Daniel Dauber

R for Non-Programmers: A Guide for Social Scientists

Table of contents

References 213

Welcome

Welcome to *R for Non-Programmers: A Guide for Social Scientists (R4NP)*. This book provides a helpful resource for anyone looking to use *R* for their research projects, regardless of their level of programming or statistical analysis experience. Each chapter presents essential concepts in data analysis in a concise, thorough, and user-friendly manner. *R4NP* will guide you through your first steps on a possibly endless journey full of 'awe and wonder'.

This book is designed to be accessible to beginners while also providing valuable insights for experienced analysts looking to transition to *R* from other computational software. You don't need any prior knowledge or programming skills to get started. *R4NP* provides a comprehensive entry into R programming, regardless of your background.

About the author

I am an Associate Professor at the University of Warwick. My research interfaces with multiple disciplines, such as International Management, Organisational Behaviour, Cross-cultural Psychology and Intercultural Communication. Most of my research investigates real-life issues in organisational contexts using a diagnostic angle with an emphasis to help improve organisational members' experiences and organisational performance.

For more than 13 years, I enjoy teaching students how to turn data into meaningful recommendations for businesses, universities and the broader society by conveying relevant theory-driven knowledge as well as research methods skills. In addition, I am enthusiastic about new approaches to analysing data, whether it is quantitative or qualitative.

My most recent publications can be found on ResearchGate[1], and I regularly post tutorials about R, productivity and other hidden gems of academic life on my blog: The Social Science Sofa[2]. If you like to connect with me, you can find me on Twitter[3].

[1] https://www.researchgate.net/profile/Daniel-Dauber
[2] https://www.thesocialsciencesofa.com
[3] https://twitter.com/daniel_dauber

1

`Readme.` *before you get started*

1.1 A starting point and reference book

My journey with *R* began rather suddenly one night. After many months of contemplating learning *R* and too many excuses not to get started, I put all logic aside and decided to use *R* on my most significant project to date. Admittedly, at the time, I had some knowledge of other programming languages and felt maybe overconfident learning a new tool. This is certainly not how I would recommend learning *R*. In hindsight, though, it enabled me to do things that pushed the project much further than I could have imagined. However, I invested a lot of additional time on top of my project responsibilities to learn this programming language. In many ways, *R* opened the door to research I would not have thought of a year earlier. Today, I completely changed my style of conducting research, collaborating with others, composing my blog posts and writing my research papers.

It is true what everyone told me back then: *R* programming has a steep learning curve and is challenging. To this date, I would agree with this sentiment if I ignored all the available resources. The one thing I wish I had to help me get started was a book dedicated to analysing data from a Social Scientist perspective and which guides me through the analytical steps I partially knew already. Instead, I spent hours searching on different blogs and YouTube to find solutions to problems in my code. However, at no time I was tempted to revert to my trusted statistics software of choice, i.e. SPSS. I certainly am an enthusiast of learning programming languages, and I do not expect that this is true for everyone. Nevertheless, the field of Social Sciences is advancing rapidly and learning at least one programming language can take you much further than you think.

Thus, the aim of this book is narrowly defined: A springboard into the world of *R* without having to become a full-fledged *R* programmer or possess abundant knowledge in other programming languages. This book will guide you through the most common challenges in empirical research in the Social Sciences. Each chapter is dedicated to a common task we have to achieve to answer our research questions. At the end of each chapter, exercises are provided to hone your skills and allow you to revisit key aspects. In addition, the appendix offers

several in-depth case studies that showcase how a research project would be carried out from start to finish in R using real datasets.

This book likely caters to your needs irrespective of whether you are a seasoned analyst and want to learn a new tool or have barely any knowledge about data analysis. However, it would be wrong to assume that the book covers everything you possibly could know about *R* or the analytical techniques covered. There are dedicated resources available to you to dig deeper. Several of these resources are cited in this book or are covered in Chapter @ref(next-steps). The primary focal point of the book is on learning *R* in the context of Social Sciences research projects. As such, it serves as a starting point on what hopefully becomes an enriching, enjoyable, adventurous, and lasting journey.

1.2 Download the companion R package

The learning approach of this book is twofold: Convey knowledge and offer opportunities to practice. Therefore, more than 50% of the book are dedicated to examples, case studies, exercises and code you can directly use yourself. To facilitate this interactive part, this book is accompanied by a so-called 'R package' (see Chapter @ref(r-packages)), which contains all datasets used in this book and enables you to copy and paste any code snippet[1] and work along with the book.

Once you worked through Chapter @ref(setting-up-r-and-rstudio) you can easily download the package `r4np` using the following code snippet in your console (see Chapter @ref(the-console-window)):

```
devtools::install_github("ddauber/r4np")
```

1.3 A 'tidyverse' approach with some basic R

As you likely know, every language has its flavours in the form of dialects. This is no different to programming languages. The chosen 'dialect' of *R* in this book is the `tidyverse` approach. Not only is it a modern way of programming in *R*, but it is also a more accessible entry point. The code written with `tidyverse` reads almost like a regular sentence and, therefore, is much easier to

[1] A *code snippet* is a piece of programming code that can be used directly as is.

read, understand, and remember. Unfortunately, if you want a comprehensive introduction to learning the underlying basic *R* terminology, you will have to consult other books. While it is worthwhile to learn different ways of conducting research in *R*, basic *R* syntax is much harder to learn, and I opted against covering it as an entry point for novice users. After working through this book, you will find exploring some of the *R* functions from other 'dialects' relatively easy, but you likely miss the ease of use from the `tidyverse` approach.

1.4 Understanding the formatting of this book

The formatting of this book carries special meaning. For example, you will find actual *R* code in boxes like these.

```
name <- "Daniel"
food <- "Apfelstrudel"

paste("My name is ", name, ", and I love ", food, ".", sep = "")
```

```
[1] "My name is Daniel, and I love Apfelstrudel."
```

You can easily copy this *code chunk* by using the button in the top-right corner. Of course, you are welcome to write the code from scratch, which I would recommend because it accelerates your learning.

Besides these blocks of code, you sometimes find that certain words are formatted in a particular way. For example, datasets, like `imdb_top_250`, included in the *R* package `r4np`, are highlighted. Every time you find a highlighted word, it refers to one of the following:

- A dataset,
- A variable,
- A data type,
- The output of code,
- The name of an *R* package,
- The name of a function or one of its components.

This formatting style is consistent with other books and resources on *R* and, therefore, easy to recognise when consulting other content, such as those covered in Chapter @ref(next-steps).

2

Why learn a programming language as a non-programmer?

'R', it is not just a letter you learn in primary school, but a powerful programming language. While it is used for a lot of quantitative data analysis, it has grown over the years to become a powerful tool that excels (#no-pun-intended) in handling data and performing customised computations with quantitative and qualitative data.

R is now one of my core tools to perform various types of work, for example,

- Statistical analysis,
- Corpus analysis,
- Development of online dashboards for interactive data visualisations and exploration,
- Connection to social media APIs for data collection,
- Creation of reporting systems to provide individualised feedback to research participants,
- Writing research articles, books, and blog posts,
- etc.

> *Learning R is like learning a foreign language. If you enjoy learning languages, then 'R' is just another one.*

While *R* has become a comprehensive tool for data scientists, it has yet to find its way into the mainstream field of Social Sciences. Why? Well, learning programming languages is not necessarily something that feels comfortable to everyone. It is not like Microsoft Word, where you can open the software and explore it through trial and error. Learning a programming language is like learning a foreign language: You have to learn vocabulary, grammar and syntax. Similar to learning a new language, programming languages also have steep learning curves and require quite some commitment.

For this reason, most people do not even dare to learn it because it is time-consuming and often not considered a *'core method'* in Social Sciences disciplines. Apart from that, tools like SPSS have very intuitive interfaces, which seem much easier to use (or not?). However, the feeling of having *'mastered' R* (although one might never be able to claim this) can be extremely rewarding.

I guess this introduction was not necessarily helpful in convincing you to learn any programming language. However, despite those initial hurdles, there are a series of advantages to consider. Below I list some good reasons to learn a programming language as they pertain to my own experiences.

2.1 Learning new tools to analyse your data is always essential

Theories change over time, and new insights into certain social phenomena are published every day. Thus, your knowledge might get outdated quite quickly. This is not so much the case for research methods knowledge. Typically, analytical techniques remain over many years. We still use the mean, mode, quartiles, standard deviation, etc., to describe our quantitative data. However, there are always new computational methods that help us to crunch the numbers even more. *R* is a tool that allows you to venture into new analytical territory because it is open source. Thousands of developers provide cutting-edge research methods free of charge for you to try with your data. You can find them on platforms like GitHub[1]. *R* is like a giant supermarket, where all products are available for free. However, to read the labels on the product packaging and understand what they are, you have to learn the language used in this supermarket.

2.2 Programming languages enhance your conceptual thinking

While I have no empirical evidence for this, I am very certain it is true. I would argue that my conceptual thinking is quite good, but I would not necessarily say that I was born with it. Programming languages are very logical. Any error in your code will make you fail to execute it properly. Sometimes you face challenges in creating the correct code to solve a problem. Through creative abstract thinking (I should copyright this term), you start to approach your

[1] https://github.com

problems differently, whether it is a coding problem or a problem in any other context. For example, I know many students enjoy the process of qualitative coding. However, they often struggle to detach their insights from the actual data and synthesise ideas on an abstract and more generic level. Qualitative researchers might refer to this as challenges in '*second-order deconstruction of meaning'*. This process of abstraction is a skill that needs to be honed, nurtured and practised. From my experience, programming languages are one way to achieve this, but they might not be recognised for this just yet. Programming languages, especially functions (see Chapter @ref(functions)), require us to generalise from a particular case to a generic one. This mental mechanism is also helpful in other areas of research or work in general.

2.3 Programming languages allow you to look at your data from a different angle

There are commonly known and well-established techniques regarding how you should analyse your data rigorously. However, it can be quite some fun to try techniques outside your discipline. This does not only apply to programming languages, of course. Sometimes, learning about a new research method enables you to look at your current tools in very different ways too. One of the biggest challenges for any researcher is to reflect on one's own work. Learning new and maybe even *'strange'* tools can help with this. Admittedly, sometimes you might find out that some new tools are also a dead-end. Still, you likely have learned something valuable through the process of engaging with your data differently. So shake off the rust of your analytical routine and blow some fresh air into your research methods.

2.4 Learning any programming language will help you learn other programming languages.

Once you understand the logic of one language, you will find it relatively easy to understand new programming languages. Of course, if you wanted to, you could become the next '*Neo'* (from 'The Matrix')[2] and change the reality of your research forever. On a more serious note, though, if you know any programming language already, learning *R* will be easier because you have accrued some basic understanding of these particular types of languages.

[2] https://www.imdb.com/title/tt0133093/?ref_=ext_shr_lnk

Having considered everything of the above, do you feel ready for your next foreign language?

3

Setting up R *and RStudio*

Every journey starts with gathering the right equipment. This intellectual journey is not much different. The first step that every *R* novice has to face is to set everything up to get started. There are essentially two strategies:

- Install *R*[1] and RStudio[2]

or

- Run RStudio in a browser via RStudio Cloud[3]

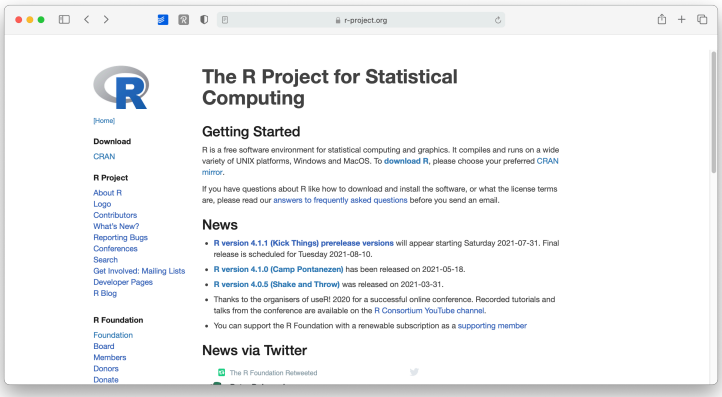
While installing *R* and RStudio requires more time and effort, I strongly recommend it, especially if you want to work offline or make good use of your computer's CPU. However, if you are unsure whether you enjoy learning *R*, you might wish to look at RStudio Cloud first. Either way, you can follow the examples of this book no matter which choice you make.

3.1 Installing R

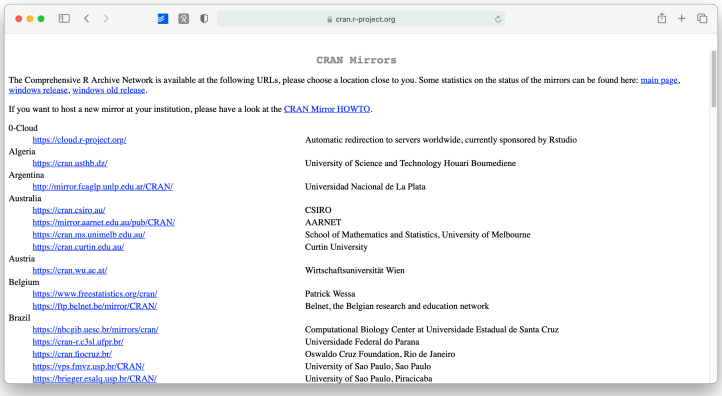
The core module of our programming is *R* itself, and since it is an open-source project, it is available for free on Windows, Mac and Linux computers. So, here is what you need to do to install it properly on your computer of choice:

1. Go to www.r-project.org[4]

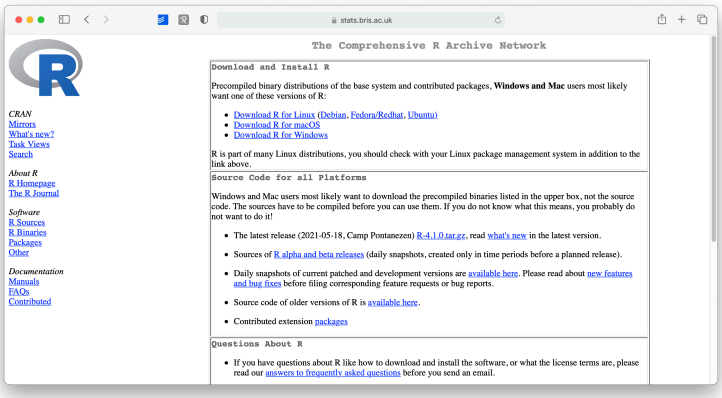
[1] https://www.r-project.org
[2] https://www.rstudio.com
[3] https://rstudio.cloud
[4] https://www.r-project.org%5D(https://www.r-project.org)



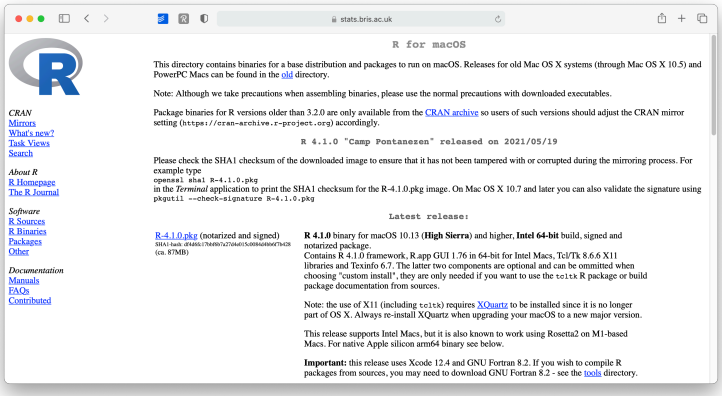
2. Click on `CRAN` where it says `Download`.
3. Choose a server in your country (all of them work, but downloads will perform quicker if you choose your country or one that is close to where you are).



4. Select the operating system for your computer, for example `Download R for macOS`.



5. Select the version you want to install (I recommend the latest version)



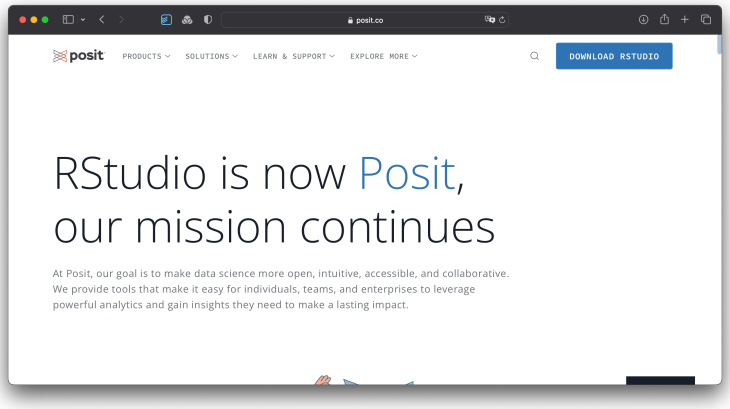
6. Open the downloaded file and follow the installation instructions. I recommend leaving the suggested settings as they are.

This was relatively easy. You now have *R* installed. Technically you can start using *R* for your research, but there is one more tool I strongly advise installing: RStudio.

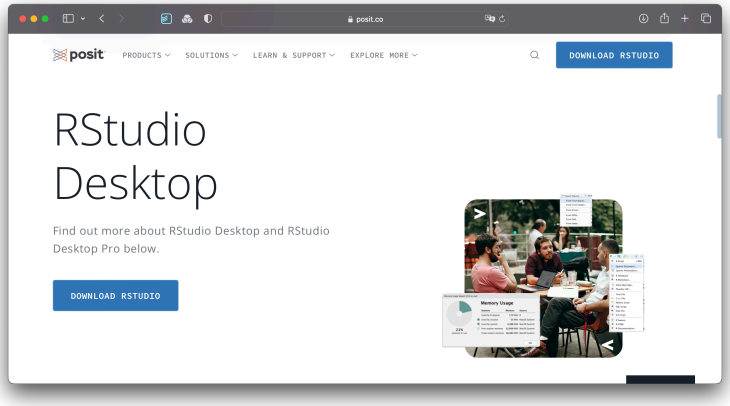
3.2 Installing RStudio

R by itself is just the '*beating heart*' of *R* programming, but it has no particular user interface. If you want buttons to click and actually '*see*' what you are doing, there is no better way than RStudio. RStudio is an *integrated development environment* (IDE) and will be our primary tool to interact with *R*. It is the only software you need to do all the fun parts and, of course, to follow along with the examples of this book. To install RStudio perform the following steps:

1. Go to http://posit.co and click on `DOWNLOAD RSTUDIO`.

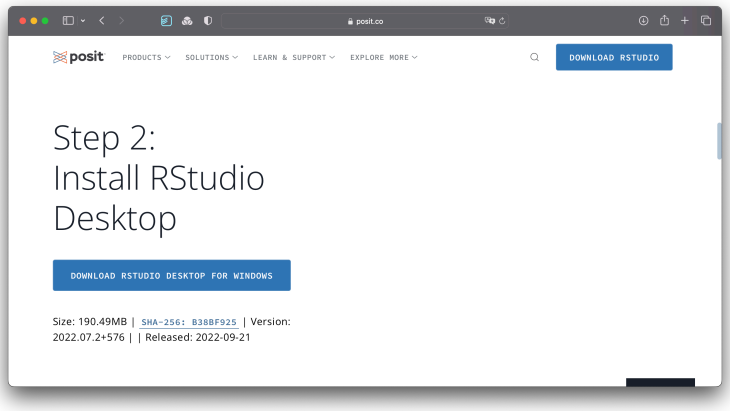


2. Click on `DOWNLOAD RSTUDIO` on this page.

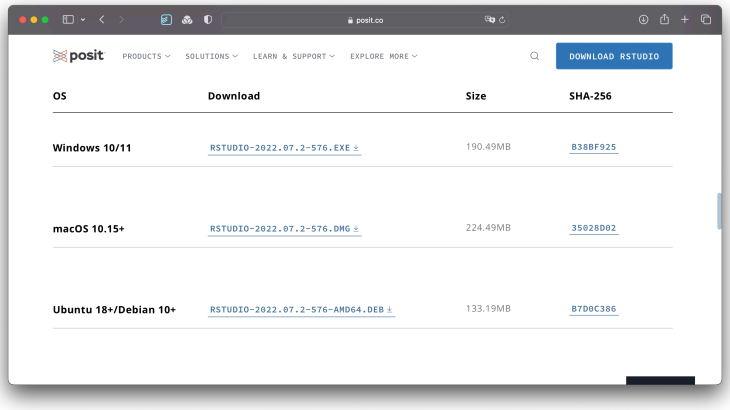


3. This leads you to the page where you can install *R* as a first step

and RStudio as a second step. Since we installed *R* already, we can click on `DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS` (or for Mac if you are not using a PC).



4. If for some reason the version for your operating system is not showing up correctly, you can scroll down and find other version ready to be installed.



5. Open the downloaded file and follow the installation instructions. Again, keep it to the default settings as much as possible.

Congratulations, you are all set up to learn *R*. From now on you only need to start RStudio and not *R*. Of course, if you are the curious type, nothing shall stop you to try *R* without RStudio.

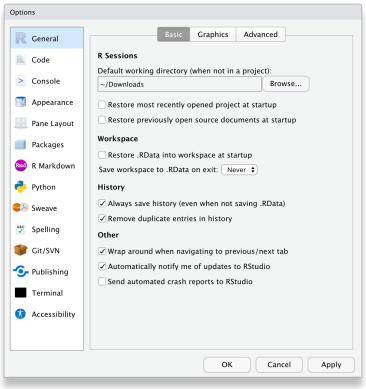
3.3 When you first start RStudio

Before you start programming away, you might want to make some tweaks to your settings right away to have a better experience (in my humble opinion). To open the Rstudio settings you have to click on

- `RStudio > Tools > Global Options` or press `⌘ + ,` if you are on a Mac.
- `RStudio > Tools > Global Options` or press `Ctrl + ,` if you work on a Windows computer.

I recommend to at least make the following changes to set yourself up for success right from the beginning:

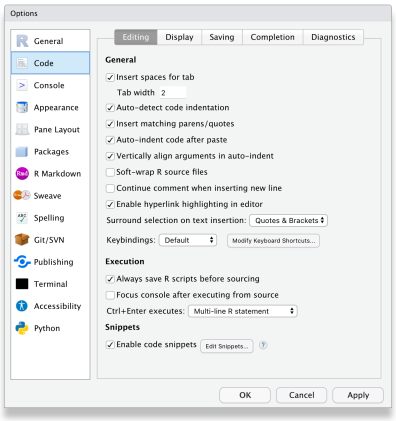
1. Already on the first tab, i.e. `General > Basic`, we should make one of the most significant changes. Deactivate every option that starts with `Restore`. This will ensure that every time you start RStudio, you begin with a clean slate. At first sight, it might sound counter-intuitive not to restart everything where you left off, but it is essential to make all your projects easily reproducible. Furthermore, if you work together with others, not restoring your personal settings also ensures that your programming works across different computers. Therefore, I recommend having the following unticked:
    - `Restore most recently opened project at startup`,
    - `Restore previsouly open source documents at startup`,
    - `Restore .Rdata into workspace at startup`



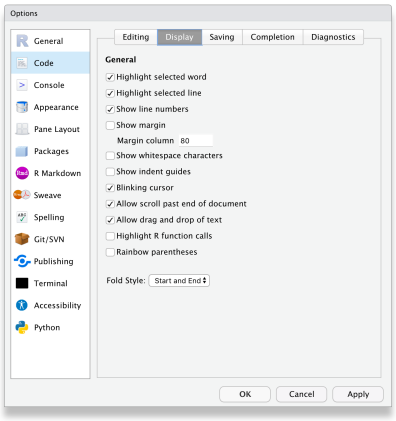
2. In the same tab under `Workspace`, select `Never` for the setting `Save workspace to .RData on exit`. One might think it is wise to keep intermediary results stored from one R session to another. However,

I often found myself fixing issues due to this lazy method, and my code became less reliable and, therefore, reproducible. With experience, you will find that this avoids many headaches.

3. In the `Code > Editing` tab, make sure to have at least the first five options ticked, especially the `Auto-indent code after paste`. This setting will save time when trying to format your coding appropriately, making it easier to read. Indentation is the primary way of making your code look more readable and less like a series of characters that appear almost random.

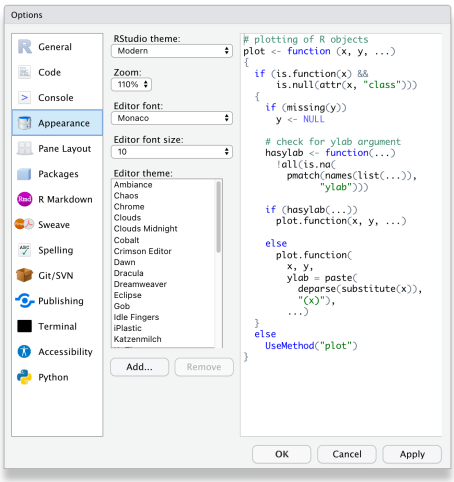


4. In the `Display` tab, you might want to have the first three options selected. In particular, `Highlight selected line` is helpful because, in more complicated code, it is helpful to see where your cursor is.



Of course, if you wish to customise your workspace further, you can do so. The visually most impactful way to alter the default appearance of RStudio

is to select `Appearance` and pick a completely different colour theme. Feel free to browse through various options and see what you prefer. There is no right or wrong here. Just make it your own.



3.4 Updating R and RStudio: Living at the pulse of innovation

While not strictly something that helps you become a better programmer, this advice might come in handy to avoid turning into a frustrated programmer. When you update your software, you need to update *R* and RStudio separately from each other. While both *R* and RStudio work closely with each other, they still constitute separate pieces of software. Thus, it is essential to keep in mind that updating RStudio will not automatically update R. This can become problematic if specific tools you installed via RStudio (like a fancy learning algorithm) might not be compatible with earlier versions of *R*. Also, additional *R* packages (see Chapter @ref(r-packages)) developed by other developers are separate pieces which require updating too, independently from *R* and RStudio.

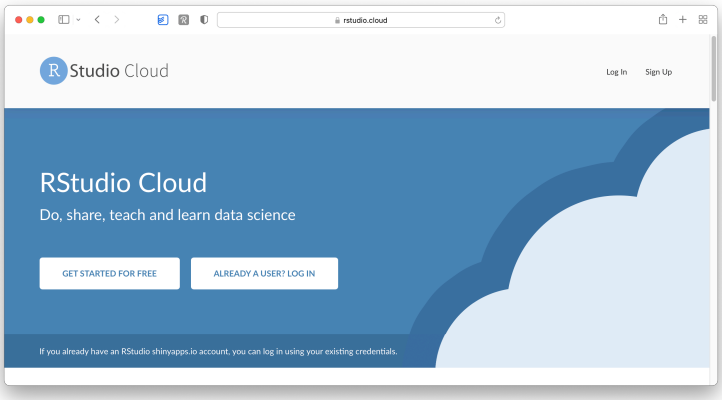
I know what you are thinking: This already sounds complicated and cumbersome. However, rest assured, we take a look at how you can easily update all your packages with RStudio. Thus, all you need to remember is: *R* needs to be updated separately from everything else.

3.5 RStudio Cloud

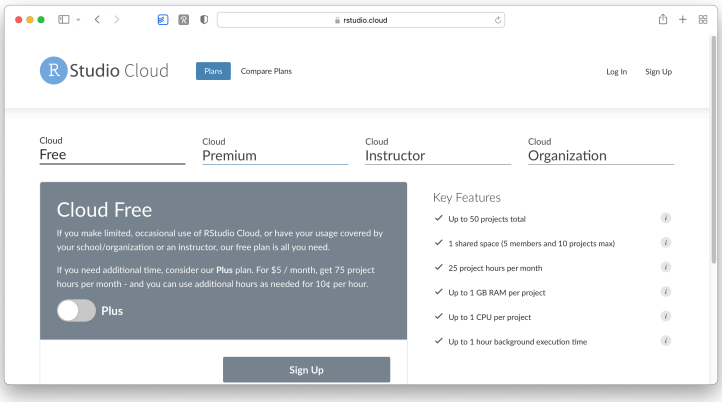
RStudio Cloud is an application that runs in your web browser. It allows you to write and access *R* code wherever you go. It even works on your tablet because it does not require installation. However, do expect it to run slower if your internet connection is not fast or stable. To work with RStudio Cloud, you also need an internet connection. However, there are many benefits to using RStudio in the cloud, such as running time-consuming scripts without using your own device. Also, it is much easier to collaborate with others, and since no installation is required, you can work on projects on any device as long as you are connected to the internet. However, RStudio Cloud's most significant advantage is that you can get started with programming within seconds compared to a desktop installation. Still, I prefer my locally-run offline version of RStudio, because I appreciate working offline as much as online. Nevertheless, I recommend setting up an account because you never know when you need it.

To get started with RStudio Cloud, we have to undertake a couple of steps:

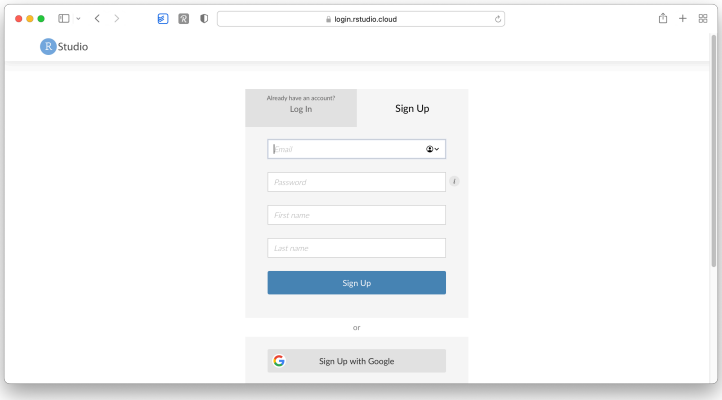
1. Open your web browser of choice and navigate to https://rstudio.cloud.



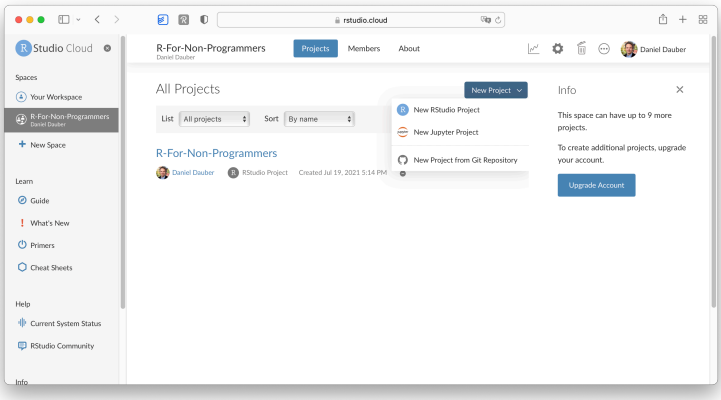
2. Click on `Sign Up` to create your account.
3. On the next page, make sure you have the free plan selected and click on `Sign up`.



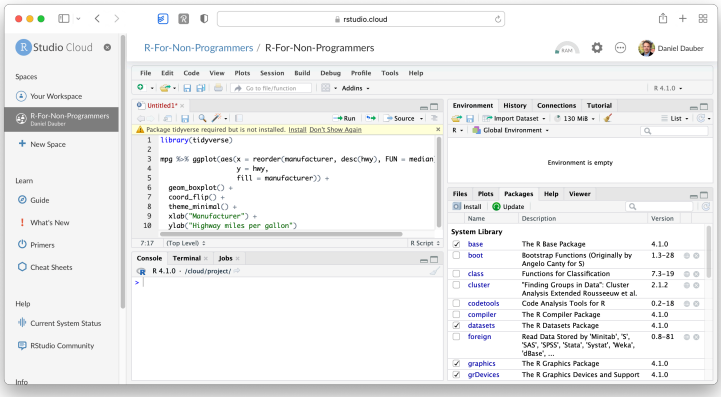
4. To finalise the registration process, you are required to provide your credentials.



5. Once you complete your registration, you are redirected to `Your Workspace`, the central hub for all your projects. As you can tell, I already added another workspace called `R for Non-Programmers`. However, it is fine to use the default one.

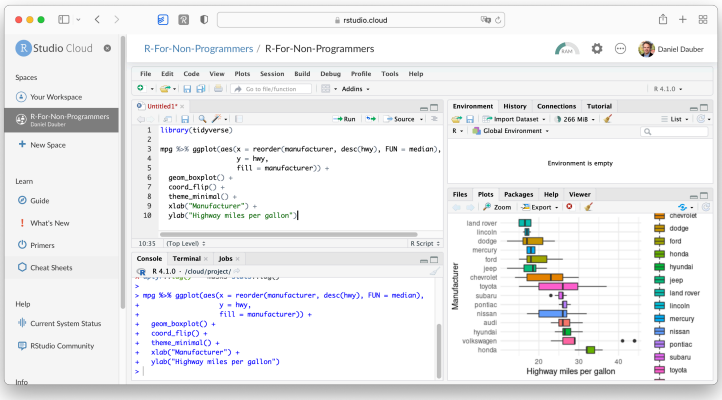


6. To start a new *R* project, you can click on `New Project > New RStudio Project`. This will open RStudio in the Cloud, which looks identical to the desktop version. You can immediately start writing your code. The example below computes a plot.[5]



7. Now we can execute the code as we would on the desktop version (see Chapter @ref(r-basics-the-very-fundamentals)).

[5] RStudio and RStudio Cloud warn you if you need to install certain R packages to successfully run code. More about R packages and what they are can be found in Chapter @ref(r-packages).



No matter whether you choose to use a desktop version of RStudio or RStudio Cloud, you will be able to follow along in this book with no problem.

4

The RStudio Interface

The RStudio interface is composed of quadrants, each of which fulfils a unique purpose:

- The `Console` window,
- The `Source` window,
- The `Environment / History / Connections / Tutorial` window, and
- The `Files / Plots / Packages / Help / Viewer` window

Sometimes you might only see three windows and wonder where the `Source` window has gone in your version of RStudio. In order to use it you have to either open a file or create a new one. You can create a new file by selecting `File > New File > R Script` in the menu bar, or use the keyboard shortcut `Ctrl + Shift + N` on PC and `Cmd + Shift + N` on Mac.

I will briefly explain the purpose of each window/pane and how they are relevant to your work in *R*.

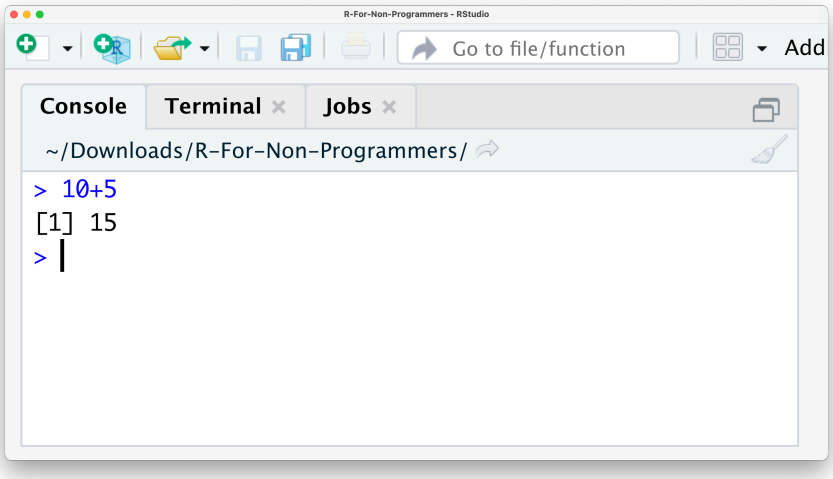
4.1 The Console window

The console is located in the bottom-left, and it is where you often will find the output of your coding and computations. It is also possible to write code directly into the console. Let's try the following example by calculating the sum of `10 + 5`. Click into the console with your mouse, type the calculation into your console and hit `Enter/Return ⏎` on your keyboard. The result should be pretty obvious:

```
# We type the below into the console
10 + 5
```

```
[1] 15
```

Here is a screenshot of how it should look like at your end in RStudio:



You just successfully performed your first successful computation. I know, this is not quite impressive just yet. *R* is undoubtedly more than just a giant calculator.

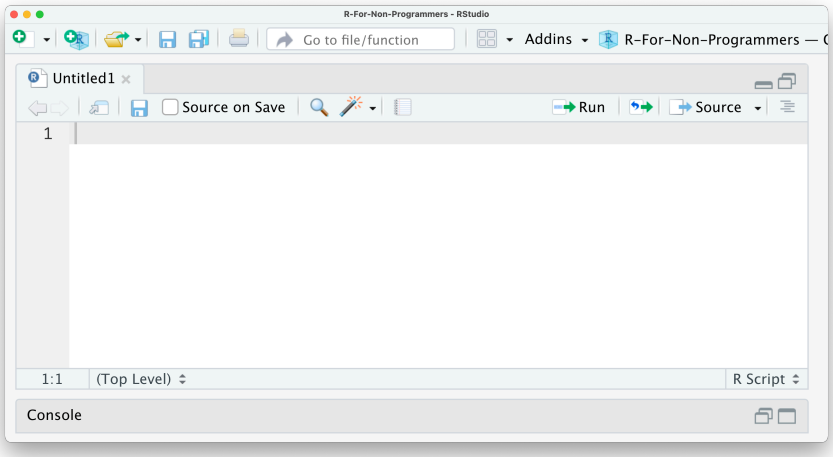
In the top right of the console, you find a symbol that looks like a broom. This one is quite an important one because it clears your console. Sometimes the console can become very cluttered and difficult to read. If you want to remove whatever you computed, you can click the broom icon and clear the console of all text. I use it so frequently that I strongly recommend learning the keyboard shortcut, which is `Ctrl + L` on PC and Mac.

4.2 The Source window

In the top left, you can find the source window. The term 'source' can be understood as any type of file, e.g. data, programming code, notes, etc. The source panel can fulfil many functions, such as:

- Inspect data in an Excel-like format (see also Chapter @ref(import-your-data))
- Open programming code, e.g. an R Script (see Chapter @ref(creating-an-r-script))
- Open other text-based file formats, e.g.
  - Plain text (.txt),
  - Markdown (.md),

  - Websites (.html),
  - LaTeX (.tex),
  - BibTex (.bib),
- Edit scripts with code in it,
- Run the analysis you have written.

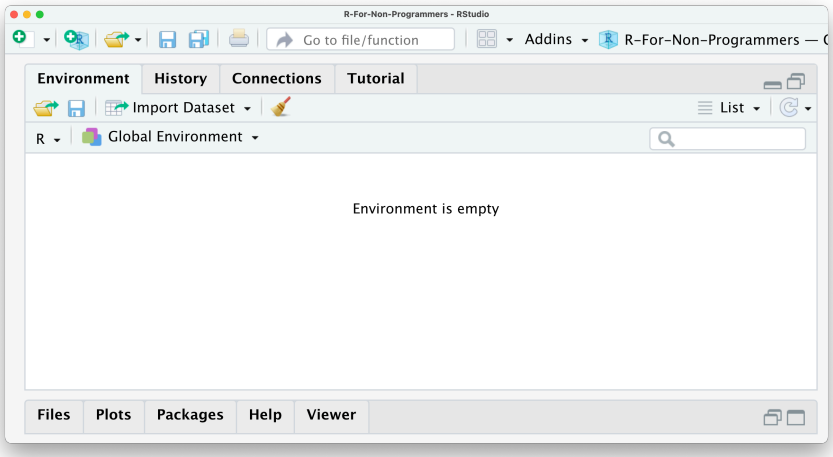


In other words, the source window will show you whatever file you are interested in, as long as RStudio can read it - and no, Microsoft Office Documents are not supported. Another limitation of the source window is that it can only show text-based files. So opening images, etc. would not work.

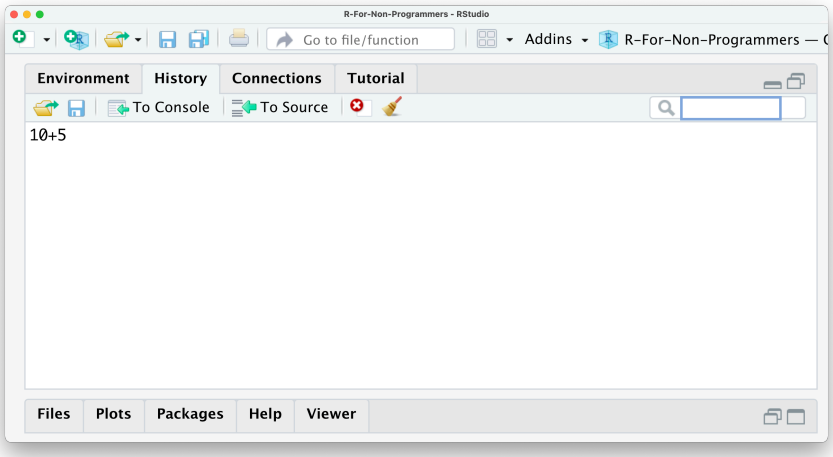
4.3 The Environment / History / Connections / Tutorial window

The window in the top right shows multiples panes. The first pane is called *Environment* and shows you objects which are available for computation. One of the first objects you will create is your dataset because, without data, we cannot perform any analysis. Another object could be a plot showing the number of male and female participants in your study. To find out how to create objects yourself, you can take a glimpse at it (see Chapter @ref(inspecting-raw-data)). Besides datasets and plots, you will also find other objects here, e.g. lists, vectors and functions you created yourself. Don't worry if none of these words makes sense at this point. We will cover each of them in the

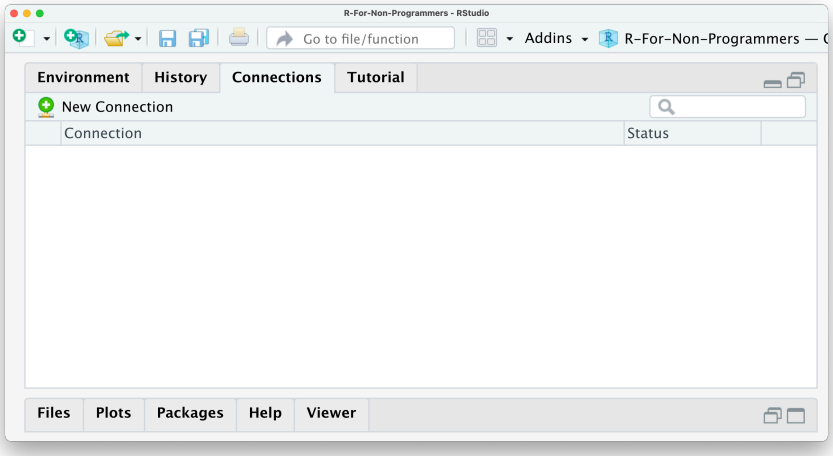
upcoming chapters. For now, remember this is a place where you can find different objects you created.



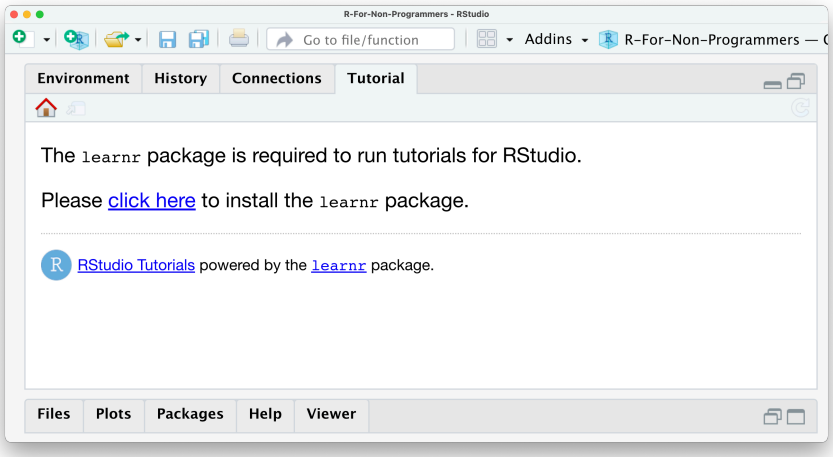
The *History* pane is very easy to understand. Whatever computation you run in the console will be stored. So you can go back and see what you coded and rerun that code. Remember the example from above where we computed the sum of `10+5`? This computation is stored in the history of RStudio, and you can rerun it by clicking on `10+5` in the history pane and then click on `To Console`. This will insert `10+5` back into the console, and we can hit `Return` to retrieve the result. You also have the option to copy the code into an existing or new *R* Script by clicking on `To Source`. By doing this, you can save this computation on your computer and reuse it later. Finally, if you would like to store your history, you can do so by clicking on the `floppy disk symbol`. There are two more buttons in this pane, one allows you to delete individual entries in the history (the white page with the red circle on it), and the last one, a `broom`, clears the entire history (irrevocably).



The pane *Connections* allows you to tab into external databases directly. This can come in handy when you work collaboratively on the same data or want to work with extensive datasets without having to download them. However, for an introduction to *R*, we will not use this feature of RStudio.

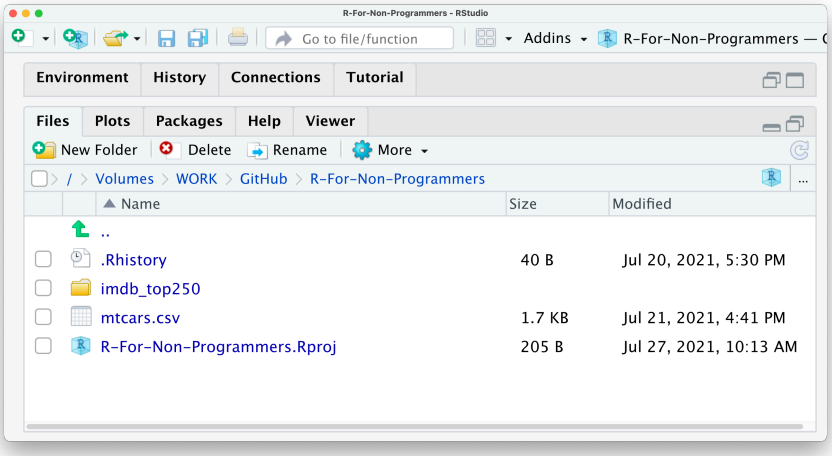


The last pane is called *Tutorial*. Here you can find additional materials to learn *R* and RStudio. If you search for more great content to learn *R*, this serves as a great starting point.



4.4 The Files / Plots / Packages / Help / Viewer window

The last window consists of five essential panes. The first one is the *Files* pane. As the name indicates, it lists all the files and folders in your root directory. A root directory is the default directory where RStudio saves your files, for example, your analysis. However, we will look at how you can properly set up your projects in RStudio in Chapter @ref(starting-your-r-projects). Thus, the *Files* pane is an easy way to load data into RStudio and create folders to keep your research project well organised.



Since the Console cannot reproduce data visualisations, RStudio offers a way to do this very easily. It is through the *Plots* pane. This pane is exclusively designed to show you any plots you have created using *R*. Here is a simple example that you can try. Type into your console `boxplot(mtcars$hp)`.

```
# Here we create a nice boxplot using a dataset called 'mtcars'
boxplot(mtcars$hp)
```

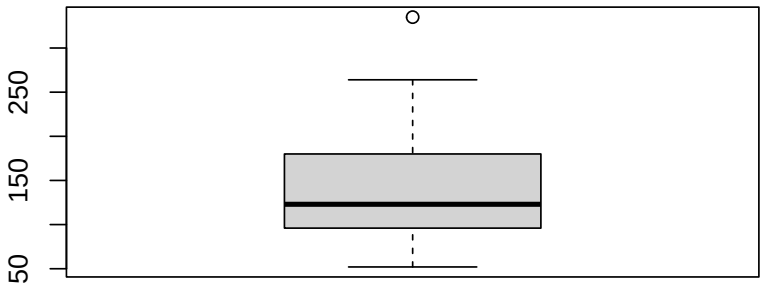
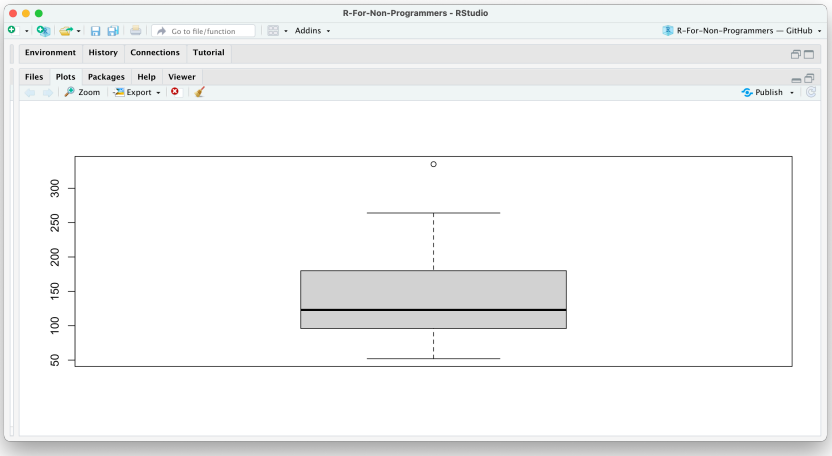


Figure 4.1

Although this is a short piece of coding, it performs quite a lot of steps:

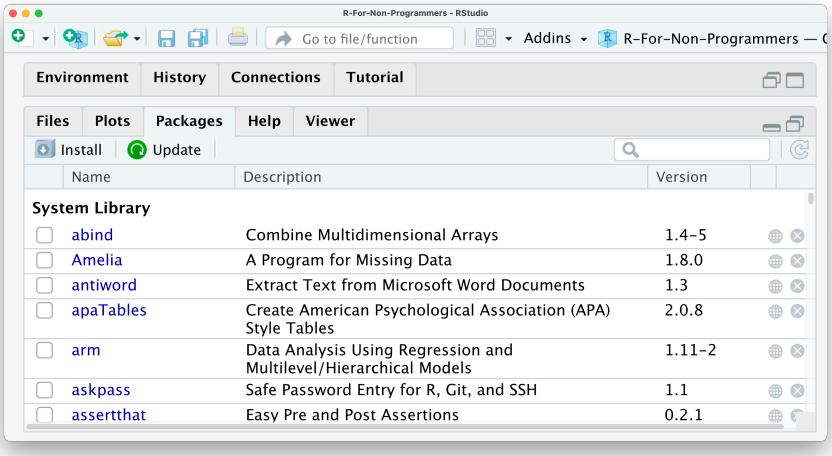
- it uses a function called `boxplot()` to draw a boxplot of
- a variable called `hp` (for horsepower), which is located in
- a dataset named `mtcars`,
- and it renders the graph in your *Plots* pane

This is how the plot should look like in your RStudio *Plots* pane.

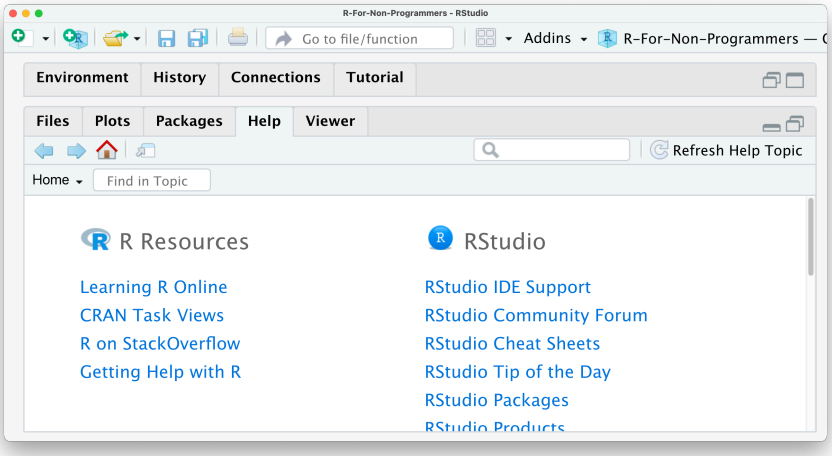


If you wish to delete the plot, you can click on the `red circle with a white x` symbol. This will delete the currently visible plot. If you wish to remove all plots from this pane, you can use the `broom`. There is also an option to export your plot and move back and forth between different plots.

The next pane is called *Packages*. Packages are additional tools you can import and use when performing your analysis. A frequent analogy people use to explain packages is your phone and the apps you install. Each package you download is equivalent to an app on your phone. It can enhance different aspects of working in *R*, such as creating animated plots, using unique machine learning algorithms, or simply making your life easier by doing multiple computations with just one single line of code. You will learn more about *R packages* in Chapter @ref(r-packages).



If you are in dire need of help, RStudio provides you with a *Help* pane. You can search for specific topics, for example how certain computations work. The *Help* pane also has documentation on different datasets that are included in *R*, RStudio or *R packages* you have installed. If you want a more comprehensive overview of how you can find help, have a look at CRAN's 'Getting Help with R'[1] webpage.

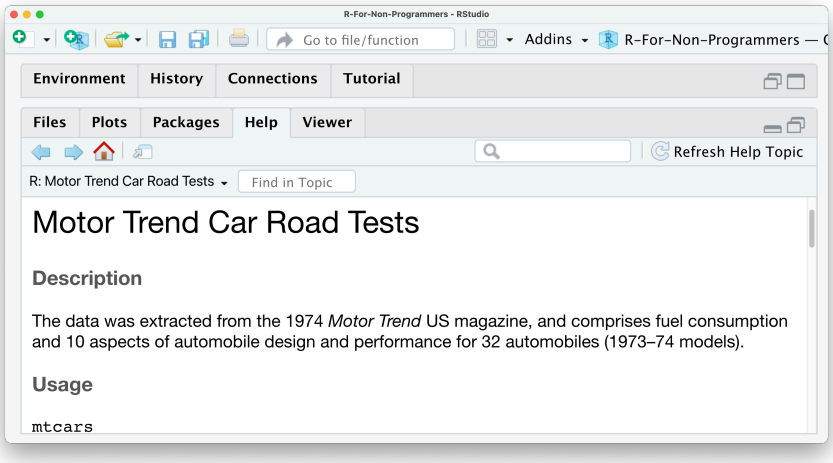


So, for example, if you want to know what the `mtcars` dataset is, you can either use the search window in the *Help* pane or, much easier, use a `?` in the console to search for it:

[1] https://www.r-project.org/help.html

```
# Type a '?' followed by the name of a dataset/function/etc.
# to look up helpful information about it.
?mtcars
```

This will open the *Help* pane and give you more information about this dataset:

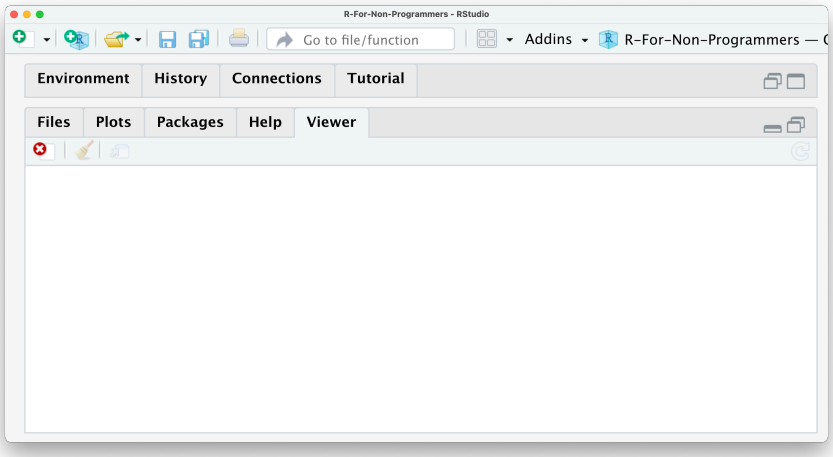


There are many different ways of how you can find help with your coding beyond RStudio and this book. My top three platforms to find solutions to my programming problems are:

- Google[2]
- stackoverflow.com[3]
- Twitter[4] (with #RStats[5])

Lastly, we have the *Viewer* pane. Not every data visualisation we create in *R* is a static image. You can create dynamic data visualisations or even websites with *R*. This type of content is displayed in the *Viewer* pane rather than in the *Plots* pane. Often these visualisations are based on HTML and other web-based programming languages. As such, it is easy to open them in your browser as well. However, in this book, we mainly focus on two-dimensional static plots, which are the ones you likely need most of the time, either for your assignments, thesis, or publication.

[2] https://www.google.com
[3] https://stackoverflow.com
[4] https://twitter.com/home
[5] https://twitter.com/hashtag/rstats



4.5 Customise your user interface

As a last remark in this chapter, I would like to make you aware that you can modify each window. There are three basic adjustments you can make:

- Hide panes by clicking on the window symbol in the top right corner of each window,
- Resize panes by dragging the border of a window horizontally or vertically, or
- Add and remove panes by going to `RStudio > Preferences > Pane Layout`, or use the keyboard shortcut ⌘ + , if you are on a Mac. Unfortunately, there is no default shortcut for PC users.

If you want a fully customised experience you can also alter the colour scheme of RStudio itself (`RStudio > Preferences > Appearance`) and if the themes offered are not enough for you, you can create a custom theme here[6].

[6] https://tmtheme-editor.herokuapp.com/#!/editor/theme/Monokai

5

R *Basics: The very fundamentals*

After a likely tedious installation of *R* and RStudio, as well as a somewhat detailed introduction to the RStudio interface, you are finally ready to *'do'* things. By *'doing'*, I mean *'coding'*. The term *'coding'* in itself can instil fear in some of you, but you only need one skill to do it: Writing. As mentioned earlier, learning coding or programming means learning a new language. However, once you have the basic grammar down, you already can communicate quite a bit. In this section, we will explore the fundamentals of *R*. These build the foundation for everything that follows. After that, we dive right into some analysis.

5.1 Basic computations in *R*

The most basic computation you can do in *R* is arithmetic operations. In other words, addition, subtraction, multiplication, division, exponentiation and extraction of roots. In short, *R* can be used like your pocket calculator, or more likely the one you have on your phone. For example, in Chapter @ref(the-console-window) we already performed an addition. Thus, it might not come as a surprise how their equivalents work in *R*. Let's take a look at the following examples:

```
# Addition
10 + 5
```

```
[1] 15
```

```
# Subtraction
10 - 5
```

```
[1] 5
```

```
# Multiplication
10 * 5
```

```
[1] 50
```

```
# Division
10 / 5
```

```
[1] 2
```

```
# Exponentiation
10 ^ 2
```

```
[1] 100
```

```
# Square root
sqrt(10)
```

```
[1] 3.162278
```

They all look fairly straightforward except for the extraction of roots. As you probably know, extracting the root would typically mean we use the symbol $\sqrt{}$ on our calculator. To compute the square root in *R*, we have to use a function instead to perform the computation. So we first put the name of the function `sqrt` and then the value `10` within parenthesis `()`. This results in the following code: `sqrt(10)`. If we were to write this down in our report, we would write $\sqrt[2]{10}$.

Functions are an essential part of *R* and programming in general. You will learn more about them in Chapter @ref(functions). Besides arithmetic operations, there are also logical queries you can perform. Logical queries always return either the value `TRUE` or `FALSE`. Here are some examples which make this clearer:

```
#1 Is it TRUE or FALSE?
1 == 1
```

```
[1] TRUE
```

```
#2 Is 45 bigger than 55?
45 > 55
```

```
[1] FALSE
```

```
#3 Is 1982 bigger or equal to 1982?
1982 >= 1982
```

```
[1] TRUE
```

```
#4 Are these two words NOT the same?
"Friends" != "friends"
```

```
[1] TRUE
```

```
#5 Are these sentences the same?
"I love statistics" == "I love statistícs"
```

```
[1] FALSE
```

Reflecting on these examples, you might notice three important aspects of logical queries:

1. We have to use `==` instead of `=`,
2. We can compare non-numerical values, i.e. text, which is also known as `character` values, with each other,
3. The devil is in the details (consider #5).

One of the most common mistakes of *R* novices is the confusion around the `==` and `=` notation. While `==` represents `equal to`, `=` is used to assign a value to an object (for more details on assignments see Chapter @ref(assigning-values-to-objects)). However, in practice, most *R* programmers tend to avoid `=` since it can easily lead to confusion with `==`. As such, you can strike `=` out of your R vocabulary for now.

There are many different logical operations you can perform. Table @ref(tab:logical-operators-r) lists the most frequently used logical operators for your reference. These will become important, for example when we filter our data for analysis, e.g. include only female or male participants.

Table 5.1: Logical operators in R

Operator	Description
==	is equal to
!=	is not equal to
>=	is bigger or equal to
>	is bigger than
<=	is smaller or equal to
<	is smaller than

a | b	a or b
a & b	a and b
!a	is not a

5.2 Assigning values to objects: '<-'

Another common task you will perform is assigning values to an object. An object can be many different things:

- a dataset,
- the results of a computation,
- a plot,
- a series of numbers,
- a list of names,
- a function,
- etc.

In short, an object is an umbrella term for many different things which form part of your data analysis. For example, objects are handy when storing results that you want to process further in later analytical steps. We have to use the assign operator `<-` to assign a value to an object. Let's have a look at an example.

```
# I have a friend called "Fiona"
friends <- "Fiona"
```

In this example, I created an object called `friends` and added `"Fiona"` to it. Since `"Fiona"` represents a `string`, we need to use `""`. So, if you wanted to read this line of code, you would say, '`friends` gets the value `"Fiona"`'. Alternatively, you could also say '`"Fiona"` is assigned to `friends`'.

If you look into your environment pane, you will find the object `friends`. You can see it carries the value `"Fiona"`. We can also print values of an object in the console by simply typing the name of the object `friends` and hit `Return` ⏎.

```
# Who are my friends?
friends
```

```
[1] "Fiona"
```

Sadly, it seems I only have one friend. Luckily we can add some more, not the least to make me feel less lonely. To create objects with multiple values, we can use the function `c()`, which stands for 'concatenate'. The Cambridge Dictionary (2021) defines this word as follows:

> '***concatenate***',
>
> *to put things together as a connected series.*

Let's concatenate some more friends into our `friends` object.

```
# Adding some more friends to my life
friends <- c("Fiona",
             "Lukas",
             "Ida",
             "Georg",
             "Daniel",
             "Pavel",
             "Tigger")

# Here are all my friends
friends
```

```
[1] "Fiona"  "Lukas"  "Ida"    "Georg"  "Daniel" "Pavel"  "Tigger"
```

To concatenate values into a single object, we need to use a comma `,` to separate each value. Otherwise, *R* will report an error back.

```
friends <- c("Fiona" "Ida")
```

R's error messages tend to be very useful and give meaningful clues to what went wrong. In this case, we can see that something '*unexpected*' happened, and it shows where our mistake is. You can also concatenate numbers

Btw, if you add `()` around your code, you can automatically print the content of the object to the console. Thus,

- `(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))` is the same as
- `milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)` followed by `milestones_of_my_life`.

You can copy and paste the following example to illustrate what I explained.

```
# Important years in my life
milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)
milestones_of_my_life
```

```
[1] 1982 2006 2011 2018 2020
```

```
# The same as above - no second line of code needed
(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))
```

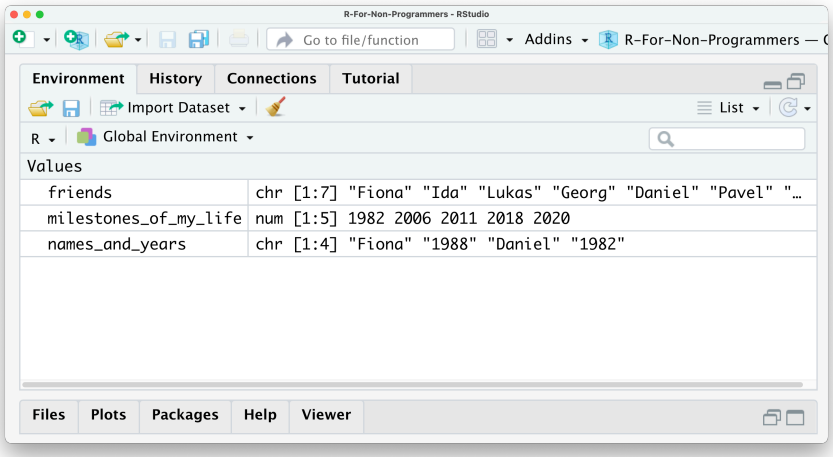
```
[1] 1982 2006 2011 2018 2020
```

Finally, we can also concatenate numbers and character values into one object:

```
(names_and_years <- c("Fiona", 1988, "Daniel", 1982))
```

```
[1] "Fiona"  "1988"   "Daniel" "1982"
```

This last example is not necessarily something I would recommend to do, because it likely leads to undesirable outcomes. For example, if you look into your environment pane you currently have three objects: `friends`, `milestones_of_my_life`, and `names_and_years`.



The `friends` object shows that all the values inside the object are classified as `chr`, which denominates `character`. For this object, this is correct because it only includes the names of my friends. On the other hand, the object `milestones_of_my_life` only includes `numeric` values, and therefore it says `num` in the environment pane. However, for the object `names_and_years` we know we

want to have `numeric` and `character` values included. Still, *R* recognises them as `character` values because values inside objects are meant to be of the same type. Remember that in *R,* numbers can always be interpreted as `character` and `numeric`, but text only can be considered as `character`.

Consequently, mixing different types of data into one object is likely a bad idea. This is especially true if you want to use the `numeric` values for computation. We will return to data types in Chapter @ref(change-data-types) where we learn how to change them, but for now we should ensure that our objects are all of the same data type.

However, there is an exception to this rule. 'Of course', you might say. There is one object that can have values of different types: a `list`. As the name indicates, a `list` object holds several items. These items are usually other objects. In the spirit of 'Inception[1]', you can have lists inside lists, which contain more lists or other objects.

Let's create a `list` called `x_files` using the `list` function and place all our objects inside.

```
# This creates our list of objects
x_files <- list(friends,
                milestones_of_my_life,
                names_and_years)

# Let's have a look what is hidden inside the x_files
x_files
```

```
[[1]]
[1] "Fiona"  "Lukas"  "Ida"    "Georg"  "Daniel" "Pavel"  "Tigger"

[[2]]
[1] 1982 2006 2011 2018 2020

[[3]]
[1] "Fiona"  "1988"   "Daniel" "1982"
```

You will notice in this example that I do not use `""` for each value in the list. This is because `friends` is not a `character` value, but an object. When we refer to objects, we do not need quotation marks.

We will encounter `list` objects quite frequently when we perform our analysis. Some functions return the results in the format of lists. This can be very helpful because otherwise our environment pane will be littered with objects and we would not necessarily know how they relate to each other, or worse, to

[1] https://www.imdb.com/title/tt1375666/?ref_=ext_shr_lnk

which analysis they belong. Looking at the list item in the environment page (Figure 5.1), you can see that the object `x_files` is classified as a `List of 3`, and if you click on the blue icon, you can inspect the different objects inside.

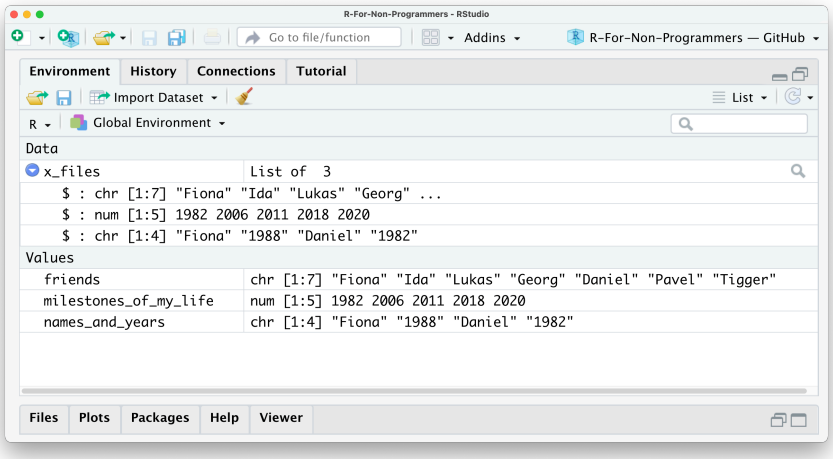


Figure 5.1: The environment pane showing our objects and our list `x_files`

In Chapter (**basic-computations-in-r?**) , I mentioned that we should avoid using the = operator and explained that it is used to assign values to objects. You can, if you want, use = instead of <-. They fulfil the same purpose. However, as mentioned before, it is not wise to do so. Here is an example that shows that, in principle, it is possible.

```
# DO
(avengers1 <- c("Iron Man",
                "Captain America",
                "Black Widow",
                "Vision"))
```

```
[1] "Iron Man"        "Captain America" "Black Widow"     "Vision"
```

```
# DON'T
(avengers2 = c("Iron Man",
               "Captain America",
               "Black Widow",
               "Vision"))
```

```
[1] "Iron Man"        "Captain America" "Black Widow"     "Vision"
```

On a final note, naming your objects is limited. You cannot chose any name. First, every name needs to start with a letter. Second, you can only use letters, numbers _ and . as valid components of the names for your objects (see also Chapter 4.2 in Wickham and Grolemund 2016). I recommend to establish a naming convention that you adhere to. Personally I prefer to only user lowercase letters and _ to separate/connect words. Ideally, you want to keep names informative, succinct and precise. Here are some examples of what some might consider good and bad choices for names.

```
# Good choices
income_per_annum
open_to_exp          # for 'openness to new experiences'
soc_int              # for 'social integration'

# Bad choices
IncomePerAnnum
measurement_of_boredom_of_watching_youtube
Sleep.per_monthsIn.hours
```

Ultimately, you need to be able to effectively work with your data and output. Ideally, this should be true for others as well who want or need to work with your analysis and data as well, e.g. your co-investigator or supervisor. The same is applies to column names in datasets (see Chapter @ref(colnames-cleaning)). Some more information about coding style and coding etiquette can be found in Chapter @ref(coding-etiquette).

5.3 Functions

I used the term '*function*' multiple times, but I never thoroughly explained what they are and why we need them. In simple terms, functions are objects. They contain lines of code that someone has written for us or we have written ourselves. One could say they are code snippets ready to use. Someone else might see them as shortcuts for our programming. Functions increase the speed with which we perform our analysis and write our computations and make our code more readable and reliable. Consider computing the `mean` of values stored in the object `pocket_money`.

```
# First we create an object that stores our desired values
pocket_money <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

#1 Manually compute the mean
```

```
sum <- 0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 + 89
sum / 12 # There are 12 items in the object
```

```
[1] 19.33333
```

```
#2 Use a function to compute the mean
mean(pocket_money)
```

```
[1] 19.33333
```

```
#3 Let's make sure #1 and #2 are actually the same
sum / 12 == mean(pocket_money)
```

```
[1] TRUE
```

If we manually compute the mean, we first calculate the sum of all values in the object `pocket_money`[2]. Then we divide it by the number of values in the object, which is `12`. This is the traditional way of computing the mean as we know it from primary school. However, by simply using the function `mean()`, we not only write considerably less code, but it is also much easier to understand because the word *'mean'* does precisely what we would expect. Which approach do you find easier: Using the function `mean()` or compute it by hand?

To further illustrate how functions look like, let's create one ourselves and call it `my_mean`.

```
my_mean <- function(numbers){
  # Compute the sum of all values in 'numbers'
  sum <- sum(numbers)

  # Divide the sum by the number of items in 'numbers'
  result <- sum/length(numbers)

  # Return the result in the console
  return(result)
}

my_mean(pocket_money)
```

```
[1] 19.33333
```

[2]If you find the order of numbers suspicious, it is because it represents the famous Fibonacci sequence[3].

Don't worry if half of this code does not make sense to you. Writing functions is an advanced *R* skill. However, it is good to know how functions look on the 'inside'. You certainly can see the similarities between the code we have written before, but instead of using actual numbers, we work with placeholders like `numbers`. This way, we can use a function for different data and do not have to rewrite it every time. Writing and using functions relates to the skill of abstraction mentioned in Chapter @ref(programming-languages-enhance-your-conceptual-thinking).

All functions in *R* share the same structure. They have a `name` followed by `()`. Within these parentheses, we put `arguments`, which have specific `values`. For example, a generic structure of a function would look something like this:

```
name_of_function(argument_1 = value_1,
                 argument_2 = value_2,
                 argument_3 = value_3)
```

How many arguments there are and what kind of values you can provide is very much dependent on the function you use. Thus, not every function takes every value. In the case of `mean()`, the function takes an object which holds a sequence of `numeric` values. It would make very little sense to compute the mean of our `friends` object, because it only contains names. *R* would return an error message:

```
mean(friends)
```

```
Warning in mean.default(friends): argument is not numeric or logical: returning
NA
```

```
[1] NA
```

`NA` refers to a value that is *'not available'*. In this case, *R* tries to compute the mean, but the result is not available, because the values are not `numeric` but a `character`. In your dataset, you might find values that are `NA`, which means there is data missing. If a function attempts a computation that includes even just a single value that is `NA`, *R* will return `NA`. However, there is a way to fix this. You will learn more about how to deal with `NA` values in Chapter @ref(dealing-with-missing-data).

Sometimes you will also get a message from *R* that states `NaN`. `NaN` stands for *'not a number'* and is returned when something is not possible to compute, for example:

```
# Example 1
0 / 0
```

```
[1] NaN
```

```
# Example 2
sqrt(-9)
```

```
Warning in sqrt(-9): NaNs produced
```

```
[1] NaN
```

5.4 R packages

R has many built-in functions that we can use right away. However, some of the most interesting ones are developed by different programmers, data scientists and enthusiasts. To add more functions to your repertoire, you can install *R* packages. *R* packages are a collection of functions that you can download and use for your own analysis. Throughout this book, you will learn about and use many different *R* packages to accomplish various tasks. To give you another analogy,

- *R* is like a global supermarket where everyone can offer their products,
- RStudio is like my shopping cart where I can put the products I like, and
- *R* packages are the products I can pick from the shelves.

Luckily, *R* packages are free to use, so I do not have to bring my credit card. For me, these additional functions, developed by some of the most outstanding scientists, is one of many reasons that keeps me addicted to performing my research in *R*.

R packages do not only include functions but often include datasets and documentation of what each function does. This way, you can easily try every function right away, even without your own dataset and read through what each function in the package does. Figure 5.2 shows the documentation of an *R* package called `ggplot2`.

However, how do you find those *R* packages? They are right at your fingertips. You have two options:

1. Use the function `install.packages()`, or
2. Use the packages pane in RStudio.

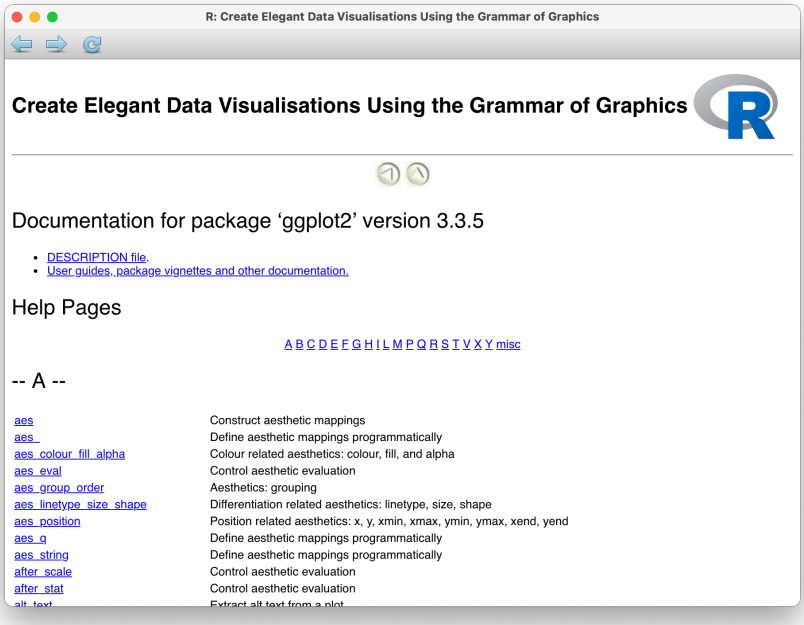


Figure 5.2: The R package documentation for 'ggplot2'

5.4.1 Installing packages using `install.packages()`

The simplest and fastest way to install a package is calling the function `install.packages()`. You can either use it to install a single package or install a series of packages all at once using our trusty `c()` function. All you need to know is the name of the package. This approach works for all packages that are on CRAN (remember CRAN from Chapter @ref(installing-r)?).

```
# Install a single package
install.packages("tidyverse")

# Install multiple packages at once
install.packages(c("tidyverse", "naniar", "psych"))
```

If a package is not available from CRAN, chances are you can find them on GitHub[4]. GitHub is probably the world's largest global platform for programmers from all walks of life, and many of them develop fantastic *R* packages that make programming and data analysis not just easier but a lot more fun.

[4] https://github.com

As you continue to work in *R*, you should seriously consider creating your own account to keep backups of your projects (see also Chapter @ref(next-steps-github)).

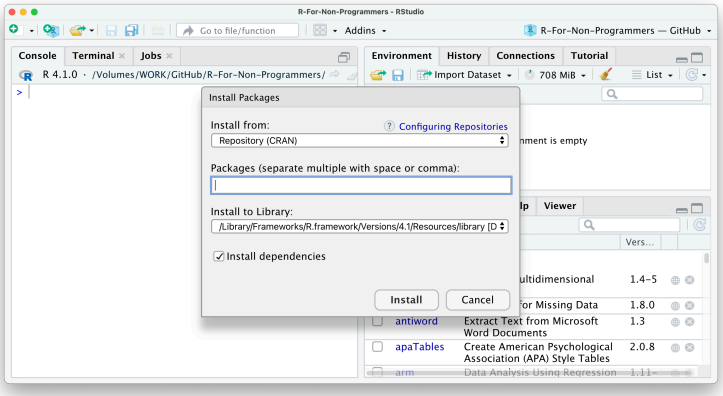
An essential companion for this book is `r4np`, which contains all datasets for this book and some useful functions to get you up and running in no time. Since it is currently only available on Github, you can use the following code snippet to install it. However, you have to install the package `devtools` first to use the function `install_github()`.

```
# Install the 'devtools' package first
install.packages("devtools")

# Then install the 'r4np' package from GitHub
devtools::install_github("ddauber/r4np")
```

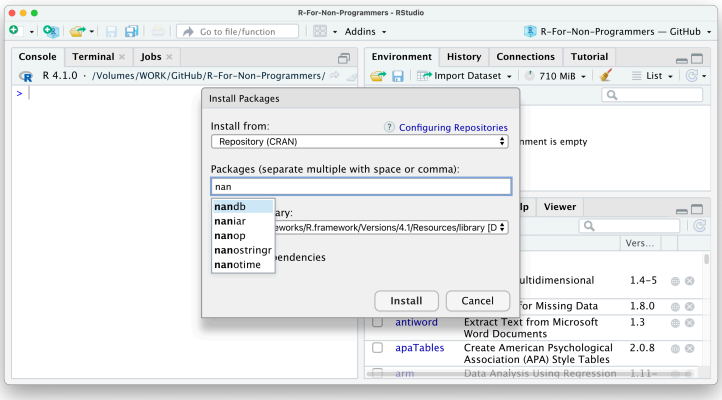
5.4.2 Installing packages via RStudio's package pane

RStudio offers a very convenient way of installing packages. In the packages pane, you cannot only see your installed packages, but you have two more buttons: `Install` and `Update`. The names are very self-explanatory. To install an *R* package you can follow the following steps:

1. Click on `Install`.
2. In most cases, you want to make sure you have `Repository (CRAN)` selected.

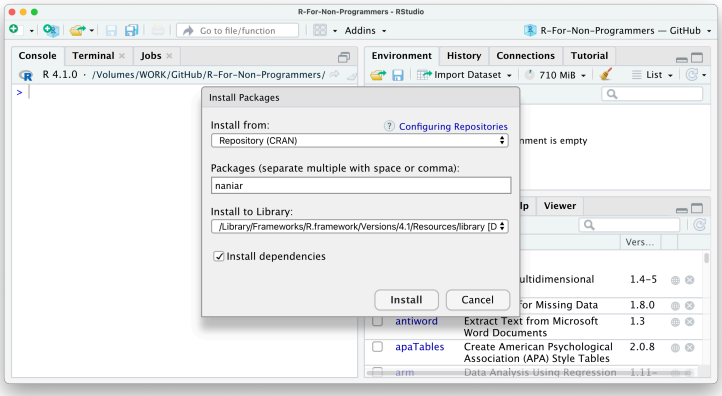


3. Type in the name of the package you wish to install. RStudio offers an auto-complete feature to make it even easier to find the package you want.



4. I recommend NOT to change the option which says `Install to library`. The default library settings will suffice.

5. Finally, I recommend to select `Install dependencies`, because some packages need other packages to function properly. This way, you do not have to do this manually.



The only real downside of using the packages pane is that you cannot install packages hosted on GitHub only. However, you can download them from there and install them directly from your computer using this option. This is particularly useful if you do not have an internet connection but you already downloaded the required packages onto a hard drive. However, in 99.9% of cases it is much easier to use the function `devtools::install_github()`.

5.4.3 Install all necessary *R* packages for this book

Lastly, if you would like to install the necessary packages to follow the examples in this book, the `r4np` package comes with a handy function to install them all at once. Of course, you need to have `r4np` installed first, as shown above.

```
# Install all R packages used in this book
r4np::install_r4np()
```

5.4.4 Using *R* Packages

Now that you have a nice collection of *R* packages, the next step would be to use them. While you only have to install *R* packages once, you have to 'activate' them every time you start a new session in RStudio. This process is also called 'loading an *R* package'. Once an *R* package is loaded, you can use all its functions. To load an *R* package, we have to use the function `library()`.

```
library(tidyverse)
```

The `tidyverse` package is a special kind of package. It contains multiple packages and loads them all at once. Almost all included packages you will use at some point when working through this book.

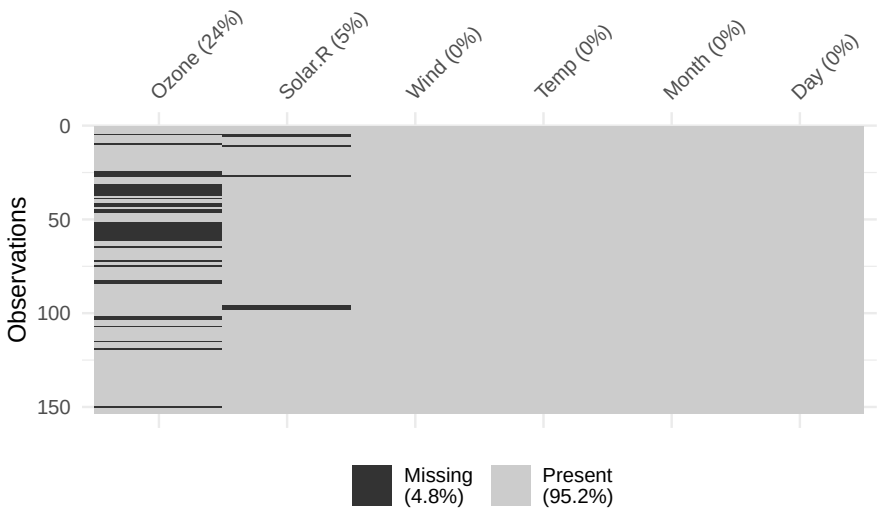
I know what you are thinking. Can you use `c()` to load all your packages at once? Unfortunately not. However, there is a way to do this, but it goes beyond the scope of this book to fully explain this. If you are curious, you can take a peek at the code here on stackoverflow.com[5].

Besides, it is not always advisable to load all functions of an entire package. One reason could be that two packages contain a function with the same name but with a different purpose. Two functions with the same name create a conflict between these two packages, and one of the functions would not be usable. Another reason could be that you only need to use the function once, and loading the whole package to use only one specific function seems excessive. Instead, you can explicitly call functions from packages without loading the package. For example, we might want to use the `vis_miss()` function from the `naniar` package to show where data is missing in our dataset `airquality`. Using functions without `library()` is also much quicker than loading the package and then calling the function if you don't use it repeatedly. Make sure you have `naniar` installed (see above). We will work with this package when we explore missing data in Chapter @ref(dealing-with-missing-data). To use a function from an *R* package without loading it, we have to use `::` between the

[5] https://stackoverflow.com/questions/8175912/load-multiple-packages-at-once

package's name and the function we want to use. Copy the code below and try it yourself.

```
# Here I use the dataset 'airquality', which comes with R
naniar::vis_miss(airquality)
```



5.5 Coding etiquette

Now you know everything to get started, but before we jump into our first project, I would like to briefly touch upon coding etiquette. This is not something that improves your analytical or coding skills directly, but is essential in building good habbits and making your life and those of others a little easier. Consider writing code like growing plants in your garden. You want to nurture the good plants, remove the weed and add labels that tell you which plant it is that you are growing. At the end of the day, you want your garden to be well-maintained. Treat you programming code the same way.

A script (see Chapter @ref(creating-an-r-script)) of programming code should always have at least the following qualities:

- Only contains code that is necessary,
- Is easy to read and understand,
- Is self-contained.

With simple code this is easily achieved. However, what about more complex and longer code representing a whole set of analytical steps?

```
# Very messy code

library(tidyverse)
library(jtools)
model1 <- lm(covid_cases_per_1m ~ idv, data = df)
summ(model1, scale = TRUE, transform.response = TRUE, vifs = TRUE)
df %>% ggplot(aes(x = covid_cases_per_1m, y = idv, col = europe, label = country))+
theme_minimal()+ geom_label(nudge_y = 2) + geom_point()
model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe, data = df)
summ(model2, scale = TRUE, transform.response = TRUE, vifs = TRUE)
anova(model1, model2)
```

How about the following in comparison?

```
# Nicely structured code

# Load required R packages
library(tidyverse)
library(jtools)

# ---- Modelling COVID-19 cases ----

## Specify and run a regression
model1 <- lm(covid_cases_per_1m ~ idv, data = df)

## Retrieve the summary statistics of model1
summ(model1,
     scale = TRUE,
     transform.response = TRUE,
     vifs = TRUE)

# Does is matter whether a country lies in Europe?

## Visualise rel. of covid cases, idv and being a European country
df %>%
  ggplot(aes(x = covid_cases_per_1m,
             y = idv,
             col = europe,
             label = country)) +
  theme_minimal() +
  geom_label(nudge_y = 2) +
```

```
  geom_point()

## Specify and run a revised regression
model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe,
                 data = df)

## Retrieve the summary statistics of model2
summ(model2,
     scale = TRUE,
     transform.response = TRUE,
     vifs = TRUE)

## Test whether model2 is an improvement over model1
anova(model1, model2)
```

I hope we can agree that the second example is much easier to read and understand even though you probably do not understand most of it yet. For once, I separated the different analytical steps from each other like paragraphs in a report. Apart from that, I added comments with `#` to provide more context to my code for someone else who wants to understand my analysis. Admittedly, this example is a little excessive. Usually, you might have fewer comments. Commenting is an integral part of programming because it allows you to remember what you did. Ideally, you want to strike a good balance between commenting on and writing your code. How many comments you need will likely change throughout your *R* programming journey. Think of comments as headers for your programming script that give it structure.

We can use `#` not only to write comments but also to tell R not to run particular code. This is very helpful if you want to keep some code but do not want to use it yet. There is also a handy keyboard shortcut you can use to 'deactivate' multiple lines of code at once. Select whatever you want to 'comment out' in your script and press `Ctrl+Shift+C` (PC) or `Cmd+Shift+C` (Mac).

```
# mean(pocket_money) # R will NOT run this code
mean(pocket_money)   # R will run this code
```

RStudio helps a lot with keeping your coding tidy and properly formatted. However, there are some additional aspects worth considering. If you want to find out more about coding style, I highly recommend to read through *'The tidyverse style guide'*[6] (Wickham 2021).

[6] https://style.tidyverse.org

5.6 Exercises

It is time to practice the newly acquired skills. The `r4np` package comes with interactive tutorials. In order to start them, you have two options. If you have installed the `r4np` package already and used the function `install_r4np()` you can use `Option 1`:

```
# Option 1:
learnr::run_tutorial("ex_r_basics", package = "r4np")
```

If you have not installed the `r4np` package and/or not run the function `install_r4np()`, you will have to do this first using `Option 2`:

```
# Option 2:
## Install 'r4np' package
devtools::install_github("ddauber/r4np")

## Install all relevant packages for this book
r4np::install_r4np()

## Start the tutorial for this chapter
learnr::run_tutorial("ex_r_basics", package = "r4np")
```

6

Starting your R *projects*

Every new project likely fills you with enthusiasm and excitement. And it should. You are about to find answers to your research questions, and you hopefully come out more knowledgeable due to it. However, there are likely certain aspects of data analysis that you find less enjoyable. I can think of two:

- Keeping track of all the files my project generates
- Data wrangling

While we cover data wrangling in great detail in the next Chapter (Chapter @ref(data-wrangling)), I would like to share some insights from my work that helped me stay organised and, consequently, less frustrated. The following applies to small and large research projects, which makes it very convenient no matter the situation and the scale of the project. Of course, feel free to tweak my approach to whatever suits you. However, consistency is king.

6.1 Creating an *R* Project file

When working on a project, you likely create many different files for various purposes, especially *R* Scripts (see Chapter @ref(creating-an-r-script)). If you are not careful, this file is stored in your system's default location, which might not be where you want them to be. RStudio allows you to manage your entire project intuitively and conveniently through *R* Project files. Using *R* Project files comes with a couple of perks, for example:

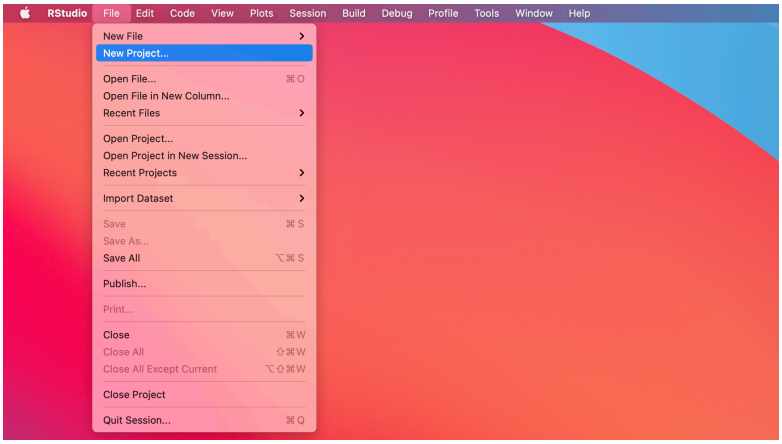
- All the files that you generate are in the same place. Your data, your coding, your exported plots, your reports, etc., all are in one place together without you having to manage the files manually. This is due to the fact that RStudio sets the root directory to whichever folder your project is saved to.
- If you want to share your project, you can share the entire folder, and others can quickly reproduce your research or help fix problems. This is because all file paths are relative and not absolute.
- You can, more easily, use GitHub for backups and so-called 'version control',

which allows you to track changes you have made to your code over time (see also Chapter @ref(next-steps-github)).

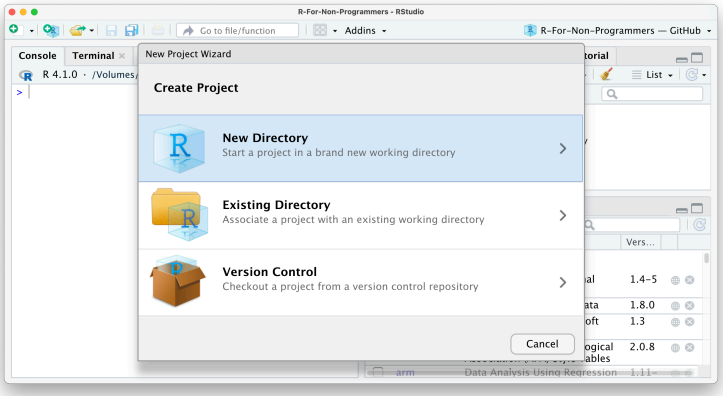
For now, the most important reason to use *R* Project files is the convenience of the organisation of files and the ability to share it easily with co-investigators, your supervisor, or your students.

To create an R Project, you need to perform the following steps:

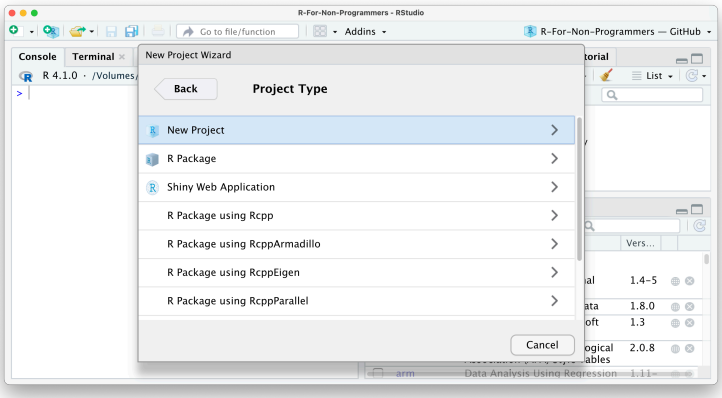
1. Select `File > New Project…` from the menu bar.



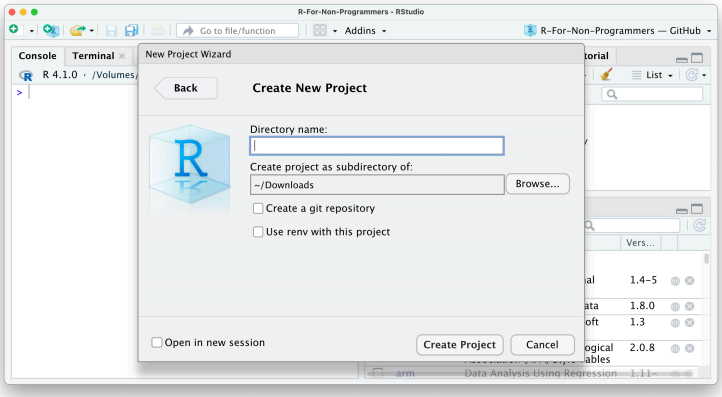
2. Select `New Directory` from the popup window.



3. Next, select `New Project`.

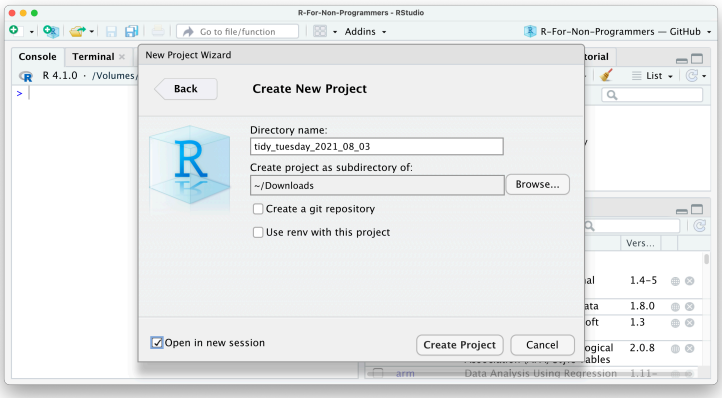


4. Pick a meaningful name for your project folder, i.e. the `Directory Name`. Ensure this project folder is created in the right place. You can change the `subdirectory` by clicking on `Browse...`. Ideally the subdirectory is a place where you usually store your research projects.

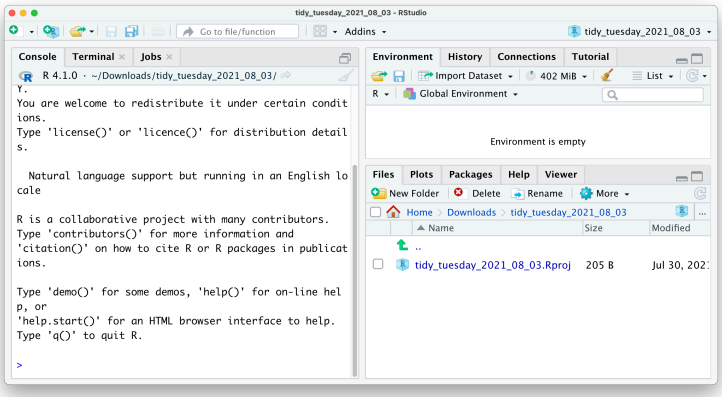


5. You have the option to `Create a git repository`. This is only relevant if you already have a GitHub account and wish to use version control. For now, you can happily ignore it if you do not use GitHub.

6. Lastly, tick `Open in new session`. This will open your *R* Project in a new RStudio window.



7. Once you are happy with your choices, you can click `Create Project`. This will open a new *R* Session, and you can start working on your project.



If you look carefully, you can see that your RStudio is now 'branded' with your project name. At the top of the window, you see the project name, the files pane shows the root directory where all your files will be, and even the console shows on top the file path of your project. You could set all this up manually, but I would not recommend it, not the least because it is easy and swift to work with *R* Projects.

6.2 Organising your projects

This section is not directly related to RStudio, *R* or data analysis in general. Instead, I want to convey to you that a good folder structure can go a long way. It is an excellent habit to start thinking about folder structures before you start working on your project. Placing your files into dedicated folders, rather than keeping them loosely in one container, will speed up your work and save you from the frustration of not finding the files you need. I have a template that I use regularly. You can either create it from scratch in RStudio or open your file browser and create the folders there. RStudio does not mind which way you do it. If you want to spend less time setting this up, you might want to use the function `create_project_folder()` from the `r4np` package. It creates all the folders as shown in Figure @ref(fig:folder-structure).

```
# Install 'r4np' from GitHub
devtools::install_github("ddauber/r4np")

# Create the template structure
r4np::create_project_folder()
```

To create a folder, click on `New Folder` in the *Files* pane. I usually have at least the following folders for every project I am involved in:

- A folder for my raw data. I store 'untouched' datasets in it. With 'untouched', I mean they have not been processed in any way and are usually files I downloaded from my data collection tool, e.g. an online questionnaire platform.
- A folder with 'tidy' data. This is usually data I exported from *R* after cleaning it, i.e. after some basic data wrangling and cleaning (see Chapter @ref(data-wrangling)).
- A folder for my *R* scripts
- A folder for my plots
- A folder for reports.

Thus, in RStudio, it would look something like this:

You probably noticed that my folders have numbers in front of them. I do this to ensure that all folders are in the order I want them to be, which is usually not the alphabetical order my computer suggests. I use two digits because I may have more than nine folders for a project, and folder ten would otherwise be listed as the third folder in this list. With this filing strategy in place, it will be easy to find whatever I need. Even others can easily understand what I stored where. It is simply 'tidy', similar to how we want our data to be.

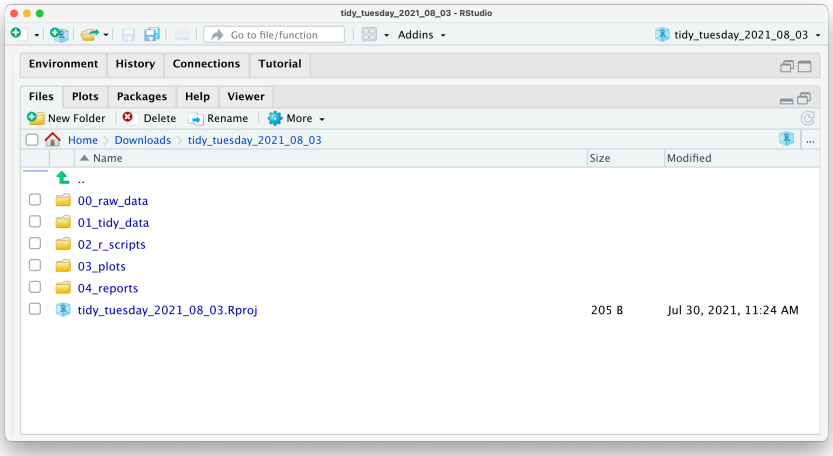


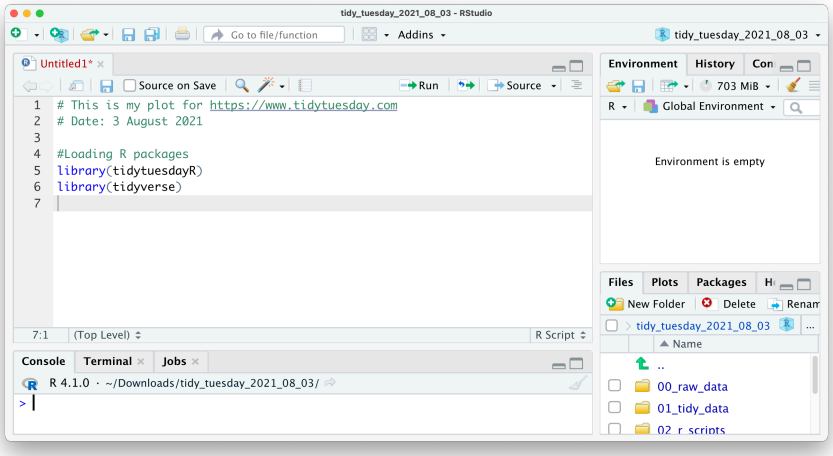
Figure 6.1: An example of a scalable folder structure for your project

6.3 Creating an *R* Script

Code quickly becomes long and complex. Thus, it is not very convenient to write it in the console. So, instead, we can write code into an *R* Script. An *R* Script is a document that RStudio recognises as *R* programming code. Files that are not *R* Scripts, like `.txt`, `.rtf` or `.md`, can also be opened in RStudio, but any code written in it will not be automatically recognised.

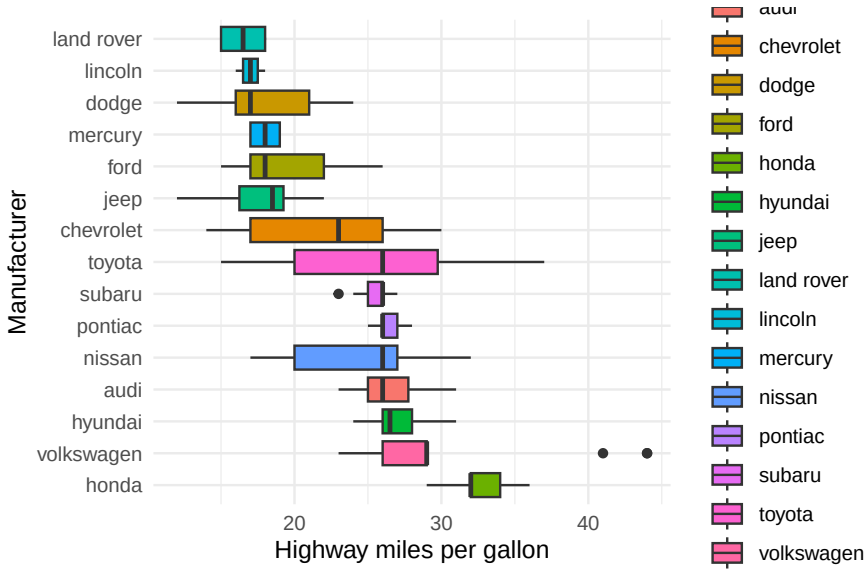
When opening an *R* script or creating a new one, it will display in the *Source* window (see Chapter @ref(the-source-window)). Some refer to this window as the 'script editor'. An *R* script starts as an empty file. Good coding etiquette (see Chapter @ref(coding-etiquette)) demands that we use the first line to indicate what this file does by using a comment `#`. Here is an example for a 'TidyTuesday'[1] *R* Project.

[1] https://www.tidytuesday.com

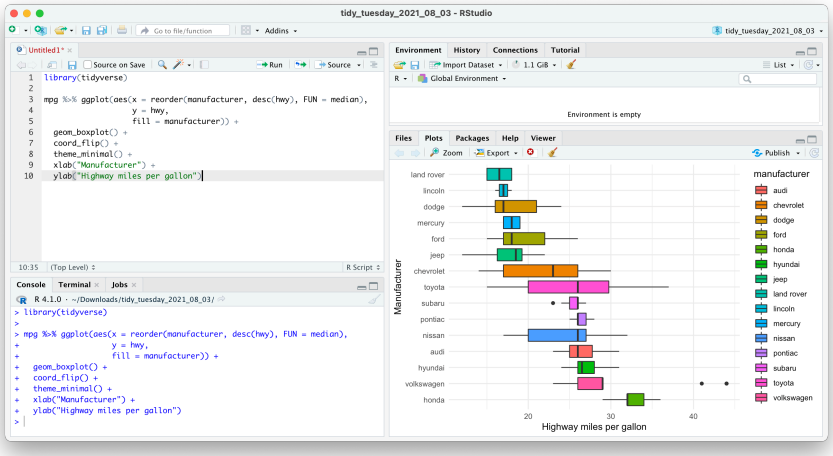


All examples in this book can easily be copied and pasted into your own *R* script. However, for some code you will have to install the *R* package `r4np` (see above). Let's try it with the following code. The plot this code creates reveals which car manufacturer produces the most efficient cars. Copy and paste this code into your *R* script.

```
library(tidyverse)

mpg %>% ggplot(aes(x = reorder(manufacturer, desc(hwy), FUN = median),
                   y = hwy,
                   fill = manufacturer)) +
  geom_boxplot() +
  coord_flip() +
  theme_minimal() +
  xlab("Manufacturer") +
  ylab("Highway miles per gallon")
```



You are probably wondering where your plot has gone. Copying the code will not automatically run it in your *R* script. However, this is necessary to create the plot. If you tried pressing `Return` ⏎, you would only add a new line. Instead, you need to select the code you want to run and press `Ctrl+Return` ⏎ (PC) or `Cmd+Return` ⏎ (Mac). You can also use the `Run` command at the top of your source window, but it is much more efficient to press the keyboard shortcut. Besides, you will remember this shortcut quickly, because we need to use it very frequently. If all worked out, you should see the following:



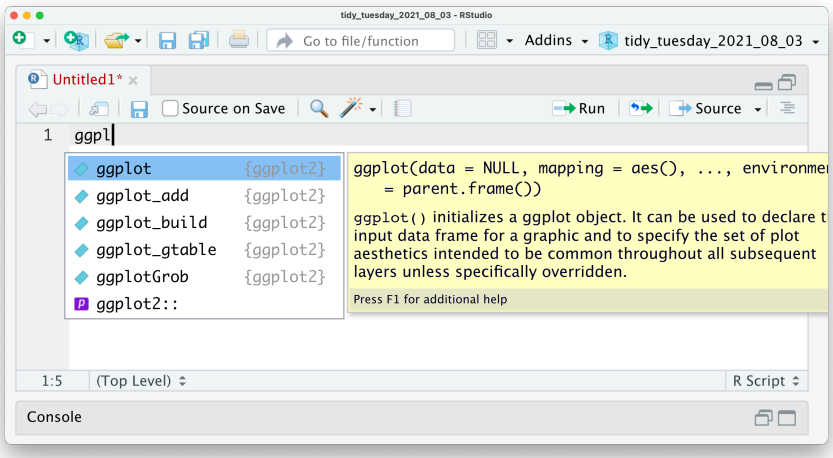
As you can see, cars from `Honda` appear to drive furthest with the same amount

of fuel (a gallon) compared to other vehicles. Thus, if you are looking for very economical cars, you now know where to find them.

The *R* script editor has some conveniences for writing your code that are worth pointing out. You probably noticed that some of the code we have pasted is blue, and some other code is in green. These colours help to make your code more readable because they carry a specific meaning. In the default settings, green stands for any values in `""`, which usually stands for `characters`. Automatic colouring of our progamming code is also called *'syntax highlighting'*.

Moreover, code in *R* scripts will be automatically indented to facilitate reading. If, for whatever reason, the indentation does not happen, or you accidentally undo it, you can reindent a line with `Ctrl+I` (PC) or `Cmd+I` (Mac).

Lastly, the console and the *R* script editor both feature code completion. This means that when you start typing the name of a function, *R* will provide suggestions. These are extremely helpful and make programming a lot faster. Once you found the function you were looking for, you press `Return ⌧` to insert it. Here is an example of what happens when you have the package `tidyverse` loaded and type `ggpl`. Only functions that are loaded via packages or any object in your environment pane benefit from code completion.



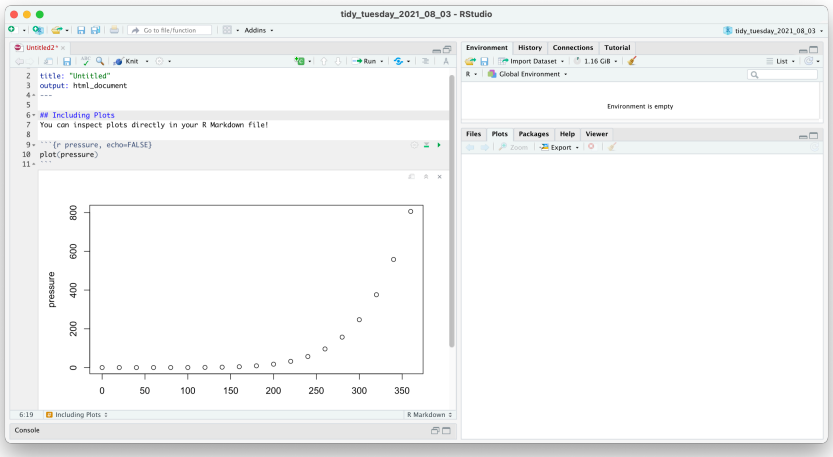
Not only does RStudio show you all the available options, but it also tells you which package this function is from. In this case, all listed functions are from the `ggplot2` package. Furthermore, when you select one of the options but have not pressed `Return ⌧` yet, you also get to see a yellow box, which provides you with a quick reference of all the arguments that this function accepts. So you do not have to memorise all the functions and their arguments.

6.4 Using *R* Markdown

There is too much to say about *R* Markdown, which is why I only will highlight that it exists and point out the one feature that might convince you to choose this format over plain *R* scripts: They look like a Word document (almost).

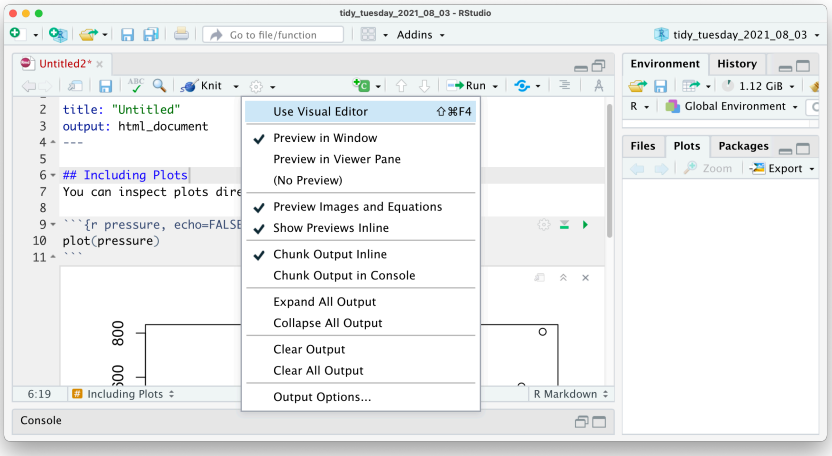
As the name indicates, *R* Markdown files are a combination of *R* scripts and '*Markdown*'. '*Markdown*' is a way of writing and formatting text documents without needing software like MS Word. Instead, you write everything in plain text. Such plain text can be converted into many different document types such as HTML websites, PDF or Word documents. If you would like to see how it works, I recommend looking at the R Markdown Cheatsheet[2]. To create an *R* Markdown file click on `File > New File > R Markdown...`.

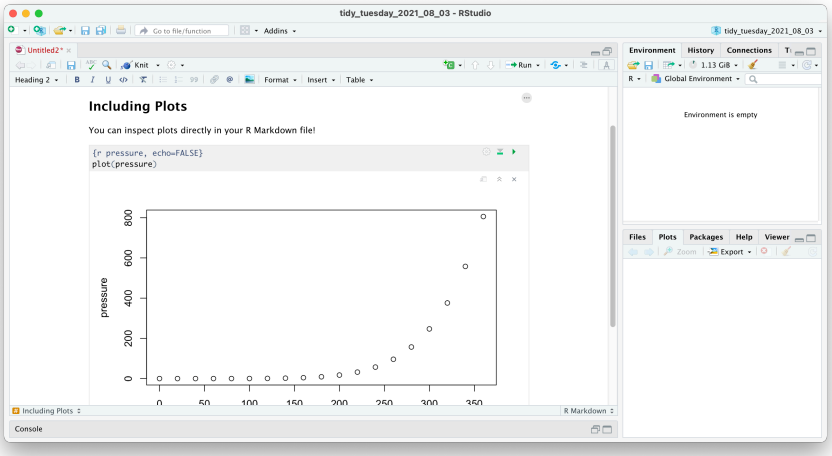
An *R* Markdown file works oppositely to an *R* script. By default, an *R* script considers everything as code and only through commenting `#` we can include text to describe what the code does. This is what you have seen in all the coding examples so far. On the other hand, an *R* Markdown file considers everything as text, and we have to specify what is code. We can do so by inserting *'code chunks'*. Therefore, there is less of a need to use comments `#` in *R* Markdown files because you can write about it. Another convenience of *R* Markdown files is that results from your analysis are immediately shown underneath the code chunk and not in the console.



If you switch your view to the `Visual Editor`, it almost looks like you are writing a report in MS Word.

[2] https://www.rstudio.com/resources/cheatsheets/





So, when should you use an *R* script, and when should you use *R* Markdown. The rule-of-thumb is that if you intend to write a report, thesis or another form of publication, it might be better to work in an *R* Markdown file. If this does not apply, you might want to use an *R* Script. As mentioned above, *R* Markdown files emphasise text, while *R* scripts primarily focus on code. In my projects, I often have a mixture of both. I use *R* Scripts to carry out data wrangling and my primary analysis and then use *R* Markdown files to present the findings, e.g. creating plots, tables, and other components of a report. By the way, this book is written in *R* Markdown using the `bookdown` package.

No matter your choice, it will neither benefit nor disadvantage you in your *R* journey or when working through this book. The choice is all yours. You likely

will come to appreciate both formats for what they offer. If you want to find out more about *R* Markdown and how to use it, I highly recommend taking a look at 'R Markdown: The Definitive Guide'[3] (Xie, Allaire, and Grolemund 2018).

[3] https://bookdown.org/yihui/rmarkdown/

7

Data Wrangling

> **Data wrangling** — also called **data cleaning**, **data remediation**, or **data munging**—refers to a variety of processes designed to transform raw data into more readily used formats. The exact methods differ from project to project depending on the data you're leveraging and the goal you're trying to achieve. (Stobierski 2021)

You collected your data over months (and sometimes years), and all you want to know is whether your data makes sense and reveals something nobody would have ever expected. However, before we can truly go ahead with our analysis, it is essential to understand whether our data is 'tidy'. What I mean is that our data is at the quality we would expect it to be and can be reliably used for data analysis. After all, the results of our research are only as good as our data and the methods we deploy to achieve the desired insights.

Very often, the data we receive is everything else but clean, and we need to check whether our data is fit for analysis and ensure it is in a format that is easy to handle. For small datasets, this is usually a brief exercise. However, I found myself cleaning data for a month because the dataset was spread out into multiple spreadsheets with different numbers of columns and odd column names that were not consistent. Thus, data wrangling is an essential first step in any data analysis. It is a step that cannot be skipped and has to be performed on every new dataset. If your background is in qualitative research, you likely conducted interviews which you had to transcribe. The process of creating transcriptions is comparable to data wrangling in quantitative datasets - and equally tedious.

Luckily, *R* provides many useful functions to make our lives easier. You will be in for a treat if you are like me and used to do this in Excel. It is a lot simpler using *R* to achieve a clean dataset.

Here is an overview of the different steps we usually work through before starting with our primary analysis. This list is certainly not exhaustive:

- Import data,
- Check data types,
- Recode and arrange factors, i.e. categorical data, and
- Run missing data diagnostics.

7.1 Import your data

The `r4np` package hosts several different datasets to work with, but at some point, you might want to apply your *R* knowledge to your own data. Therefore, an essential first step is to import your data into RStudio. There are three different methods, all of which are very handy:

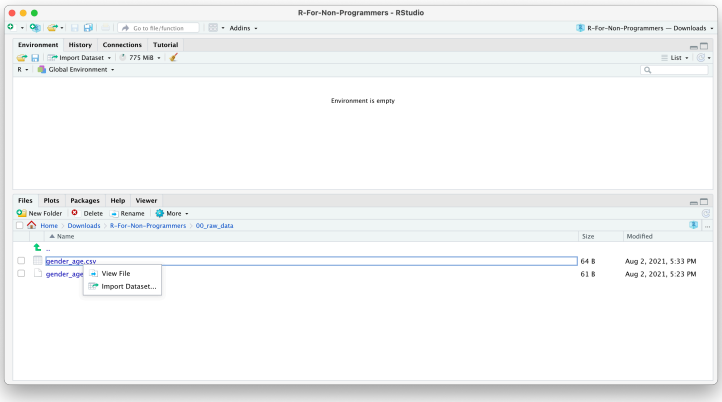
1. Click on your data file in the *Files* pane and choose `Import Dataset`.
2. Use the `Import Dataset` button in the *Environment* pane.
3. Import your data calling one of the `readr` functions in the *Console* or *R* script.

We will use the `readr` package or all three options. Using this package we can import a range of different file formats, including `.csv`, `.tsv`, `.txt`. If you want to import data from an `.xlsx` file, you need to use another package called `readxl`. Similarly, if you used SPSS, Stata or SAS files before and want to use them in *R* you can use the `haven` package. The following sections will primarily focus on using `readr` via RStudio or directly in your *Console* or *R* script.

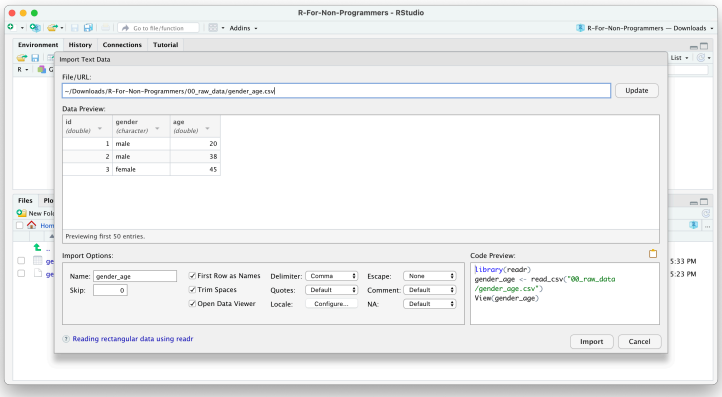
7.1.1 Import data from the *Files* pane

This approach is by far the easiest. Let's assume you have a dataset called `gender_age.csv` in your `00_raw_data` folder. If you wish to import it, you can do the following:

1. Click on the name of the file.
2. Select `Import Dataset`.



3. A new window will open, and you can choose different options. You also see a little preview of how the data looks like. This is great if you are not sure whether you did it correctly.

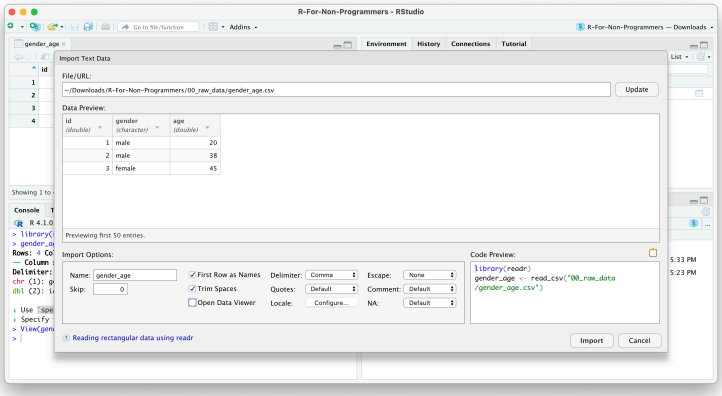


4. You can change how data should be imported, but the default should be fine in most cases. Here is a quick breakdown of the most important options:

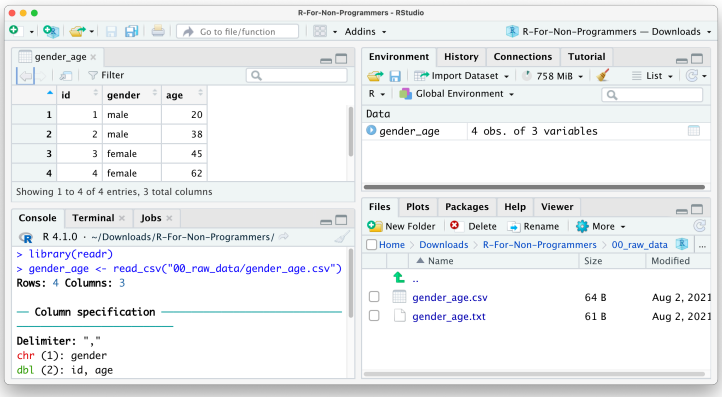
    - `Name` allows you to change the object name, i.e. the name of the object this data will be assigned to. I often use `df_raw` (`df` stand for `data frame`, which is how *R* calls such rectangular datasets).
    - `Skip` is helpful if your data file starts with several empty rows at the top. You can remove them here.
    - `First Row as Names` is ticked by default. In most Social Science

projects, we tend to have the name of the variables as the first row in our dataset.

- `Trim Spaces` removes any unnecessary whitespace in your dataset. Leave it ticked.
- `Open Data Viewer` allows you to look at your imported dataset. I use it rarely, but it can be helpful at times.
- `Delimiter` defines how your columns are separated from each other in your file. If it is a `.csv` file, it would imply it is a 'comma-separated value', i.e. `,`. This setting can be changed for different files, depending on how your data is delimited. You can even use the option `Other…` to specify a custom separation option.
- `NA` specifies how missing values in your data are acknowledged. By default, empty cells in your data will be recognised as missing data.



5. Once you are happy with your choices, you can click on `Import`.
6. You will find your dataset in the *Environment* pane.



In the *Console*, you can see that *R* also provides the `Column specification`, which we need later when inspecting 'data types'. `readr` automatically imports all text-based columns as `chr`, i.e. `character` values. However, this might not always be the correct data type. We will cover more of this aspect of data wrangling in Chapter @ref(change-data-types).

7.1.2 Importing data from the *Environment* pane

The process of importing datasets from the *Environment* pane follows largely the one from the *Files* pane. Click on `Import Dataset > From Text (readr)...`. The only main difference lies in having to find the file using the `Browse...` button. The rest of the steps are the same as above.

Be aware that you will have to use the *Environment* pane for importing data from specific file types, e.g. `.txt`. The *File* pane would only open the file but not import the data for further processing.

7.1.3 Importing data using functions directly

If you organised your files well, it could be effortless and quick to use all the functions from `readr` directly. Here are two examples of how you can use `readr` to import your data. Make sure you have the package loaded.

```
# Import data from '.csv'
read_csv("00_raw_data/gender_age.csv")

# Import data from any text file by defining the separator
read_delim("00_raw_data/gender_age.txt", delim = "|")
```

You might be wondering whether you can use `read_delim()` to import `.csv`

files too. The answer is 'Yes, you can!'. In contrast to `read_delim()`, `read_csv()` sets the delimiter to `,` by default. This is mainly for convenience because `.csv` files are one of the most popular file formats used to store data.

You might also be wondering what a *'delimiter'* is. When you record data in a plain-text file, it is easy to see where a new observation starts and ends because it is defined by a row in your file. However, we also need to tell our software where a new column starts, i.e. where a cell begins and ends. Consider the following example. We have a file that holds our data which looks like this:

```
idagegender
124male
256female
333male
```

The first row we probably can still decipher as `id`, `age`, `gender`. However, the next row makes it difficult to understand which value represents the `id` of a participant and which value reflects the `age` of that participant. Like us, computer software would find it hard too to decide on this ambiguous content. Thus, we need to use delimiters to make it very clear which value belongs to which column. For example, in a `.csv` file, the data would be separated by a `,`.

```
id,age,gender
1,24,male
2,56,female
3,33,male
```

Considering our example from above, we could also use `|` as a delimiter, but this is very unusual.

```
id|age|gender
1|24|male
2|56|female
3|33|male
```

There is a lot more to `readr` than could be covered in this book. If you want to know more about this *R* package, I highly recommend looking at the readr webpage[1].

7.2 Inspecting your data

For the rest of this chapter, we will use the `wvs` dataset from the `r4np` package. However, we do not know much about this dataset, and therefore we cannot

[1] https://readr.tidyverse.org

ask any research questions worth investigating. Therefore, we need to look at what it contains. The first method of inspecting a dataset is to type the name of the object, i.e. `wvs`.

```
# Ensure you loaded the 'r4np' package first
library(r4np)

# Show the data in the console
wvs
```

```
# A tibble: 69,578 x 7
   `Participant ID` `Country name` Gender   Age relationship_status
              <dbl> <chr>           <dbl> <dbl> <chr>
 1         20070001 Andorra             1    60 married
 2         20070002 Andorra             0    47 living together as married
 3         20070003 Andorra             0    48 separated
 4         20070004 Andorra             1    62 living together as married
 5         20070005 Andorra             0    49 living together as married
 6         20070006 Andorra             1    51 married
 7         20070007 Andorra             1    33 married
 8         20070008 Andorra             0    55 widowed
 9         20070009 Andorra             1    40 single
10         20070010 Andorra             1    38 living together as married
# i 69,568 more rows
# i 2 more variables: Freedom.of.Choice <dbl>, `Satisfaction-with-
life` <dbl>
```

The result is a series of rows and columns. The first information we receive is: `A tibble: 69,578 x 7`. This indicates that our dataset has 69,578 observations (i.e. rows) and 9 columns (i.e. variables). This rectangular format is the one we encounter most frequently in Social Sciences (and probably beyond). If you ever worked in Microsoft Excel, this format will look familiar. However, rectangular data is not necessarily `tidy` data. Wickham (2014) (p. 4) defines tidy data as follows:

---

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Even though it might be nice to look at a tibble in the console, it is not particularly useful. Depending on your monitor size, you might only see a small number of columns, and therefore we do not get to see a complete list of all variables. In short, we hardly ever will find much use in inspecting data this way. Luckily other functions can help us.

If you want to see each variable covered in the dataset and their data types, you can use the function `glimpse()` from the `dplyr` package which is loaded as part of the `tidyverse` package.

```
glimpse(wvs)
```

```
Rows: 69,578
Columns: 7
$ `Participant ID`       <dbl> 20070001, 20070002, 20070003, 20070004, 20070~
$ `Country name`         <chr> "Andorra", "Andorra", "Andorra", "Andorra", "~
$ Gender                 <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, ~
$ Age                    <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 3~
$ relationship_status     <chr> "married", "living together as married", "sep~
$ Freedom.of.Choice      <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7~
$ `Satisfaction-with-life` <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7,~
```

The output of glimpse shows us the name of each column/variable after the `$`, for example, `` `Participant ID` ``. The `$` is used to lookup certain variables in our dataset. For example, if we want to inspect the column `relationship_status` only, we could write the following:

```
wvs$relationship_status
```

```
    [1] "married"                   "living together as married"
    [3] "separated"                 "living together as married"
    [5] "living together as married" "married"
    [7] "married"                   "widowed"
....
```

When using `glimpse()` we find the recognised data type for each column, i.e. each variable, in `<...>`, for example `<chr>`. We will return to data types in Chapter @ref(change-data-types). Lastly, we get examples of the values included in each variable. This output is much more helpful.

I use `glimpse()` very frequently for different purposes, for example:

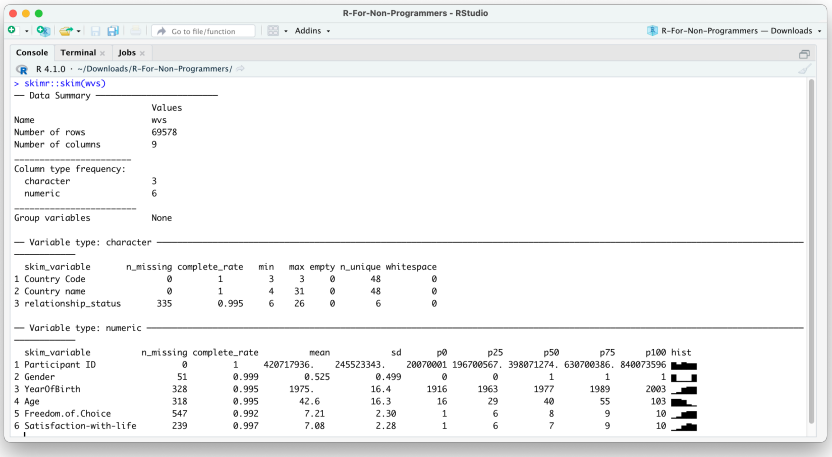
- to understand what variables are included in a dataset,
- to check the correctness of data types,
- to inspect variable names for typos or unconventional names,

- to look up variable names.

There is one more way to inspect your data and receive more information about it by using a specialised *R* package. The `skimr` package is excellent in 'skimming' your dataset. It provides not only information about variable names and data types but also provides some descriptive statistics. If you installed the `r4np` package and called the function `install_r4np()`, you will have `skimr` installed already.

```
skimr::skim(wvs)
```

The output in the *Console* should look like this:



As you can tell, there is a lot more information in this output. Many descriptive statistics that could be useful are already displayed. `skim()` provides a summary of the dataset and then automatically sorts the variables by data type. Depending on the data type, you also receive different descriptive statistics. As a bonus, the function also provides a histogram for numeric variables. However, there is one main problem: Some of the numeric variables are not numeric: `Participant ID` and `Gender`. Thus, we will have to correct the data types in a moment.

Inspecting your data in this way can be helpful to get a better understanding of what your data includes and spot problems with it. In addition, if you receive data from someone else, these methods are an excellent way to familiarise yourself with the dataset relatively quickly. Since I prepared this particular dataset for this book, I also made sure to provide documentation for it. You can access it by using `?wvs` in the *Console*. This will open the documentation

in the *Help* pane. Such documentation is available for every dataset we use in this book.

7.3 Cleaning your column names: Call the `janitor`

If you have eagle eyes, you might have noticed that most of the variable names in `wvs` are not consistent or easy to read/use.

```
# Whitespace and inconsistent capitalisation
Participant ID
Country name
Gender
Age

# Difficult to read
Freedom.of.Choice
Satisfaction-with-life
```

From Chapter @ref(coding-etiquette), you will remember that being consistent in writing your code and naming your objects is essential. The same applies, of course, to variable names. *R* will not break using the existing names, but it will save you a lot of frustration if we take a minute to clean the names and make them more consistent. Since we will use column names in almost every line of code we produce, it is best to standardise them and make them easier to read and write.

You are probably thinking: "This is easy. I just open the dataset in Excel and change all the column names." Indeed, it would be a viable and easy option, but it is not very efficient, especially with larger datasets with many more variables. Instead, we can make use of the `janitor` package. By definition, `janitor` is a package that helps to clean up whatever needs cleaning. In our case, we want to tidy our column names. We can use the function `clean_names()` to achieve this. We store the result in a new object called `wvs` to keep those changes. The object will also show up in our *Environment* pane.

```
wvs_clean <- janitor::clean_names(wvs)

glimpse(wvs_clean)
```

```
Rows: 69,578
Columns: 7
$ participant_id      <dbl> 20070001, 20070002, 20070003, 20070004, 2007000~
```

```
$ country_name        <chr> "Andorra", "Andorra", "Andorra", "Andorra", "An~
$ gender              <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0,~
$ age                <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39,~
$ relationship_status   <chr> "married", "living together as married", "separ~
$ freedom_of_choice    <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, ~
$ satisfaction_with_life <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 1~
```

Now that `janitor` has done its magic, we suddenly have easy to read variable names that are consistent with the *'Tidyverse style guide'* (Wickham 2021).

If, for whatever reason, the variable names are still not looking the way you want, you can use the function `rename()` from the `dplyr` package to manually assign new variable names.

```
wvs_clean <-
  wvs_clean %>%
  rename(satisfaction = satisfaction_with_life,
         country = country_name)

glimpse(wvs_clean)
```

```
Rows: 69,578
Columns: 7
$ participant_id    <dbl> 20070001, 20070002, 20070003, 20070004, 20070005, ~
$ country         <chr> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
$ gender           <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
$ age             <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39, 44~
$ relationship_status <chr> "married", "living together as married", "separate~
$ freedom_of_choice  <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
$ satisfaction      <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

You are probably wondering what `%>%` stands for. This symbol is called a *'piping operator'* , and it allows us to chain multiple functions together by considering the output of the previous function. So, do not confuse `<-` with `%>%`. Each operator serves a different purpose. The `%>%` has become synonymous with the `tidyverse` approach to *R* programming and is the chosen approach for this book. Many functions from the `tidyverse` are designed to be chained together.

If we wanted to spell out what we just did, we could say:

1. `wvs <-`: We assigned whatever happened to the right of the assignment operator to the object `wvs`.
2. `wvs %>%`: We defined the dataset we want to use with the functions defined after the `%>%`.
3. `rename(satisfaction = satisfcation_with_life)`: We define a new

name `satisfaction` for the column `satisfaction_with_life`. Notice that the order is `new_name = old_name`. Here we also use `=`. A rare occasion where it makes sense to do so.

Just for clarification, the following two lines of code accomplish the same task. The only difference is that with `%>%` we could chain another function right after it. So, you could say, it is a matter of taste which approach you prefer. However, in later chapters, it will become apparent why using `%>%` is very advantageous.

```
# Renaming a column using '%>%'
wvs_clean %>% rename(satisfaction_new = satisfaction)

# Renaming a column without '%>%'
rename(wvs_clean, satisfaction_new = satisfaction)
```

Since you will be using the pipe operator very frequently, it is a good idea to remember the keyboard shortcut for it: `Ctrl+Shift+M` for PC and `Cmd+Shift+M` for Mac.

7.4 Data types: What are they and how can you change them

When we inspected our data, I mentioned that some variables do not have the correct data type. You might be familiar with different data types by classifying them as:

- *Nominal data*, which is categorical data of no particular order,
- *Ordinal data*, which is categorical data with a defined order, and
- *Quantitative data*, which is data that usually is represented by numeric values.

In *R* we have a slightly different distinction:

- `character` / `<chr>`: Textual data, for example the text of a tweet.
- `factor` / `<fct>`: Categorical data with a finite number of categories with no particular order.
- `ordered` / `<ord>`: Categorical data with a finite number of categories with a particular order.
- `double` / `<dbl>`: Numerical data with decimal places.

- `integer` / `<int>`: Numerical data with whole numbers only (i.e. no decimals).
- `logical` / `<lgl>`: Logical data, which only consists of values `TRUE` and `FALSE`.
- `date` / `date`: Data which consists of dates, e.g. `2021-08-05`.
- `date-time` / `dttm`: Data which consists of dates and times, e.g. `2021-08-05 16:29:25 BST`.

For a complete list of data types, I recommend looking at 'Column Data Types'[2] (Müller and Wickham 2021).

R has a fine-grained categorisation of data types. The most important distinction, though, lies between `<chr>`, `<fct>`/`<ord>` and `<dbl>` for most datasets in the Social Sciences. Still, it is good to know what the abbreviations in your `tibble` mean and how they might affect your analysis.

Now that we have a solid understanding of different data types, we can look at our dataset and see whether `readr` classified our variables correctly.

```
glimpse(wvs_clean)
```

```
Rows: 69,578
Columns: 7
$ participant_id    <dbl> 20070001, 20070002, 20070003, 20070004, 20070005, ~
$ country         <chr> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
$ gender           <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
$ age             <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39, 44~
$ relationship_status <chr> "married", "living together as married", "separate~
$ freedom_of_choice  <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
$ satisfaction      <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

`readr` did a great job in identifying all the numeric variables. However, by default, `readr` imports all variables that include text as `<chr>`. It appears as if this is not entirely correct for our dataset. The variables `country`, `gender` and `relationship_status` specify a finite number of categories. Therefore they should be classified as a `factor`. The variable `participant_id` is represented by numbers, but its meaning is also rather categorical. We would not use the ID numbers of participants to perform additions or multiplications. This would make no sense. Therefore, it might be wise to turn them into a `factor`, even though we likely will not use it in our analysis and would make no difference. However, I am a stickler for those kinds of things.

To perform the conversion, we need to use two new functions from `dplyr`:

- `mutate()`: Changes, i.e. 'mutates', a variable.
- `as_factor()`: Converts data from one type into a `factor`.

[2] https://tibble.tidyverse.org/articles/types.html

If we want to convert all variables in one go, we can put them into the same function, separated by a ,.

```
wvs_clean <-
  wvs_clean %>%
  mutate(country = as_factor(country),
         gender = as_factor(gender),
         relationship_status = as_factor(relationship_status),
         participant_id = as_factor(participant_id)
         )

glimpse(wvs_clean)
```

```
Rows: 69,578
Columns: 7
$ participant_id    <fct> 20070001, 20070002, 20070003, 20070004, 20070005, ~
$ country         <fct> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
$ gender           <fct> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
$ age             <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39, 44~
$ relationship_status <fct> married, living together as married, separated, li~
$ freedom_of_choice  <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
$ satisfaction      <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

The output in the console shows that we successfully performed the transformation and our data types are as we intended them to be. Mission accomplished.

If you need to convert all `<chr>` columns you can use `mutate_if(is.character, as_factor)` instead. This function will look at each column and if it is a `character` type variable, it will convert it into a `factor`. However, use this function only if you are certain that all `character` columns need converting. If this is the case, it can be a huge time-saver. Here is the corresponding code snippet:

```
wvs_clean %>%
  mutate_if(is.character, as_factor)
```

Lastly, there might be occasions where you want to convert data into other formats than from `<chr>` to `<fct>`. Similar to `as_factor()` there are more functions available to transform your data into the correct format of your choice:

- `as.integer()`: Converts values into integers, i.e. without decimal places. Be aware that this function does not round values as you might expect. Instead, it removes any decimals as if they never existed.

- `as.numeric()`: Converts values into numbers with decimals if they are present.
- `as.character()`: Converts values to string values, even if they are numbers. If you use `readr` to import your data, you will not often use this function because `readr` imports unknown data types as `<chr>` by default.

All these function can be used in the same as we did before. Here are three examples to illustrated their use.

```
# Conversion to character values
converted_data <-
  wvs_clean %>%
  mutate(participant_id = as.character(participant_id),
         age = as.character(age))

glimpse(converted_data)
```

```
Rows: 69,578
Columns: 7
$ participant_id    <chr> "20070001", "20070002", "20070003", "20070004", "2~
$ country         <fct> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
$ gender           <fct> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
$ age             <chr> "60", "47", "48", "62", "49", "51", "33", "55", "4~
$ relationship_status <fct> married, living together as married, separated, li~
$ freedom_of_choice  <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
$ satisfaction      <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

```
# Conversion to numeric values
converted_data %>%
  mutate(age = as.numeric(age))
```

```
# A tibble: 69,578 x 7
   participant_id country gender   age relationship_status    freedom_of_choice
   <chr>          <fct>   <fct>  <dbl> <fct>                           <dbl>
 1 20070001       Andorra 1         60 married                            10
 2 20070002       Andorra 0         47 living together as mar~             9
 3 20070003       Andorra 0         48 separated                           9
 4 20070004       Andorra 1         62 living together as mar~             9
 5 20070005       Andorra 0         49 living together as mar~             8
 6 20070006       Andorra 1         51 married                            10
 7 20070007       Andorra 1         33 married                            10
 8 20070008       Andorra 0         55 widowed                             8
 9 20070009       Andorra 1         40 single                              8
10 20070010       Andorra 1         38 living together as mar~            10
```

```
# i 69,568 more rows
# i 1 more variable: satisfaction <dbl>
```

```
# Convert numeric values to integers
numeric_values <- tibble(values = c(1.3, 2.6, 3.8))

numeric_values %>%
  mutate(integers = as.integer(values))
```

```
# A tibble: 3 x 2
  values integers
   <dbl>    <int>
1    1.3        1
2    2.6        2
3    3.8        3
```

7.5 Handling factors

Factors are an essential way to classify observations in our data in different ways. In terms of data wrangling, there are usually at least two steps we take to prepare them for analysis:

- Recoding factors, and
- Reordering factor levels.

7.5.1 Recoding factors

Another common problem we have to tackle when working with data is their representation in the dataset. For example, `gender` could be measured as `male` and `female`[3] or as `0` and `1`. *R* does not mind which way you represent your data, but some other software does. Therefore, when we import data from somewhere else, the values of a variable might not look the way we want. The practicality of having your data represented accurately as what they are, becomes apparent when you intend to create tables and plots.

For example, we might be interested in knowing how many participants in `wvs_clean` were `male` and `female`. The function `count()` from `dplyr` does precisely that. We can sort the results, i.e. `n`, in descending order by including `sort = TRUE`. By default, `count()` arranges our variable `gender` in alphabetical order.

[3] A sophisticated research project would likely not rely on a dichotomous approach to gender, but appreciate the diversity in this regard.

```
wvs_clean %>% count(gender, sort = TRUE)
```

```
# A tibble: 3 x 2
  gender     n
  <fct>  <int>
1 1      36478
2 0      33049
3 <NA>      51
```

Now we know how many people were `male` and `female` and how many did not disclose their `gender`. Or do we? The issue here is that you would have to know what `0` and `1` stand for. Surely you would have a coding manual that gives you the answer, but it seems a bit of a complication. For `gender`, this might still be easy to remember, but can you recall the ID numbers for 48 countries?

It certainly would be easier to replace the `0`s and `1`s with their corresponding labels. This can be achieved with a simple function called `fct_recode()` from `forcats`. However, since we 'mutate' a variable into something else, we also have to use the `mutate()` function.

```
wvs_clean <-
  wvs_clean %>%
  mutate(gender = fct_recode(gender,
                             "male" = "0",
                             "female" = "1"))
```

If you have been following along very carefully, you might spot one oddity in this code: `"0"` and `"1"`. You likely recall that in Chapter @ref(r-basics-the-very-fundamentals), I mentioned that we use `""` for `character` values but not for numbers. So what happens if we run the code and remove `""`.

```
wvs_clean %>%
  mutate(gender = fct_recode(gender,
                             "male" = 0,
                             "female" = 1))
```

```
Error in `mutate()`:
i In argument: `gender = fct_recode(gender, male = 0, female = 1)`.
Caused by error in `fct_recode()`:
! Each element of `...` must be a named string.
i Problems with 2 arguments: male and female
```

The error message is easy to understand: `fct_recode()` only expects `strings` as input and not numbers. R recognises `0` and `1` as numbers, but `fct_recode()` only converts a `factor` value into another `factor` value and since we 'mutated'

gender from a `numeric` variable to a `factor` variable, all values of gender are no longer numbers. To refer to a factor level (i.e. one of the categories in our factor), we have to use `""`. In other words, data types matter and are often a source of problems with your code. Thus, always pay close attention to them. For now, it is important to remember that factor levels are always string values, i.e. text.

Now that we have change the factor levels to our liking, we can rerun our analysis and generate a frequency table for `gender`. This time we receive a much more readable output.

```
wvs_clean %>% count(gender)
```

```
# A tibble: 3 x 2
  gender     n
  <fct>  <int>
1 male   33049
2 female 36478
3 <NA>      51
```

Another benefit of going through the trouble of recoding your factors is the readability of your plots. For example, we could quickly generate a bar plot based on the above table and have appropriate labels instead of `0` and `1`.

```
wvs_clean %>%
  count(gender) %>%
  ggplot(aes(gender, n)) +
  geom_col()
```

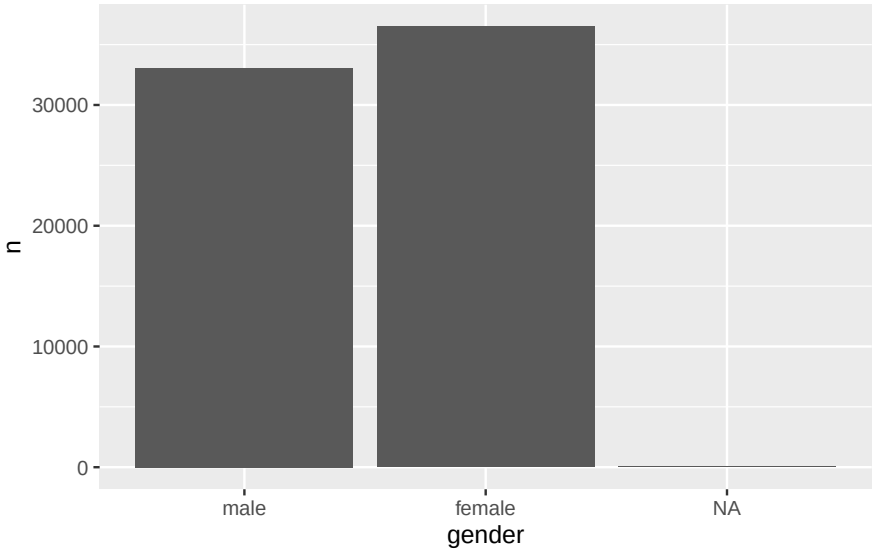


Figure 7.1: A bar plot of the factor levels `male` and `female`

Plots are an excellent way to explore your data and understand relationships between variables. We will learn more about this when we start to perform analytical steps on our data (see Chapter @ref(descriptive-statistics) and beyond).

Another use case for recoding factors could be for purely cosmetic reasons. For example, when looking through our dataset, we might notice that some country names are very long and do not look great in data visualisations or tables. Thus, we could consider shortening them which is a slightly more advanced process.

First, we need to find out which country names are particularly long. There are 48 countries in this dataset, so it could take some time to look through them all. Instead, we could use the function `filter()` from `dplyr` to pick only countries with a long name. However, this poses another problem: How can we tell the filter function to pick only country names with a certain length? Ideally, we would want a function that does the counting for us. As you probably anticipated, there is a package called `stringr`, which also belongs to the `tidyverse`, and has a function that counts the number of characters that represent a value in our dataset: `str_length()`. This function takes any `character` variable and returns the length of it. This also works with `factors` because this function can 'coerce' it into a `character`, i.e. it just ignores that it is a `factor` and looks at it as if it was a regular `character` variable. Good news for us, because now we can put the puzzle pieces together.

```
wvs_clean %>%
  filter(str_length(country) >= 15) %>%
  count(country)
```

```
# A tibble: 3 x 2
  country                             n
  <fct>                           <int>
1 Bolivia, Plurinational State of  2067
2 Iran, Islamic Republic of        1499
3 Korea, Republic of               1245
```

I use the value `15` arbitrarily after some trial and error. You can change the value and see which other countries would show up with a lower threshold. However, this number seems to do the trick and returns three countries that seem to have longer names. All we have to do is replace these categories with new ones the same way we recoded `gender`.

```
wvs_clean <-
  wvs_clean %>%
  mutate(country = fct_recode(country,
                              "Bolivia" = "Bolivia, Plurinational State of",
                              "Iran" = "Iran, Islamic Republic of",
                              "Korea" = "Korea, Republic of"))
```

Now we have country names that nicely fit into tables and plots we want to create later and do not have to worry about it later.

7.5.2 Reordering factor levels

Besides changing factor levels, it is also common that we want to reconsider the arrangement of these levels. To inspect the order of factor levels, we can use the function `fct_unique()` from the `forcats` package and provide a factor variable, e.g. `gender`, in the `wvs_clean` dataset.

```
fct_unique(wvs_clean$gender)
```

```
[1] male   female <NA>
Levels: male female
```

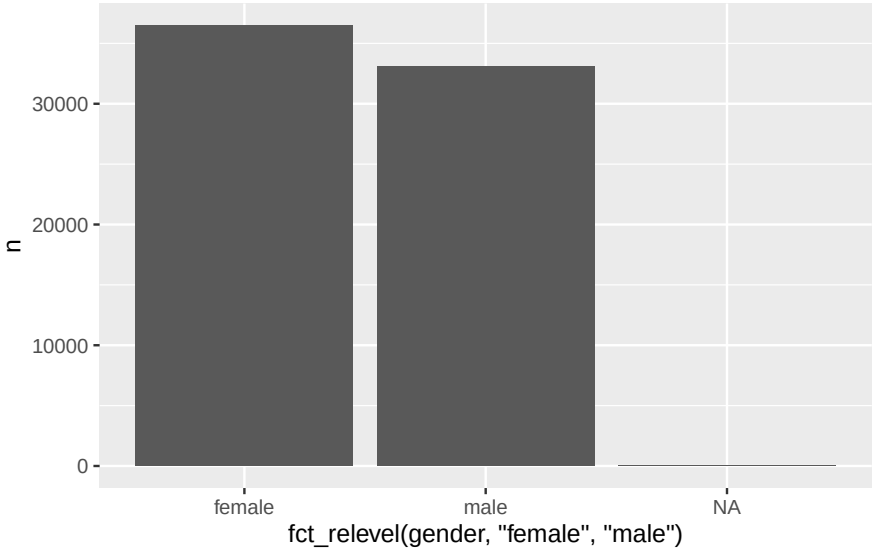
A reason for changing the order of factor levels could be to arrange how factor levels are displayed in plots. For example, Figure @ref(fig:gender-bar-plot) shows the bar of `male` participants first, but instead, we might want to show `female` participants first. The `forcats` package offers several options to rearrange factor levels based on different conditions. Depending on the

circumstances, one or the other function might be more efficient. Three of these functions[4] I tend to use fairly frequently:

- `fct_relevel()`: To reorder factor levels by hand.
- `fct_reorder()`: To reorder factor levels based on another variable of interest.
- `fct_rev()`: To reverse the order of factor levels.

Each function requires slightly different arguments. If we wanted to change the order of our factor levels, we could use all three of them to achieve the same result (in our case):

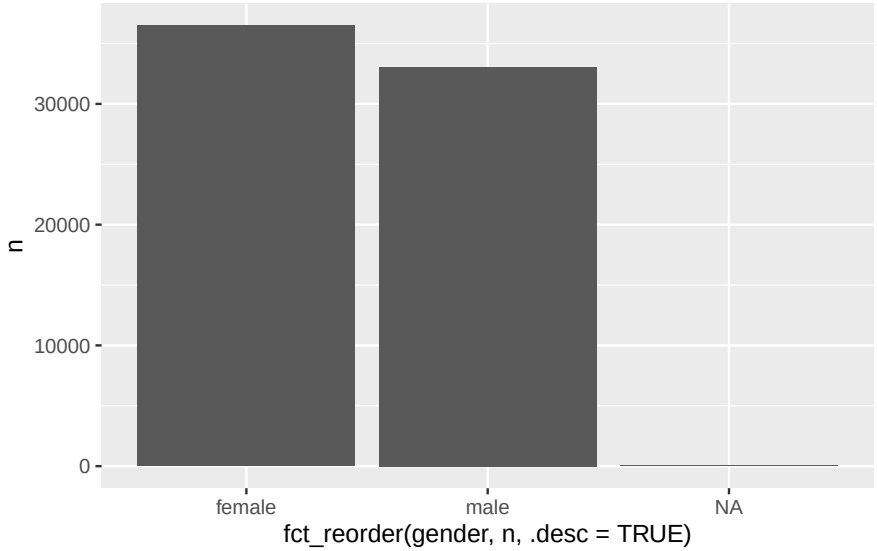
```
# Reorder by hand
wvs_clean %>%
  count(gender) %>%
  ggplot(aes(x = fct_relevel(gender, "female", "male"),
             y = n)) +
  geom_col()
```



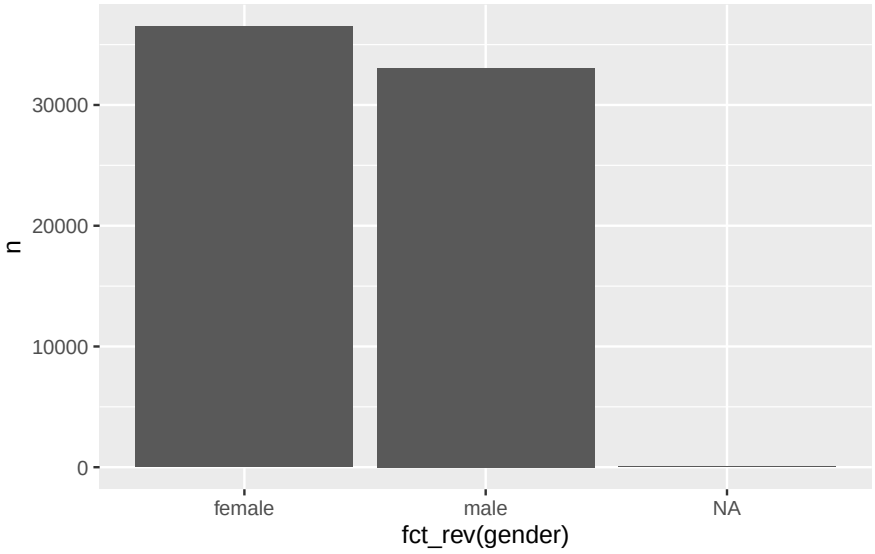

```
# Reorder by another variable, e.g. the frequency 'n'
wvs_clean %>%
  count(gender) %>%
  ggplot(aes(x = fct_reorder(gender, n, .desc = TRUE),
```

[4] https://forcats.tidyverse.org/reference/index.html

```
            y = n)) +
  geom_col()
```




```
# Reverse the order of factor levels
wvs_clean %>%
  count(gender) %>%
  ggplot(aes(x = fct_rev(gender),
             y = n)) +
  geom_col()
```



There are many ways to rearrange factors, and it very much depends on the data you have and what you wish to achieve. For most of my plots, I tend to revert to a basic R function, i.e. `reorder()`, which is much shorter to type and does essentially the same as `fct_reorder()`. There is no right or wrong in this regard, just personal taste.

7.6 Handling dates, times and durations

There are many reasons why we encounter dates, times and durations in our research. The most apparent reason for handling dates and times is longitudinal studies that focus on change over time. Therefore, at least one of our variables would have to carry a timestamp. When it comes to data cleaning and wrangling, there are usually two tasks we need to perform:

- Convert data into a suitable format for further analysis, and
- Perform computations with dates and times, e.g. computing the duration between two time points.

The following sections will provide some guidance on solving commonly occurring issues and how to solve them.

7.6.1 Converting dates and times for analysis

No matter which data collection tool you use, you likely encounter differences in how various software report dates and times back to you. Not only is there a great variety in formats in which dates and times are reported, but there are also other issues that one has to take care of, e.g. time zone differences. Here are some examples of how dates could be reported in your data:

```
26 November 2022
26 Nov 2022
November, 26 2022
Nov, 26 2022
26/11/2022
11/26/2022
2022-11-26
20221126
```

It would be nice if there was a standardised method of recording dates and times and, in fact, there is one: ISO 8601[5]. But unfortunately, not every software, let alone scholar, necessarily adheres to this standard. Luckily there is a dedicated *R* package that solely deals with these issues called `lubridate`, which is also part of the `tidyverse`. It helps reformat data into a coherent format and takes care of other essential aspects, such as '*time zones, leap days, daylight savings times, and other time related quirks*' (Grolemund and Wickham 2011). Here, we will primarily focus on the most common issues, but I highly recommend looking at the 'lubridate' website[6] to see what else the package has to offer.

Let's consider the dates (i.e. the data type `date`) shown above and try to convert them all into the same format. To achieve this we can use three different functions, i.e. `dmy()`, `mdy()` and `ymd()`. It is probably very obvious what these letters stand for:

- `d` for day,
- `m` for month, and
- `y` for year.

All we have to do is identify the order in which the dates are presented in your dataset, and the `lubridate` package will do its magic to standardise the format.

```
library(lubridate)
```

[5] https://www.iso.org/iso-8601-date-and-time-format.html
[6] https://lubridate.tidyverse.org/

```
# All of the below will yield the same output
dmy("26 November 2022")
```

```
[1] "2022-11-26"
```

```
dmy("26 Nov 2022")
```

```
[1] "2022-11-26"
```

```
mdy("November, 26 2022")
```

```
[1] "2022-11-26"
```

```
mdy("Nov, 26 2022")
```

```
[1] "2022-11-26"
```

```
dmy("26/11/2022")
```

```
[1] "2022-11-26"
```

```
mdy("11/26/2022")
```

```
[1] "2022-11-26"
```

```
ymd("2022-11-26")
```

```
[1] "2022-11-26"
```

```
ymd("20221126")
```

```
[1] "2022-11-26"
```

Besides dates, we sometimes also have to deal with times. For example, when did someone start a survey and finish it. If you frequently work with software like Qualtrics, SurveyMonkey, etc., you are often provided with such information. From a diagnostic perspective, understanding how long it took someone to complete a questionnaire can provide valuable insights into survey optimisation and design or show whether certain participants properly engaged with your data collection tool. In *R*, we can use either `time` or `dttm` (date-time) as a data type to capture hours, minutes and seconds. The `lubridate` package handles `time` similarly to `date` with easily recognisable abbreviations:

- `h` for hour,
- `m` for minutes, and
- `s` for seconds.

Here is a simple example that demonstrates how conveniently `lubridate` manages to handle even the most outrages ways of specifying times:

```
dinner_time <- tibble(time = c("17:34:21",
                               "17:::34:21",
                               "17 34 21",
                               "17 : 34 : 21")
                      )

dinner_time %>% mutate(time_new = hms(time))
```

```
# A tibble: 4 x 2
  time         time_new
  <chr>        <Period>
1 17:34:21     17H 34M 21S
2 17:::34:21   17H 34M 21S
3 17 34 21     17H 34M 21S
4 17 : 34 : 21 17H 34M 21S
```

Lastly, `dttm` data types combine both: `date` and `time`. To illustrate how we can work with this format we can look at some actual data from a questionnaire. For example, the `` `quest_time` `` dataset has information about questionnaire completion dates and times..

```
quest_time
```

```
# A tibble: 84 x 3
      id start            end
   <dbl> <chr>            <chr>
 1     1 12/05/2016 20:46 12/05/2016 20:49
 2     2 19/05/2016 22:08 19/05/2016 22:25
 3     3 20/05/2016 07:05 20/05/2016 07:14
 4     4 13/05/2016 08:59 13/05/2016 09:05
 5     5 13/05/2016 11:25 13/05/2016 11:29
 6     6 13/05/2016 11:31 13/05/2016 11:37
 7     7 20/05/2016 11:29 20/05/2016 11:45
 8     8 20/05/2016 14:07 20/05/2016 14:21
 9     9 20/05/2016 14:55 20/05/2016 15:01
10    10 20/05/2016 16:12 20/05/2016 16:25
# i 74 more rows
```

While it all looks neat and tidy, keeping dates and times as a `chr` is not ideal, especially if we want to use it to conduct further analysis. Therefore, it would be important to transform these variables into the appropriate `dttm` format. We can use the function `dmy_hm()` similar to previous functions we used.

```
quest_time_clean <-
  quest_time %>%
  mutate(start = dmy_hm(start),
         end = dmy_hm(end)
         )

quest_time_clean
```

```
# A tibble: 84 x 3
      id start               end
   <dbl> <dttm>              <dttm>
 1     1 2016-05-12 20:46:00 2016-05-12 20:49:00
 2     2 2016-05-19 22:08:00 2016-05-19 22:25:00
 3     3 2016-05-20 07:05:00 2016-05-20 07:14:00
 4     4 2016-05-13 08:59:00 2016-05-13 09:05:00
 5     5 2016-05-13 11:25:00 2016-05-13 11:29:00
 6     6 2016-05-13 11:31:00 2016-05-13 11:37:00
 7     7 2016-05-20 11:29:00 2016-05-20 11:45:00
 8     8 2016-05-20 14:07:00 2016-05-20 14:21:00
 9     9 2016-05-20 14:55:00 2016-05-20 15:01:00
10    10 2016-05-20 16:12:00 2016-05-20 16:25:00
# i 74 more rows
```

While the difference between this format and the previous one is relatively marginal and unimpressive, the main benefit of this conversation becomes apparent when we perform operations on these variables, e.g. computing the time it took participants to complete the questionnaire.

7.6.2 Computing durations with dates, times and date-time variables

Once we have 'corrected' our variables to show as dates, times and date-times, we can perform further analysis and ask important questions about durations, periods and intervals of observations.

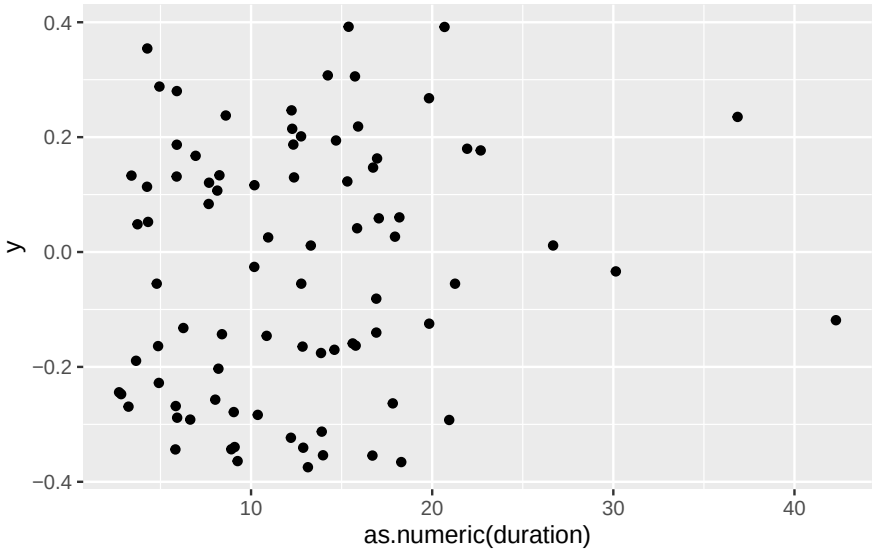
Let's consider the `quest_time` dataset which we just prepared. As mentioned earlier, it is often essential to understand how long it took participants to complete a given questionnaire. Therefore we would want to compute the duration based on the `start` and `end` time.

```
df_duration <-
  quest_time_clean %>%
  mutate(duration = end - start)

df_duration
```

```
# A tibble: 84 x 4
      id start               end                 duration
   <dbl> <dttm>              <dttm>              <drtn>
 1     1 2016-05-12 20:46:00 2016-05-12 20:49:00  3 mins
 2     2 2016-05-19 22:08:00 2016-05-19 22:25:00 17 mins
 3     3 2016-05-20 07:05:00 2016-05-20 07:14:00  9 mins
 4     4 2016-05-13 08:59:00 2016-05-13 09:05:00  6 mins
 5     5 2016-05-13 11:25:00 2016-05-13 11:29:00  4 mins
 6     6 2016-05-13 11:31:00 2016-05-13 11:37:00  6 mins
 7     7 2016-05-20 11:29:00 2016-05-20 11:45:00 16 mins
 8     8 2016-05-20 14:07:00 2016-05-20 14:21:00 14 mins
 9     9 2016-05-20 14:55:00 2016-05-20 15:01:00  6 mins
10    10 2016-05-20 16:12:00 2016-05-20 16:25:00 13 mins
# i 74 more rows
```

From these results, it seems that some participants have either super-humanly fast reading skills or did not engage with the questionnaire. To get a better overview, we can plot `duration` to see the distribution better.

```
df_duration %>%
  ggplot(aes(x = as.numeric(duration),
             y = 0)) +
  geom_jitter()
```



For now, you should not worry about what this code means but rather consider the interpretation of the plot, i.e. that most participants (represented by a black dot) needed less than 20 minutes to complete the questionnaire. Depending on the length of your data collection tool, this could either be good or bad news. Usually, the shorter the questionnaire can be, the higher the response rate, unless you have incentives for participants to complete a longer survey.

in Chapter @ref(descriptive-statistics), we will cover data visualisations like this boxplot in greater detail. If you cannot wait any longer and want to create your own visualisations, you are welcome to skip ahead. However, several data visualisations are awaiting you in the next chapter as well.

7.7 Dealing with missing data

There is hardly any Social Sciences project where researchers do not have to deal with missing data. Participants are sometimes unwilling to complete a questionnaire or miss the second round of data collection entirely, e.g. in longitudinal studies. It is not the purpose of this chapter to delve into all aspects of analysing missing data but provide a solid starting point. There are mainly three steps involved in dealing with missing data:

1. Mapping missing data,

2. Identifying patterns of missing data, and
3. Replacing or removing missing data

7.7.1 Mapping missing data

Every study that intends to be rigorous will have to identify how much data is missing because it affects the interpretability of our available data. In *R*, this can be achieved in multiple ways, but using a specialised package like `naniar` does help us to do this very quickly and systematically. First, we have to load the `naniar` package, and then we use the function `vis_miss()` to visualise how much and where exactly data is missing.

```
library(naniar)

vis_miss(wvs_clean)
```

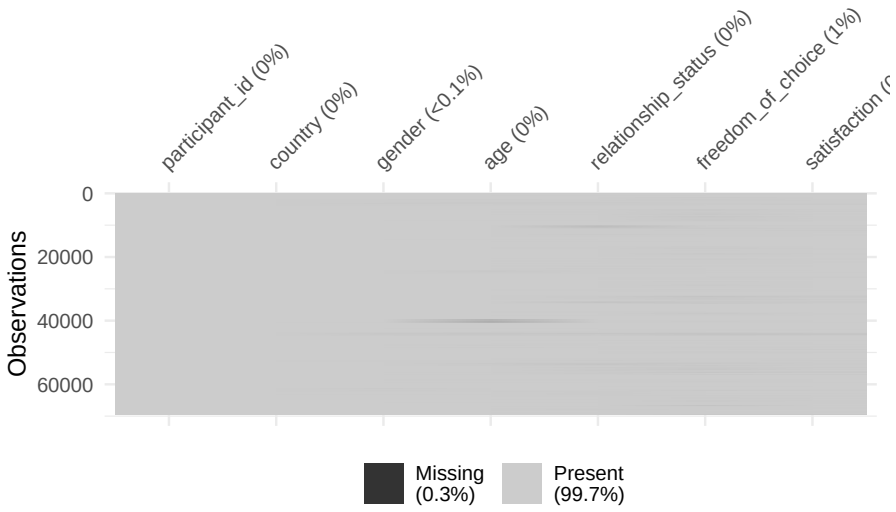


Figure 7.2: Mapping missing data with `naniar`

`naniar` plays along nicely with the `tidyverse` approach of programming. As such, it would also be possible to write `wvs %>% vis_miss()`.

As we can see, 99,7% of our dataset is complete, and we are only missing 0.3%. The dark lines (actually blocks) refer to missing data points. On the x-axis, we can see all our variables, and on the y-axis, we see our observations. This is the same layout as our rectangular dataset: Rows are observations,

and columns are variables. Overall, this dataset appears relatively complete (luckily). In addition, we can see the percentage of missing data per variable. `freedom_of_choice` is the variable with the most missing data, i.e. 0.79%. Still, the amount of missing data is not very large.

When working with larger datasets, it might also be helpful to rank variables by their degree of missing data to see where the most significant problems lie.

```
gg_miss_var(wvs_clean)
```

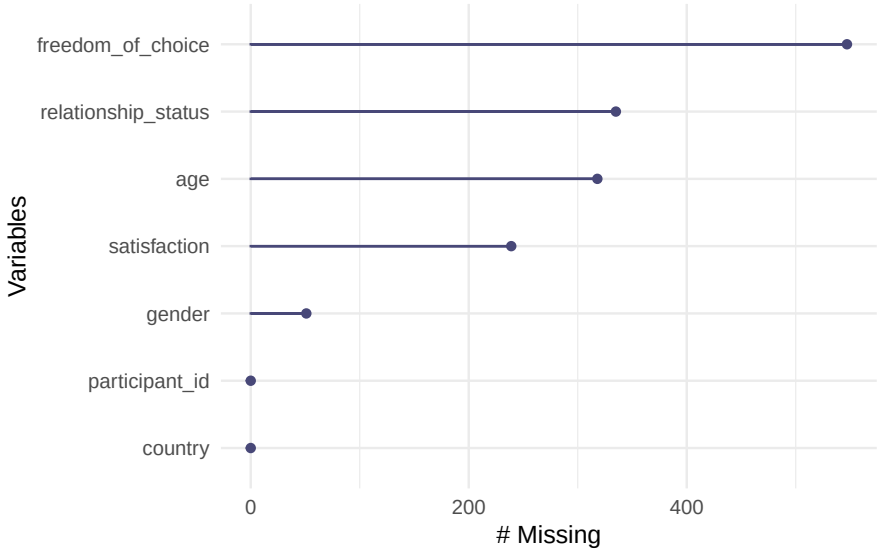


Figure 7.3: Missing data per variable

As we already know from the previous data visualisation, `freedom_of_choice` has the most missing data points, while `participant_id`, and `country_name` have no missing values. If you prefer to see the actual numbers instead, we can use a series of functions that start with `miss_` (for a complete list of all functions, see the reference page of naniar[7]). For example, to retrieve the numeric values which are reflected in the plot above, we can write the following:

```
# Summarise the missingness in each variable
miss_var_summary(wvs_clean)
```

```
# A tibble: 7 x 3
```

[7] https://naniar.njtierney.com/reference/index.html

```
  variable            n_miss pct_miss
  <chr>                <int>    <dbl>
1 freedom_of_choice      547   0.786
2 relationship_status    335   0.481
3 age                    318   0.457
4 satisfaction           239   0.343
5 gender                  51   0.0733
6 participant_id           0   0
7 country                  0   0
```

I tend to prefer data visualisations over numerical results for mapping missing data, especially in larger datasets with many variables. This also has the benefit that patterns of missing data can be more easily identified as well.

7.7.2 Identifying patterns of missing data

If you find that your data 'suffers' from missing data, it is essential to answer another question: Is data missing systematically? This is quite an important diagnostic step since systematically missing data would imply that if we remove these observations from our dataset, we likely produce wrong results. The following section elaborate on this aspect of missing data. In general, we can distinguish missing data based on how it is missing, i.e.

- missing completely at random (MCAR),
- missing at random (MAR), and
- missing not at random (MNAR). (Rubin 1976)

7.7.2.1 Missing completely at random (MCAR)

Missing completely at random (MCAR) means that neither observed nor missing data can systematically explain why data is missing. It is a pure coincidence how data is missing, and there is no underlying pattern.

The `naniar` package comes with the very popular Little's MCAR test (Little 1988), which provides insights into whether our data is missing completely at random. Thus, we can call the function `mcar_test()` and inspect the result.

```
wvs_clean %>%
  # Remove variables which do not reflect a response
  select(-participant_id) %>%
  mcar_test()
```

```
# A tibble: 1 x 4
  statistic    df p.value missing.patterns
      <dbl> <dbl>   <dbl>            <int>
1      411.    67       0               19
```

When you run such a test, you have to ensure that variables that are not part of the data collection process are removed. In our case, the `participant_id` is generated by the researcher and does not represent an actual response by the participants. As such, we need to remove it using `select()` before we can run the test. A `-` inverts the meaning of `select()`. While `select(participant_id)` would do what it says, i.e. include it as the only variable in the test, `select(-participant_id)` results in selecting everything but this variable in our test. You will find it is sometimes easier to remove a variable with `select()` rather than listing all the variables you want to keep.

Since the `p.value` of the test is so small that it got rounded down to `0`, i.e. $p < 0.0001$, we have to assume that our data is not missing completely at random. If we found that $p > 0.05$, we would have confirmation that data are missing completely at random. We will cover p-values and significance tests in more detail in Chapter @ref(significance).

7.7.2.2 Missing at random (MAR)

Missing at random (MAR) refers to a situation where the observed data can explain missing data, but not any unobserved data. Dong and Peng (2013) (p. 2) provide a good example when this is the case:

> *Let's suppose that students who scored low on the pre-test are more likely to drop out of the course, hence, their scores on the post-test are missing. If we assume that the probability of missing the post-test depends only on scores on the pre-test, then the missing mechanism on the post-test is MAR. In other words, for students who have the same pre-test score, the probability of (them) missing the post-test is random.*

Thus, the main difference between MCAR and MAR data lies in the fact that we can observe some patterns of missing data if data is MAR. These patterns are only based on data we have, i.e. observed data. We also assume that no unobserved variables can explain these or other patterns.

Accordingly, we first look into variables with no missing data and see whether they can explain our missing data in other variables. For example, we could investigate whether missing data in `freedom_of_choice` is attributed to specific countries.

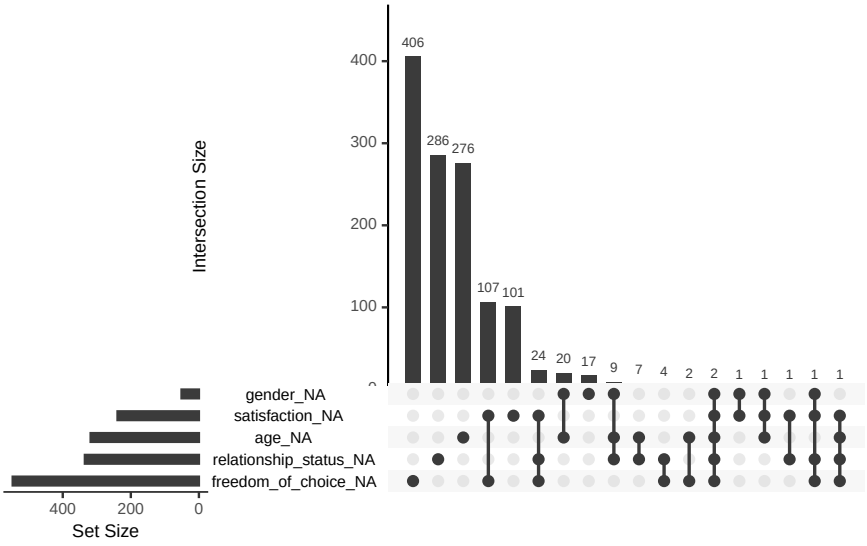
```
wvs_clean %>%
  group_by(country) %>%
  filter(is.na(freedom_of_choice)) %>%
  count(sort = TRUE)
```

```
# A tibble: 32 x 2
# Groups:   country [32]
   country         n
   <fct>       <int>
 1 Japan          47
 2 Brazil         44
 3 New Zealand    44
 4 Russia         43
 5 Bolivia        40
 6 Romania        29
 7 Kazakhstan     27
 8 Turkey         24
 9 Egypt          23
10 Serbia         20
# i 22 more rows
```

It seems four countries have exceptionally high numbers of missing data for `freedom_of_choice`: Japan, Brazil, New Zealand, Russia and Bolivia. Why this is the case lies beyond this dataset and is something only the researchers themselves could explain. Collecting data in different countries is particularly challenging, and one is quickly faced with different unfavourable conditions. Furthermore, the missing data is not completely random because we have some first evidence that the location of data collection might have affected its completeness.

Another way of understanding patterns of missing data can be achieved by looking at relationships between missing values, for example, the co-occurrence of missing values across different variables. This can be achieved by using *upset plots*. An *upset plot* consists of three parts: Set size, intersection size and a Venn diagram which defines the intersections.

```
gg_miss_upset(wvs_clean)
```



The most frequent combination of missing data in our dataset occurs when only `freedom_of_choice` is missing (the first column), but nothing else. Similar results can be found for `relationships_status` and `age`. The first combination of missing data is defined by two variables: `satisfaction` and `freedom_of_choice`. In total, 107 participants had `satisfaction` and `freedom_of_choice` missing but nothing else.

The 'set size' shown in the upset plot refers to the number of missing values for each variable in the diagram. This corresponds to what we have found when looking at Figure @ref(fig:missing-data-per-variable).

Our analysis also suggests that values are not completely randomly missing but that we have data to help explain why they are missing.

7.7.2.3 Missing not at random (MNAR)

Lastly, missing not at random (MNAR) implies that data is missing systematically and that other variables or reasons exist that explain why data is missing. Still, they are not fully known to us. In questionnaire-based research, an easily overlooked reason that can explain missing data is the 'page-drop-off' phenomenon. In such cases, participants stop completing a questionnaire once they advance to another page. Figure @ref(fig:mnar-example) shows this very clearly for a large scale project where an online questionnaire was used. After almost every page break in the questionnaire, some participants decided to discontinue. Finding these types of patterns is difficult when only working with numeric values. Thus, it is always advisable to visualise your data as well. Such missing data is linked to the design of the data collection tool.

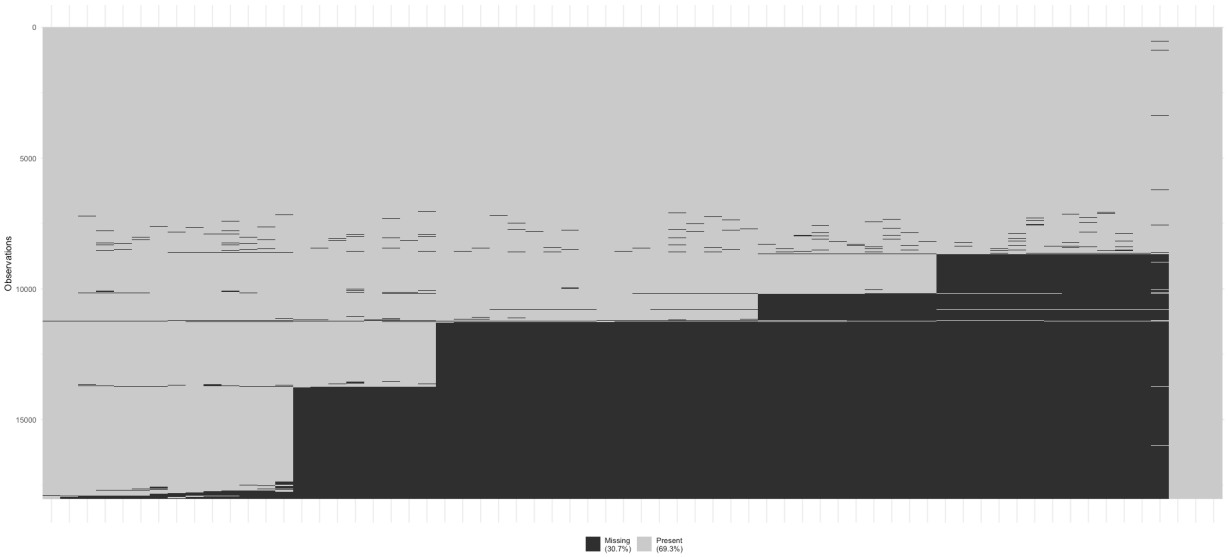


Figure 7.4: MNAR pattern in a dataset due to 'page-drop-offs'

Defining whether a dataset is MNAR or not is mainly achieved by ruling out MCAR and MAR assumptions. It is not possible to test whether missing data is MNAR, unless we have more information about the underlying population available (Ginkel et al. 2020).

We have sufficient evidence that our data is MAR as was shown above, because we managed to identify some relationships between unobserved and observed data. In practice, it is very rare to find datasets that are truly MCAR (Buuren 2018). Therefore we might consider 'imputation' as a possible strategy to solve our missing data problem. More about imputation in the next Chapter.

If you are looking for more inspiration of how you could visualise and identify patterns of missingness in your data, you might find the 'Gallery[8]' of the `naniar` website particularly useful.

7.7.3 Replacing or removing missing data

Once you determined which pattern of missing data applies to your dataset, it is time to evaluate how we want to deal with those missing values. Generally, you can either keep the missing values as they are, replace them or remove them entirely.

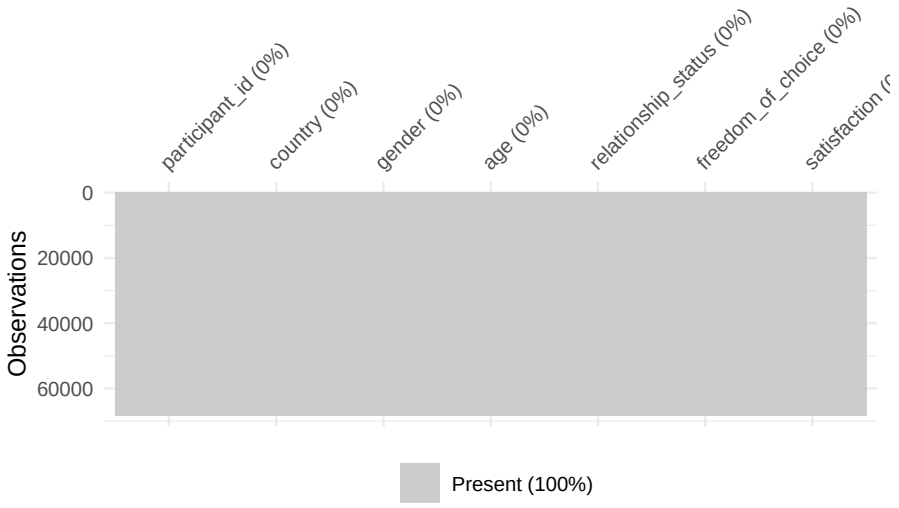
Jakobsen et al. (2017) provide a rule of thumb of 5% where researchers can consider missing data as negligible, i.e. we can ignore the missing values be-

[8] https://naniar.njtierney.com/articles/naniar-visualisation.html

cause they won't affect our analysis in a significant way. They also argue that if the proportion of missing data exceeds 40%, we should also only work with our observed data. However, with such a large amount of missing data, it is questionable whether we can rely on our analysis as much as we want to. If the missing data lies somewhere in-between this range, we need to consider the missing data pattern at hand.

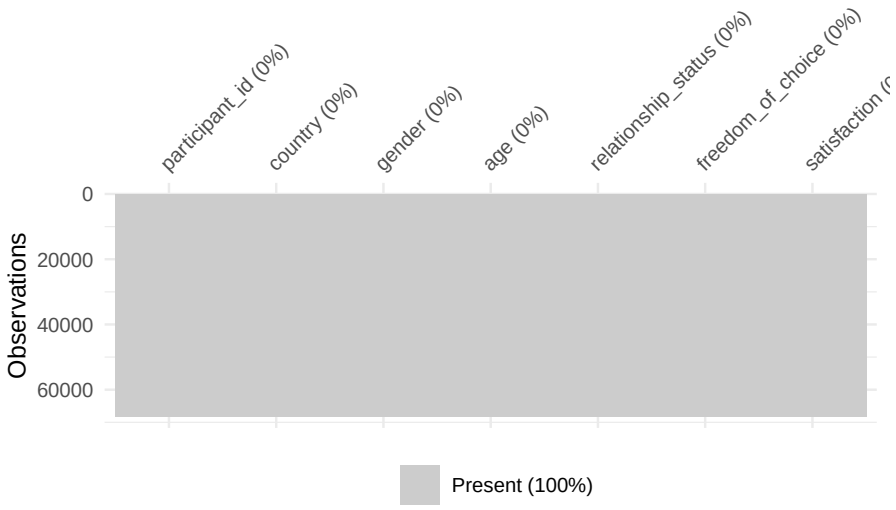
If data is MCAR, we could remove missing data. This process is called *'listwise deletion'*, i.e. you remove all data from a participant with missing data. As just mentioned, removing missing values is only suitable if you have relatively few missing values in your dataset (Schafer 1999), as is the case with the `wvs` dataset. There are additional problems with deleting observations listwise, many of which are summarised by Buuren (2018) in his introduction[9]. Usually, we try to avoid removing data as much as possible. If you wanted to perform listwise deletion, it can be done with a single function call: `na.omit()` from the built-in `stats` package. Alternatively, you could also use `drop_na()` from the `tidyr` package, which is automatically loaded when used `library(tidyverse)`. Here is an example of how we can apply these two functions to achieve the same effect.

```
# No more missing data in this plot
wvs_clean %>% na.omit() %>% vis_miss()
```



[9] https://stefvanbuuren.name/fimd/sec-simplesolutions.html

```
# Here is the alternative method with drop_na()
wvs_clean %>% drop_na() %>% vis_miss()
```




```
# How many observations are left after we removed all missing data?
wvs_clean %>% na.omit() %>% count()
```

```
# A tibble: 1 x 1
      n
  <int>
1 68312
```

If you wanted to remove the missing data without the plot, you could use `wvs_no_na <- na.omit(wvs)` or `wvs_no_na <- drop_na()`. However, I always recommend 'saving' the result in a new object to ensure we keep the original data available. This can be helpful when trying to compare how this decision affects my analysis, i.e. I can run the analysis with and without missing data removed. For the `wvs` dataset, using `na.omit()` does not seem to be the best option.

Based on our analysis of the `wvs` dataset (by no means complete!), we could assume that data is MAR. In such cases, it is possible to 'impute' the missing values, i.e. replacing the missing data with computed scores. This is possible because we can model the missing data based on variables we have. We cannot model missing data based on variables we have not measured in our study for obvious reasons.

You might be thinking: "Why would it make data better if we 'make up' numbers? Is this not cheating?" Imputation of missing values is a science in itself. There is plenty to read about the process of handling missing data, which would reach far beyond the scope of this book. However, the seminal work of Buuren (2018), Dong and Peng (2013) and Jakobsen et al. (2017) are excellent starting points. In short: Simply removing missing data can lead to biases in your data, e.g. if we removed missing data in our dataset, we would mainly exclude participants from Japan, Brazil, New Zealand, Russia and Bolivia (as we found out earlier). While imputation is not perfect, it can produce more reliable results than not using imputation at all, assuming our data meets the requirements for such imputation.

Even though our dataset has only a minimal amount of missing data (relative to the entire size), we can still use imputation. There are many different ways to approach this task, one of which is 'multiple imputation'. As highlighted by Ginkel et al. (2020), multiple imputation has not yet reached the same popularity as listwise deletion, despite its benefits. The main reason for this lies in the complexity of using this technique. There are many more approaches to imputation, and going through them all in detail would be impossible and distract from the book's main purpose. Still, I would like to share some interesting packages with you that use different imputation methods:

- mice[10] (Uses multivariate imputation via chained equations)
- Amelia[11] (Uses a bootstrapped-based algorithm)
- missForest[12] (Uses a random forest algorithm)
- Hmisc[13] (Uses additive regression, bootstrapping, and predictive mean matching)
- mi[14] (Uses posterior predictive distribution, predictive mean matching, mean-imputation, median-imputation, or conditional mean-imputation)

Besides multiple imputation, there is also the option for single imputation. The `simputation` package offers a great variety of imputation functions, one of which also fits our data quite well `impute_knn()`. This function makes use of a clustering technique called *'K-nearest neighbour'*. In this case, the function will look for observations closest to the one with missing data and take the value of that observation. In other words, it looks for participants who answered the questionnaire in a very similar way. The great convenience of this approach is that it can handle all kinds of data types simultaneously, which is not true for all imputation techniques.

[10] https://cran.r-project.org/web/packages/Amelia/index.html
[11] https://cran.r-project.org/web/packages/Amelia/index.html
[12] https://cran.r-project.org/web/packages/missForest/index.html
[13] https://cran.r-project.org/web/packages/Hmisc/index.html
[14] https://cran.r-project.org/web/packages/mi/index.html

If we apply this function to our dataset, we have to write the following:

```
wvs_nona <-
  wvs_clean %>%
  select(-participant_id) %>%

  # Transform our tibble into a data frame
  as.data.frame() %>%
  simputation::impute_knn(. ~ .)
```

```
# Be aware that our new dataframe has different datatypes
glimpse(wvs_nona)
```

```
Rows: 69,578
Columns: 6
$ country         <fct> Andorra, Andorra, Andorra, Andorra, Andorra, Andor~
$ gender          <fct> female, male, male, female, male, female, female, ~
$ age            <chr> "60", "47", "48", "62", "49", "51", "33", "55", "4~
$ relationship_status <fct> married, living together as married, separated, li~
$ freedom_of_choice  <chr> "10", "9", "9", "9", "8", "10", "10", "8", "8", "1~
$ satisfaction      <chr> "10", "9", "9", "8", "7", "10", "5", "8", "8", "10~
```

```
# Let's fix this
wvs_nona <-
  wvs_nona %>%
  mutate(age = as.numeric(age),
         freedom_of_choice = as.numeric(freedom_of_choice),
         satisfaction = as.numeric(satisfaction))
```

The function `impute_knn(. ~ .)` might look like a combination of text with an emoji `(. ~ .)`. This imputation function requires us to specify a model to impute the data. Since we want to impute all missing values in the dataset and use all variables available, we put `.` on both sides of the equation, separated by a ~. The `.` reflects that we do not specify a specific variable but instead tell the function to use all variables that are not mentioned. In our case, we did not mention any variables at all, and therefore it chooses all of them. For example, if we wanted to impute only `freedom_of_choice`, we would have to put `impute_knn(freedom_of_choice ~ .)`. We will elaborate more on how to specify models when we cover regression models (see Chapter @ref(regression)).

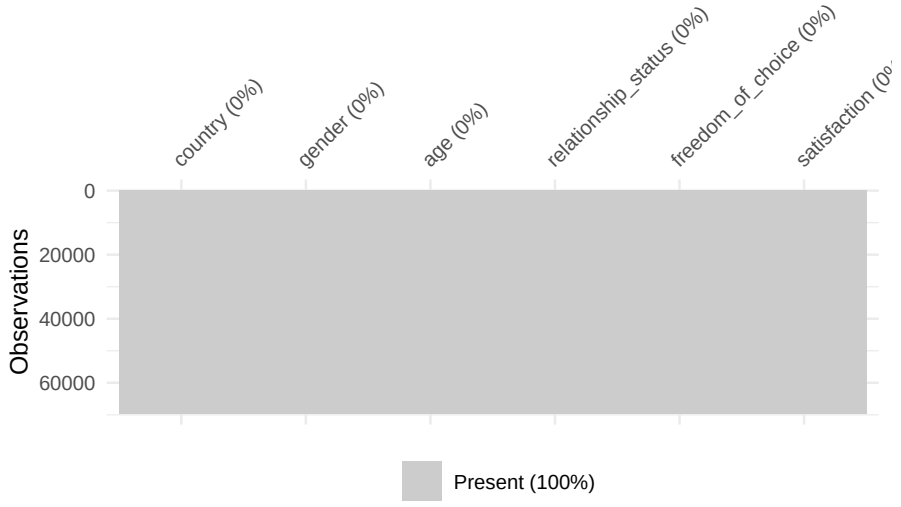
As you will have noticed, we also had to convert our `tibble` into a `data frame` using `as.data.frame()`. As mentioned in Chapter @ref(functions), some functions require a specific data type or format. The `simputation` package works with `dplyr`, but it prefers `data frames`. Based on my experiences, the wrong

data type or format is one of the most frequent causes of errors that novice *R* programmers report. So, keep an eye on it and read the documentation carefully. Also, once you inspect the new data frame, you will notice that some of our `double` variables have now turned into `character` values. Therefore, I strongly recommend checking your newly created data frames to avoid any surprises further down the line of your analysis. *R* requires us to be very precise in our work, which I think is rather an advantage than an annoyance[15].

Be aware that imputation of any kind can take a long time. For example, my MacBook Pro took about 4.22 seconds to complete `impute_knn()` with the `wvs` dataset. If we used multiple imputation, this would have taken considerably longer, i.e. several minutes and more.

We now should have a dataset that is free of any missing values. To ensure this is the case we can create the missing data matrix that we made at the beginning of this chapter (see Figure @ref(fig:mapping-missing-data)).

```
vis_miss(wvs_nona)
```



As the plot demonstrates, we managed to obtain a dataset which is now free of missing values and we can perform our next steps of data wrangling or analysis.

[15] Still, I understand that it often feels annoying.

7.7.4 Two more methods of replacing missing data you hardly ever need

There are two more options for replacing missing data[16], but these I would hardly ever recommend. Still, this section would not be complete without mentioning them, and they have their use-cases.

The first method involves the function `replace_na()`. It allows you to replace any missing data with a value of your choice. For example, let's consider the following dataset which contains feedback from students regarding one of their lectures.

```
# The dataset which contains the feedback
student_feedback <- tibble(name = c("Thomas", NA, "Helen", NA),
                           feedback = as.numeric(c(6, 5, NA, 4)),
                           overall = as_factor(c("excellent", "very good", NA, NA))
                           )

student_feedback
```

```
# A tibble: 4 x 3
  name   feedback overall
  <chr>     <dbl> <fct>
1 Thomas        6 excellent
2 <NA>          5 very good
3 Helen        NA <NA>
4 <NA>          4 <NA>
```

We have different types of data to deal with in `student_feedback`, and there are several missing data points, i.e. `NA` in each column. We can use the `replace_na()` function to decide how to replace the missing data. For `feedback`, we might want to insert a number, for example, the average of all available ratings (`(6 + 5+ 4) / 3 = 5`). However, we might wish to add some string values for `name` and `overall`. We have to keep in mind that `replace_na()` requires a `chr` variable as input, which is not the case for `overall`, because it is a `factor`. Consequently, we need to go back to Chapter @ref(change-data-types) and change the data type before replacing `NA`s. Here is how we can achieve all this in a few lines of code:

```
student_feedback %>%
  mutate(name = replace_na(name, "Unkown"),
         feedback = replace_na(feedback, 5),
         overall = replace_na(as.character(overall), "unrated"),
```

[16] Technically, there are many more options, but most of them work similar to the demonstrated techniques and are equally less needed in most scenarios.

```
         overall = as_factor(overall)
         )
```

```
# A tibble: 4 x 3
  name   feedback overall
  <chr>     <dbl> <fct>
1 Thomas        6 excellent
2 Unkown        5 very good
3 Helen         5 unrated
4 Unkown        4 unrated
```

We successfully replaced all the missing values with values of our choice. Only for `overall` we have to perform two steps. If you feel adventurous, you could also warp the `as_factor()` function around the `replace_na()` one. This would result in using only one line of code for this variable.

```
student_feedback %>%
  mutate(name = replace_na(name, "Unkown"),
         feedback = replace_na(feedback, 0),
         overall = as_factor(replace_na(as.character(overall), "unrated"))
         )
```

```
# A tibble: 4 x 3
  name   feedback overall
  <chr>     <dbl> <fct>
1 Thomas        6 excellent
2 Unkown        5 very good
3 Helen         0 unrated
4 Unkown        4 unrated
```

This approach to handling missing data can be beneficial when addressing these values explicitly, e.g. in a plot. While one could keep the label of `NA`, it sometimes makes sense to use a different label that is more meaningful to your intended audience. Still, I would abstain from replacing `NA`s with an arbitrary value unless there is a good reason for it.

The second method of replacing missing values is less arbitrary and requires the function `fill()`. It replaces missing values with the previous or next non-missing value. The official help page[17] for this function offers some excellent examples of when one would need it, i.e. when specific values are only recorded once for a set of data points. For example, we might want to know how many students are taking certain lectures, but the module ID is only recorded for the first student on each module.

[17] https://tidyr.tidyverse.org/reference/fill.html

```
# The dataset which contains student enrollment numbers
student_enrollment <- tibble(module = c("Management", NA, NA,
                                        NA, "Marketing", NA),
                             name = c("Alberto", "Frances", "Martin",
                                      "Alex", "Yuhui", "Himari")
                             )
student_enrollment
```

```
# A tibble: 6 x 2
  module     name
  <chr>      <chr>
1 Management Alberto
2 <NA>       Frances
3 <NA>       Martin
4 <NA>       Alex
5 Marketing  Yuhui
6 <NA>       Himari
```

Here it is unnecessary to guess which values we have to insert because we know that `Alberto`, `Frances`, `Martin`, and `Alex` are taking the `Management` lecture. At the same time, `Yuhui` and `Himari` opted for Marketing. The `fill()` function can help us complete the missing values accordingly. By default, it completes values top-down, i.e. it takes the last value that was not missing and copies it down until we reach a non-missing value.

```
student_enrollment %>% fill(module)
```

```
# A tibble: 6 x 2
  module     name
  <chr>      <chr>
1 Management Alberto
2 Management Frances
3 Management Martin
4 Management Alex
5 Marketing  Yuhui
6 Marketing  Himari
```

If you have to reverse the direction of how *R* should fill in the blank cells in your dataset, you can use `fill(.direction = "up")`. In our example, this would make not much sense, but for the sake of demonstration, here is what would happen:

```
student_enrollment %>% fill(module, .direction = "up")
```

```
# A tibble: 6 x 2
  module     name
  <chr>      <chr>
1 Management Alberto
2 Marketing  Frances
3 Marketing  Martin
4 Marketing  Alex
5 Marketing  Yuhui
6 <NA>       Himari
```

This method can be helpful if the system you use to collect or produce your data is only recording changes in values, i.e. it does not record all values for all observations in your dataset.

7.7.5 Main takeaways regarding dealing with missing data

Handling missing data is hardly ever a simple process. Do not feel discouraged if you get lost in the myriad of options. While there is some guidance on how and when to use specific strategies to deal with missing values in your dataset, the most crucial point to remember is: Be transparent about what you did. As long as you can explain why you did something and how you did it, everyone can follow your thought process and help improve your analysis. However, ignoring the fact that data is missing and not acknowledging it is more than just unwise.

7.8 Latent constructs and their reliability

Social Scientists commonly face the challenge that we want to measure something that cannot be measured directly. For example, *'happiness'* is a feeling that does not naturally occur as a number we can observe and measure. The opposite is true for *'temperature'* which is naturally measured in numbers. At the same time, *'happiness'* is much more complex of a variable than *'temperature'*, because *'happiness'* can unfold in various ways and be caused by different triggers (e.g. a joke, an unexpected present, tasty food, etc.). To account for this, we often work with *'latent variables'*. These are defined as variables that are not directly measured but are inferred from other variables. In practice, we often use multiple questions in a questionnaire to measure one latent variable, usually by computing the mean of those questions.

7.8.1 Computing latent variables

The `gep` dataset from the `r4np` package includes data about students' social integration experience (`si`) and communication skills development (`cs`). Data were obtained using the Global Education Profiler (GEP)[18]. I extracted three different questions (also called *'items'*) from the questionnaire, which measure `si`, and three items that measure `cs`. This dataset only consists of a randomly chosen set of responses, i.e. 300 out of over 12,000.

```
glimpse(gep)
```

```
Rows: 300
Columns: 12
$ age                       <dbl> 22, 26, 21, 23, 25, 27, 24, 23, 21, 24, ~
$ gender                   <chr> "Female", "Female", "Female", "Female", ~
$ level_of_study            <chr> "UG", "PGT", "UG", "UG", "PGT", "PGT", "~
$ si_socialise_with_people_exp  <dbl> 6, 3, 4, 3, 4, 2, 5, 1, 6, 6, 3, 3, 4, 5~
$ si_supportive_friends_exp    <dbl> 4, 4, 5, 4, 4, 2, 5, 1, 6, 6, 3, 3, 4, 6~
$ si_time_socialising_exp     <dbl> 5, 2, 4, 4, 3, 2, 6, 3, 6, 6, 3, 2, 4, 6~
$ cs_explain_ideas_imp       <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp    <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_explain_ideas_exp       <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
$ cs_find_clarification_exp    <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
$ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
```

For example, if we wanted to know how each student scored with regards to social integration (`si`), we have to

- compute the mean (`mean()`) of all related items (i.e. all variables starting with `si_`),
- for each row (`rowwise()`) because each row presents one participant.

The same is true for communication skills (`cs_imp` and `cs_exp`). We can compute all three variables in one go. For each variable, we compute `mean()` and use `c()` to list all the variables that we want to include in the mean:

```
# Compute the scores for the latent variable 'si' and 'cs'
gep <-
  gep %>%
  rowwise() %>%
  mutate(si = mean(c(si_socialise_with_people_exp,
                     si_supportive_friends_exp,
                     si_time_socialising_exp
```

[18] https://warwick.ac.uk/gep

```
                    )
                  ),
         cs_imp = mean(c(cs_explain_ideas_imp,
                         cs_find_clarification_imp,
                         cs_learn_different_styles_imp
                         )
                       ),
         cs_exp = mean(c(cs_explain_ideas_exp,
                         cs_find_clarification_exp,
                         cs_learn_different_styles_exp
                         )
                       )
         )

glimpse(gep)
```

```
Rows: 300
Columns: 15
Rowwise:
$ age                       <dbl> 22, 26, 21, 23, 25, 27, 24, 23, 21, 24, ~
$ gender                   <chr> "Female", "Female", "Female", "Female", ~
$ level_of_study            <chr> "UG", "PGT", "UG", "UG", "PGT", "PGT", "~
$ si_socialise_with_people_exp <dbl> 6, 3, 4, 3, 4, 2, 5, 1, 6, 6, 3, 3, 4, 5~
$ si_supportive_friends_exp    <dbl> 4, 4, 5, 4, 4, 2, 5, 1, 6, 6, 3, 3, 4, 6~
$ si_time_socialising_exp     <dbl> 5, 2, 4, 4, 3, 2, 6, 3, 6, 6, 3, 2, 4, 6~
$ cs_explain_ideas_imp       <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp    <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_explain_ideas_exp       <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
$ cs_find_clarification_exp    <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
$ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
$ si                       <dbl> 5.000000, 3.000000, 4.333333, 3.666667, ~
$ cs_imp                   <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
$ cs_exp                   <dbl> 6.000000, 4.666667, 3.666667, 5.000000, ~
```

Compared to dealing with missing data, this is a fairly straightforward task. However, there is a caveat. Before we can compute the mean across all these variables, we need to know and understand whether all these scores reliably contribute to one single latent variable. If not, we would be in trouble and make a significant mistake. This is commonly known as *internal consistency* and is a measure of reliability, i.e. how much we can rely on the fact that multiple questions in a questionnaire measure the same underlying, i.e. latent, construct.

7.8.2 Checking internal consistency of items measuring a latent variable

By far, the most common approach to assessing *internal consistency* of latent variables is Cronbach's α. This indicator looks at how strongly a set of items (i.e. questions in your questionnaire) are related to each other. For example, the stronger the relationship of all items starting with `si_` to each other, the more likely we achieve a higher Cronbach's α. The `psych` package has a suitable function to compute it for us.

Instead of listing all the items by hand, I use the function `starts_with()` to pick only the variables whose names start with `si_`. It certainly pays off to think about your variable names more thoroughly in advance to benefit from such shortcuts (see also Chapter @ref(colnames-cleaning)).

```
gep %>%
  select(starts_with("si_")) %>%
  psych::alpha()
```

```
Reliability analysis
Call: psych::alpha(x = .)

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean  sd median_r
      0.85      0.85     0.8      0.66 5.8 0.015  3.5 1.3     0.65

    95% confidence boundaries
         lower alpha upper
Feldt     0.82  0.85  0.88
Duhachek  0.82  0.85  0.88

 Reliability if an item is dropped:
                     raw_alpha std.alpha G6(smc) average_r S/N alpha se
si_socialise_with_people_exp    0.82    0.82   0.70    0.70 4.6   0.021
si_supportive_friends_exp       0.77    0.77   0.63    0.63 3.4   0.027
si_time_socialising_exp         0.79    0.79   0.65    0.65 3.7   0.025
                              var.r med.r
si_socialise_with_people_exp     NA  0.70
si_supportive_friends_exp        NA  0.63
si_time_socialising_exp          NA  0.65

 Item statistics
                                n raw.r std.r r.cor r.drop mean  sd
si_socialise_with_people_exp 300  0.86  0.86  0.75   0.69  3.7 1.4
si_supportive_friends_exp    300  0.90  0.89  0.81   0.75  3.5 1.6
si_time_socialising_exp      300  0.88  0.88  0.79   0.73  3.2 1.5
```

```
Non missing response frequency for each item
                                1    2    3    4    5    6 miss
si_socialise_with_people_exp 0.06 0.17 0.22 0.24 0.19 0.12    0
si_supportive_friends_exp    0.13 0.18 0.20 0.18 0.18 0.13    0
si_time_socialising_exp      0.15 0.21 0.22 0.20 0.12 0.10    0
```

The function `alpha()` returns a lot of information. The most important part, though, is shown at the very beginning:

```
 raw_alpha std.alpha G6(smc) average_r  S/N  ase mean   sd median_r
      0.85      0.85     0.8      0.66 5.76 0.01 3.46 1.33     0.65
```

In most publications, researchers would primarily report the `raw_alpha` value. This is fine, but it is not a bad idea to include at least `std.alpha` and `G6(smc)` as well.

In terms of interpretation, Cronbrach's α scores can range from `0` to `1`. The closer the score to `1`, the higher we would judge its reliability. Nunally (1967) originally provided the following classification for Cronbach's α:

- between `0.6` and `0.5` can be sufficient during the early stages of development,
- `0.8` or higher is sufficient for most basic research,
- `0.9` or higher is suitable for applied research, where the questionnaires are used to make critical decisions, e.g. clinical studies, university admission tests, etc., with a 'desired standard' of `0.95`.

However, a few years later, Nunally (1978) revisited his original categorisation and considered `0.7` or higher as a suitable benchmark in more exploratory-type research. This gave grounds for researchers to pick and choose the 'right' threshold for them (Henson 2001). Consequently, depending on your research field, the expected reliability score might lean more towards `0.7` or `0.8`. Still, the higher the score, the more reliable you latent variable is.

Our dataset shows that `si` scores a solid $\alpha = 0.85$, which is excellent. We should repeat this step for `cs_imp` and `cs_exp` as well. However, we have to adjust `select()`, because we only want variables included that start with `cs_` and end with the facet of the respective variable, i.e. `_imp` or `_exp`. Otherwise we compute the Cronbach's α for a combined latent construct.

```
# Select variables which start with 'cs_'
gep %>%
  select(starts_with("cs_")) %>%
  glimpse()
```

```
Rows: 300
Columns: 8
```

```
Rowwise:
$ cs_explain_ideas_imp       <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp    <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_explain_ideas_exp       <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
$ cs_find_clarification_exp    <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
$ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
$ cs_imp                 <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
$ cs_exp                 <dbl> 6.000000, 4.666667, 3.666667, 5.000000, ~
```

Therefore it is necessary to define two conditions to select the correct items in our dataset using `ends_with()` and combine these two with the logical operator `&` (see also Table @ref(tab:logical-operators-r)).

```
# Do the above but include only variables that also end with '_imp'
gep %>%
  select(starts_with("cs_") & ends_with("_imp")) %>%
  glimpse()
```

```
Rows: 300
Columns: 4
Rowwise:
$ cs_explain_ideas_imp       <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp    <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_imp                 <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
```

We also created a variable that starts with `cs_` and ends with `_imp`, i.e. the latent variable we computed, which also has to be removed.

```
# Remove 'cs_imp', because it is the computed latent variable
gep %>%
  select(starts_with("cs_") & ends_with("_imp"), -cs_imp) %>%
  glimpse()
```

```
Rows: 300
Columns: 3
Rowwise:
$ cs_explain_ideas_imp       <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp    <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
```

Thus, we end up computing the Cronbach's α for both variables in the following way:

```
gep %>%
  select(starts_with("cs_") & ends_with("_imp"), -cs_imp) %>%
  psych::alpha()
```

```
Reliability analysis
Call: psych::alpha(x = .)

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean   sd median_r
      0.86      0.87    0.82      0.68 6.5 0.014    5 0.98     0.65

    95% confidence boundaries
         lower alpha upper
Feldt     0.83  0.86  0.89
Duhachek  0.84  0.86  0.89

 Reliability if an item is dropped:
                             raw_alpha std.alpha G6(smc) average_r S/N
cs_explain_ideas_imp              0.78      0.78    0.65      0.65 3.6
cs_find_clarification_imp         0.76      0.76    0.62      0.62 3.2
cs_learn_different_styles_imp     0.88      0.88    0.79      0.79 7.4
                              alpha se var.r med.r
cs_explain_ideas_imp              0.025    NA  0.65
cs_find_clarification_imp         0.028    NA  0.62
cs_learn_different_styles_imp     0.014    NA  0.79

 Item statistics
                                n raw.r std.r r.cor r.drop mean  sd
cs_explain_ideas_imp          300  0.90  0.90  0.85   0.77  5.2 1.0
cs_find_clarification_imp     300  0.91  0.91  0.87   0.79  5.1 1.1
cs_learn_different_styles_imp 300  0.86  0.85  0.71   0.67  4.8 1.2

Non missing response frequency for each item
                                   1    2    3    4    5    6 miss
cs_explain_ideas_imp          0.01 0.01 0.06 0.14 0.27 0.51    0
cs_find_clarification_imp     0.01 0.02 0.05 0.14 0.30 0.47    0
cs_learn_different_styles_imp 0.01 0.03 0.09 0.19 0.31 0.36    0
```

```
gep %>%
  select(starts_with("cs_") & ends_with("_exp"), -cs_exp) %>%
  psych::alpha()
```

```
Reliability analysis
```

```
Call: psych::alpha(x = .)

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean  sd median_r
      0.81      0.82    0.76       0.6 4.4 0.019  4.2 1.1     0.58

    95% confidence boundaries
         lower alpha upper
Feldt     0.77  0.81  0.85
Duhachek  0.78  0.81  0.85

 Reliability if an item is dropped:
                            raw_alpha std.alpha G6(smc) average_r S/N
cs_explain_ideas_exp             0.69      0.69    0.53      0.53 2.3
cs_find_clarification_exp        0.73      0.74    0.58      0.58 2.8
cs_learn_different_styles_exp    0.81      0.81    0.68      0.68 4.2
                             alpha se var.r med.r
cs_explain_ideas_exp             0.035    NA  0.53
cs_find_clarification_exp        0.031    NA  0.58
cs_learn_different_styles_exp    0.022    NA  0.68

 Item statistics
                               n raw.r std.r r.cor r.drop mean  sd
cs_explain_ideas_exp          300  0.88  0.88  0.80   0.71  4.2 1.3
cs_find_clarification_exp     300  0.85  0.86  0.76   0.68  4.5 1.2
cs_learn_different_styles_exp 300  0.84  0.82  0.67   0.61  4.0 1.4

Non missing response frequency for each item
                                 1    2    3    4    5    6 miss
cs_explain_ideas_exp          0.04 0.09 0.11 0.33 0.26 0.17    0
cs_find_clarification_exp     0.02 0.05 0.13 0.27 0.32 0.21    0
cs_learn_different_styles_exp 0.06 0.09 0.21 0.24 0.22 0.17    0
```

Similarly, to `si`, `cs_imp` and `cs_exp` show very good internal consistency scores: $\alpha_{cs\_imp} = 0.86$ and $\alpha_{cs\_exp} = 0.81$. Based on these results we could be confident to use our latent variables for further analysis.

7.8.3 Confirmatory factor analysis

However, while Cronbach's α is very popular due to its simplicity, there is plenty of criticism. Therefore, it is often not enough to report the Cronbach's α, but undertake additional steps. Depending on the stage of development of your measurement instrument (e.g. your questionnaire), you likely have to perform one of the following before computing the α scores:

- *Exploratory factor analysis (EFA)*: Generally used to identify latent variables

in a set of questionnaire items. Often these latent variables are not yet known. This process is often referred to as *dimension reduction*.

- *Confirmatory factor analysis (CFA)*: This approach is used to confirm whether a set of items truly reflect a latent variable which we defined ex-ante.

During the development of a new questionnaire, researchers often perform EFA on a pilot dataset to see which factors would emerge without prior assumptions. Once this process has been completed, another round of data would be collected to perform a CFA. This CFA ideally confirms the results from the pilot study. It is also not unusual to split a larger dataset into two parts and then perform EFA and CFA on separate sub-samples. However, this latter approach has the disadvantage that researchers would not have the opportunity to improve the questionnaire between EFA and CFA.

Since the `gep` data is based on an established measurement tool, we perform a CFA. To perform a CFA, we use the popular `lavaan` (*Latent Variable Analysis*) package. The steps of running a CFA in *R* include:

1. Define which variables are supposed to measure a specific latent variable (i.e. creating a model).
2. Run the CFA to see whether our model fits the data we collected.
3. Interpret the results based on various indicators.

The code chunk below breaks down these steps in the context of the `lavaan` package.

```
library(lavaan)

#1: Define the model which explains how items relate to latent variables
model <- '
social_integration =~
si_socialise_with_people_exp +
si_supportive_friends_exp +
si_time_socialising_exp

comm_skills_imp =~
cs_explain_ideas_imp +
cs_find_clarification_imp +
cs_learn_different_styles_imp

comm_skills_exp =~
cs_explain_ideas_exp +
cs_find_clarification_exp +
cs_learn_different_styles_exp
```

```
'

#2: Run the CFA to see how well this model fits our data
fit <- cfa(model, data = gep)

#3a: Extract the performance indicators
fit_indices <- fitmeasures(fit)

#3b: We tidy the results with enframe() and
#    pick only those indices we are most interested in
fit_indices %>%
  enframe() %>%
  filter(name == "cfi" |
         name == "srmr" |
         name == "rmsea") %>%
  mutate(value = round(value, 3))   # Round to 3 decimal places
```

```
# A tibble: 3 x 2
  name  value
  <chr> <lvn.vctr>
1 cfi   0.967
2 rmsea 0.081
3 srmr  0.037
```

The `enframe()` function is useful to turn a named vector (i.e. a named sequence of values) into a `tibble`, which we can further manipulate using the functions we already know. However, it is not required to take this step to inspect the results. This can be done out of pure convenience. If we want to know where `name` and `value` came from we have to inspect the output from `enframe(fit_indices)`.

```
enframe(fit_indices)
```

```
# A tibble: 46 x 2
   name            value
   <chr>           <lvn.vctr>
 1 npar            2.100000e+01
 2 fmin            1.177732e-01
 3 chisq           7.066390e+01
 4 df              2.400000e+01
 5 pvalue          1.733810e-06
 6 baseline.chisq  1.429729e+03
 7 baseline.df     3.600000e+01
 8 baseline.pvalue 0.000000e+00
```

```
 9 cfi             9.665187e-01
10 tli             9.497780e-01
# i 36 more rows
```

These column names were generated when we called the function `enframe()`. I often find myself working through chains of analytical steps iteratively to see what the intermediary steps produce. This also makes it easier to spot any mistakes early on. Therefore, I recommend slowly building up your `dplyr` chains of function calls, especially when you just started learning *R* and the `tidyverse` approach of data analysis.

The results of our CFA appear fairly promising:

- The `cfi` (*Comparative Fit Index*) lies above `0.95`,
- The `rmsea` (*Root Mean Square Error of Approximation*) appears slightly higher than desirable, which usually is lower than `0.6`, and
- The `srmr` (*Standardised Root Mean Square Residual*) lies well below `0.08` (West et al. 2012; Hu and Bentler 1999).

Overall, the model seems to suggest a good fit with our data. Combined with the computed Cronbach's α, we can be reasonably confident in our latent variables and perform further analytical steps.

7.8.4 Reversing items with opposite meaning

Questionnaires like the `gep` consist of items that are all phrased in the 'same direction'. What I mean by this is that all questions are asked so that a high score would carry the same meaning. However, this is not always true. Consider the following three questions as an example:

- Q1: *'I enjoy talking to people I don't know.'*
- Q2: *'I prefer to stay at home.'*
- Q3: *'Making new friends is exciting.'*

Q1 and Q3 both refer to aspects of socialising with or getting to know others. Q2, however, seems to ask for the opposite, i.e. not engaging with others and staying at home. Intuitively, we would assume that a participant who scores high on Q1 and Q2 would score low on Q3. Q3 is a so-called *reversed item*, i.e. the meaning of the question is reversed compared to other questions in this set. This strategy is commonly used to test for respondents' engagement with a questionnaire and break a possible response pattern. For example, if a participant scores 5 out of 6 on Q1, Q2 and Q3, we have to assume that this person might not have paid much attention to the statements because someone who scores high on Q1 and Q3 would likely not score high on Q2. However, their usefulness is debated across disciplines. When designing your

data collection tool, items must be easy to understand, short, not double-barreled and reflect a facet of a latent variable appropriately. These aspects are far more important than reversing the meaning of items.

Suppose we work with a dataset containing reversed items. In that case, we have to recode the answers for each participant to reflect the same meaning as the other questions before computing the Cronbach's α or performing an EFA or CFA.

Let's assume we ask some of our friends to rate Q1, Q2 and Q3 on a scale from 1-6. We might obtain the following results:

```
df_social <- tibble(name = c("Agatha Harkness", "Daren Cross", "Hela",
                             "Aldrich Killian", "Malekith", "Ayesha"),
                    q1 = c(6, 2, 4, 5, 5, 2),
                    q2 = c(2, 5, 1, 2, 3, 6),
                    q3 = c(5, 1, 5, 4, 5, 1))

df_social
```

```
# A tibble: 6 x 4
  name               q1    q2    q3
  <chr>           <dbl> <dbl> <dbl>
1 Agatha Harkness     6     2     5
2 Daren Cross         2     5     1
3 Hela                4     1     5
4 Aldrich Killian     5     2     4
5 Malekith            5     3     5
6 Ayesha              2     6     1
```

While, for example, `Agatha Harkness` and `Aldrich Killian` are both very social, `Ayesha` seems to be rather the opposite. For all of our respondents, we can notice a pattern that if the value of `q1` is high, usually `q3` is high too, but `q2` is low. We would be very disappointed and probably puzzled if we now computed the Cronbach's α of our latent variable `social`.

```
# Cronbach's alpha without reverse coding q2
df_social %>%
  select(-name) %>%
  psych::alpha()
```

```
Warning in psych::alpha(.): Some items were negatively correlated with the first principal component ar
should be reversed.
To do this, run the function again with the 'check.keys=TRUE' option

Some items ( q2 ) were negatively correlated with the first principal component and
```

```
probably should be reversed.
To do this, run the function again with the 'check.keys=TRUE' option


Reliability analysis
Call: psych::alpha(x = .)

  raw_alpha std.alpha G6(smc) average_r   S/N ase mean   sd median_r
      -2.2      -1.7     0.7     -0.27 -0.64 1.7  3.6 0.69     -0.8

    95% confidence boundaries
          lower alpha upper
Feldt    -12.47 -2.18  0.52
Duhachek  -5.49 -2.18  1.13

 Reliability if an item is dropped:
   raw_alpha std.alpha G6(smc) average_r   S/N alpha se var.r med.r
q1    -21.00    -21.04   -0.91     -0.91 -0.95   17.947    NA -0.91
q2      0.94      0.95    0.91      0.91 19.70    0.042    NA  0.91
q3     -7.61     -8.03   -0.80     -0.80 -0.89    6.823    NA -0.80

 Item statistics
   n raw.r std.r r.cor r.drop mean  sd
q1 6  0.93  0.94  0.95   0.29  4.0 1.7
q2 6 -0.58 -0.61 -0.89  -0.88  3.2 1.9
q3 6  0.83  0.84  0.93  -0.22  3.5 2.0

Non missing response frequency for each item
      1    2    3    4    5    6 miss
q1 0.00 0.33 0.00 0.17 0.33 0.17    0
q2 0.17 0.33 0.17 0.00 0.17 0.17    0
q3 0.33 0.00 0.00 0.17 0.50 0.00    0
```

The function `alpha()` is clever enough to make us aware that something went wrong. It even tells us that `Some items ( q2 ) were negatively correlated with the total scale and probably should be reversed`. Also, the Cronbach's α is now negative, which is another indicator that reversed items might be present in the dataset. To remedy this problem, we have to recode `q2` to reflect the opposite meaning. Inverting the coding usually implies we have to mirror the scores, i.e. the largest value become the smallest one and vice versa. Since we use a 6-point Likert scale, we have to recode values in the following way:

- 6 -> 1
- 5 -> 2
- 4 -> 3
- 3 -> 4

- `2 -> 5`
- `1 -> 6`

As is so often the case in *R*, there is more than just one way of achieving this:

- Compute a new variable with `mutate()` only,
- Apply `recode()` to the variable in question, or
- Use the `psych` package.

I will introduce you to all three methods, because each comes with their own benefits and drawbacks. Thus, for me, it is very context-specific.

7.8.4.1 Reversing items with `mutate()`

Let us begin with the method that is my favourite because it is very quick to execute and only requires one function. With `mutate()`, we can calculate the reversed score for an item by applying the following formula:

$$score_{reversed} = score_{max} + score_{min} - score_{obs}$$

In other words, the reversed score $score_{reversed}$ of an item equals the theoretical maximum score on a given scale $score_{max}$ plus the theoretical minimum score on this scale $score_{min}$, and then we subtract the observed score $score_{obs}$, i.e. the actual response by our participant. For example, if someone in our study scored a `5` on `q2`, we could compute the reversed score as follows: `6 + 1 - 5 = 2` because our Likert scale ranges from `1` to `6`. In *R*, we can apply this idea to achieve this for multiple observations at once.

```
df_social_r <-
  df_social %>%
  mutate(q2_r = 7 - q2)

df_social_r %>%
  select(q2, q2_r)
```

```
# A tibble: 6 x 2
     q2  q2_r
  <dbl> <dbl>
1     2     5
2     5     2
3     1     6
4     2     5
5     3     4
6     6     1
```

This technique is simple, flexible and easy to read and understand. We can change this formula to fit any scale length and make adjustments accordingly.

I strongly recommend starting with this approach before opting for one of the other two.

7.8.4.2 Reversing items with `recode()`

Another option to reverse items is `recode()`. Conceptually, `recode()` functions similarly to `fct_recode()`, only that it can handle numbers and factors. Therefore, we can apply a similar strategy as we did before with factors (see Chapter @ref(recoding-factors)).

```
df_social_r <-
  df_social %>%
  mutate(q2_r = recode(q2,
                       "6" = "1",
                       "5" = "2",
                       "4" = "3",
                       "3" = "4",
                       "2" = "5",
                       "1" = "6")
         )

df_social_r
```

```
# A tibble: 6 x 5
  name               q1    q2    q3 q2_r
  <chr>           <dbl> <dbl> <dbl> <chr>
1 Agatha Harkness     6     2     5 5
2 Daren Cross         2     5     1 2
3 Hela                4     1     5 6
4 Aldrich Killian     5     2     4 5
5 Malekith            5     3     5 4
6 Ayesha              2     6     1 1
```

While this might seem convenient, it requires more coding and is, therefore, more prone to errors. If we had to do this for multiple items, this technique is not ideal. However, it is a great option whenever you need to change values to something entirely customised, e.g. every `6` needs to be come a `10` and every `3` needs to become a `6`. However, such cases are rather rare in my experience.

Be aware, that after using `recode()`, your variable becomes a `chr`, and you likely have to convert it back to a numeric data type. We can adjust the above code to achieve recoding and converting in one step.

```
df_social_r <-
  df_social %>%
```

```
  mutate(q2_r = as.numeric(recode(q2,
                                  "6" = "1",
                                  "5" = "2",
                                  "4" = "3",
                                  "3" = "4",
                                  "2" = "5",
                                  "1" = "6"))
  )

df_social_r
```

```
# A tibble: 6 x 5
  name               q1    q2    q3  q2_r
  <chr>           <dbl> <dbl> <dbl> <dbl>
1 Agatha Harkness     6     2     5     5
2 Daren Cross         2     5     1     2
3 Hela                4     1     5     6
4 Aldrich Killian     5     2     4     5
5 Malekith            5     3     5     4
6 Ayesha              2     6     1     1
```

7.8.4.3 Reversing items with the `psych` package

The last option I would like to share with you is with the `psych` package. This option can be helpful when reversing multiple items at once without having to create additional variables. The function `reverse.code()` takes a data set with items and requires the specification of a coding key, i.e. `keys`. This coding key indicates whether an item in this dataset should be reversed or not by using `1` for items that should not be reversed and `-1` for items you wish to reverse.

```
items <- df_social %>% select(-name)

df_social_r <- psych::reverse.code(keys = c(1, -1, 1),
                      items = items,
                      mini = 1,
                      maxi = 6)

df_social_r
```

```
     q1 q2- q3
[1,]  6   5  5
[2,]  2   2  1
[3,]  4   6  5
[4,]  5   5  4
```

```
[5,]  5   4  5
[6,]  2   1  1
```

7.8.4.4 A final remark about reversing item scores

No matter which method we used, we successfully converted the scores from `q2` and stored them in a new variable `q2_r` or, in the case of the `psych` package, as `q2-`. We can now perform the internal consistency analysis again.

```
# Cronbach's alpha after reverse coding q2
df_social_r %>%
  select(-name, -q2) %>%
  psych::alpha()
```

```
Reliability analysis
Call: psych::alpha(x = .)

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean  sd median_r
      0.95      0.95    0.95      0.87  21 0.033  3.8 1.8     0.91

    95% confidence boundaries
         lower alpha upper
Feldt     0.80  0.95  0.99
Duhachek  0.89  0.95  1.02

 Reliability if an item is dropped:
     raw_alpha std.alpha G6(smc) average_r S/N alpha se var.r med.r
q1        0.95      0.95    0.91      0.91  21    0.037    NA  0.91
q3        0.88      0.89    0.80      0.80   8    0.092    NA  0.80
q2_r      0.94      0.95    0.91      0.91  20    0.042    NA  0.91

 Item statistics
     n raw.r std.r r.cor r.drop mean  sd
q1   6  0.94  0.94  0.91   0.87  4.0 1.7
q3   6  0.98  0.98  0.98   0.96  3.5 2.0
q2_r 6  0.95  0.95  0.91   0.88  3.8 1.9

Non missing response frequency for each item
        1    2    4    5    6 miss
q1   0.00 0.33 0.17 0.33 0.17    0
q3   0.33 0.00 0.17 0.50 0.00    0
q2_r 0.17 0.17 0.17 0.33 0.17    0
```

Now everything seems to make sense, and the internal consistency is excellent. We can assume that all three questions reflect one latent construct. From here,

we can return to computing our latent variables (see Chapter @ref(computing-latent-variables)) and continue with our analysis.

When working with datasets, especially those you have not generated yourself, it is always important to check whether any items are meant to be reverse coded. Missing this aspect will likely throw many surprising results and can completely ruin your analysis. I vividly remember my PhD supervisor's story about a viva where a student forgot about reversing their scores. The panellists noticed that the correlations presented were counter-intuitive, and only then it was revealed that the entire analysis had to be redone. Tiny mistakes can often have an enormous ripple effect. Make sure you have 'reversed items' on your checklist when performing data cleaning and wrangling.

7.9 Once you finished with data wrangling

Once you finished your data cleaning, I recommend writing (i.e. exporting) your cleaned data out of *R* into your *'01_tidy_data'* folder. You can then use this dataset to continue with your analysis. This way, you do not have to run all your data wrangling and cleaning code every time you open your *R* project. To write your tidy dataset onto your hard drive of choice, we can use `readr` in the same way as we did at the beginning to import data. However, instead of `read_csv()` we have to use another function that writes the file into a folder. The function `write_csv()` first takes the name of the object we want to save and then the folder and file name we want to save it to. We only made changes to the `wvs` dataset and after cleaning and dealing with missing data, we saved it in the object `wvs_nona`. So we should save it to our hard drive and name it, for example, `wvs_nona.csv`, because it is not only clean, but also has *'no NA'*, i.e. no missing data.

```
write_csv(wvs_clean, "01_tidy_data/wvs_nona.csv")
```

This chapter has been a reasonably long one. Nonetheless, it only covered the basics of what can and should be done when preparing data for analysis. These steps should not be rushed or skipped. It is essential to have the data cleaned appropriately. This process helps you familiarise yourself with the dataset in great depth, and it makes you aware of limitations or even problems of your data. In the spirit of Open Science[19], using *R* also helps to document the steps you undertook to get from a raw dataset to a tidy one. This is also beneficial if you intend to publish your work later. Reproducibility has become an essential

[19] https://osf.io

aspect of transparent and impactful research and should be a guiding principle of any empirical research.

Now that we have learned the basics of data wrangling, we can finally start our data analysis.

7.10 Exercises

If you would like to test your knowledge or simply practice what we covered in this chapter you can copy and paste the following code if you have the `r4np` installed already.

```
# Option 1:
learnr::run_tutorial("ex_data_wrangling", package = "r4np")
```

If you have not yet installed the `r4np` package, you will have to do this first using by using this code chunk:

```
# Option 2:
devtools::install_github("ddauber/r4np")
learnr::run_tutorial("ex_data_wrangling", package = "r4np")
```

8

Descriptive Statistics

The best way to understand how participants in your study have responded to various questions or experimental treatments is to use *descriptive statistics*. As the name indicates, their main purpose is to 'describe'. Most of the time, we want to describe the composition of our sample and how the majority (or minority) of participants performed.

In contrast, we use *inferential statistics* to make predictions. In Social Sciences, we are often interested in predicting how people will behave in certain situations and scenarios. We aim to develop models that help us navigate the complexity of social interactions that we all engage in but might not fully understand. We cover *inferential statistics* in later chapters of this book.

In short, descriptive statistics are an essential component to understand your data. To some extent, one could argue that we were already describing our data when we performed various data wrangling tasks (see Chapter @ref(data-wrangling)). The following chapters focus on essential descriptive statistics, i.e. those you likely want to investigate in 99.9 out of 100 research projects.

This book takes a 'visualised' approach to data analysis. Therefore, each section will entail data visualisations and statistical computing. A key learning outcome of this chapter is to plot your data using the package `ggplot2` and present your data's characteristics in different ways. Each chapter will ask questions about our dataset that we aim to answer visually and computationally. However, first, we need to understand how to create plots in *R*.

8.1 Plotting in *R* with `ggplot2`

Plotting can appear intimidating at first but is very easy and quick once you understand the basics. The `ggplot2` package is a very popular package to generate plots in *R*, and many other packages are built upon it. This makes it a very flexible tool to create almost any data visualisation you could imagine. If you want to see what is possible with `ggplot2`, you might want to consider

looking at #tidytuesday[1] on Twitter, where novices and veterans share their data visualisations every week.

To generate any plot, we need to define three components at least:

- a dataset,
- variables we want to plot, and
- a function to indicate how we want to plot them, e.g. as lines, bars, points, etc.

Admittedly, this is a harsh oversimplification, but it will serve as a helpful guide to get us started. The function `ggplot()` is the one responsible for creating any type of data visualisation. The generic structure of a `ggplot()` looks like this:

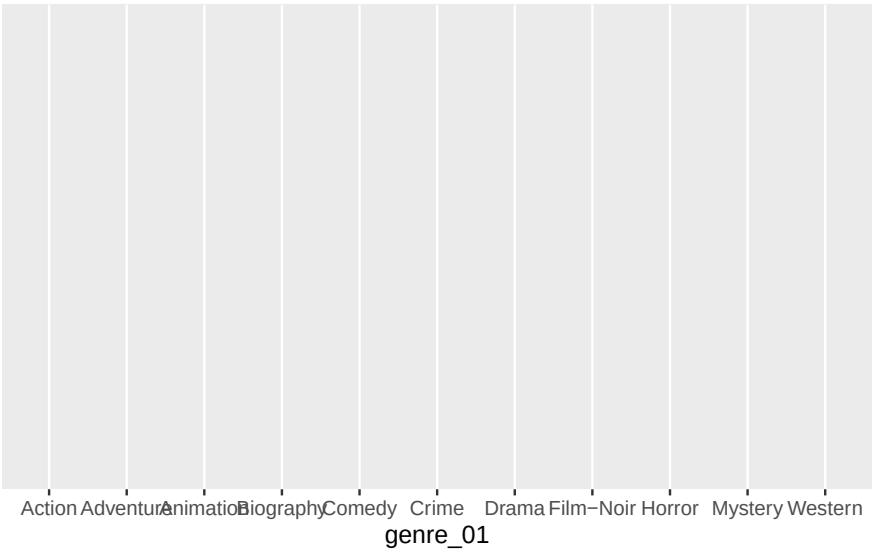
ggplot(data, aes(x = variable_01, y = variable_02))

In other words, we first need to provide the dataset `data`, and then the aesthetics (`aes()`). Think of `aes()` as the place where we define our variables (i.e. `x` and `y`). For example, we might be interested to know which movie genre is the most popular among the top 250 IMDb movies. The dataset `imdb_top_250` from the `r4np` package allows us to find an answer to this question. Therefore we define the components of the plot as follows:

- our data is `imdb_top_250`, and
- our variable of interest is `genre_01`.

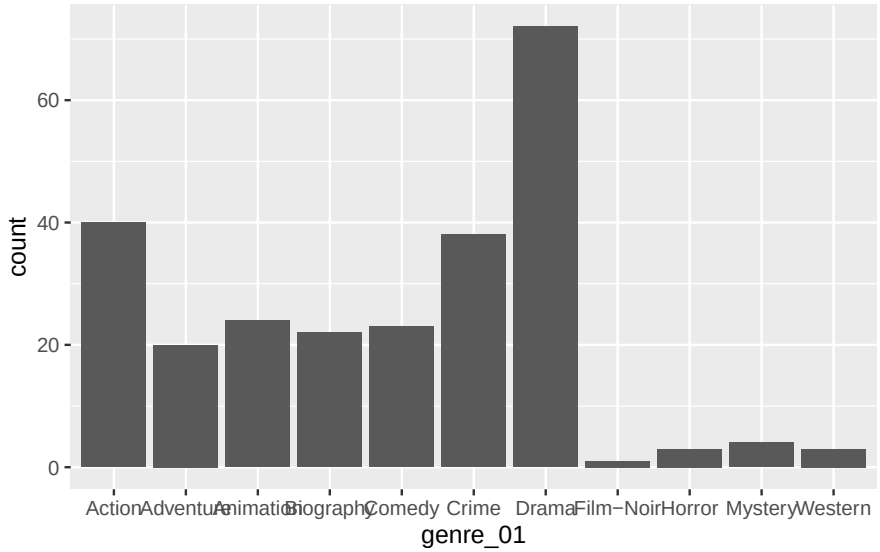
```
ggplot(imdb_top_250, aes(x = genre_01))
```

[1] https://twitter.com/search?q=%23tidytuesday



Running this line of code will produce an empty plot. We only get labels for our x-axis since we defined it already. However, we have yet to tell `ggplot()` how we want to represent the data on this canvas. Your choice for how you want to plot your data is usually informed by the type of data you use and the statistics you want to represent. For example, plotting the mean of a factor is not meaningful, e.g. computing the mean of movie genres. On the other hand, we can count how often specific genres appear in our dataset. One way of representing a factor's count (or frequency) is to use a bar plot. To add an element to `ggplot()`, i.e. bars, we use `+` and append the function `geom_bar()`, which draws bars. The `+` operator works similar to `%>%` and allows to chain multiple functions one after the other as part of a `ggplot()`.

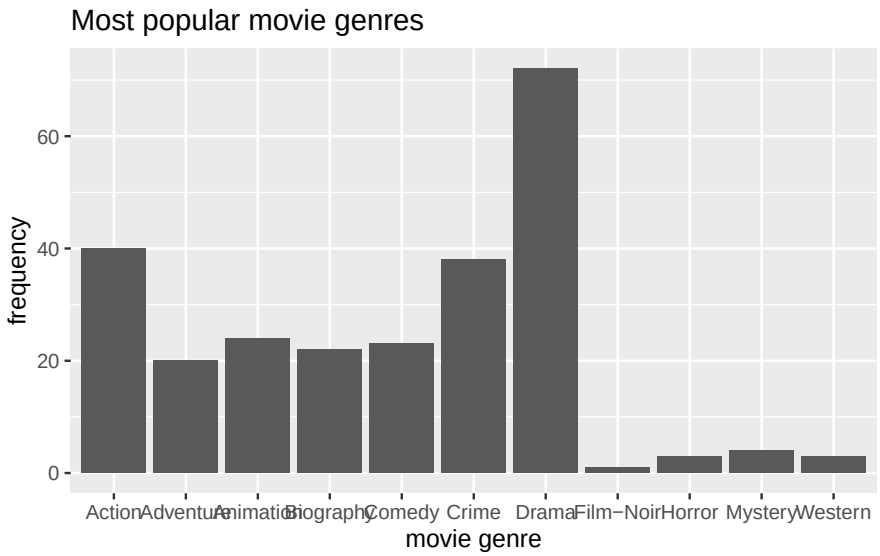
```
ggplot(imdb_top_250, aes(x = genre_01)) +
  geom_bar()
```



With only two lines of coding, we created a great looking plot. We can see that `Drama` is by far the most popular genre, followed by `Action` and `Crime`. Thus, we successfully found an answer to our question. Still, there are more improvements necessary to use it in a publication.

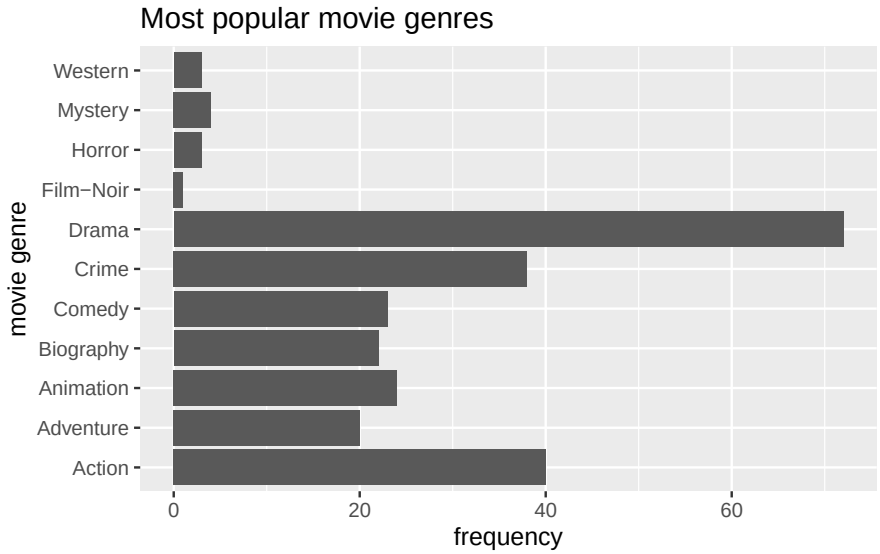
We can use `+` to add other elements to our plot, such as a title and proper axes labels. Here are some common functions to further customise our plot:

```
ggplot(imdb_top_250, aes(x = genre_01)) +
  geom_bar() +
  ggtitle("Most popular movie genres") +  # Add a title
  xlab("movie genre") +                   # Rename x-axis
  ylab("frequency")                       # Rename y-axis
```



When working with factors, the category names can be rather long. In this plot, we have lots of categories, and the labels `Adventure`, `Animation`, and `Biography` are a bit too close to each other for my taste. This might be an excellent opportunity to use `coord_flip()`, which rotates the entire plot by 90 degrees, i.e. turning the x-axis into the y-axis and vice versa. This makes the labels much easier to read.

```
ggplot(imdb_top_250, aes(x = genre_01)) +
  geom_bar() +
  ggtitle("Most popular movie genres") +
  xlab("movie genre") +
  ylab("frequency") +
  coord_flip()
```



Our plot is almost perfect, but we should take one more step to make reading and understanding this plot even easier. At the moment, the bars are ordered alphabetically by movie genre. Unfortunately, this is hardly ever a useful way to order your data. Instead, we might want to sort the data by frequency, showing the most popular genre at the top. To achieve this, we could either sort the movies by hand (see Chapter @ref(reordering-factor-levels)) or slightly amend what we have coded so far.

The problem you encounter when rearranging a `geom_bar()` with only one variable is that we do not have an explicit value to indicate how we want to sort the bars. Our current code is based on the fact that `ggplot` does the counting for us. So, instead, we need to do two things:

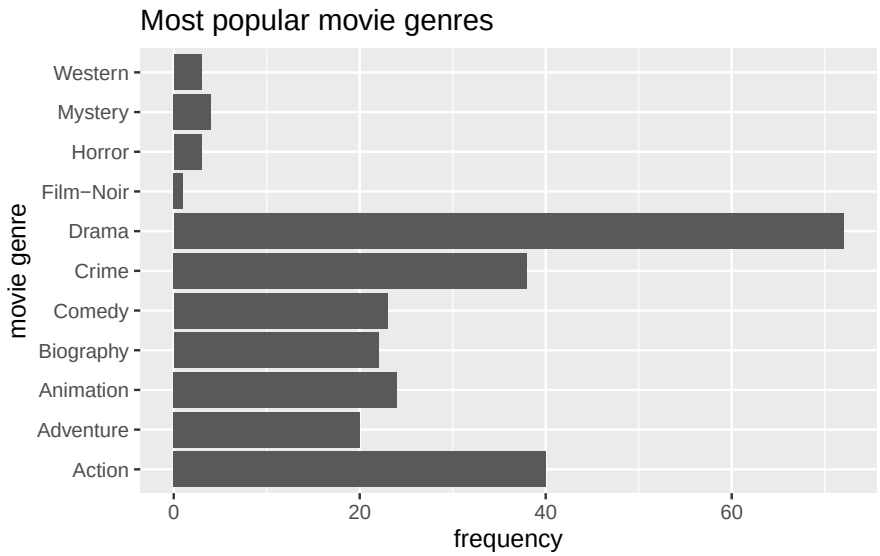
- create a table with all genres and their frequency, and
- use this table to plot the genres by the frequency we computed

```
# Step 1: The frequency table only
imdb_top_250 %>%
  count(genre_01)
```

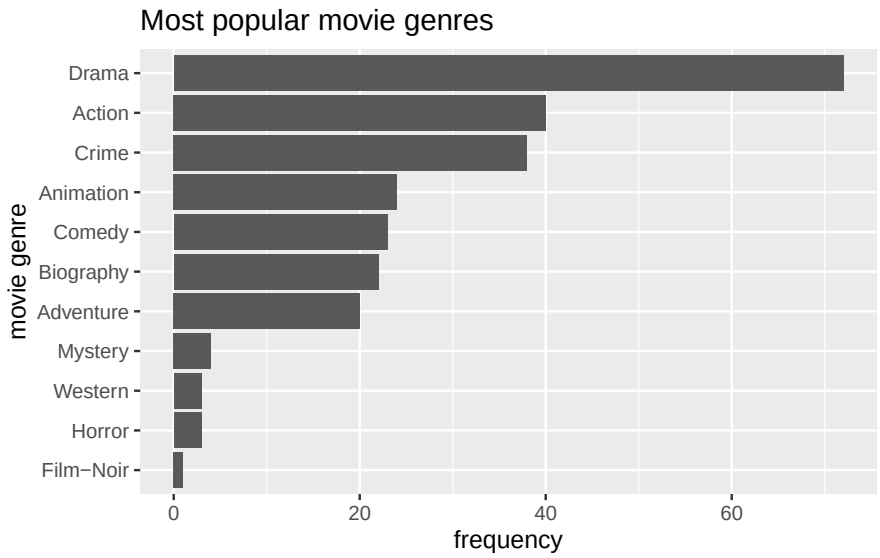
```
# A tibble: 11 x 2
   genre_01      n
   <fct>     <int>
 1 Action       40
 2 Adventure    20
 3 Animation    24
 4 Biography    22
```

```
 5 Comedy       23
 6 Crime        38
 7 Drama        72
 8 Film-Noir     1
 9 Horror        3
10 Mystery       4
11 Western       3
```

```r
# Step 2: Plotting a barplot based on the frequency table
imdb_top_250 %>%
  count(genre_01) %>%
  ggplot(aes(x = genre_01, y = n)) +

  # Use geom_col() instead of geom_bar()
  geom_col() +

  # Add titles for plot
  ggtitle("Most popular movie genres") +
  xlab("movie genre") +
  ylab("frequency") +

  # Rotate plot by 180 degrees
  coord_flip()
```



```
#Step 3: reorder() genre_01 by frequency, i.e. by 'n'
imdb_top_250 %>%
  count(genre_01) %>%
  ggplot(aes(x = reorder(genre_01, n), y = n)) +  # Use 'reorder()'
  geom_col() +
  ggtitle("Most popular movie genres") +
  xlab("movie genre") +
  ylab("frequency") +
  coord_flip()
```



Step 3 is the only code you need to create the desired plot. The other two steps only demonstrate how one can slowly build this plot, step-by-step. You might have noticed that I used `dplyr` to chain all these functions together (i.e. `%>%`), and therefore, it was not necessary to specify the dataset in `ggplot()`.

There are also two new functions we had to use: `geom_col()` and `reorder()`. The function `geom_col()` performs the same step as `geom_bar()` which can be confusing. The easiest way to remember how to use them is: If you use a frequency table to create your barplot, use `geom_col()`, if not, use `geom_bar()`. The function `geom_col()` requires that we specify an a-axis and a y-axis (our frequency scores), while `geom_bar()` works with one variable.

In many cases, when creating plots, you have to perform two steps:

- generate the statistics you want to plot, e.g. a frequency table, and
- plot the data via `ggplot()`

Now you have learned the fundamentals of plotting in *R*. We will create a lot more plots throughout the next chapters. They all share the same basic structure, but we will use different `geom_`s to describe our data. By the end of this book, you will have accrued enough experience in plotting in *R* that it will feel like second nature. If you want to deepen your knowledge of `ggplot`, you should take a look at the book 'ggplot: Elegant Graphics for Data Analysis'[2] or 'R Graphics Cookbook'[3] which moves beyond `ggplot2`. Apart from that, you can also find fantastic packages which extend the range of *'geoms'* you can use in the 'ggplot2 extensions gallery'[4].

8.2 Frequencies and relative frequencies: Counting categorical variables

One of the most fundamental descriptive statistics includes frequencies, i.e. counting the number of occurrences of a factor level, string or numeric value. For example, we might be interested to know which director produced most movies listed in the IMDb Top 250. Therefore multiple movies might belong to the same value in our factor `director`. We already used the required function to compute such a frequency when we looked at movie genres (see Chapter @ref(plotting-in-r-with-ggplot2)), i.e. `count()`.

```
imdb_top_250 %>% count(director)
```

```
# A tibble: 155 x 2
   director                 n
   <fct>                <int>
 1 Aamir Khan               1
 2 Adam Elliot              1
 3 Akira Kurosawa           6
 4 Alejandro G. Inarritu    1
 5 Alfred Hitchcock         6
 6 Andrei Tarkovsky         2
 7 Andrew Stanton           2
 8 Anthony Russo            2
 9 Anurag Kashyap           1
10 Asghar Farhadi           1
# i 145 more rows
```

The result indicates that 155 directors produced 250 movies. Thus, some

[2] https://ggplot2-book.org
[3] https://r-graphics.org
[4] https://exts.ggplot2.tidyverse.org/gallery/

movies have to be from the same director. As the frequency table shows, `Akira Kurosawa` directed `6` films, and `Alfred Hitchcock` also produced `6` movies. To keep the table more manageable, we can `filter()` our dataset and include only directors who made at least `6` movies because we know there are at least two already.

```
imdb_top_250 %>%
  count(director) %>%
  filter(n >= 6)
```

```
# A tibble: 6 x 2
  director              n
  <fct>             <int>
1 Akira Kurosawa        6
2 Alfred Hitchcock      6
3 Christopher Nolan     7
4 Martin Scorsese       7
5 Stanley Kubrick       7
6 Steven Spielberg      6
```

There are in total 6 directors who produced several excellent movies, and if you are a bit of a movie fan, all these names will sound familiar.

Besides *absolute frequencies*, i.e. the actual count of occurrences of a value, we sometimes wish to compute *relative frequencies* to make it easier to compare categories more easily. *Relative frequencies* are more commonly known as *percentages* which indicate the ratio or proportion of an occurrence relative to the total number of observations. Consider the following example, which reflects a study where participants were asked to rate a new movie after watching it.

```
glimpse(movie_ratings)
```

```
Rows: 26
Columns: 3
$ participant <chr> "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L"~
$ gender      <chr> "female", "female", "female", "female", "male", "female", ~
$ rating      <int> 5, 8, 4, 8, 3, 4, 10, 5, 2, 8, 4, 3, 7, 9, 3, 6, 4, 8, 10,~
```

It might be interesting to know whether our movie ratings were mainly from `male` or `female` viewers. Thus, we can compute the absolute and relative frequencies at once by creating a new variable with `mutate()` and the formula to calculate percentages, i.e. `n / sum(n)`.

```
movie_ratings %>%
  count(gender) %>%
```

```
  # add the relative distributions, i.e. percentages
  mutate(perc = n / sum(n))
```

```
# A tibble: 2 x 3
  gender     n  perc
  <chr>  <int> <dbl>
1 female    22 0.846
2 male       4 0.154
```

It seems we have a very uneven distribution of `male` and `female` participants. Thus, our analysis is affected by having more `female` participants. Consequently, any further computation we perform with this dataset would reflect the opinion of `female` viewers more than those of `male` ones. This is important to know because it limits the representativeness of our results. The insights gained from primarily `female` participants in this study will not be helpful when trying to understand `male` audiences. Therefore, understanding who the participants are in your study is essential before undertaking any further analysis. Sometimes, it might even imply that we have to change the focus of our study entirely.

8.3 Central tendency measures: Mean, Median, Mode

The *mean*, *median* and *mode* (the 3 Ms) are all measures of central tendency, i.e. they provide insights into how our data is distributed. Measures of central tendency help to provide insights into the most frequent/typical scores we can observe in the data for a given variable. All the 3Ms summarise our data/variable by a single score which can be helpful but sometimes also terribly misleading.

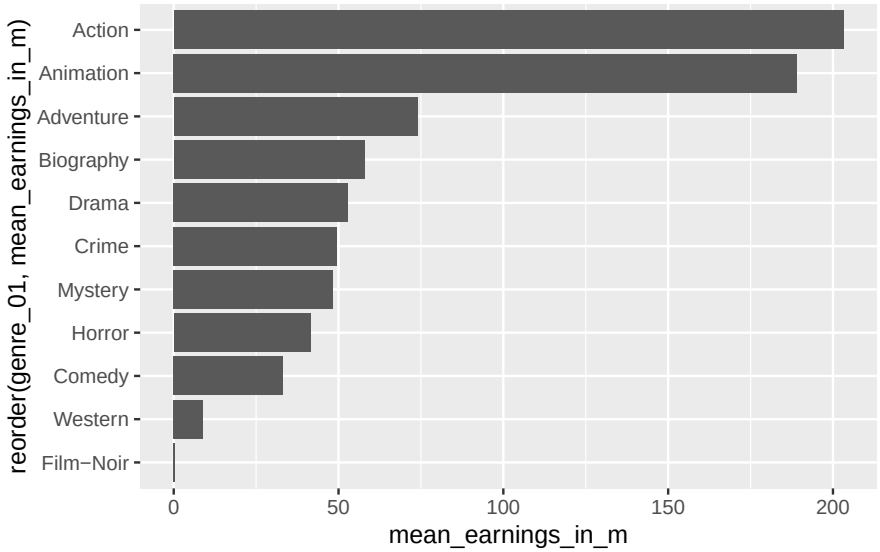
8.3.1 Mean

The mean is likely the most known descriptive statistics and, at the same time, a very powerful and influential one. For example, the average ratings of restaurants on Google might influence our decision on where to eat out. Similarly, we might consider the average rating of movies to decide which one to watch in the cinema with friends. Thus, what we casually refer to as the *'average'* is equivalent to the *'mean'*.

In *R*, it is simple to compute the mean using the function `mean()`. We used this function in Chapter @ref(functions) and Chapter @ref(latent-constructs) already. However, we have not looked at how we can plot the mean.

Assume we are curious to know how successful movies are in each of the genres. The mean could be a good starting point to answer this question because it provides the *'average'* success of movies in a particular genre. The simplest approach to investigating this is using a bar plot, like in Chapter @ref(plotting-in-r-with-ggplot2). So, we first create a tibble that contains the means of `gross_in_m` (i.e. the financial success fo a movie) for each genre in `genre_01`. Then we use this table to plot a bar plot with `geom_col()`.

```
imdb_top_250 %>%

  # Group data by genre
  group_by(genre_01) %>%

  # Compute the mean for each group (remove NAs via na.rm = TRUE)
  summarise(mean_earnings_in_m = mean(gross_in_m, na.rm = TRUE)) %>%

  # Create the plot
  ggplot(aes(x = reorder(genre_01, mean_earnings_in_m), y = mean_earnings_in_m)) +
  geom_col() +
  coord_flip()
```



You will have noticed that we use the function `summarise()` instead of `mutate()`. The `summarise()` function is a special version of `mutate()`. While `mutate()` returns a value for each row, summarise condenses our dataset, e.g. turning each row of observations into scores for each genre. Here is an example of a simple dataset which illustrates the difference between these two functions.

```
# Create a simple dataset from scratch with tibble()
(data <- tibble(number = c(1, 2, 3, 4, 5),
                group = factor(c("A", "B", "A", "B", "A"))))
```

```
# A tibble: 5 x 2
  number group
   <dbl> <fct>
1      1 A
2      2 B
3      3 A
4      4 B
5      5 A
```

```
# Return the group value for each observation/row
data %>%
  group_by(group) %>%
  mutate(sum = sum(number))
```

```
# A tibble: 5 x 3
# Groups:   group [2]
  number group   sum
   <dbl> <fct> <dbl>
1      1 A         9
2      2 B         6
3      3 A         9
4      4 B         6
5      5 A         9
```

```
# Returns the group value once for each group
data %>%
  group_by(group) %>%
  summarise(sum = sum(number))
```

```
# A tibble: 2 x 2
  group   sum
  <fct> <dbl>
1 A         9
2 B         6
```

Considering the results from our plot, it appears as if `Action` and `Animation` are far ahead of the rest. On average, both make around 200 million per movie. `Adventure` ranks third with only approximately 70 million. We can retrieve the exact earnings by removing the plot from the above code.

```
imdb_top_250 %>%
  group_by(genre_01) %>%
  summarise(mean_earnings_in_m = mean(gross_in_m, na.rm = TRUE)) %>%
  arrange(desc(mean_earnings_in_m))
```

```
# A tibble: 11 x 2
   genre_01  mean_earnings_in_m
   <fct>                  <dbl>
 1 Action                203.
 2 Animation             189.
 3 Adventure              73.9
 4 Biography              57.9
 5 Drama                  52.9
 6 Crime                  49.6
 7 Mystery                48.4
 8 Horror                 41.6
 9 Comedy                 33.2
10 Western                 8.81
11 Film-Noir               0.45
```

In the last line, I used a new function called `arrange()`. It allows us to sort rows in our dataset by a specified variable (i.e. a column). By default, `arrange()` sorts values in ascending order, putting the top genre last. Therefore, we have to use another function to change the order to descending with `desc()` to see the top-genre listed first. The function `arrange()` works similarly to `reorder()` but is used for sorting variables in data frames and less so for plots.

Based on this result, we might believe that `Action` and `Animation` movies are the most successful genres. However, we have not taken into account how many movies there are in each genre. Consider the following example:

```
# Assume there are 2 Action movies in the top 250
# both of which earn 203 million
2 * 203
```

```
[1] 406
```

```
# Assume there are 10 Drama movies in the top 25
# each of which earns 53 million
10 * 53
```

```
[1] 530
```

Thus, the *'mean'* alone might not be a good indicator. It can tell us which

genre is most successful based on a single movie, but we also should consider how many movies there are in each genre.

Let's add the number of movies (`n`) in each genre to our table. We can achieve this by using the function `n()`. Of course, we could also use the function `count()` first and then `summarise()` the earnings. There is hardly ever one single way to achieve the same result. This time it is truly a matter of personal taste.

```
imdb_top_250 %>%
  group_by(genre_01) %>%
  summarise(mean_earnings_in_m = mean(gross_in_m, na.rm = TRUE),
            n = n()) %>%
  arrange(desc(mean_earnings_in_m))
```

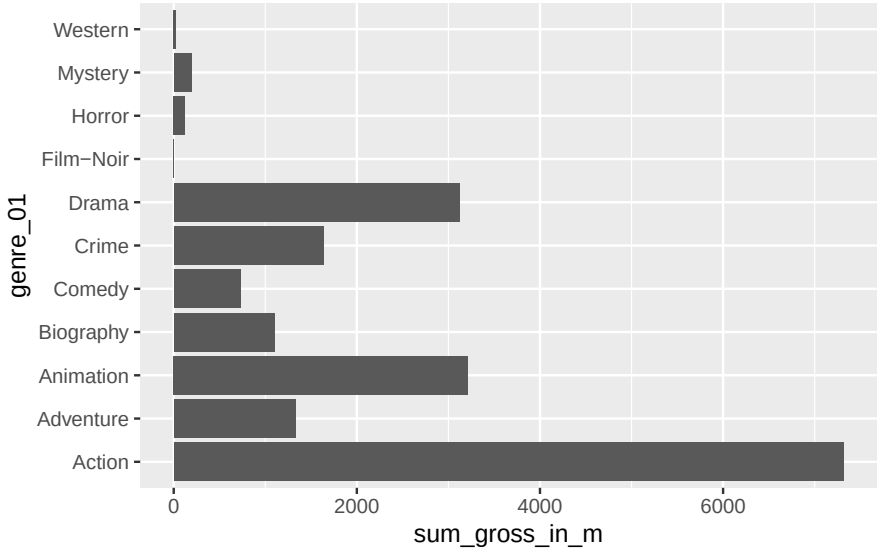
```
# A tibble: 11 x 3
   genre_01  mean_earnings_in_m     n
   <fct>                  <dbl> <int>
 1 Action                203.      40
 2 Animation             189.      24
 3 Adventure              73.9     20
 4 Biography              57.9     22
 5 Drama                  52.9     72
 6 Crime                  49.6     38
 7 Mystery                48.4      4
 8 Horror                 41.6      3
 9 Comedy                 33.2     23
10 Western                 8.81     3
11 Film-Noir               0.45     1
```

Our interpretation might slightly change based on these findings. There are considerably more movies in the genre `Drama` than in `Action` or `Animation`. We already plotted the frequency of movies per genre in Chapter @ref(plotting-in-r-with-ggplot2). Accordingly, we would have to think that `Drama` turns out to be the most successful genre if we consider the number of movies listed in the IMDb top 250.

As a final step, we can plot the sum of all earnings per genre as yet another indicator for the 'most successful genre'.

```
imdb_top_250 %>%
  filter(!is.na(gross_in_m)) %>%
  group_by(genre_01) %>%
  summarise(sum_gross_in_m = sum(gross_in_m)) %>%
  ggplot(aes(x = genre_01, y = sum_gross_in_m)) +
```

```
  geom_col() +
  coord_flip()
```



These results confirm that the `Action` genre made the most money out of all genres covered by the top 250 IMDb movies. I am sure you are curious to know which action movie contributed the most to this result. We can achieve this easily by drawing on functions we already know.

```
imdb_top_250 %>%
  select(title, genre_01, gross_in_m) %>%
  filter(genre_01 == "Action") %>%
  slice_max(order_by = gross_in_m,
            n = 5)
```

```
# A tibble: 5 x 3
  title                 genre_01 gross_in_m
  <chr>                 <fct>         <dbl>
1 Avengers: Endgame     Action         858.
2 Avengers: Infinity War Action        679.
3 The Dark Knight       Action         535.
4 The Dark Knight Rises Action         448.
5 Jurassic Park         Action         402.
```

`Avengers: Endgame` and `Avengers: Infinity` war rank the highest out of all `Action` movies. Two incredible movies if you are into Marvel comics.

In the last line, I sneaked in another new function from `dplyr` called `slice_max()`. This function allows us to pick the top 5 movies in our data. So, if you have many rows in your data frame (remember there are 40 action movies), you might want to be 'picky' and report only the top 3, 4 or 5. As you can see, `slice_max()` requires at least to arguments: `order_by` which defines the variable your dataset should be sorted by, and `n` which defines how many rows should be selected, e.g. `n = 3` for the top 3 movies or `n = 10` for the Top 10 movies. If you want to pick the lowest observations in your dataframe, you can use `slice_min()`. There are several other 'slicing' functions that can be useful and can be found on the corresponding dplyr website[5].

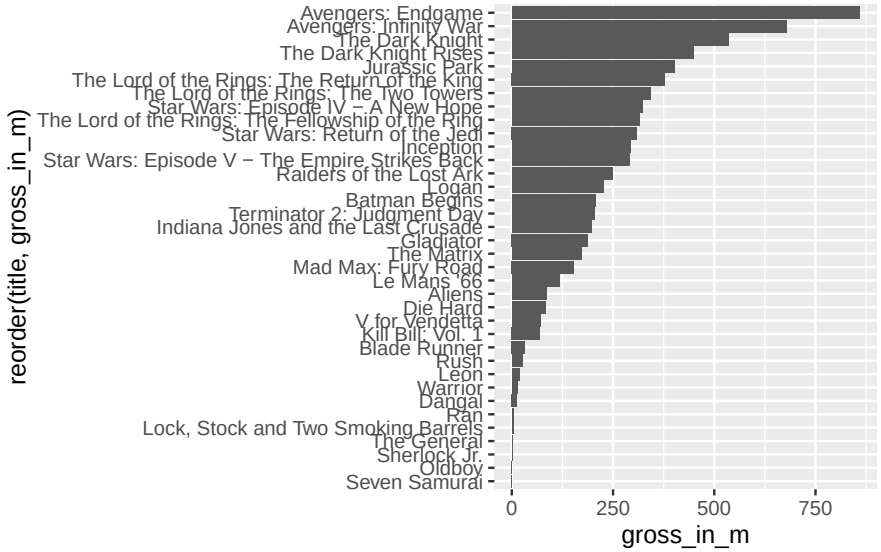
In conclusion, the mean helps understand how each movie, on average, performed across different genres. However, the mean alone provides a somewhat incomplete picture. Thus, we need to always look at means in the context of other information to gain a more comprehensive insight into our data.

8.3.2 Median

The *'median'* is the little, for some, lesser-known and used brother of the *'mean'*. However, it can be a powerful indicator for central tendency because it is not so much affected by outliers. With outliers, I mean observations that lie way beyond or below the average observation in our dataset. Let's inspect the `Action` genre more closely and see how each movie in this category performed relative to each other. We first `filter()` our dataset to only show movies in the genre `Action` and also remove responses that have no value for `gross_in_m` using the opposite (i.e. `!`) of the function `is.na()`.

```
imdb_top_250 %>%
  filter(genre_01 == "Action" & !is.na(gross_in_m)) %>%
  ggplot(aes(x = reorder(title, gross_in_m),
             y = gross_in_m)) +
  geom_col() +
  coord_flip()
```

[5] https://dplyr.tidyverse.org/reference/slice.html



`Avengers: Endgame` and `Avengers: Infinity War` are far ahead of any other movie. We can compute the mean with and without these two movies by adding the titles of the movies as a filter criterion.

```
# Mean earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action") %>%
  summarise(mean = mean(gross_in_m, na.rm = TRUE))
```

```
# A tibble: 1 x 1
   mean
  <dbl>
1  203.
```

```
# Mean earnings for Action genre without Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action" &
           title != "Avengers: Endgame" &
           title != "Avengers: Infinity War") %>%
  summarise(mean = mean(gross_in_m, na.rm = TRUE))
```

```
# A tibble: 1 x 1
   mean
  <dbl>
1  170.
```

The result is striking. Without these two movies, the `Action` genre would have fewer earnings per movie than the `Animation` genre. However, if we computed the median instead, we would not notice such a massive difference in results. This is because the median sorts a dataset, e.g. by `gross_in_m` and then picks the value that would cut the data into two equally large halves. It does not matter which value is the highest or lowest in our dataset. What matters is the value that is ranked right in the middle of all values. The median splits your dataset into two equally large datasets.

```
# Median earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action") %>%
  summarise(median = median(gross_in_m, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  median
   <dbl>
1   180.
```

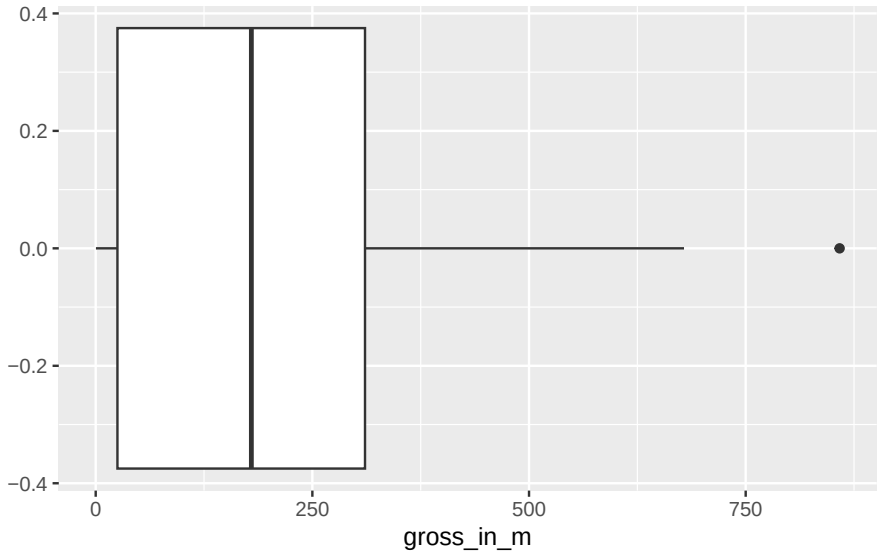
```
# Median earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action" &
           title != "Avengers: Endgame" &
           title != "Avengers: Infinity War") %>%
  summarise(median = median(gross_in_m, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  median
   <dbl>
1   163.
```

Both medians are much closer to each other, showing how much better suited the median is in our case. In general, when we report means, it is advisable to report the median as well. If a mean and median differ substantially, it could imply that your data 'suffers' from outliers. So we have to detect them and think about whether we should remove them for further analysis (see Chapter @ref(dealing-with-outliers)).

We can visualise medians using boxplots. Boxplots are a very effective tool to show how your data is distributed in one single data visualisation, and it offers more than just the mean. It also shows the spread of your data (see Chapter @ref(spread-of-data)).

```
imdb_top_250 %>%
  filter(genre_01 == "Action" & !is.na(gross_in_m)) %>%
  ggplot(aes(gross_in_m)) +
  geom_boxplot()
```



To interpret this boxplot consider Figure @ref(fig:anatomy-of-a-boxplot). Every boxplot consists of a 'box' and two whiskers (i.e. the lines leading away from the box). The distribution of data can be assessed by looking at different ranges:

- `MINIMUM` to `Q1` represents the bottom 25% of our observations,
- `Q1` to `Q3`, also known as the `interquartile range` (IQR) defines the middle 50% of our observations, and
- `Q3` to `MAXIMUM`, contains the top 25% of all our data.

Thus, with the help of a boxplot, we can assess whether our data is evenly distributed across these ranges or not. If observations fall outside a certain range (i.e. `1.5 IQR`) they are classified as outliers. We cover outliers in great detail in Chapter @ref(dealing-with-outliers).

Considering our boxplot, it shows that we have one observation that is an outlier, which is likely `Avengers: Endgame`, but what happened to the other Avengers movie? Is it not an outlier as well? It seems we need some more information. We can overlay another `geom_` on top to visualise where precisely each movie lies on this boxplot. We can represent each movie as a point by using `geom_point()`. This function requires us to define the values for the x

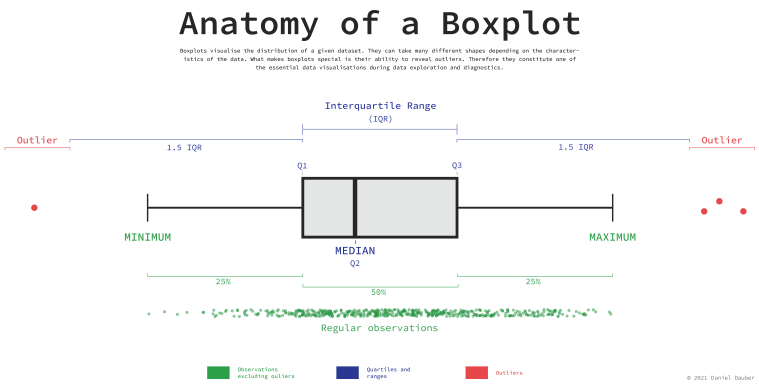
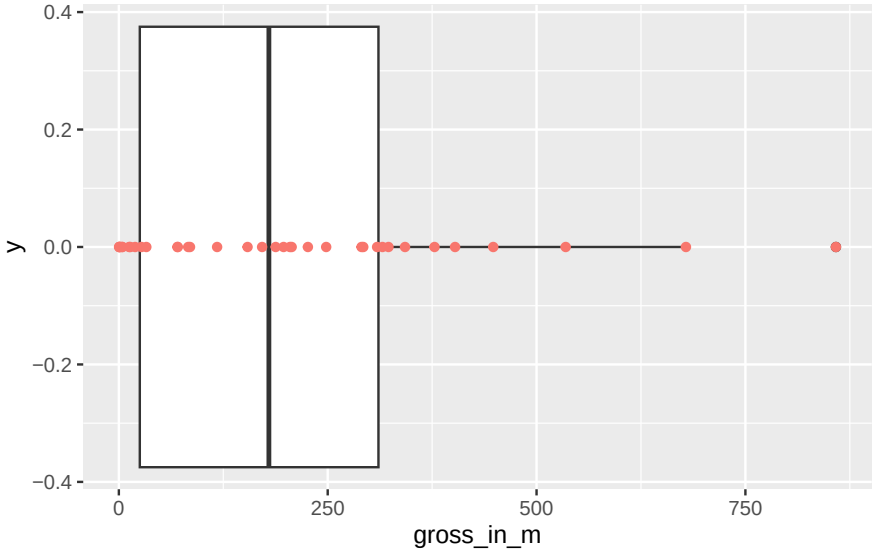


Figure 8.1: Anatomy of a Boxplot

and y-axis. Here it makes sense to set `y = 0`, which aligns all the dots in the middle of the boxplot.

```
imdb_top_250 %>%
  filter(genre_01 == "Action" & !is.na(gross_in_m)) %>%
  ggplot(aes(gross_in_m)) +
  geom_boxplot() +
  geom_point(aes(y = 0, col = "red"),
             show.legend = FALSE)
```



To make the dots stand out more, I changed the colour to `"red"`. By adding the `col` attributed (`color` and `colour` also work), `ggplot2` would automatically generate a legend. Since we do not need it, we can specify it directly in the `geom_point()` function. If you prefer the legend, remove `show.legend = FALSE`.

The two dots on the right are the two Avengers movies. This plot also nicely demonstrates why boxplots are so popular and helpful: They provide so many insights, not only into the central tendency of a variable, but also highlight outliers and, more generally, give a sense of the spread of our data (more about this in Chapter @ref(spread-of-data)).

The median is an important descriptive and diagnostic statistic and should be included in most empirical quantitative studies.

8.3.3 Mode

Finally, the 'mode' indicates which value is the most frequently occurring value for a specific variable. For example, we might be interested in knowing which IMDb rating was most frequently awarded to the top 250 movies.

When trying to compute the mode in *R*, we quickly run into a problem because there is no function available to do this straight away unless you search for a package that does it for you. However, before you start searching, let's reflect on what the mode does and why we can find the mode without additional packages or functions. The mode is based on the frequency of the occurrence of a value. Thus, the most frequently occurring value would be the one that is listed at the top of a frequency table. We have already created several

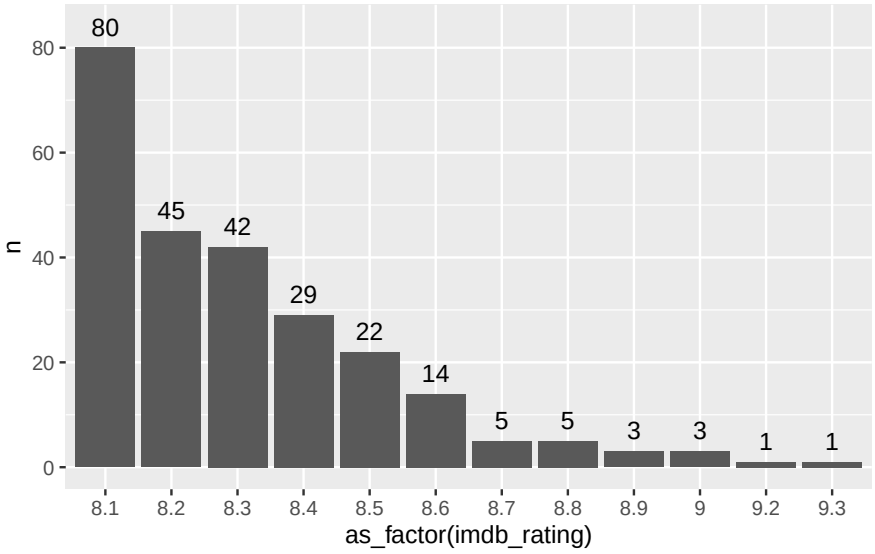
frequency tables in this book, and we can create another one to find the answer to our question.

```
imdb_top_250 %>% count(imdb_rating)
```

```
# A tibble: 12 x 2
   imdb_rating     n
         <dbl> <int>
 1         8.1    80
 2         8.2    45
 3         8.3    42
 4         8.4    29
 5         8.5    22
 6         8.6    14
 7         8.7     5
 8         8.8     5
 9         8.9     3
10         9       3
11         9.2     1
12         9.3     1
```

We can also easily visualise this frequency table in the same way as before. To make the plot a bit more 'fancy', we can add labels to the bar which reflect the frequency of the rating. We need to add the attribute `label` and also add a `geom_text()` layer to display them. Because the numbers would overlap with the bars, I 'nudge' the labels up by `4` units on the y-axis. Finally, because we treat `imdb_rating` as categories, we should visualise them `as_factor()`s.

```
imdb_top_250 %>%
  count(imdb_rating) %>%
  ggplot(aes(x = as_factor(imdb_rating),
             y = n,
             label = n)) +
  geom_col() +
  geom_text(nudge_y = 4)
```



The frequency table and the plot reveal that the mode for `imdb_rating` is `8.1`. In addition, we also get to know how often this rating was applied, i.e. `80` times. This provides much more information than receiving a single score and helps better interpret the importance of the mode as an indicator for a central tendency. Consequently, there is generally no need to compute the mode if you can have a frequency table instead. Still, if you are keen to have a function that computes the mode, you will have to write your own function, e.g. as shown in this post on stackeroverflow.com[6] or search for a package that coded one already.

As a final remark, it is also possible that you can find two or more modes in your data. For example, if the rating `8.1` appears `80` times and the rating `9.0` appears `80` times, both would be considered a mode.

8.4 Indicators and visualisations to examine the spread of data

Understanding how your data is spread out is essential to get a better sense of what your data is composed of. We already touched upon the notion of spread in Chapter @ref(median) through plotting a boxplot. Furthermore, the spread of data provides insights into how homogeneous or heterogeneous our participants' responses are. The following will cover some essential techniques to

[6] https://stackoverflow.com/questions/2547402/how-to-find-the-statistical-mode

investigate the spread of your data and investigate whether our variables are normally distributed, which is often a vital assumption for specific analytical methods (see Chapter @ref(sources-of-bias). In addition, we will aim to identify outliers that could be detrimental to subsequent analysis and significantly affect our modelling and testing in later stages.

8.4.1 Boxplot: So much information in just one box

The boxplot is a staple in visualising descriptive statistics. It offers so much information in just a single plot that it might not take much to convince you that it has become a very popular way to show the spread of data.

For example, we might be interested to know how long most movies run. Our gut feeling might tell us that most movies are probably around two hours long. One approach to finding out is a boxplot, which we used before.

```
# Text for annotations
longest_movie <-
  imdb_top_250 %>%
  filter(runtime_min == max(runtime_min)) %>%
  select(title)

shortest_movie <-
  imdb_top_250 %>%
  filter(runtime_min == min(runtime_min)) %>%
  select(title)

# Create the plot
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_boxplot() +
  annotate("text",
           label = glue::glue("{longest_movie}
                               ({max(imdb_top_250$runtime_min)} min)"),
           x = 310,
           y = 0.05,
           size = 2.5) +
  annotate("text",
           label = glue::glue("{shortest_movie}
                               ({min(imdb_top_250$runtime_min)} min)"),
           x = 45,
           y = 0.05,
           size = 2.5)
```

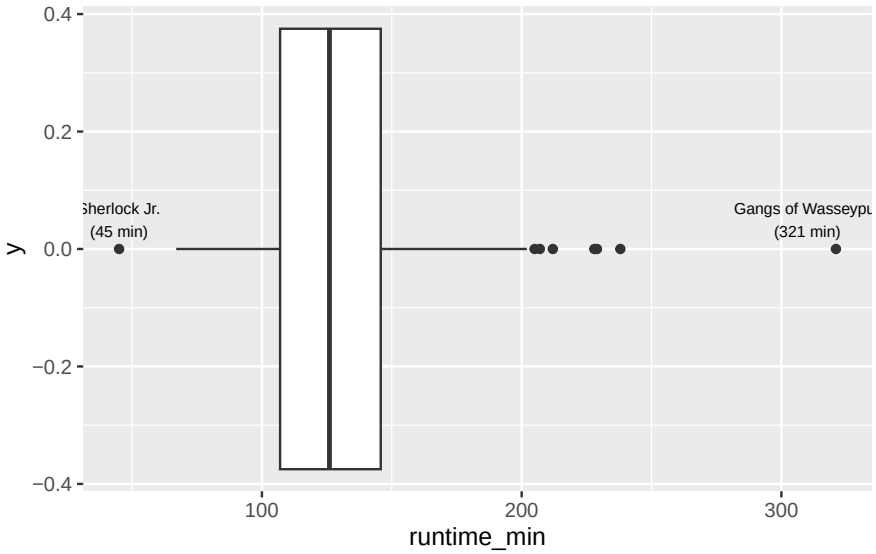


Figure 8.2: A boxplot

The results indicate that most movies are between 100 to 150 minutes long. Our intuition was correct. We find that one movie is even over 300 minutes long, i.e. over 5 hours: `Gangs of Wasseypur`. In contrast, the shortest movie only lasts 45 minutes and is called `Sherlock Jr.`. I added annotations using `annotate()` to highlight these two movies in the plot. A very useful package for annotations is `glue`, which allows combining text with data to label your plots. So, instead of looking up the longest and shortest movie, I used the `filter()` and `select()` functions to find them automatically. This has the great advantage that if I wanted to update my data, the name of the longest movie might change. However, I do not have to look it up again by hand.

As we can see from our visualisation, both movies would also count as outliers in our dataset (see Chapter @ref(outliers-iqr) for more information on the computation of such outliers).

8.4.2 Histogram: Do not mistake it as a bar plot

Another frequently used approach to show the spread (or distribution) of data is the histogram. The histogram easily gets confused with a bar plot. However, you would be very mistaken to assume that they are the same. Some key differences between these two types of plots is summarised in Table @ref(tab:histogram-vs-bar-plot).

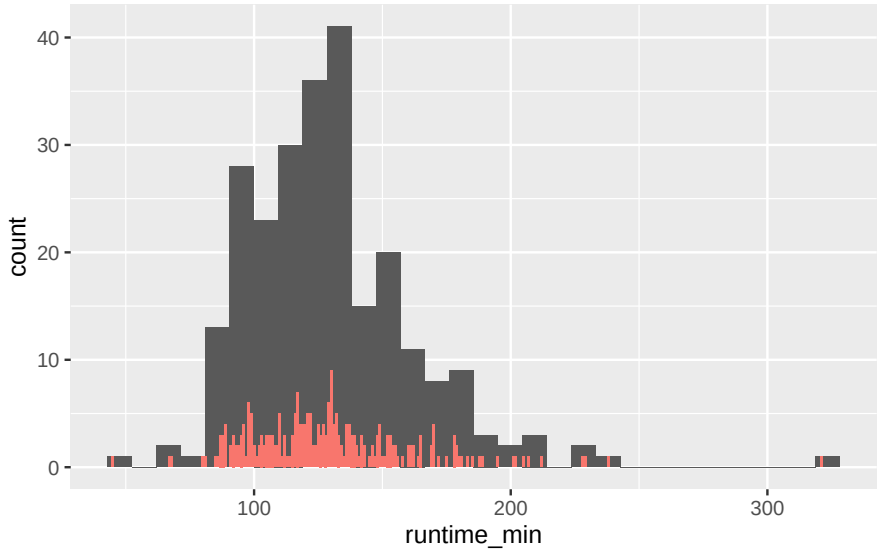
Table 8.1: (#tab:histogram-vs-bar-plot) Histogram vs bar plot

Histogram	Bar plot
Used to show the distribution of non-categorical data	Used for showing the frequency of categorical data, i.e. `factors`
Each bar (also called 'bin') represents a group of observations.	Each bar represents one category (or level) in our `factor`.
The order of the bars is important and cannot/should not be changed.	The order of bars is arbitrary and can be reordered if meaningful.

Let's overlap a bar plot with a histogram for the same variable to make this difference even more apparent.

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram() +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



There are a couple of important observations to be made:

- The bar plot has much shorter bars because each bar represents the frequency of a single unique score. Thus, the runtime of `151` is represented as a bar, as is the runtime of `152`. Only identical observations are grouped together.

As such, the bar plot is based on a frequency table, similar to what we computed before.

- In contrast, the histogram's 'bars' are higher because they group together individual observations based on a specified range, e.g. one bar might represent movies that have a runtime between 150-170 minutes. These ranges are called `bins`.

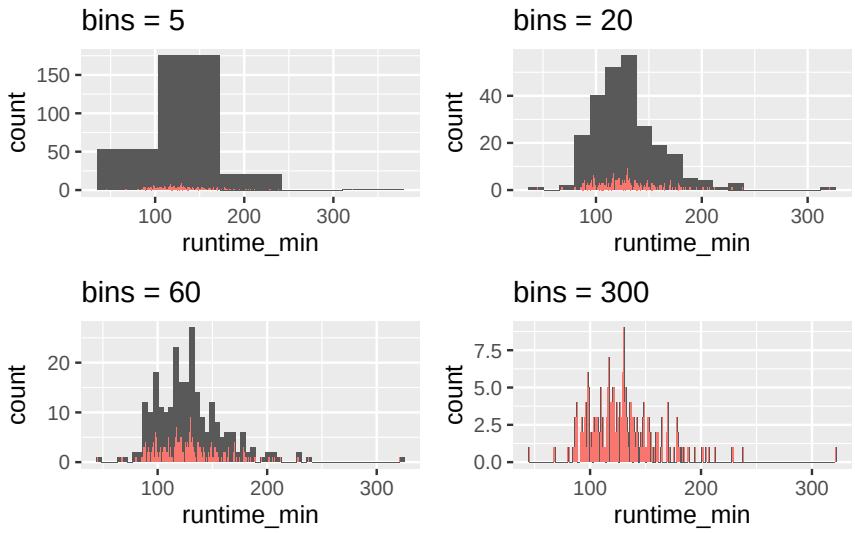
We can control the number of bars in our histogram using the `bins` attribute. `ggplot` even reminds us in a warning that we should adjust it to represent more details in the plot. Let's experiment with this setting to see how it would look like with different numbers of `bins`.

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 5) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
```

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 20) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
```

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 60) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
```

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 300) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
```



As becomes evident, if we select a large enough number of bins, we can achieve the same result as a bar plot. This is the closest a bar plot can become to a histogram, i.e. if you define the number of `bins` so that each observation is captured by one `bin`. While theoretically possible, practically, this rarely makes much sense.

We use histograms to judge whether our data is normally distributed. A normal distribution is often a requirement for assessing whether we can run certain types of analyses or not (for more details, see Chapter @ref(normality)). In short: It is imperative to know about it in advance. The shape of a normal distribution looks like a bell (see Figure @ref(fig:normal-distribution)). If our data is equal to a normal distribution, we find that

- the mean and the median are the same value and we can conclude that
- the mean is a good representation for our data/variable.

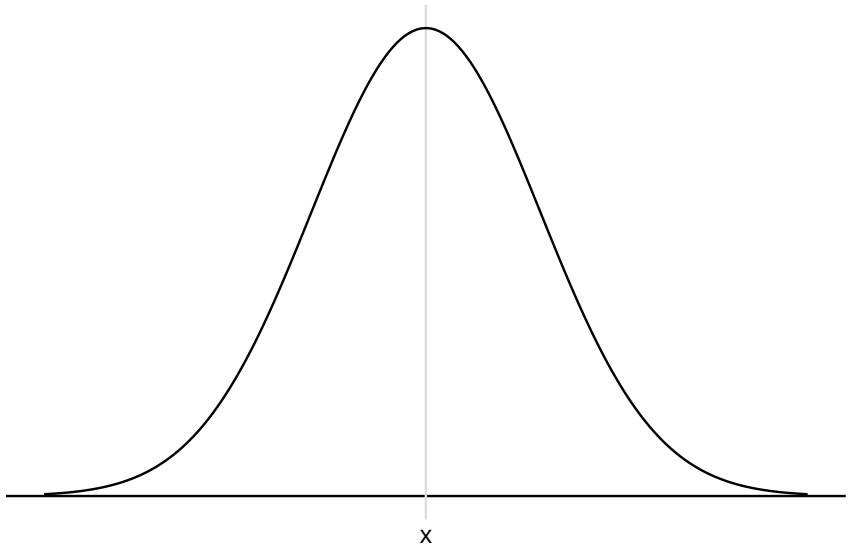


Figure 8.3: A normal distribution

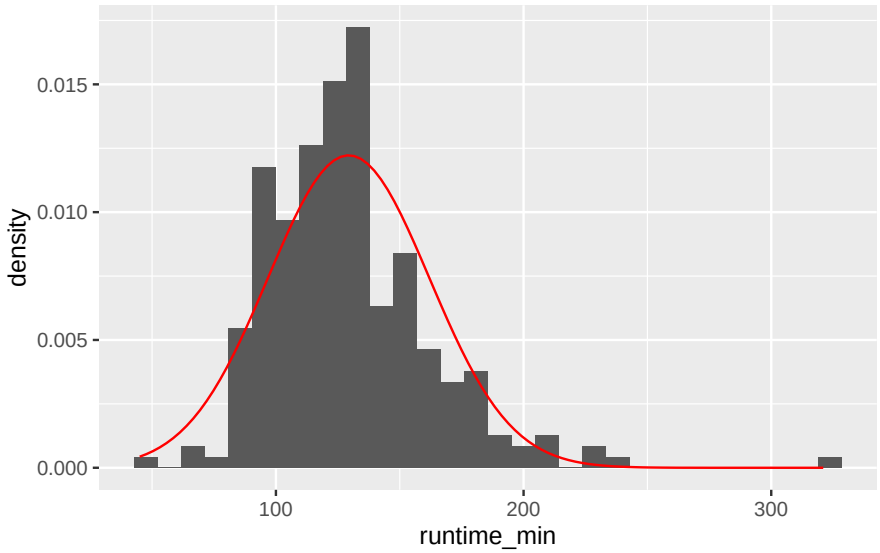
Let's see whether our data is normally distributed using the histogram we already plotted and overlay a normal distribution. The coding for the normal distribution is a little more advanced. Do not worry if you cannot fully decipher its meaning just yet. To draw such a reference plot, we need to:

- compute the `mean()` of our variable,
- calculate the standard deviation `sd()` of our variable (see Chapter @ref(standard-deviation)),
- use the function `geom_func()` to plot it and,
- define the function `fun` as `dnorm`, which stands for 'normal distribution'.

It is more important to understand what the aim of this task is, rather than fully comprehending the computational side in *R*: we try to compare our distribution with a normal one.

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(aes(y = ..density..), bins = 30) +

  # The following part creates a normal curve based on
  # the mean and standard deviation of our data
```

```
geom_function(fun = dnorm,
              n = 103,
              args = list(mean = mean(imdb_top_250$runtime_min),
                          sd = sd(imdb_top_250$runtime_min)),
              col = "red")
```



However, there are two problems if we use histograms in combination with a normal distribution reference plot: First, the y-axis needs to be transformed to fit the normal distribution (which is a density plot and not a histogram). Second, the shape of our histogram is affected by the number of `bins` we have chosen, which is an arbitrary choice we make. Besides, the lines of code might be tough to understand because we have to 'hack' the visualisation to make it work, i.e. using `aes(y = ..density..)`. There is, however, a better way to do this: Density plots.

8.4.3 Density plots: Your smooth histograms

Density plots are a special form of the histogram. It uses 'kernel smoothing', which turns our blocks into a smoother shape. Better than trying to explain what it does, it might help to see it. We use the same data but replace `geom_histogram()` with `geom_density()`. I also saved the plot with the normal distribution in a separate object, so we can easily reuse throughout this chapter.

```r
# Ingredients for our normality reference plot
mean_ref <- mean(imdb_top_250$runtime_min)
sd_ref <- sd(imdb_top_250$runtime_min)

# Create a plot with our reference normal distribution
n_plot <-
  imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_function(fun = dnorm,
                n = 103,
                args = list(mean = mean_ref,
                            sd = sd_ref),
                col = "red")

# Add our density plot
n_plot +
  geom_density()
```

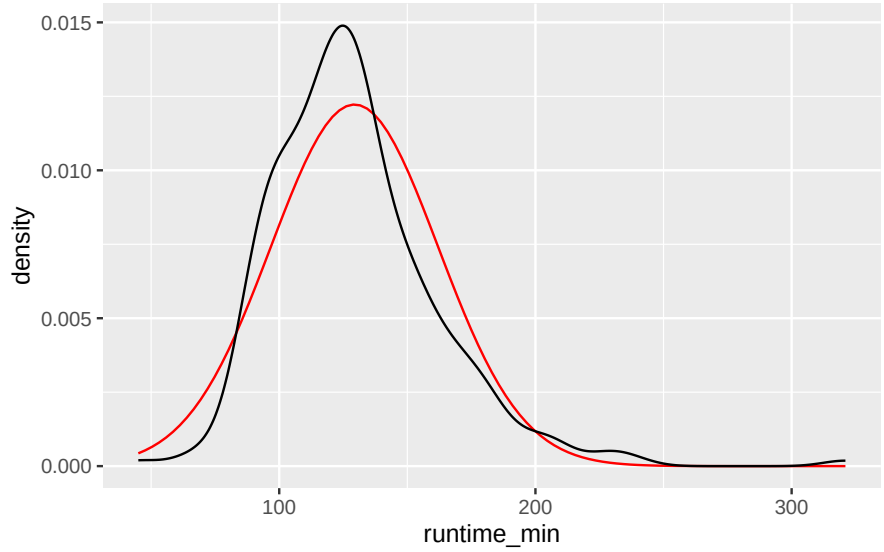
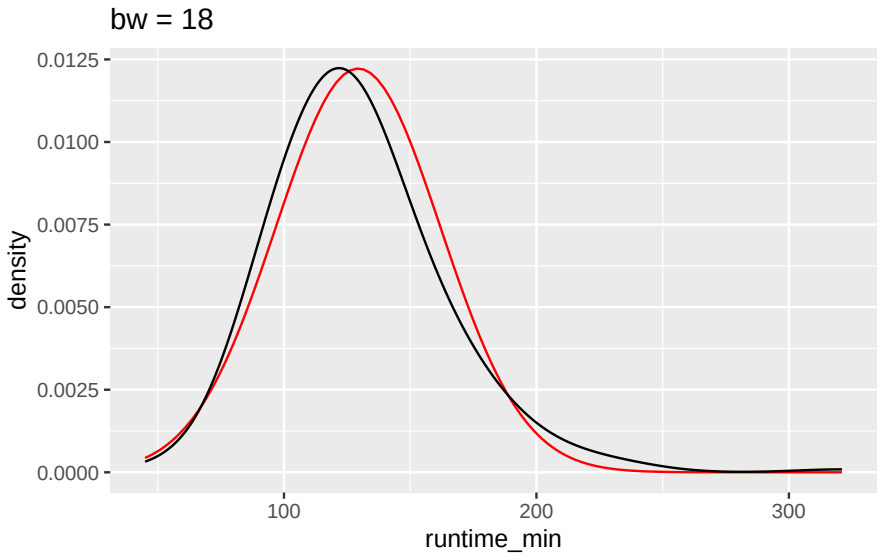


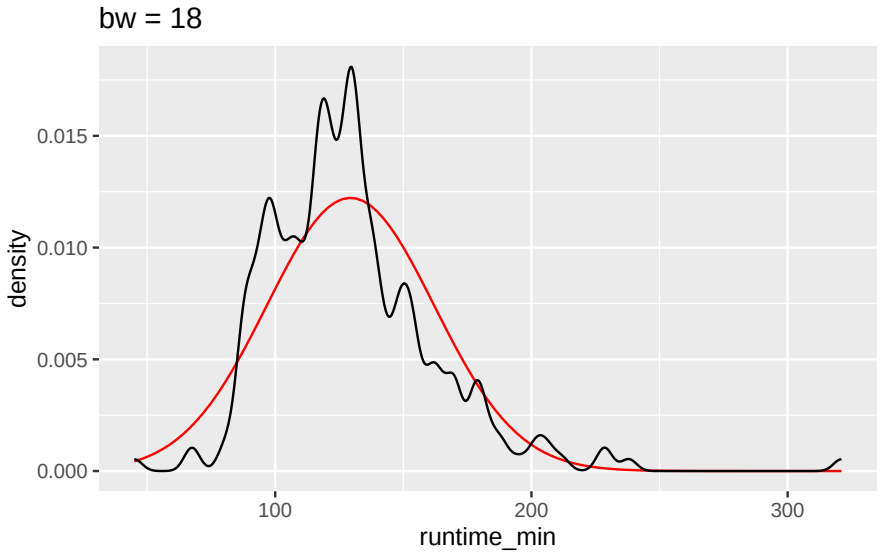
Figure 8.4: Density plot vs normal distribution

We can also define how big the `bins` are for density plots, which make the plot more or less smooth. After all, the density plot is a histogram, but the transitions from one bin to the next are 'smoothed'. Here is an example of how different `bw` settings affect the plot.

```
# bw = 18
n_plot +
  geom_density(bw = 18) +
  ggtitle("bw = 18")
```




```
# bw = 3
n_plot +
  geom_density(bw = 3) +
  ggtitle("bw = 18")
```

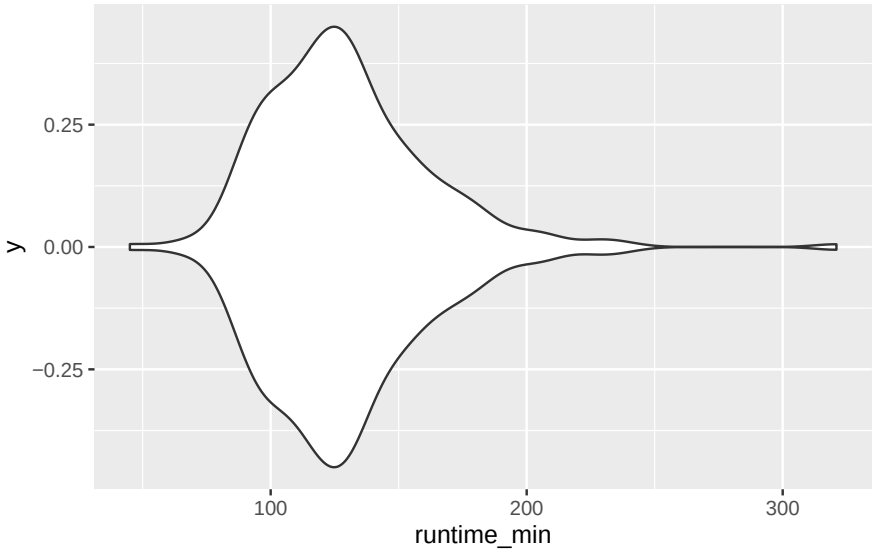


The benefits of the density plot in this situation are obvious: It is much easier to see whether our data is normally distributed or not when compared to a reference plot. However, we still might struggle to determine normality just by these plots because it depends on how high or low we set `bw`.

8.4.4 Violin plot: Your smooth boxplot

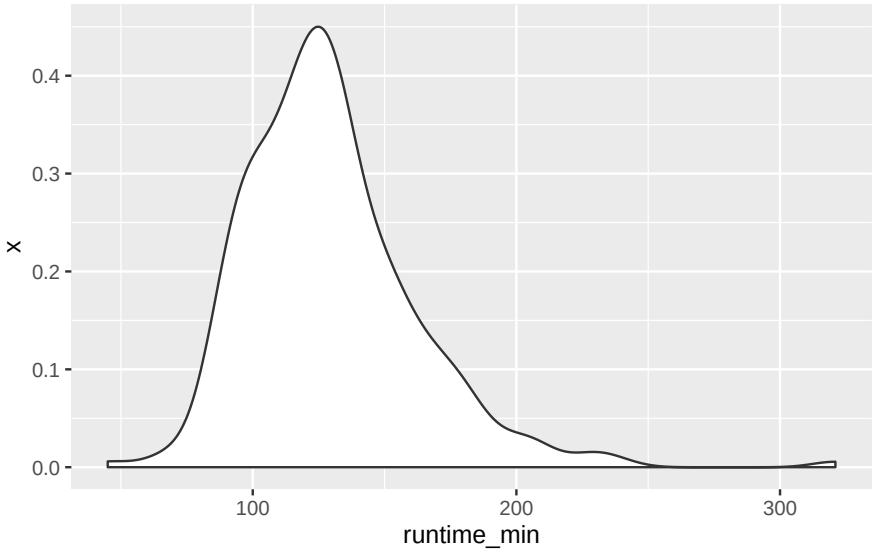
If a density plot is the sibling of a histogram, the violin plot would be the sibling of a boxplot, but the twin of a density plot. Confused? If so, then let's use the function `geom_volin()` to create one.

```
imdb_top_250 %>%
  ggplot(aes(x = runtime_min, y = 0)) +
  geom_violin()
```



Looking at our plot, it becomes evident where the violin plot got its name from, i.e. its shape. The reason why it is also a twin of the density plot becomes clear when we only plot half of the violin with `geom_violinghalf()` from the `see` package.

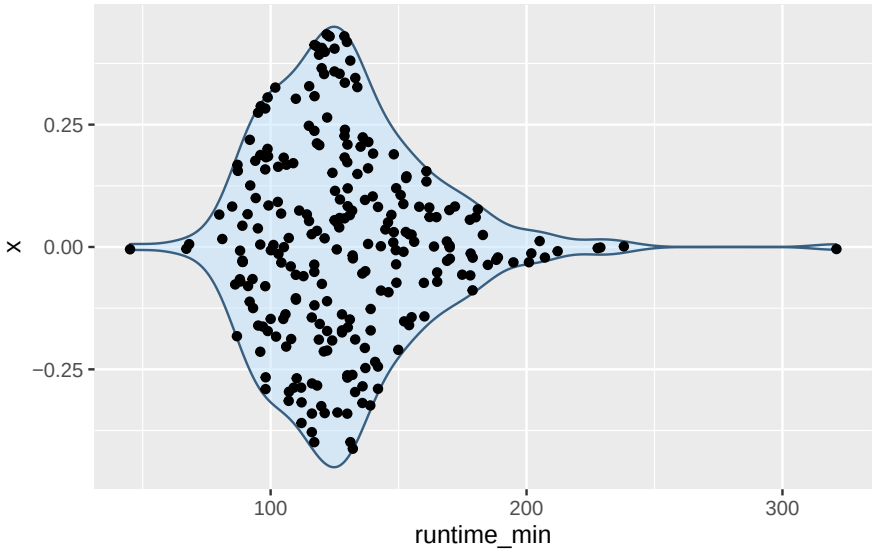
```
imdb_top_250 %>%
  ggplot(aes(x = 0, y = runtime_min)) +
  see::geom_violinhalf() +
  coord_flip()
```



It looks exactly like the density plot we plotted earlier (see Figure @ref(fig:density-vs-normal)). The interpretation largely remains the same to a density plot as well. The relationship between the boxplot and the violin plot lies in the symmetry of the violin plot.

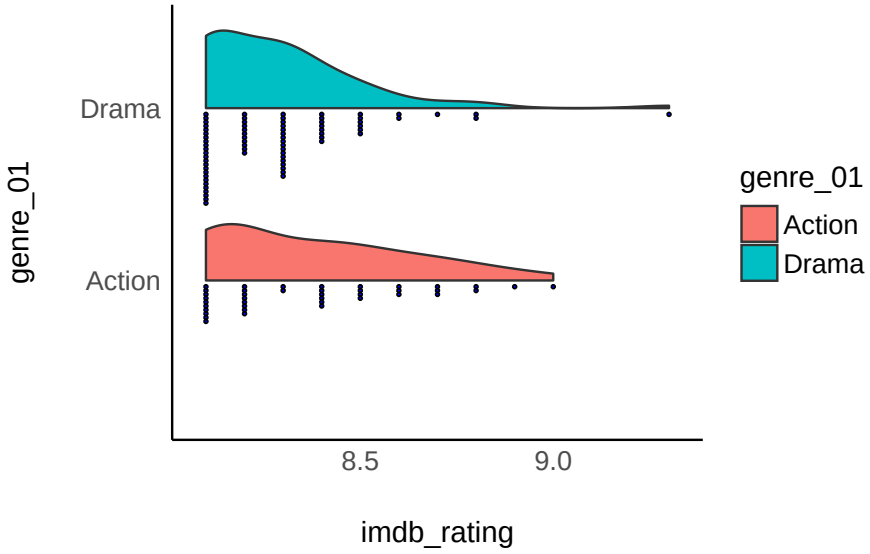
At this point, it is fair to say that we enter 'fashion' territory. It is really up to your taste which visualisation you prefer because they are largely similar but offer nuances that some data sets might require. Each visualisation, though, comes with caveats, which results in the emergence of new plot types. For example, *'sina'* plots which address the following problem: A boxplot by itself will never be able to show bimodal distributions like a density plot. On the other hand, a density plot can show observations that do not exist, because they are smoothed. The package `ggforce` enables us to draw a *'sina'* plot which combines a violin plot with a dot plot. Here is an example of overlaying a sina plot (black dots) and a violin plot (faded blue violin plot).

```
imdb_top_250 %>%
  ggplot(aes(y = runtime_min, x = 0)) +
  geom_violin(alpha = 0.5, col = "#395F80", fill = "#C0E2FF") +
  ggforce::geom_sina() +
  coord_flip()
```



Lastly, I cannot finish this chapter without sharing with you one of the most popular uses of half-violin plots: The rain cloud plot. It combines a dot plot with a density plot, and each dot represents a movie in our dataset. This creates the appearance of a cloud with raindrops. There are several packages available that can make such a plot. Here I used the `see` package.

```
imdb_top_250 %>%
  filter(genre_01 == "Action" | genre_01 == "Drama") %>%
  ggplot(aes(x = genre_01, y = imdb_rating, fill = genre_01)) +
  see::geom_violindot(fill_dots = "blue", size_dots = 0.2) +
  see::theme_modern() +
  coord_flip()
```



8.4.5 QQ plot: A 'cute' plot to check for normality in your data

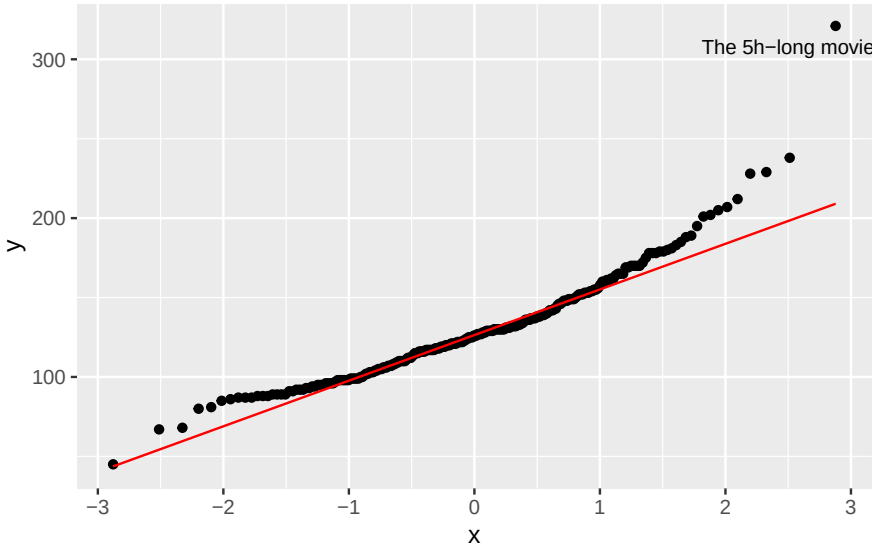
The QQ plot is an alternative to comparing distributions to a normality curve. Instead of a curve, we plot a line that represents our data's quantiles against the quantiles of a normal distribution. The term 'quantile' can be somewhat confusing, especially after learning about the boxplot, which shows quartiles. However, there is a relationship between these terms. Consider the following comparison:

- 1^{st} Quartile = 25^{th} percentile = 0.25 quantile
- 2^{nd} Quartile = 50^{th} percentile = 0.50 quantile = median
- 3^{rd} Quartile = 75^{th} percentile = 0.75 quantile

With these definitions out of the way, let's plot some quantiles against each other.

```
imdb_top_250 %>%
  ggplot(aes(sample = runtime_min)) +
  geom_qq() +
  geom_qq_line(col = "red") +
  annotate("text",
           label = "The 5h-long movie",
           x = 2.5,
```

```
          y = 308,
          size = 3)
```



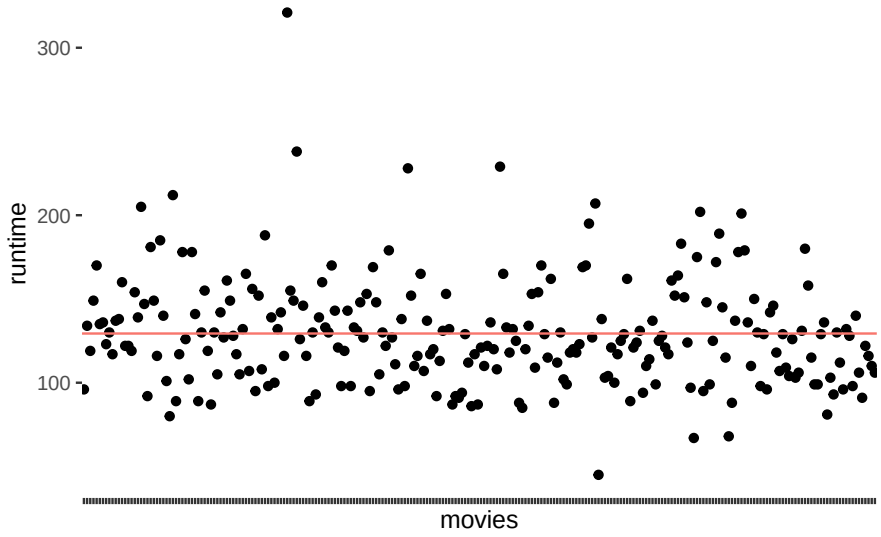
The function `geom_qq()` creates the dots, while `geom_qq_line` establishes a reference for a normal distribution. The reference line is drawn in such a way that it touches the quartiles of our distribution. Ideally, we would want that all dotes are firmly aligned with each other. Unfortunately, this is not the case in our dataset. At the top and the bottom, we have points that deviate quite far from a normal distribution. Remember the five-hour-long movie? It is very far away from the rest of the other movies in our dataset.

8.4.6 Standard deviation: Your average deviation from the mean

I left the most commonly reported statistics for the spread of data last. The main reason for this is that one might quickly jump ahead to look at the standard deviation without ever considering plotting the distribution of variables in the first place. Similar to the mean and other numeric indicators, they could potentially convey the wrong impression. Nevertheless, the standard deviation is an important measure.

To understand what the standard deviation is, we can consider the following visualisation:

```r
runtime_mean <- mean(imdb_top_250$runtime_min)

imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = title, y = runtime_min)) +
  geom_point() +
  geom_hline(aes(yintercept = runtime_mean, col = "red"), show.legend = FALSE) +

  # Making the plot a bit more pretty
  theme(axis.text.x = element_blank(),         # Removes movie titles
        panel.grid.major = element_blank(),    # Removes grid lines
        panel.background = element_blank()     # Turns background white
        ) +
  ylab("runtime") +
  xlab("movies")
```



The red line (created with `geom_hline()` represents the mean runtime for all movies in the dataset, which is 129 minutes. We notice that the points are falling above and below the mean, but not directly on it. In other words, there are not many movies that are about 129 minutes long. We make this visualisation even more meaningful if we sorted the movies by their runtime. We can also change the shape of the dots (see also Chapter @ref(correlations)) by using the attribute shape. This helps to plot many dots without having them overlap.

```
imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = reorder(title, runtime_min), y = runtime_min)) +
  geom_point(shape = 124) +
  geom_hline(aes(yintercept = runtime_mean, col = "red"), show.legend = FALSE) +

  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
        ) +
  ylab("runtime") +
  xlab("movies")
```

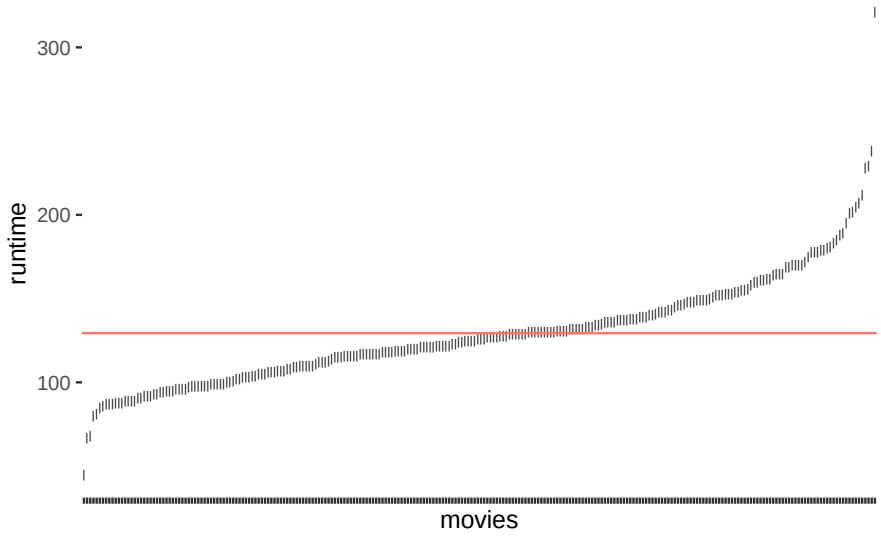


Figure 8.5: Deviation of `runtime_min` from the mean run time of movies.

As we can see, only a tiny fraction of movies are close to the mean. If we now consider the distance of each observation from the red line, we know how much each of them, i.e. each movie, deviates from the mean. The standard deviation tells us how much the runtime deviates on average. To be more specific, to compute the standard deviation by hand, you would:

- compute the difference between each observed value of `runtime_min` and the mean of `runtime_min` in your dataset, which is also called *'deviance'*, i.e. $deviance = runtime\_min - mean_{all\ movies}$

- square the deviance to turn all scores into positive ones, i.e. $deviance_{squared} = (deviance)^2$,
- then we take the sum of all deviations (also known as *'sum of squared errors'*) and divide it by the number of movies minus 1, which results in the *'variance'*, i.e. $variance = \frac{\sum(deviations_{squared})}{250-1}$. The variance reflects the average dispersion of our data.
- Lastly, we take the square root of this score to obtain the standard deviation, i.e. $sd = \sqrt{variance}$.

While the deviation and variance are interesting to look at, the standard deviation has the advantage that it provides us with the average deviation (also called *'error'*) based on the units of measurement of our variable. Thus, it is much easier to interpret its size.

We could compute this by hand if we wanted, but it is much simpler to use the function `sd()` to achieve the same.

```
sd(imdb_top_250$runtime_min)
```

```
[1] 32.63701
```

The result shows that movies tend to be about 32 minutes longer or shorter than the average movie. This seems quite long. However, we must be aware that this score is also influenced by the outliers we detected before. As such, if standard deviations in your data appear quite large, it can be due to outliers, and you should investigate further. Plotting your data will undoubtedly help to diagnose any outliers.

8.5 Packages to compute descriptive statistics

There is no one right way of how you can approach the computation of descriptive statistics. There are only differences concerning convenience and whether you have intentions to use the output from descriptive statistics to plot them or use them in other ways. Often, we only want to take a quick look at our data to understand it better. If this is the case, I would like to introduce you to two packages that are essential in computing descriptive statistics and constitute an excellent starting point to understanding your data:

- `psych`
- `skimr` (see also Chapter @ref(inspecting-raw-data))

8.5.1 The `psych` package for descriptive statistics

As its name indicates, the `psych` package is strongly influenced by how research is conducted in the field of Psychology. The package is useful in many respects, and we already used it in Chapter @ref(latent-constructs) to compute Cronbach's α.

Another useful function is `describe()`. We can use this function to compute summary statistics for a variable. For example, we might wish to inspect the descriptive statistics for `runtime_min`.

```
psych::describe(imdb_top_250$runtime_min)
```

```
   vars   n   mean    sd median trimmed   mad min max range skew kurtosis   se
X1    1 250 129.37 32.64    126  126.44 28.17  45 321   276 1.36     4.66 2.06
```

If we want to see descriptive statistics per group, we use `describeBy()`, which takes a factor as a second argument. So, for example, we could compute the descriptive statistics for `runtime_min` separately for each `genre_01`.

```
descriptives <- psych::describeBy(imdb_top_250$runtime_min,
                  imdb_top_250$genre_01)

# The output for the first three genres
descriptives[1:3]
```

```
$Action
   vars  n   mean   sd median trimmed   mad min max range skew kurtosis  se
X1    1 40 141.98 42.4  136.5  138.75 24.46  45 321   276  1.6     6.21 6.7

$Adventure
   vars  n   mean    sd median trimmed   mad min max range skew kurtosis   se
X1    1 20 141.85 39.09    142  138.56 39.29  89 228   139 0.46    -0.62 8.74

$Animation
   vars  n   mean    sd median trimmed  mad min max range skew kurtosis se
X1    1 24 102.96 14.67     99     102 12.6  81 134    53 0.57    -0.85  3
```

The advantage of using `psych` is the convenience of retrieving several descriptive statistics with just one function instead of chaining together several functions to achieve the same.

8.5.2 The `skimr` package for descriptive statistics

While the `psych` package returns descriptive statistics for a single numeric variable, `skimr` takes this idea further and allows us to generate descriptive

statistics for all variables of all types in a data frame with just one function, i.e. `skim()`. We already covered this package in Chapter @ref(inspecting-raw-data), but here is a brief reminder of its use for descriptive statistics.

```
skimr::skim(imdb_top_250)
```

```
> skimr::skim(imdb_top_250)
── Data Summary ────────────────────
                           Values
Name                       imdb_top_250
Number of rows             250
Number of columns          16
_______________________
Column type frequency:
  character                2
  factor                   7
  numeric                  7
________________________
Group variables            None

── Variable type: character ──────────────────────────────────────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate   min   max empty n_unique whitespace
1 title                 0             1     1    68     0      250          0
2 synopsis              0             1    50   298     0      250          0

── Variable type: factor ─────────────────────────────────────────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate ordered n_unique top_counts
1 director              0             1 FALSE        155 Chr: 7, Mar: 7, Sta: 7, Aki: 6
2 star_01               0             1 FALSE        185 Rob: 6, Cha: 5, Leo: 5, Tom: 5
3 star_02               0             1 FALSE        230 Har: 3, Mat: 3, Rob: 3, Ale: 2
4 star_03               0             1 FALSE        239 Car: 3, Joe: 3, Mar: 3, Geo: 2
5 star_04               0             1 FALSE        241 Bil: 2, Chr: 2, Dia: 2, Mic: 2
6 genre                 0             1 FALSE        105 Dra: 21, Cri: 13, Cri: 11, Bio: 10
7 genre_01              0             1 FALSE         11 Dra: 72, Act: 40, Cri: 38, Ani: 24

── Variable type: numeric ────────────────────────────────────────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate      mean         sd       p0       p25       p50      p75      p100 hist
1 rank                  0             1       126.       72.3       1       63.2     126.      188.      250 ▇▇▇▇▇
2 year                  0             1      1987.       25.2    1921      1966.     1994      2007      2021 ▁▂▃▆▇
3 runtime_min           0             1       129.       32.6      45       107       126       146.      321 ▂▇▂▁▁
4 imdb_rating           0             1         8.31      0.224     8.1       8.1       8.25      8.4       9.3 ▇▂▁▁▁
5 metascore            34         0.864      82.6       10.9      55        75        84        90       100 ▁▃▅▇▇
6 votes                 0             1    556212.   480414.    26562    163057.   411843    858892.  2421026 ▇▃▂▁▁
7 gross_in_m           35         0.86       87.4      128.        0.01      5.32     32        127.      858. ▇▁▁▁▁
```

In addition to providing descriptive statistics, `skim()` sorts the variables by data type and provides different descriptive statistics meaningful to each kind. Apart from that, the column `n_missing` can help spot missing data. Lastly, we also find a histogram in the last column for `numeric` data, which I find particularly useful.

When just starting to work on a new dataset, the `psych` and the `skimr` package are solid starting points to get an overview of the main characteristics of your data, especially in larger datasets. On your *R* journey, you will encounter many other packages which provide useful functions like these.

9

Sources of bias: Outliers, normality and other 'conundrums'

'*Bias'* in your analysis is hardly ever a good thing unless you are a qualitative researcher. Whether you consider it positive or negative, we have to be aware of issues preventing us from performing a specific type of analysis. All of the statistical computations discussed in the following chapters can easily be affected by different sources of bias. The lack of certain biases can be an assumption of particular statistical tests. Thus, violating these assumptions would imply that the analytical technique we use will produce wrong results, i.e. biased results. Field (2013) summarises three main assumptions we have to consider:

- Linearity and additivity,
- Independence,
- Normality, and
- Homogeneity of variance, i.e. homoscedasticity.

Most *parametric tests* require that all assumptions are met. If this is not the case, we have to use alternative approaches, i.e. *non-parametric tests*. The distinction is essential since parametric and non-parametric tests are based on different computational methods, leading to different outcomes.

Lastly, Field (2013) also mentions outliers as an important source of bias. Irrespective of whether your data fulfils the assumptions for parametric tests, outliers tend to be a significant problem. They usually lead to wrong results and, consequently, to misinterpretations of our findings. We will also cover how we can identify and handle such outliers in our data at the end of this chapter.

9.1 Linearity and additivity

The assumption of linearity postulates that the relationship of variables represents a straight line and not a curve or any other shape. Figure @ref(fig:linear-

nonlinear-relationships) depicts examples of how two variables could be related to each other. Only the first one demonstrates a linear relationship, and all other plots would represent a violation of linearity.

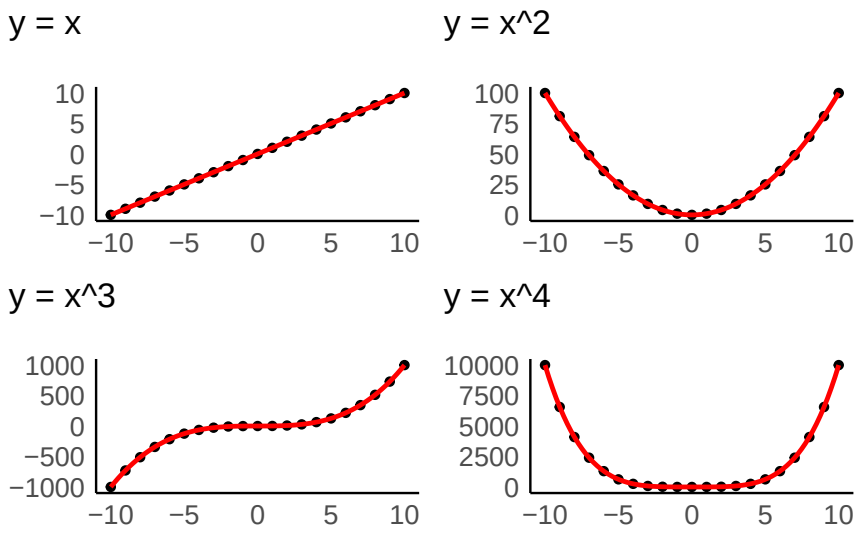


Figure 9.1: Examples of linear and non-linear relationships of two variables

Data visualisations are particularly useful to identify whether variables are related to each other in a linear fashion. The examples above were all created with `geom_point()`, which creates a dot plot that maps the relationship between two variables. In Chapter @ref(correlations), we will look more closely at the relationship of two variables in the form of *correlations*, which measure the strength of a linear relationship between variables.

Additivity is given when the effects of all independent variables can be added up to obtain the total effect they have on a dependent variable. In other words, the effect that multiple variables have on another variable can be added up to reflect their total effect.

If we assume that we have a dependent variable Y which is affected by other (independent) variables X, we could summarise additivity and linearity as a formula:

$$Y = \beta_1 * X_1 + \beta_2 * X_2 + ... + \beta_n * X_n$$

The β (beta) stands for the degree of change in a variable X causes in Y. Or, in simpler terms, β reflects the impact an independent variable has on the dependent variable. We will return to this equation in Chapter @ref(regression), where we create a linear model via regression.

9.2 Independence

The notion of independence is an important one. It assumes that each observation in our dataset is independent of other observations. For example, imagine my wife Fiona and I take part in a study that asks us to rank movies by how much we like them. Each of us has to complete the ranking by ourselves, but since we both sit in the same living room, we start chatting about these movies. By doing so, we influence each other's rankings and might even agree on the same ranking. Thus, our scores are not independent from each other. On the other hand, suppose we were both sitting in our respective offices and rank these movies. In that case, the rankings could potentially still be very similar, but this time the observations are independent of each other.

There is no statistical measurement or plot that can tell us whether observations are independent or not. Ensuring independence is a matter of data collection and not data analysis. Thus, it depends on how you, for example, designed your experiment, or when and where you ask participants to complete a survey, etc. Still, this criterion should not be downplayed as being a 'soft' one, just because there is no statistical test, but should remain on your radar throughout the planning and data collection stage of your research.

9.3 Normality

We touched upon the notion of *'normality'* and *'normal distributions'* before in Chapter @ref(spread-of-data) because it refers to the spread of our data. Figure @ref(fig:normal-distribution2) should look familiar by now.

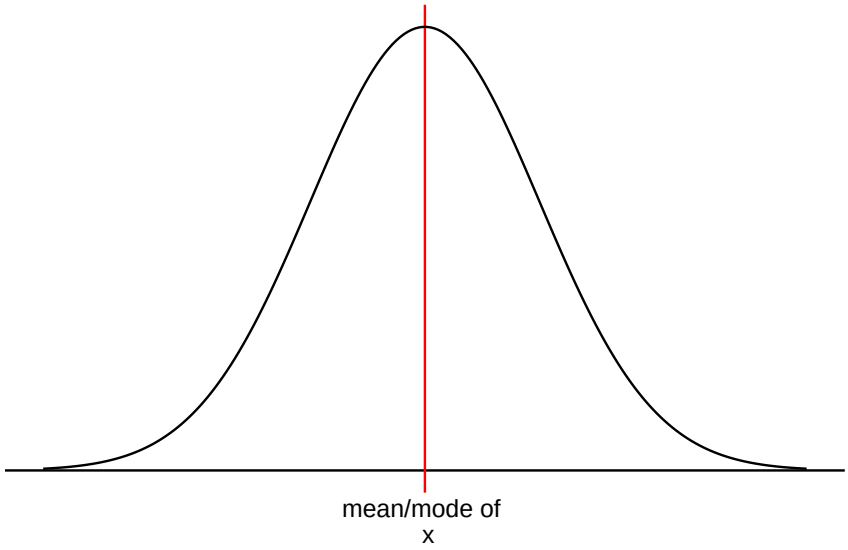
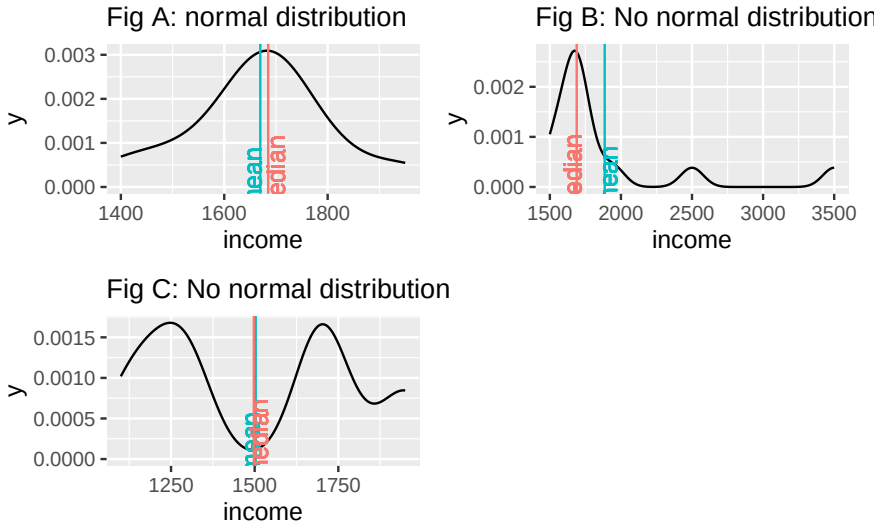


Figure 9.2: A normal distribution

However, we have yet to understand why it is essential that our data follows a normal distribution. Most parametric tests are based on means. For example, if we want to compare two groups with each other, we would compute the mean for each of them and then see whether their means differ from each other in a significant way (see Chapter @ref(comparing-groups). Of course, if our data is not very normally distributed, means are a poor reference point for most of the observations in this group. We already know that outliers heavily affect means, but even without outliers, the mean could be a poor choice. Let me provide some visual examples.



We notice that neither the median nor the mean by themselves is a reliable indicator for normality. Figure A and Figure C both show that the median and mean are almost identical, but only Figure A follows a normal distribution. The median and mean in Figure C are not reflective of the average observation in this dataset. Most scores lie below and above the mean/median. Therefore, when we analyse the normality of our data, we usually are not interested in the normality of a single variable but the normality of the sampling distribution. However, we cannot directly assess the sampling distribution in most cases. As such, we often revert to testing the normality of our data. There are also instances where we would not expect a normal distribution to exist. Consider the following plots:

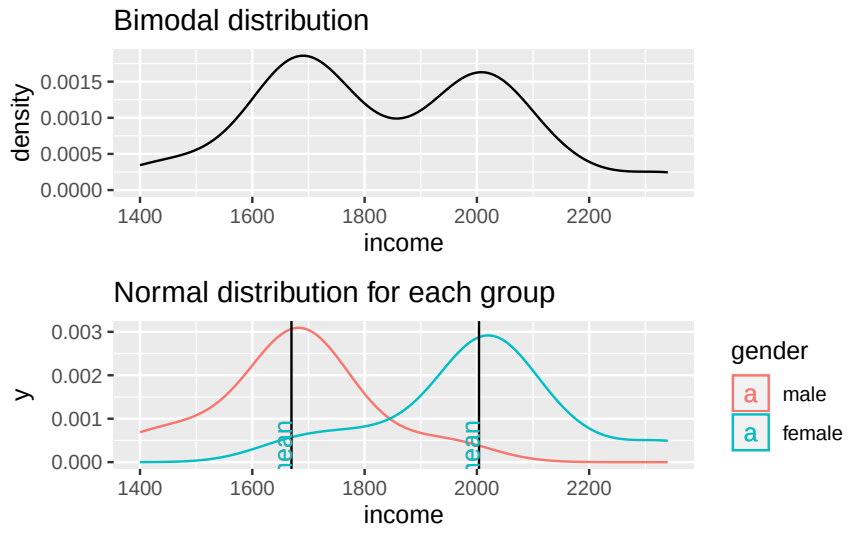


Figure 9.3: Two normal distributions in one dataset.

The first plot clearly shows that data is not normally distributed. If anything, it looks more like the back of a camel. The technical term for this distribution is called *'bimodal distribution'*. In cases where our data has even more peaks we consider it as a *'multimodal distribution'*. If we identify distributions that look remotely like this, we can assume that there must be another variable that helps explain why there are two peaks in our distribution. The plot below reveals that gender appears to play an important role. Drawing the distribution for each subset of our data reveals that `income` is now normally distributed for each group and has two different means. Thus, solely focusing on normal distributions for a single variable would not be meaningful if you do not consider the impact of other variables. If we think that `gender` plays an important role to understand `income` levels, we would have to expect that the distribution is not normal when looking at our data in its entirety.

Determining whether data is normally distributed can be challenging when only inspecting plots, e.g. histograms, density plots or QQ plots. Luckily, there is also a statistical method to test whether our data is normally distributed: *The Shapiro-Wilk test*. This test compares our distribution with a normal distribution (like in our plot) and tells us whether our distribution is **significantly different** from it. Thus, if the test is not significant, the distribution of our data is not significantly different from a normal distribution, or in simple terms: It is normally distributed. We can run the test in R as follows for the dataset that underpins Fig A above:

```
shapiro.test(data$income)
```

```
    Shapiro-Wilk normality test

data:  data$income
W = 0.94586, p-value = 0.5916
```

This result confirms that the data is normally distributed, because it is not significantly different ($p > 0.05$). The chapter on correlations looks at significance and its meaning more thoroughly (see Chapter @ref(correlations)).

In conclusion, the normality of data is an essential pre-test for any of our studies. If we violate the assumption of normality, we will have to fall back to non-parametric tests. However, this rule has two exceptions: *The Central Limit Theorem* and using *'robust'* measures of parametric tests.The Central Limit Theorem postulates that as our sample becomes larger, our sampling distribution becomes more and more normal around the mean of the underlying population. For example, Field (2013) (p.54) refers to a sample of 30 as a common rule of thumb. As such, it is possible to assume normality for larger datasets even though our visualisation and the Shapiro-Wilk test tell us otherwise. The second exception is that many parametric tests offer a 'robust' alternative, often via *bootstrapping*. *Bootstrapping* refers to the process of subsampling your data, for example, 2000 times, and look at the average outcome. An example of bootstrapping is shown in Chapter @ref(bootstrapped-regression).

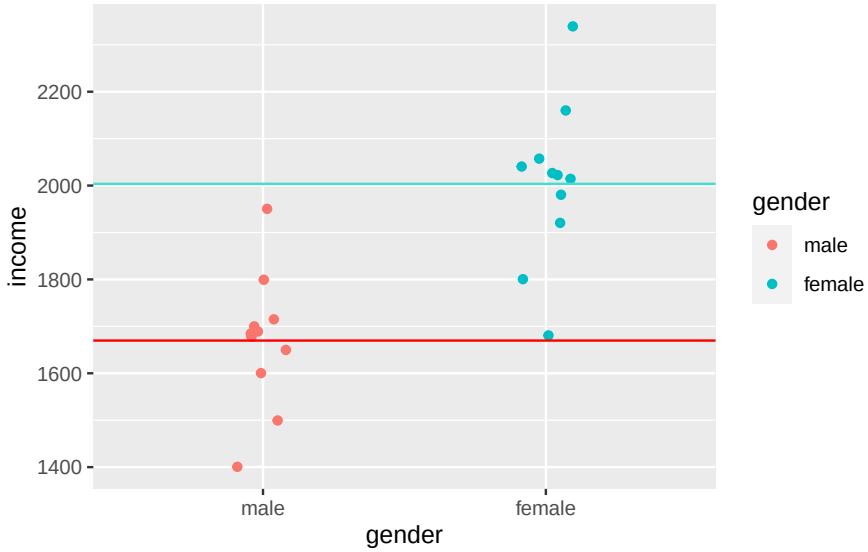
Admittedly, this raises the question: Can I ignore normality and move on with my analysis if my sample is large enough or I use bootstrapping? In short: Yes. This fact probably also explains why we hardly ever find results from normality tests in journal publications since most Social Science research involves more than 30 participants. However, if you find yourself in a situation where the sample size is smaller, all of the above needs to be check and thoroughly considered. However, the sample size also has implications with regards to the power of your test/findings (see Chapter @ref(power-analysis)). Ultimately, the answer to the above questions remains, unsatisfyingly: 'It depends'.

9.4 Homogeneity of variance (homoscedasticity)

The term *'variance'* should sound familiar, because we mentioned it in Chapter @ref(standard-deviation) where we looked at the standard deviation derived from the variance.

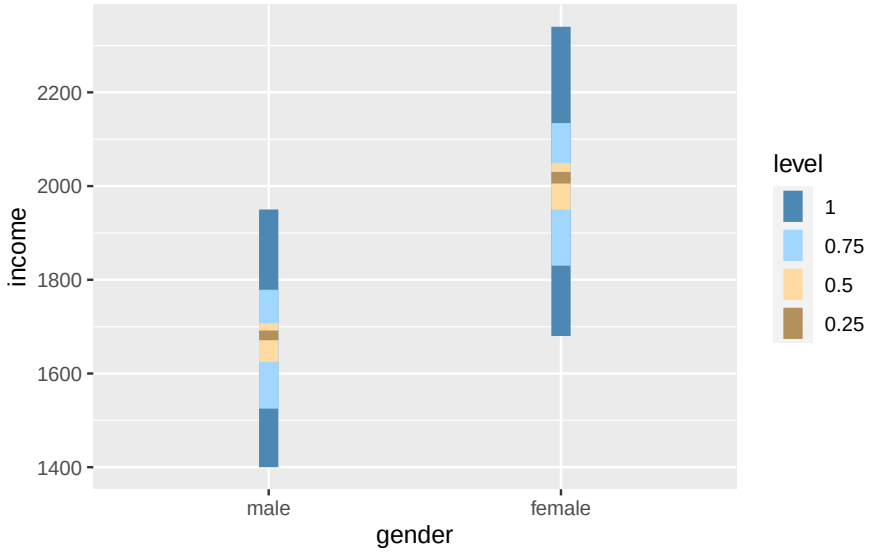
Homogeneity of variance implies that the variance of, for example, two subsets of data, is equal or close to being equal. Let's look at how close the observed values are to the mean for the two groups identified in Figure @ref(fig:two-normalities-groups).

```
data4 %>%
  ggplot(aes(x = gender, y = income, col = gender)) +
  geom_jitter(width = 0.1) +
  geom_hline(yintercept = group_means$mean[1], color = "red") +
  geom_hline(yintercept = group_means$mean[2], color = "turquoise")
```



Judging by eye, we could argue that most values lie around the mean for each respective group. However, some observations are a bit further off. Still, using this visualisation, it is tough to judge whether the spread is about the same. However, boxplots can help with this, or even better, a boxplot reflected by a bar. The package `ggdist` has an excellent plotting function called `stat_interval()`, which allows us to show a boxplot in the form of a bar.

```
data4 %>%
  ggplot(aes(x = gender, y = income, group = gender)) +
  ggdist::stat_interval(aes(y = income),
                        .width = c(0.25, 0.5, 0.75, 1)) +

  # Let's add some nice complementary colours
  scale_color_manual(values = c("#4D87B3", "#A1D6FF", "#FFDAA1", "#B3915F"))
```



If we compare the bars, we can tell that the variance in both groups looks very similar, i.e. the length of the bars appear to be about the same height. Furthermore, if we compare the IQR for both groups, we find that they are quite close to each other.

```
data4 %>%
  group_by(gender) %>%
  summarise(iqr = IQR(income))
```

```
# A tibble: 2 x 2
  gender   iqr
  <fct>  <dbl>
1 male    82.5
2 female  99
```

However, to test whether the variance between these two groups is truly similar or different, we have to perform a *Levene's test*. The Levene's test follows a similar logic as the Shapiro-Wilk test. If the test is significant, i.e. $p < 0.05$, we have to assume that the variances between these groups are significantly different from each other. However, if the test is not significant, then the variances are similar, and we can proceed with a parametric test - assuming other assumptions are not violated. The interpretation of this test follows the one for the Shapiro-Wilk test. To compute the Levene's test we can use the function `leveneTest()` from the `car` package. However, instead of using `,` to separate the two variables, this function expects a formula, which is denotated by a ~. We will cover formulas in the coming chapters.

```
car::leveneTest(gep$age ~ gep$gender)
```

```
Levene's Test for Homogeneity of Variance (center = median)
       Df F value Pr(>F)
group   2  0.8834 0.4144
      297
```

The Levene's test shows that our variances are similar and not different from each other because $p > 0.05$. This is good news if we wanted to continue and perform a group comparison, like in Chapter @ref(comparing-groups).

9.5 Outliers and how to deal with them

In Chapter @ref(descriptive-statistics), I referred to outliers many times but never eluded to the aspects of handling them. Dealing with outliers is similar to dealing with missing data. However, it is not quite as straightforward as one might think.

In a first step, we need to determine which values count as an outlier. Aguinis, Gottfredson, and Joo (2013) reviewed 232 journal articles and found that scholars had defined outliers in 14 different ways, used 39 different techniques to detect them and applied 20 different strategies to handle them. It would be impossible to work through all these options in this book. However, I want to offer two options that have been frequently considered in publications in the field of Social Sciences:

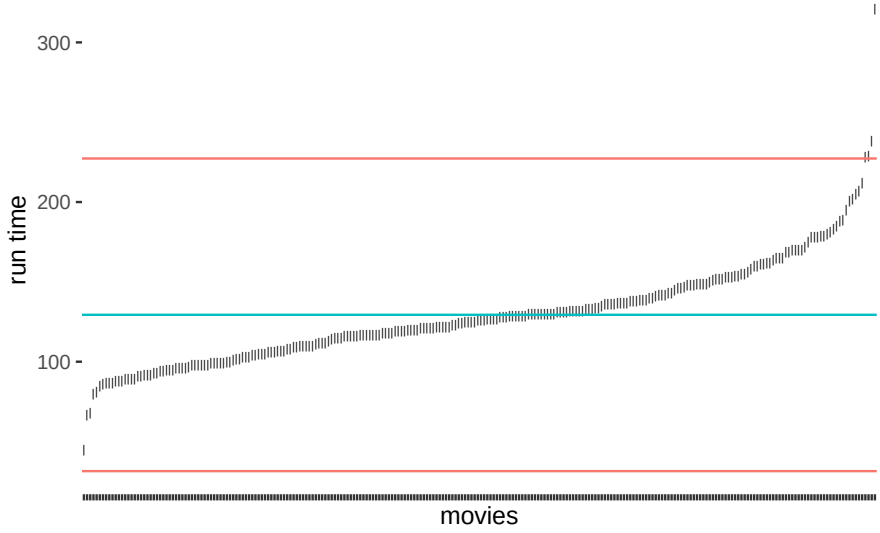
- The standard deviation (SD), and
- The inter-quartile range (IQR).

9.5.1 Detecting outliers using the standard deviation

A very frequently used approach to detecting outliers is the use of the standard deviation. Usually, scholars use multiples of the standard deviation to determine thresholds. For example, a value that lies 3 standard deviations above or below the mean could be categorised as an outlier. Unfortunately, there is quite some variability regarding how many multiples of the standard deviation counts as an outlier. Some authors might use `3`, and others might settle for `2` (see also Leys et al. (2013)). Let's stick with the definition of `3` standard deviations to get us started. We can revisit our previous plot regarding `runtime_min` (see Figure @ref(fig:runtime-movies-deviation)) and add lines that show the thresholds above and below the mean. As before, I will create a base plot `outlier_plot` first so that we do not have to repeat the same code

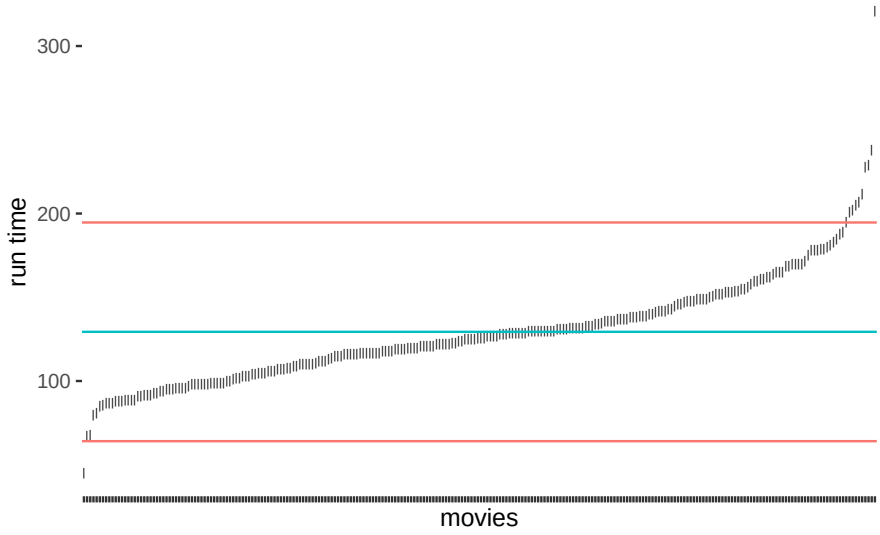
over and over again. We then use `outlier_plot` and add more layers as we see fit.

```
# Compute the mean and the thresholds
runtime_mean <- mean(imdb_top_250$runtime_min)
sd_upper <- runtime_mean + 3 * sd(imdb_top_250$runtime_min)
sd_lower <- runtime_mean - 3 * sd(imdb_top_250$runtime_min)

# Create the base plot
outlier_plot <-
  imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = reorder(title, runtime_min),
             y = runtime_min)) +
  geom_point(shape = 124) +

  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
        ) +
  ylab("run time") +
  xlab("movies")

# Add the thresholds and mean
outlier_plot +
  geom_hline(aes(yintercept = runtime_mean, col = "red"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_upper, col = "blue"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_lower, col = "blue"),
             show.legend = FALSE)
```



The results suggest that only very few outliers would be detected if we chose these thresholds. Especially `Sherlock Jr.`, the shortest movie in our dataset, would not classify as an outlier. How about we choose 2 standard deviations instead?

```
# Compute the mean and standard deviation
runtime_mean <- mean(imdb_top_250$runtime_min)
sd_upper <- runtime_mean + 2 * sd(imdb_top_250$runtime_min)
sd_lower <- runtime_mean - 2 * sd(imdb_top_250$runtime_min)

# Create our plot
outlier_plot +
  geom_hline(aes(yintercept = runtime_mean, col = "red"),
               show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_upper, col = "blue"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_lower, col = "blue"),
             show.legend = FALSE)
```



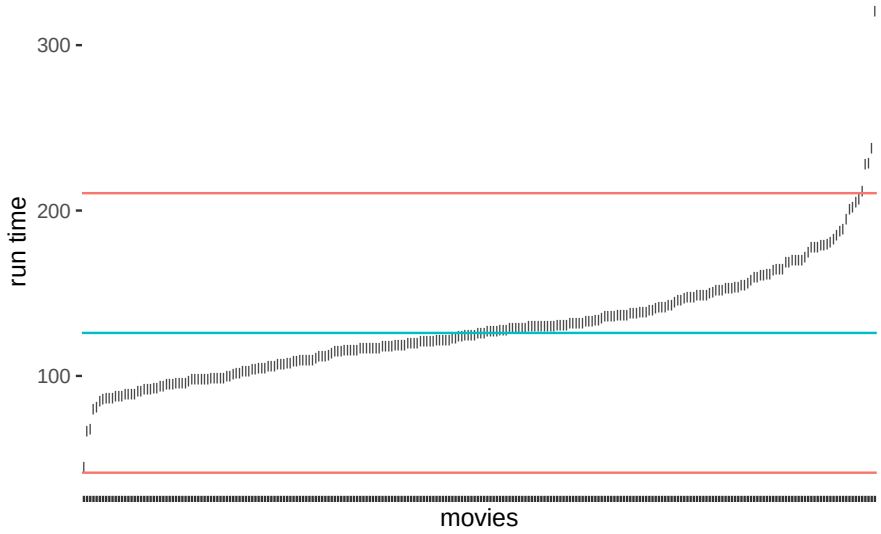
As we would expect, we identify some more movies as being outliers, but the shortest movie would still not be classified as an outlier. It certainly feels somewhat arbitrary to choose a threshold of our liking. Despite its popularity, there are additional problems with this approach:

- outliers affect our mean and standard deviation too,
- since we use the mean, we assume that our data is normally distributed, and
- in smaller samples, this approach might result in not identifying outliers at all (despite their presence) (Leys et al. 2013) (p. 764).

Leys et al. (2013) propose an alternative approach because medians are much less vulnerable to outliers than the mean. Similarly to the standard deviation, it is possible to calculate thresholds using the *median absolute deviation* (MAD). Best of all, the function `mad()` in *R* does this automatically for us. Leys et al. (2013) suggest using `2.5` times the MAD as a threshold. However, if we want to compare how well this option performs against the standard deviation, we could use `3` as well.

```
# Compute the median and thresholds
runtime_median <- median(imdb_top_250$runtime_min)
mad_upper <- runtime_median + 3 * mad(imdb_top_250$runtime_min)
mad_lower <- runtime_median - 3 * mad(imdb_top_250$runtime_min)

# Create our plot
outlier_plot +
  geom_hline(aes(yintercept = runtime_median, col = "red"),
```

```
            show.legend = FALSE) +
  geom_hline(aes(yintercept = mad_upper, col = "blue"),
            show.legend = FALSE) +
  geom_hline(aes(yintercept = mad_lower, col = "blue"),
            show.legend = FALSE)
```



Compared to our previous results, we notice that the median approach was much better in detecting outliers at the upper range of `runtim_min`. Because the median is not affected so much by the five-hour-long movie, the results have improved. Still, we would not classify the outlier at the bottom for the shortest film in the data. If we chose the criterion of `2.5 * MAD`, we would also get this outlier (see Figure @ref(fig:MAD2-outlier-detection)).

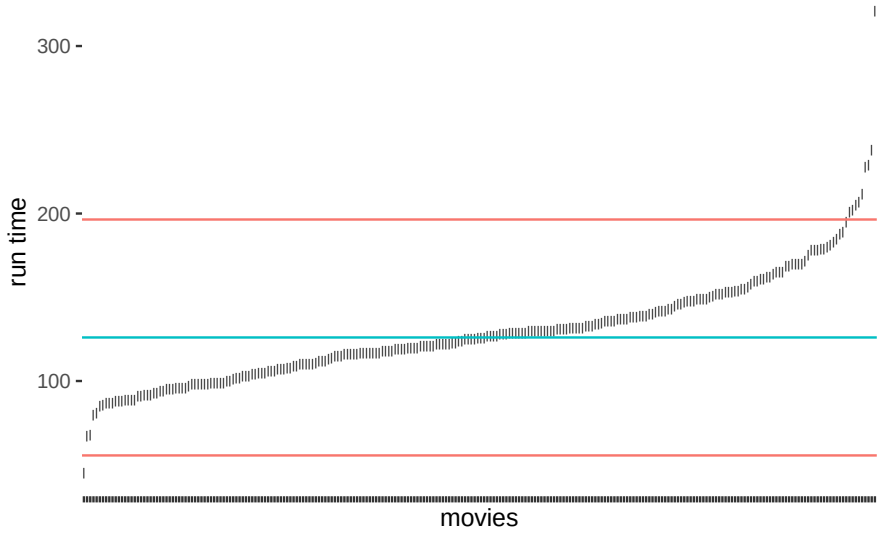


Figure 9.4: Outlier detection via MAD using `2.5 * MAD` as a threshold

Which approach to choose very much depends on the nature of your data. For example, one consideration could be that if the median and the mean of a variable are dissimilar, choosing the median might be the better option. However, we should not forget that the outliers we need to identify will affect the mean. Hence, it appears that in many cases, the median could be the better choice. Luckily, there is yet another highly popular method based on two quartiles (rather than one, i.e. the median).

9.5.2 Detecting outliers using the interquartile range (IQR)

Another approach to classifying outliers is the use of the *interquartile range (IQR)*. The IQR is used in boxplots and creates the dots at its ends to indicate any outliers. This approach is straightforward to implement because the computation of the IQR is simple:

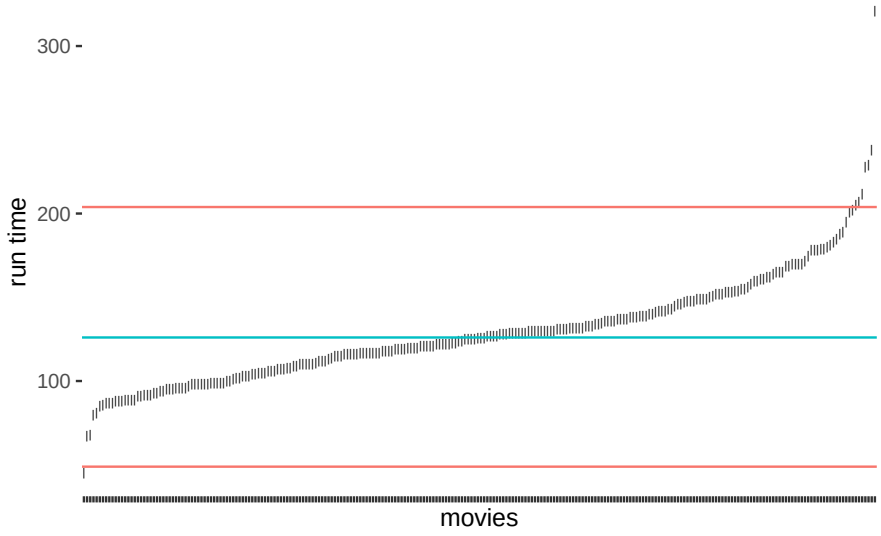
$$IQR = Q_3 - Q_1$$

Therefore, we can create new thresholds for the detection of outliers. For the IQR, it is common to use $\pm 1.5 * IQR$ as the lower and upper thresholds measured from Q_1 and Q_3 respectively. To compute the quartiles we can use the function `quantile()`, which returns $Minimum, Q_1, Q_2, Q_3, and\ Q_4$

```
# Compute the quartiles
(runtime_quantiles <- quantile(imdb_top_250$runtime_min))
```

```
    0%    25%    50%    75%   100%
 45.00 107.00 126.00 145.75 321.00
```

```
# Compute the thresholds
iqr_upper <- runtime_quantiles[4] + 1.5 * IQR(imdb_top_250$runtime_min)
iqr_lower <- runtime_quantiles[2] - 1.5 * IQR(imdb_top_250$runtime_min)

# Create our plot
outlier_plot +
  geom_hline(aes(yintercept = runtime_median, col = "red"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = iqr_upper, col = "blue"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = iqr_lower, col = "blue"),
             show.legend = FALSE)
```



As we can tell, the IQR method detects about the same outliers for our data as the median absolute deviation (MAD) approach. The outliers we find here are the same as shown in Figure @ref(fig:a-boxplot). For our data, I would argue that the IQR and MAD method by Leys et al. (2013) produced the 'best' selection of outliers. However, we have to acknowledge that these classifications will always be subjective because how we position the thresholds depends on the researcher's choice. Still, the computation is standardised, and we can plausibly explain the process of identifying outliers, we just have to make it explicit to the audience of our research.

9.5.3 Removing or replacing outliers

Now that we have identified our outliers, we are confronted with the question of what we should do with them. Like missing data (see Chapter @ref(dealing-with-missing-data)), we can either remove them or replace them with other values. While removal is a relatively simple task, replacing it with other 'reasonable' values implies finding techniques to create such values. As you may remember, we were confronted with a similar problem before when we looked into missing data (Chapter @ref(dealing-with-missing-data)). The same techniques, especially multiple imputation (see Cousineau and Chartier 2010), can be used for such scenarios.

Irrespective of whether we remove or replace outliers, we somehow need to single them out of the crowd. Since the IQR strategy worked well for our data, we can use the thresholds we defined before, i.e. `iqr_upper` and `iqr_lower`. Therefore, an observation (i.e. a movie) is considered an outlier if

- its value lies above `iqr_upper`, or
- its value lies below `iqr_lower`

It becomes clear that we somehow need to define a condition because if it is an outlier, it should be labelled as one, but if not, then obviously we need to label the observation differently. Ideally, we want a new column in our dataset which indicates whether a movie is an outlier (i.e. `outlier == TRUE`) or not (`outlier == FALSE`). *R* offers a way for us to express such conditions with the function `ifelse()`. It has the following structure:

```
ifelse(condition, TRUE, FALSE)
```

Let's formulate a sentence that describes our scenario as an `ifelse()` function:

- If a movie's `runtime_min` is longer than `iqr_upper`, or
- if a movie's `runtime_min` is lower than `iqr_lower`,
- classify this movie as an outlier (i.e. `TRUE`),
- otherwise, classify this movie as not being an outlier (i.e. `FALSE`).

We already know from Chapter @ref(basic-computations-in-r) how to use logical and arithmetic operators. All we have to do is put them together in one function call and create a new column with `mutate()`.

```
imdb_top_250 <-
  imdb_top_250 %>%
  mutate(outlier = ifelse(runtime_min > iqr_upper |
                            runtime_min < iqr_lower,
                          TRUE, FALSE))
```

Since we have now a classification, we can more thoroughly inspect our outliers

and see which movies are the ones that are lying outside our defined norm. We can `arrange()` them by `runtime_min`.

```
imdb_top_250 %>%
  filter(outlier == "TRUE") %>%
  select(title, runtime_min, outlier) %>%
  arrange(runtime_min)
```

```
# A tibble: 8 x 3
  title                       runtime_min outlier
  <chr>                             <dbl> <lgl>
1 Sherlock Jr.                         45 TRUE
2 Andrei Rublev                       205 TRUE
3 Seven Samurai                       207 TRUE
4 Ben-Hur                             212 TRUE
5 Lawrence of Arabia                  228 TRUE
6 Once Upon a Time in America         229 TRUE
7 Gone with the Wind                  238 TRUE
8 Gangs of Wasseypur                  321 TRUE
```

The list of movies contains some of the most iconic Hollywood films ever shown on screen. However, I think we can agree that most of them are truly outside the norm of regular movies, not just in terms of runtime.

From here, it is simple to remove these movies (i.e. keep the movies that are not outliers) or set their values to `NA`. If we `replace()` the values with `NA`, we can continue with one of the techniques demonstrated for missing values in Chapter @ref(dealing-with-missing-data). Both approaches are shown in the following code chunk.

```
# Keep all observations but outliers
imdb_top_250 %>%
  filter(outlier == "FALSE") %>%
  select(title, outlier)
```

```
# Replace values with NA
imdb_top_250 %>%
  mutate(runtime_min = replace(runtime_min, outlier == "TRUE", NA)) %>%
  select(title, runtime_min)
```

```
# A tibble: 250 x 2
   title                                     runtime_min
   <chr>                                           <dbl>
 1 The Shawshank Redemption                          142
```

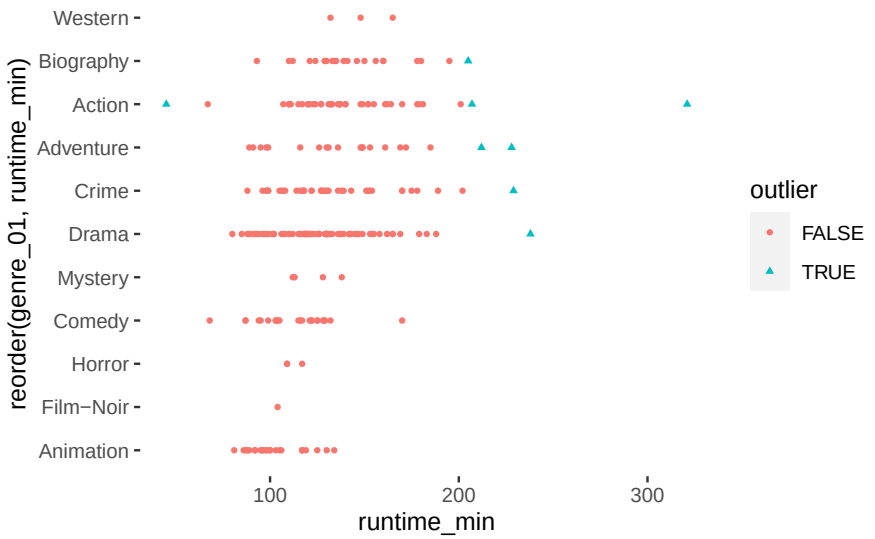
```
 2 The Godfather                                          175
 3 The Dark Knight                                        152
 4 The Godfather: Part II                                 202
 5 12 Angry Men                                            96
 6 The Lord of the Rings: The Return of the King          201
 7 Pulp Fiction                                           154
 8 Schindler's List                                       195
 9 Inception                                              148
10 Fight Club                                             139
# i 240 more rows
```

The `replace()` function is very intuitive to use. It first needs to know where you want to replace a value (`runtime_min`), then what the condition for replacing is (`outlier == "TRUE"`), and lastly, which value should be put instead of the original one (`NA`).

9.5.4 Concluding remarks about outliers

As you hopefully noticed, understanding your data requires some effort, but it is important to know your data well before proceeding to any further analysis. You can experiment with different data visualisations and design them in a way that best reflects the message you want to get across. For example, because we have added a new variable that classifies outliers, we can do more with our data visualisation than before and highlight them more clearly.

```
imdb_top_250 %>%
  ggplot(aes(x = reorder(genre_01, runtime_min),
             y = runtime_min,
             col = outlier,
             shape = outlier)
         ) +
  geom_point(size = 1) +
  theme(panel.background = element_blank()) +
  coord_flip()
```



To round things off, let me share with you one more visualisation, which demonstrates that outliers might also have to be defined based on specific groupings of your data, e.g. movie genres. Thus, it might be essential to consider outliers in light of sub-samples of your data rather than the entire dataset, especially when comparing groups (see Chapter @ref(comparing-groups)).

```
# The wesanderson package can make your plots look more
# 'Wes Anderson'
colour_pal <- wesanderson::wes_palette("Darjeeling1", 11, type = "continuous")

imdb_top_250 %>%
  ggplot(aes(x = reorder(genre_01, runtime_min),
             y = runtime_min,
             col = genre_01)
         ) +
  geom_boxplot(alpha = 0,
               show.legend = FALSE) +
  geom_jitter(width = 0.1,
              size = 0.5,
              alpha = 0.5,
              show.legend = FALSE
              ) +
  scale_color_manual(values = colour_pal) +
  coord_flip() +
  theme(panel.background = element_blank()) +
  xlab("runtime") +
```

```
  ylab("Genre") +
  ggtitle("Distribution of movie runtimes by genre")
```

10

Correlations

Sometimes counting and measuring means, medians, and standard deviations is not enough because they are all based on a single variable. Instead, we might have questions related to the relationship of two or more variables. In this section, we will explore how correlations can (and kind of cannot - see Chapter @ref(simpsons-paradox)) provide insights into the following questions:

- Do movie viewers agree with movie critics regarding the rating of movies?
- Do popular movies receive more votes from users than less popular movies?
- Do movies with more votes also make more money?

11

Power: You either have it or you don't

The most frequently asked question by my students is: How much data do I have to collect? My answer is always the same: *'It depends'*. From a student's perspective, this must be one of the most frustrating answers. Usually, I follow this sentence up with something more helpful to guide students on their way. For example, there is a way to determine how large your sample has to be to detect specific relationships reliably. This procedure is called *power analysis*.

While *power analysis* can be performed before and after data collection, it seems rather pointless to do it afterwards when collecting more data is not only challenging but sometimes impossible. Thus, my explanations in this chapter will primarily focus on the question: How much data is enough to detect relationships between variables reliably.

12

Comparing groups

Social Sciences is about the study of human beings and their interactions. As such, we frequently want to compare two or more groups of human beings, organisations, teams, countries, etc., with each other to see whether they are similar or different from each other. Sometimes we also want to track individuals over time and see how they may have changed in some way or other. In short, comparing groups is an essential technique to make inferences and helps us better understand the diversity surrounding us.

If we want to perform a group comparison, we have to consider which technique is most appropriate for our data. For example, some of it might be related to the type of data we have collected, and other aspects might be linked to the distribution of the data. More specifically, before we apply any statistical technique, we have to consider at least the following:

13

Regression: Creating models to predict future observations

Regressions are an exciting area of data analysis since it enables us to make very specific predictions, incorporating different variables simultaneously. As the name implies, regressions 'regress', i.e., draw on past observations to make predictions about future observations. Thus, any analysis incorporating a regression makes the implicit assumption that the past best explains the future.

I once heard someone refer to regressions as driving a car by looking at the rear-view mirror. As long as the road is straight, we will be able to navigate the car successfully. However, if there is a sudden turn, we might drive into the abyss. This makes it very clear when and how regressions can be helpful. Regressions are also a machine learning method, which falls under *models with supervised learning*. If you find machine learning fascinating, you might find the book *"Hands-on Machine Learning with R"* (Boehmke and Greenwell 2019) very insightful and engaging.

14

Mixed-methods research: Analysing qualitative data in R

Conducting mixed-methods research is challenging for everyone. It requires an understanding of different methods and data types and particular knowledge in determining how one can mix different methods to improve the insights compared to a single method approach. This chapter looks at one possibility of conducting mixed-methods research in *R*. It is likely the most evident use of computational software for qualitative data.

While I consider myself comfortable in qualitative and quantitative research paradigms, this chapter could be somewhat uncomfortable if you are used to only one or the other approach, i.e. only quantitative or only qualitative research. However, with advancements in big data science, it is impossible to code, for example, two million tweets qualitatively. Thus, the presented methodologies should not be considered in isolation from other forms of analysis. For some, they might serve as a screening tool to sift through large amounts of data and find those nuggets of most significant interest that deserve more in-depth analysis. For others, these tools constitute the primary research design to allow to generalise to a larger population. In short: the purpose of your research will dictate the approach.

15

Where to go from here: The next steps in your R *journey*

The first steps have been made, but we can certainly not claim we have reached our destination on this journey. If anything, this book helped us reach the first peak in a series of mountains, giving us an overview of what else there is to explore. To provide some guidance on where to go next, I collated some resources worth exploring as a natural continuation to this book.

15.1 GitHub: A Gateway to even more ingenious *R* packages

If you feel you want more *R* or you are curious to know what other *R* packages can do to complement your analysis, then GitHub brings an almost infinite amount of options for you. Besides the CRAN versions of packages, you find development versions of *R* packages on Github. These usually incorporate the latest features but are not yet available through CRAN. Having said that, if you write your next publication, it might be best to work with packages that are released on CRAN. These are the most stable versions. After all, you don't want the software to fail on you and hinder you from making that world-changing discovery.

Still, more and more *R* packages are released every day that offer the latest advancements in statistical computations and beyond. Methods covered in Chapter @ref(mixed-methods-research) are a great example of what has become possible with advanced programming languages. So if you want to live at the bleeding edge of innovative research methods, then look no further.

However, GitHub also serves another purpose: To back up your research projects and work collaboratively with others. I strongly encourage you to create your own GitHub account, even just to try it. RStudio has built-in features for working with GitHub, making it easy to keep track of your analysis and ensure regular backups. Nobody wants to clean up their data over months and lose it because their cat just spilt the freshly brewed Taiwanese High Mountain Oolong Tea over one's laptop. I use GitHub for many different

things, hosting my blog (The Social Science Sofa[1]), research projects, and this book.

There is much more to say about GitHub that cannot be covered in this book, but if you seek an introduction, you might find GitHub's Youtube Channel[2] of interest or their 'Get started'[3] guide. Best of all, setting up and using your GitHub account is free.

15.2 Books to read and expand your knowledge

There are undoubtedly many more books to read, explore and use. However, my number one recommendation to follow up with this book is 'R for Data Science'[4]. It takes your *R* coding beyond the context of Social Sciences and introduces new aspects, such as *'custom functions'* and *'looping'*. These are two essential techniques if you, for example, have to fit a regression model for +20 subsets of your data, create +100 plots to show results for different countries, or need to create +1000 of individualised reports for training participants. In short, these are very powerful (and efficient) techniques you should learn sooner than later but are less essential for a basic understanding of data analysis in *R* for the Social Sciences.

If you want to expand your repertoire regarding data visualisations, a fantastic starting point represents 'ggplot2: Elegant Graphics for Data Analysis'[5] and 'Fundamentals of Data Visualization'[6]. However, these days I am looking mostly for inspiration and new packages that help me create unique, customised plots for my presentations and publications. Therefore, looking at the source code of plots you like (for example, on GitHub) is probably the best way to learn about `ggplot2` and some new techniques of how to achieve specific effects (see also Chapter @ref(next-steps-twitter)).

If you are a qualitative researcher, you might be more interested in what else you can do with *R* to systematically analyse large amounts of textual data (as was shown in Chapter @ref(mixed-methods-research)). I started with the excellent book 'Text Mining with R: A Tidy Approach'[7], which introduces you in greater depth to sentiment analysis, correlations, n-grams, and topic modelling. The lines of qualitative and quantitative research become increasingly

[1] http://thesocialsciencesofa.com
[2] https://www.youtube.com/user/github
[3] https://docs.github.com/en
[4] https://r4ds.had.co.nz
[5] https://ggplot2-book.org
[6] https://clauswilke.com/dataviz/
[7] https://www.tidytextmining.com

blurred. Thus, learning these techniques will be essential moving forward and pushing the boundaries of what is possible with textual data.

R can do much more than just statistical computing and creating pretty graphs. For example, you can write your papers with it, even if you do not write a single line of code. From Chapter @ref(r-markdown-and-r-notebooks), you might remember that I explained that *R* Markdown files are an alternative to writing R scripts. Suppose you want to deepen your knowledge in this area and finally let go of Microsoft Word, I encourage you to take a peek at the 'R Markdown Cookbook'[8] for individual markdown files and 'bookdown: Authoring Books and Technical Documents with R Markdown'[9] for entire manuscripts, e.g. journal paper submissions or books. I almost entirely abandoned Microsoft Word, even though it served me well for so many years - thanks.

Lastly, I want to make you aware of another open source book that covers What They Forgot to Teach you About R[10] by Jennifer Bryan and Jim Hester. It is an excellent resource to get some additional insights into how one should go about working in *R* and *RStudio*.

15.3 Engage in regular online readings about *R*

A lot of helpful information for novice and expert *R* users is not captured in books but online blogs. There are several that I find inspiring and an excellent learning platform, each focusing on different aspects.

The most comprehensive hub for all things *R* is undoubtedly 'R-bloggers'[11]. It is a blog aggregator which focuses on collecting content related to *R*. I use it regularly to read more about new packages, new techniques, helpful tricks to become more 'fluent' in *R* or simply find inspiration for my own *R* packages. More than once, I found interesting blogs by just reading posts on 'R-bloggers'. For example, the day I wrote this chapter, I learned about `emayili`, which allows you to write your emails from *R* using *R* markdown. So not even the sky is the limit, it seems.

Another blog worth considering is the one from 'Tidyverse'[12]. This blog is hosted and run by RStudio and covers all packages within the `tidyverse`. Posts like 'waldo 0.3.0'[13] made my everyday data wrangling tasks a lot easier

[8] https://bookdown.org/yihui/rmarkdown-cookbook/
[9] https://bookdown.org/yihui/bookdown/
[10] https://rstats.wtf
[11] https://www.r-bloggers.com
[12] https://www.tidyverse.org/blog/
[13] https://www.tidyverse.org/blog/2021/08/waldo-0-3-0/

because finding the differences between two datasets can be like searching a needle in a haystack. For example, it is not unusual to receive two datasets that contain the same measures, but some of their column names are slightly different, which does not allow us to merge them in the way we want quickly. I previously spent days comparing and arranging multiple datasets with over 100 columns. Let me assure you, it is not really fun to do.

The blog 'Data Imaginist'[14] by Thomas Lin Pedersen covers various topics around data visualisations. He is well known for his Generative Art, i.e. art that is computationally generated. But one of my favourite packages, `patchwork`, was written by him and gained immense popularity. It has never been easier to arrange multiple plots with such little code.

Lastly, I want to share with you the blog of Cédric Scherer[15] for those of you who want to learn more about high-quality data visualisations based on `ggplot2`. His website hosts many visualisations with links to the source code on GitHub to recreate them yourself. It certainly helped me improve the visual storytelling of my research projects.

15.4 Join the Twitter community and hone your skills

Learning *R* means you also join a community of like-minded programmers, researchers, hobbyists and enthusiasts. Whether you have a Twitter account or not, I recommend looking at the #RStats[16] community there. Plenty of questions about *R* programming are raised and answered on Twitter. In addition, people like to share insights from their projects, often with source code published on GitHub. Even if you are not active on social media, it might be worth having a Twitter account just to receive news about developments in *R* programming.

As with any foreign language, if we do not use it regularly, we easily forget it. Thus, I would like to encourage you to take part in Tidy Tuesday[17]. It is a weekly community exercise around data visualisation and data wrangling. In short: You download a dataset provided by the community, and you are asked to create a visualisation as simple or complex as you wish. Even if you only manage to participate once a month, it will make you more 'fluent' in writing your code. Besides, there is a lot to learn from others because you are also asked to share the source code. This activity takes place on Twitter, and you can find contributions by using #TidyTuesday[18]. Whether you want to

[14] https://www.data-imaginist.com
[15] https://www.cedricscherer.com/top/dataviz/
[16] https://twitter.com/search?q=%23rstats
[17] https://www.tidytuesday.com
[18] https://twitter.com/search?q=%23tidytuesday

share your plots is up to you, but engaging with this activity will already pay dividends. Besides, it is fun to work with datasets that are not necessarily typical for your field.

References

Aguinis, Herman, Ryan K Gottfredson, and Harry Joo. 2013. "Best-Practice Recommendations for Defining, Identifying, and Handling Outliers." *Organizational Research Methods* 16 (2): 270–301.

Boehmke, Brad, and Brandon Greenwell. 2019. *Hands-on Machine Learning with r*. Chapman; Hall/CRC.

Buuren, Stef van. 2018. *Flexible Imputation of Missing Data*. CRC press.

Cambridge Dictionary. 2021. "Concatenate." https://dictionary.cambridge.org/dictionary/english/concatenate.

Cousineau, Denis, and Sylvain Chartier. 2010. "Outliers Detection and Treatment: A Review." *International Journal of Psychological Research* 3 (1): 58–67.

Dong, Yiran, and Chao-Ying Joanne Peng. 2013. "Principled Missing Data Methods for Researchers." *SpringerPlus* 2 (1): 1–17.

Field, Andy. 2013. *Discovering Statistics Using IBM SPSS Statistics*. Sage Publications.

Ginkel, Joost R van, Marielle Linting, Ralph CA Rippe, and Anja van der Voort. 2020. "Rebutting Existing Misconceptions about Multiple Imputation as a Method for Handling Missing Data." *Journal of Personality Assessment* 102 (3): 297–308.

Grolemund, Garrett, and Hadley Wickham. 2011. "Dates and Times Made Easy with lubridate." *Journal of Statistical Software* 40 (3): 1–25. https://www.jstatsoft.org/v40/i03/.

Henson, Robin K. 2001. "Understanding Internal Consistency Reliability Estimates: A Conceptual Primer on Coefficient Alpha." *Measurement and Evaluation in Counseling and Development* 34 (3): 177–89.

Hu, Li-tze, and Peter M Bentler. 1999. "Cutoff Criteria for Fit Indexes in Covariance Structure Analysis: Conventional Criteria Versus New Alternatives." *Structural Equation Modeling: A Multidisciplinary Journal* 6 (1): 1–55.

Jakobsen, Janus Christian, Christian Gluud, Jørn Wetterslev, and Per Winkel. 2017. "When and How Should Multiple Imputation Be Used for Handling Missing Data in Randomised Clinical Trials–a Practical Guide with Flowcharts." *BMC Medical Research Methodology* 17 (1): 1–10.

Leys, Christophe, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. 2013. "Detecting Outliers: Do Not Use Standard Deviation

Around the Mean, Use Absolute Deviation Around the Median." *Journal of Experimental Social Psychology* 49 (4): 764–66.

Little, Roderick J. A. 1988. "A Test of Missing Completely at Random for Multivariate Data with Missing Values." *Journal of the American Statistical Association* 83 (404): 1198–1202. https://doi.org/10.1080/01621459.1988.10478722.

Müller, Kirill, and Hadley Wickham. 2021. "Column Data Types." https://tibble.tidyverse.org/articles/types.html.

Nunally, J. C. 1967. *Psychometric Theory*. New york: Mc Graw-Hill.

———. 1978. *Psychometric Theory (2nd Edition)*. New york: Mc Graw-Hill.

Rubin, Donald B. 1976. "Inference and Missing Data." *Biometrika* 63 (3): 581–92.

Schafer, Joseph L. 1999. "Multiple Imputation: A Primer." *Statistical Methods in Medical Research* 8 (1): 3–15.

Stobierski, Tim. 2021. "Data Wrangling: What It Is and Why It's Important." Harvard Business School Online. https://online.hbs.edu/blog/post/data-wrangling.

West, Stephen G, Aaron B Taylor, Wei Wu, et al. 2012. "Model Fit and Model Selection in Structural Equation Modeling." *Handbook of Structural Equation Modeling* 1: 209–31.

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

———. 2021. *The Tidyverse Style Guide*. https://style.tidyverse.org.

Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc.

Xie, Yihui, J. J. Allaire, and Garrett Grolemund. 2018. *R Markdown: The Definitive Guide*. Boca Raton, Florida: Chapman; Hall/CRC. https://bookdown.org/yihui/rmarkdown.