    state == "Baden-Wrttemberg" ~ "Baden-Württemberg",
    state == "Th_ringen" ~ "Thüringen",
    TRUE ~ state
  ))

df |>
  group_by(state) |>
  summarise(
    mean(pop2011),
    sum(pop2011)
  )

df |>
  filter(state == "Saarland") |>
  print(n = 100)

df <- df |>
  filter(!(city == "Perl" & is.na(pop2011)))
```

References

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

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

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

Kieran Healy. *Data Visualization: A Practical Introduction*. Princeton University Press, 2018. URL https://socviz.co/.

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

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

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

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

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

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

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

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

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

Perry Stephenson. Data science practice. Accessed January 30, 2023, 2023. URL https://datasciencepractice.study/.

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/.

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

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

Hadley Wickham. The tidyverse style guide, 2024. URL https://style.tidyverse.org/.

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

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

A. Navigating the file system

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

A.1. The file system

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

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

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

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

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

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

A.2. Working directory

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

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

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

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

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

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

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

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

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

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

```r
getwd()
```

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

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

When you want to load or save a file in R it's important to know what the working directory is. You can find out by using the `getwd()` command. For the moment, let's assume that I'm using Mac OS or Linux, since things are different on Windows, see section Section A.5. Let's check the current active working directory:

```
getwd()
```

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

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

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

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

```
list.files()
```

A.4. R Studio projects

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

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

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

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

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

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

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

B. Operators

B.1. Assignment:

- `<-` (assignment operator)

B.2. Arithmetic:

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

B.3. Relational:

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

B.4. Logical:

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

B.5. Others:

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

C. Popular functions

C.1. Help

- `?`: Search R documentation for a specific term.
- `??` Search R help files for a word or phrase.
- RSiteSearch: Search search.r-project.org
- `help.start`: Access to html manuals and documentations implemented in R
- `browseVignettes`: view a list of all vignettes associated with your installed packages
- `vignette`: View a specified package vignette, that is, supporting material such as introductions.

C.2. Package management

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

C.3. General

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

C.4. Tools

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

C.5. Data import

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

C.6. Inspect data

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

C.7. Graphics

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

C.8. Data management

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

- `rownames`: Get or set the row names of a matrix-like object.
- `tibble`: Creates a tibble, a modern and tidy data frame.

C.9. `dplyr` functions

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

C.10. Data analysis

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

C.11. Statistical functions

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

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

D. Helpful shortcuts

Table D.1.: Different OS, different keys

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

Table D.2.: Helpful shortcuts

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