
Part I

Motivation



1

Why you should read this book

Data science is a growth area in industry, public service and academia. Although not everybody needs to become a professional data scientist, we should all strive to become critical consumers of data that are presented to us by news media, governments and lobby groups. If you want to take the leap from being a consumer to being a mindful producer of data analysis and visualisation, I invite you to join me on a journey from basic programming to publication-ready infographics.

1.1 Finding programmatic solutions for data analysis and visualisation problems

Data analysis is the process of exploring, transforming and modelling data to discover useful information, summarise the discoveries and draw informed conclusions. Data visualisation is the graphical representation of information and data. Data visualisation is also the name of a field of research that aims to develop effective visual techniques for data analysis at various stages (exploration, interpretation and reporting).

The purpose of this book is to teach data analysis and visualisation in a hands-on manner. Often, data need to be transformed before they are ready to be presented in visual form. We will learn how to apply the necessary transformations with computer programs. If you have worked with spreadsheet software before (e.g. Microsoft Excel® or Google Sheets®), you already have a basic understanding of how to store and represent data on a computer. However, spreadsheet software has limitations when data management tasks need to be automated (e.g. for producing automated reports whenever a data set is updated). Spreadsheet software also has limited support for creating bespoke customised infographics. By the end of this book, you will have learned the programming language R, which offers a principled, customisable alternative to spreadsheet software.

1.2 Becoming a responsible producer of data visualisation

Data visualisation has a long history. Maps of the night sky are among the earliest attempts by humans to represent data (positions of stars and their brightness) in graphical form, dating back at least to 1534 BC ([Spaeth, 2000](#)). While early approaches to data visualisation were mostly ad hoc, Renaissance mathematicians began to systematically describe how to present data in graphical form. For example, René Descartes popularised one of the cornerstones of modern data visualisation in 1637: the two-dimensional coordinate system that we now refer to as the ‘Cartesian’ coordinate system in his honour ([Hatfield, 2018](#)). On the basis of Cartesian coordinates, the Scottish engineer William Playfair invented many types of diagrams that are still in common use today such as bar charts in 1786 and pie charts in 1801 ([Friendly and Denis, 2001](#)).

Thanks to advances in computer technology, we are currently experiencing a proliferation of infographics, both in quantity and variety. However, not every diagram produced by a computer is automatically well crafted. I now highlight some common problems with infographics encountered in the wild.

1.2.1 Areas should be proportional to numeric data

One of the fundamental rules of data visualisation was put into words by [Tufte \(1983\)](#) as follows:

‘The representation of numbers, as physically measured on the surface of the graphic itself, should be directly proportional to the numerical quantities represented.’

This rule is often referred to as ‘area principle’: every part of a diagram should have an area in proportion to the number it represents. Diagrams that violate the area principle can be misleading as the following example shows.

In an article with the headline ‘Over 100 Million Now Receiving Federal Welfare’ ([Halper, 2012](#)), the US news magazine Washington Examiner published the diagram shown in figure 1.1. The diagram aims to show how the number of people on federal welfare increased from the first quarter of 2009 to the second quarter of 2011. The author of this figure chose to present the data in the form of a bar chart. A bar chart is a type of diagram in which data are

binned by categories. Each category is represented by one bar, and the count of data in each category is shown as the height of the bar. In figure 1.1, the categories are quarters. Because each bar is equally wide, the area principle implies that the height of each bar should be proportional to the number of people on welfare in the corresponding quarter. Figure 1.1 violates the area principle because the bar for ‘2011 Q2’ is about 4.3 times longer than the bar for ‘2009 Q1’. However, the number of welfare recipients only increased by a factor 1.1 (from 97 million to 107 million).

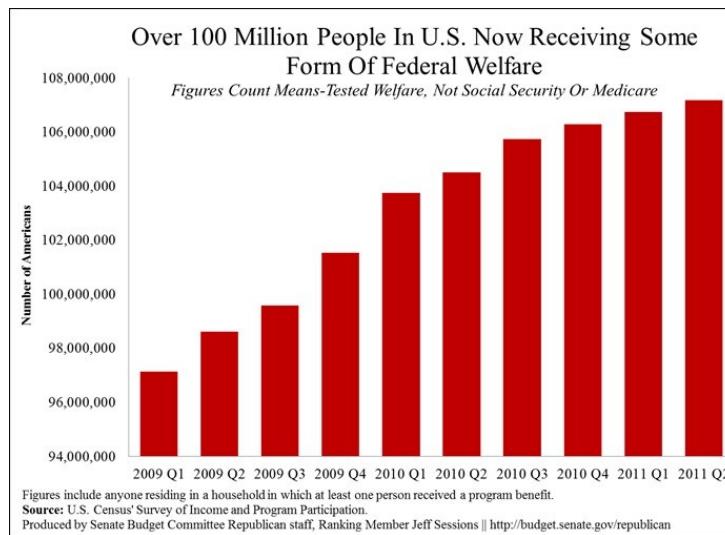


FIGURE 1.1: This diagram (Halper, 2012) is misleading because the areas of the bars are not proportional to the numbers they represent.

The problem with figure 1.1 is that the y-axis (i.e. the vertical axis) starts from 94 million instead of zero. Consequently, small differences in the number of welfare recipients appear exaggerated. A better visualisation is shown in figure 1.2, where the y-axis starts from zero. From figure 1.2, it becomes clear that the number of welfare recipients increased, but the increase is relatively small.

A critical reader may argue that the wider range of the y-axis in figure 1.2 compresses the differences and, hence, makes it more difficult to accurately infer the number of recipients compared to figure 1.1. This criticism is valid. However, if we want to scale the y-axis as in figure 1.1, we should use a point-to-point chart (also known as line chart) as shown in figure 1.3 instead of a bar chart. Points are zero-dimensional objects; thus, they have neither a length nor an area. Therefore, point-to-point charts cannot violate the area principle; they do not represent numbers by areas but by positions along an axis.

Figures 1.2 and 1.3 are both good representations of the number of welfare

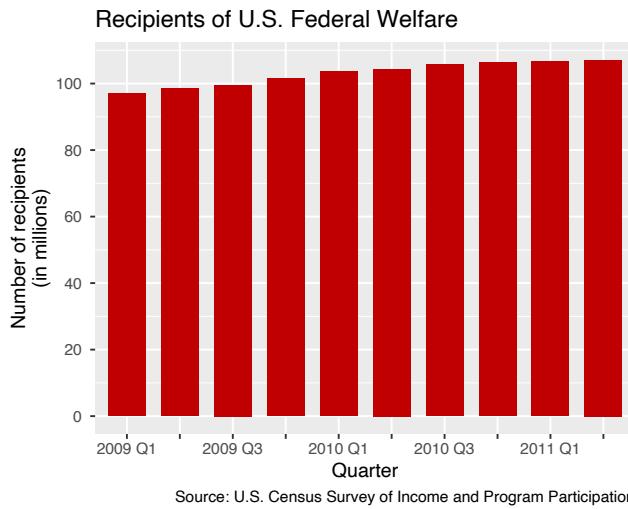


FIGURE 1.2: Unlike figure 1.1, this diagram satisfies the area principle because the y-axis starts from zero.

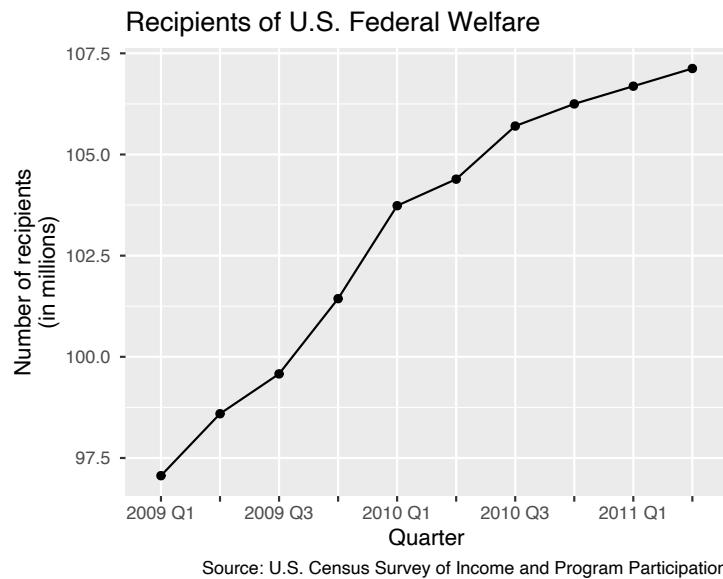


FIGURE 1.3: If we want to use a y-axis range that is close to the range of the data, we should not use a bar chart but a point-to-point chart.

recipients, and each plot has its strengths and weaknesses. While figure 1.2 emphasises the total number of welfare recipients in different quarters, figure 1.3 makes it easier to infer the number of recipients from the y-axis. Figure 1.1 combines the weaknesses of both diagrams instead of their strengths. Thereby, figure 1.1 makes readers believe that the increase in the number of welfare recipients is far more dramatic than the numbers actually imply. As designers of statistical diagrams, we should choose the type of diagram more judiciously.

1.2.2 Use diagrams for communication, not for the show effect

Diagrams attract the reader's attention. As creators of data visualisation, we need to be mindful that we should only attract the reader's attention to a diagram if we have a good reason. Some data are more efficiently presented as part of the text or in a table. For example, the pie chart in figure 1.4 (Barnes, 2015) takes up relatively much space, but it contains only two independent pieces of quantitative information:

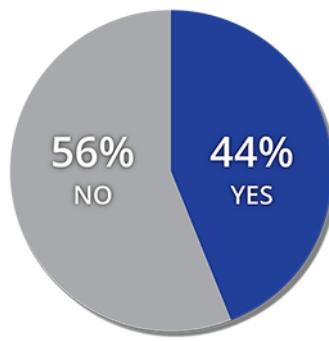
- There were 98 respondents. The number of respondents appears in small font below the pie chart; thus, readers are likely to miss this information at first glance.
- 56% of respondents answered 'No'. The pie chart is divided into two slices, so the only answer options seem to have been 'Yes' and 'No'. Hence, the percentage of respondents who answered 'Yes' must necessarily be 44%.

Instead of showing a pie chart, it would be more efficient to state the quantitative information directly in the text, for example: 'Out of 98 respondents, 56% prefer that Joe Biden would not run for the 2016 Democratic presidential nomination.'

When there are more pieces of quantitative information, it may be impractical to include all the numbers in the text. For example, figure 1.5 contains four pieces of quantitative information: the number of residents in each of the four ethnic groups that appear in the Singaporean census (Chinese, Malay, Indian and Other). If we want to express the same information in text, it would become relatively long: 'According to the Department of Statistics Singapore, Chinese residents were the largest ethnic group (3,006,769; 74.3%) followed by Malay residents (545,498; 13.5%) and Indian residents (362,274; 9.0%). Only 129,669 residents (3.2%) belonged to other ethnic groups.' In this case, the bar chart in figure 1.5 is arguably the more reader-friendly and intuitive alternative to the long-winded text.

Text and diagrams are not the only tools that are available for communicating data. Tables are often a good alternative if there are relatively few numbers. For example, figure 1.6 contains the same information as the bar chart in figure 1.5. The tabular format of figure 1.6 makes it a little bit easier to compare the exact population numbers of different ethnicities because the reader's eye can simply move down the second column in the table. As shown

Would you like to see Joe Biden run for the 2016 Democratic presidential nomination?



*Breakdown of 98 respondents

FIGURE 1.4: This pie chart (Barnes, 2015) contains little quantitative information. It would be more efficient to present the data as text instead of a diagram.

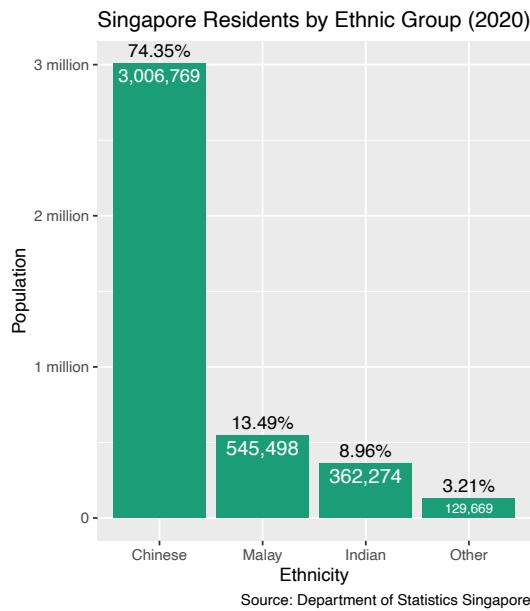


FIGURE 1.5: This bar chart contains four independent pieces of quantitative data (one number for each ethnic group). In this case, a bar chart is better than long-winded text. The percentage values shown above each bar are not strictly necessary, but they are a reader-friendly addition.

in the second column, we can even include a bar chart directly as part of the table. Compared with figure 1.5, the only graphical element that is absent from figure 1.6 are the axis tick marks ('0', '1 million', '2 million' and '3 million'). Omitting the tick marks does not come with a significant loss of information because it is easy to find the exact population numbers from the table. Therefore, the tabular format of figure 1.6 is, in my opinion, preferable to figure 1.5 because the table is a little bit less cluttered. We learn how to prepare tables in one of the exercises of part III.

Singapore Residents by Ethnic Group (2020)		
Ethnicity	Population	Percentage
Chinese	3,006,769	74.35%
Malay	545,498	13.49%
Indian	362,274	8.96%
Other	129,669	3.21%

Source: Department of Statistics Singapore

FIGURE 1.6: Table containing the same numeric information as figure 1.5. The bar chart in the second column is not strictly necessary, but it helps readers to get an impression of the data at first glance.

If you are new to data visualisation, figures 1.5 and 1.6 might appear extremely plain. The only geometric features are rectangular bars, and all of them are in the same colour. Would it not be more engaging for readers if we included more complex objects and a greater variety of colours? If you answered with yes, figure 1.7 might be more to your taste. The Korea Herald ([Chang-Duk, 2014](#)) presented figure 1.7 to visualise military spending by country in 2013. Instead of plain rectangular bars, the designer represents each country by a bullet. The larger the bullet, the higher the amount of military spending. In the top right corner, there are also other symbols of weapons systems (a plane, missiles, soldiers, a helicopter and a tank). These symbols clearly depict what the money is spent on; thus, the image arguably does its job from an artistic perspective.

However, artistic quality does not equate to successful communication. The bullets are lined up as in a bar chart, but, unlike the bars in a conventional bar chart, they have different widths. Thus, it is unclear whether the reader is supposed to compare the bullets on the basis of their heights, their surface areas or their volumes. A careful measurement of the bullet dimensions suggests that the designer intended to represent the amount of military spending by the surface areas, which is not self-explanatory given that bullets are three-dimensional objects. The shades cast by the bullets emphasise the three-dimensional impression, but they do not add more information to the figure. The world map and the weapon symbols in the top right corner add even more clutter. The flags on the bullets add a splash of colour, but they only duplicate

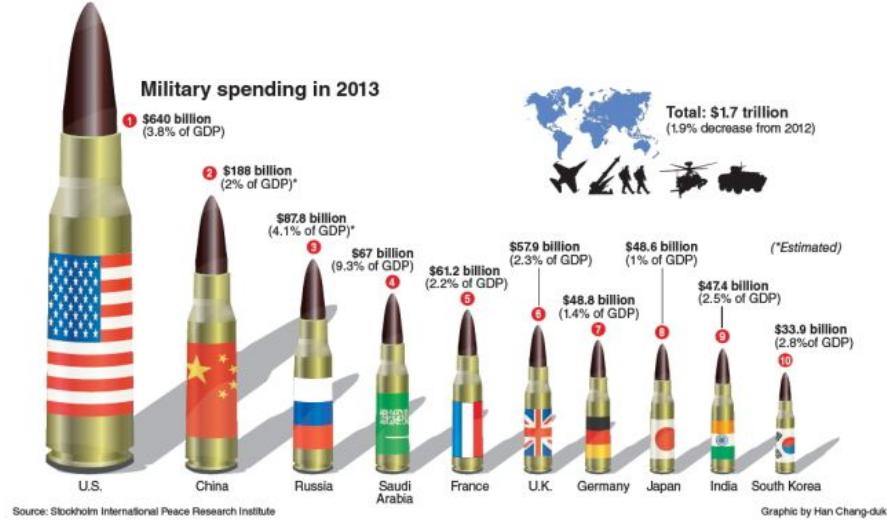


FIGURE 1.7: This plot (Chang-Duk, 2014) has artistic value, but it does not communicate data effectively.

information that is already implicit in the country names below each bullet. Moreover, it is confusing that the percentage values above the bullets refer to a different reference value (percent of national GDP) than the percentage value to the right of the world map (decrease compared to 2012).

Figure 1.8 shows an alternative visualisation of the same data. The table looks admittedly less colourful, but its restrained design puts the data at the centre of the reader’s attention. We learn from this example that we should generally refrain from unnecessary ornamentation (e.g. variations in colour and three-dimensional effects). The goal of data visualisation is to communicate data, not to impress the reader with artistic creativity.

1.3 Topics not included in this book

Technologies for data visualisation are rapidly evolving and increasingly diverse. It would be impossible to cover every possible aspect of modern data visualisation. Here are some topics that are not included in this book.

- *Interactive graphics*

Modern web technology enables designers to add a variety of interactive features to graphic displays. Some common effects found in web-based infographics are infotips (i.e. text boxes that reveal additional information

Military Spending in 2013		
Country	Spending (billion US\$)	% of GDP
U.S.	640.0	3.8
China	188.0	2.0
Russia	87.8	4.1
Saudi Arabia	67.0	9.3
France	61.2	2.2
U.K.	57.9	2.3
Germany	48.8	1.4
Japan	48.6	1.0
India	47.4	2.5
South Korea	33.9	2.8
Total	1,700.0	

Source: Stockholm International Peace Research Institute

FIGURE 1.8: table with the same data as figure 1.7.

when hovering over specific parts of the figure), morphing between different data sets and linked brushing (i.e. hovering over one part of a diagram highlights another, related part). If you are interested in interactive data visualisation, I recommend that you have a look at the R package **shiny**.

- *Animations*

Data sets that highlight changes over time (e.g. weather radar) are sometimes best shown as animations. The R package **ganimate** contains utilities for creating animations.

- *Three-dimensional plots*

Some data sets describe the relation of three continuous variables (e.g. weight, age and cholesterol level). One can think of the observations as points in a three-dimensional space. Three-dimensional scatter plots can reveal patterns in the data, especially if the coordinate axes can be interactively rotated. There are several R packages that can produce interactive three-dimensional scatter plots (e.g. **plotly** or **rgl**).

Interactivity, animations and three-dimensional plots can enrich the user experience. However, beginners tend to be carried away by their possibilities; thus, I suggest that you first become a responsible producer of non-interactive, static and two-dimensional graphics before adding more tools to your tool kit.

To develop a basic set of tools, the content of this book is generally leaning towards hands-on instructions. Consequently, it touches on statistical theory only in part XI. I recommend that you start developing a solid theoretical foundation by taking a statistics course if you have not done it yet. This book

also omits the theory behind computer programming and algorithms. In R, the complexity of the algorithms tends to be hidden behind high-level function calls. This property makes R suitable for beginners. However, to reach a higher level of proficiency, I suggest that you take an introductory computer science course as one of the next steps in your learning trajectory.

Exercise: Reflections on data visualisation

Prerequisite: chapter 1

Approximate duration: 120 minutes

Submission format: PDF or Word document

Answer each of the following questions with several paragraphs of text.

- (I) Figures 1.9–1.11 show examples of data visualisation in the media. Write a paragraph about each of them. What do you like or dislike about these diagrams? Describe how you would improve the presentation of these data.

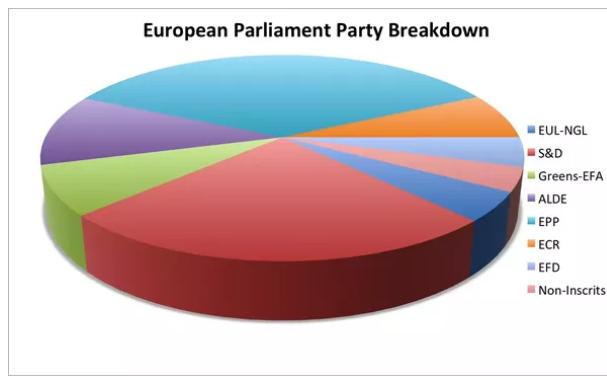


FIGURE 1.9: Diagram by [Hickey \(2013\)](#).

- (II) Find an example of (good or bad) data visualisation in the media. Explain what you like or dislike about it. How would you improve the presentation of the data?

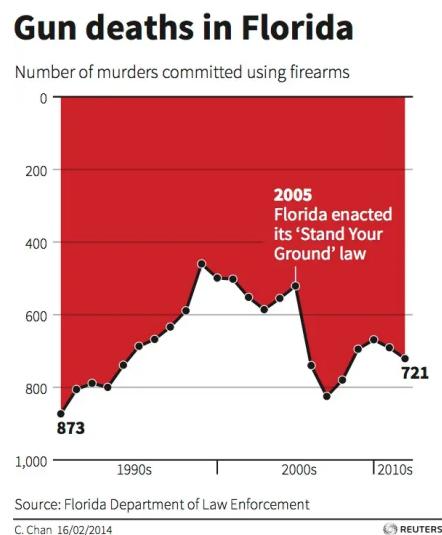


FIGURE 1.10: Diagram by [Chan \(2014\)](#).

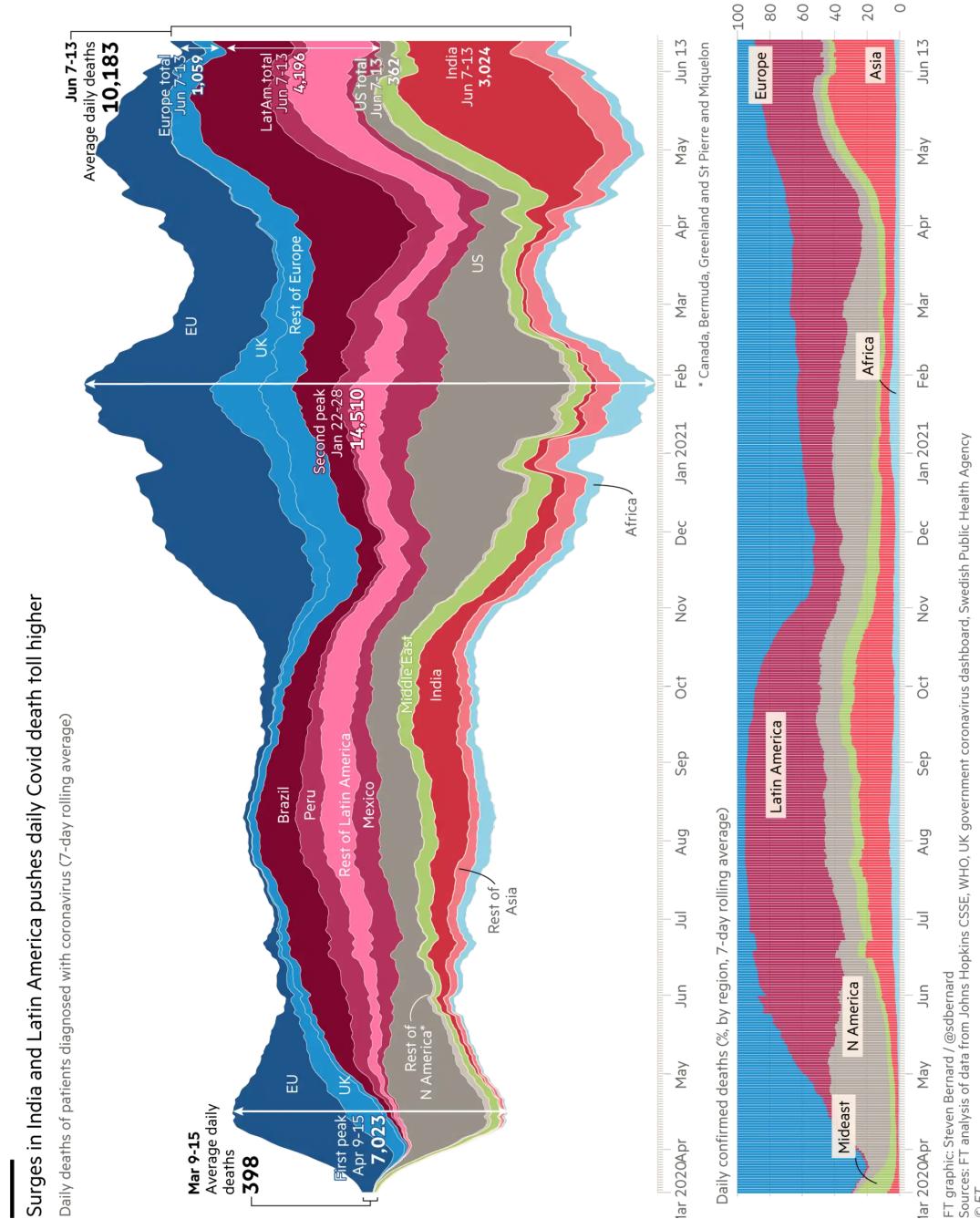


FIGURE 1.11: Diagram by Bernard (2020).



Part II

Getting to know R and RStudio



2

Exploring R and RStudio

Patrons at a restaurant usually do not experience how much effort went into making their meals. Instead, patrons judge the quality of the chef primarily by the taste and look of the food on their plates. Even if the preparation of the meal is not visible to the patrons, a chef's success relies on more than just the taste and outward appearance. Good kitchen utensils and refined knife skills are essential to create tasty food in a short time. Similarly, data scientists are ultimately judged by the quality of their products (e.g. reports or plots), but it needs good software and programming skills to work efficiently.

In this book, we use a combination of two pieces of software: the programming language R and the integrated development environment RStudio. R and RStudio possess many useful features that support a data scientist's workflow from raw data to final product. The combination of R and RStudio has become the standard software choice in statistics and data visualisation. A large community of developers is actively contributing to R and RStudio. Let us join this community and take our first steps towards learning these two pieces of software.

2.1 What is R?

R is a programming language originally created in 1996 by Ross Ihaka and Robert Gentleman, two statistics professors from the University of Auckland in New Zealand ([Vance, 2009](#)). Originally intended only as a replacement for the statistics software used for teaching at their university department, R has become enormously popular among statisticians and data scientists worldwide for the following reasons:

- R is free as in ‘free beer’ (i.e. there is no cost to the users).
- R is free as in ‘free speech’ (i.e. open-source).
- R is stable and reliable because bug fixes and improvements to the core code are publicly discussed.
- R runs on all common operating systems.
- R is versatile because users have contributed many additional features over the years in the form of ‘packages’.

Because of the aforementioned reasons, R is a great choice whenever we simply need to ‘get the job done’. R offers efficient solutions to many complex problems in statistics, data analysis, visualisation and report writing. Consequently, many data science job advertisements specify R as a desired, if not even mandatory, skill.

There are admittedly some downsides to R.

- R is an interpreted, not a compiled language (unlike, for example, C, C++ and Fortran); the machine code is generated on the fly by a program called the *interpreter*. By contrast, a *compiler* would first read the entire code and then generate an optimised executable program. As a consequence, R can be slow and often handles memory inefficiently.¹ However, these problems can sometimes be prevented by developing good R coding habits. If necessary, there are some advanced tools available to include compiled code in R (for example, the **Rcpp** package), which are beyond the scope of this book.
- There is no guarantee that packages are thoroughly maintained by the users who contributed them.
- Beginners may feel that R’s documentation is cryptic. Some commercial products offer more personalised help (e.g. over the phone), whereas R users have to fend more for themselves. However, there are R message boards that usually offer high-quality advice (see section 4.2).

If you are on a Mac or Windows computer, you can download R from the Comprehensive R Archive Network (<https://cran.r-project.org/>). If you are on Linux, it is best to use your package manager to install R.

2.2 What is RStudio?

There are many ways to write and run R programs. For example, we can write the code with any conventional text editor (e.g. the pre-installed application TextEdit on a Mac or Notepad on Windows). We can run R code from the Command Prompt in Windows or a Terminal in Mac and Linux. For certain applications, running code from the Command Prompt or Terminal may be the only available option (e.g. when we want to run an R program on a remote server).

However, most of the time it is more advantageous to use software that combines writing and running code into a single user interface. Various ‘integrated development environments’ (IDEs) have been developed for R over the years,

¹There are also other reasons why R is relatively slow, for example dynamic typing, name lookup with mutable environments and ‘lazy’ evaluation of function arguments (Wickham, 2014).

for example Tinn-R² for Windows or RKWard³ for Linux. At present, by far the most popular IDE for R is RStudio⁴ for the following reasons:

- RStudio's basic version is free (both as in 'free beer' and as in 'free speech').
- RStudio provides a consistent user interface regardless of the operating system.
- RStudio includes a code editor with many convenient features (e.g. syntax highlighting, automatic indentation and code suggestions).
- The RStudio user interface shows at one glance all the variables and functions that we have defined in our code.
- With RStudio, we can view R graphics, built-in help documents and web applications.

For many users, R and RStudio have, in fact, become practically synonymous. Although they really are two separate pieces of software, the convenience that RStudio has brought to working with R is indeed remarkable. Because it makes coding in R efficient and intuitive, RStudio is the IDE of choice for this book. To download RStudio, please go to <https://rstudio.com/products/rstudio/download/>. The free RStudio Desktop version is sufficient to follow along with the text in this book.

Before you start working with RStudio, let us customise some of its settings (figure 2.1). Please go to the menu item 'Tools' → 'Global Options'. Look for the drop-down menu that says 'Save workspace to .Rdata on exit'. Let us choose 'Never' from the Menu. Next, please remove the tick marks for all checkboxes above this drop-down menu. We do not want to reuse or restore anything at startup because these features can be confusing when learning R and RStudio. It is easier to understand R's behaviour if we always start from a blank slate.

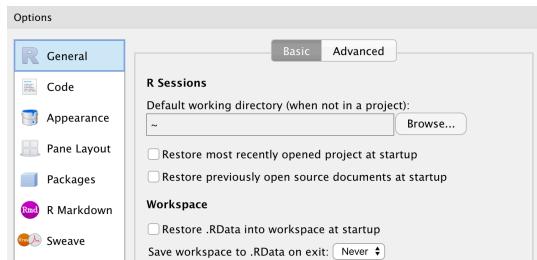


FIGURE 2.1: Recommended global options. The top three boxes are unticked, and we do not save .Rdata on exit.

There are a few more changes to the editor settings that I find useful (figure 2.2). I make these changes by first clicking on 'Code' in the sidebar on the

²<https://sourceforge.net/projects/tinn-r/>

³<https://rkward.kde.org/>

⁴<https://www.rstudio.com/>

left, and then on the tab ‘Display’. I recommend ticking the boxes for ‘Show line numbers’ and ‘Show margin’. To follow generally recommended practices, I set the margin column to 80. I also tick the box for ‘Show indent guides’. The purpose of indent guides is explained in section 7.3.5. There are many other options we can change (e.g. font size or background colour). Feel free to play with the settings, but I personally can live happily with the remaining defaults.

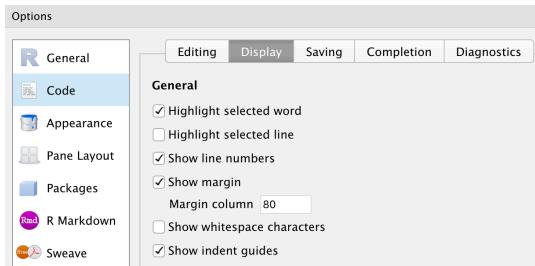


FIGURE 2.2: Recommended editor settings.

2.3 RStudio user interface

When RStudio is opened for the first time after installation, the user interface window is divided into three panes (figure 2.3):

- the console in the left half of the window.
- a pane that, by default, shows the environment tab in the top right.
- a pane with a tab that shows the files in the current directory, which is, by default, the home directory of your computer.

If any of the panes are invisible, they are hidden under another pane. The hidden pane becomes visible when you either

- click on the ‘expand pane’ symbol  in the top right corner of the pane or
- slide the separator between the panes up, down, left or right with your mouse. When the mouse pointer is inside one of the gaps between the panes, it turns into a double-sided arrow . By sliding the arrow in the desired direction, you can adjust the sizes of the panes.

Some panes serve more than one purpose. For example, the bottom right pane also has a tab for plots. As you become more familiar with the RStudio user interface over the next few chapters, the various purposes of the panes and tabs becomes more evident.

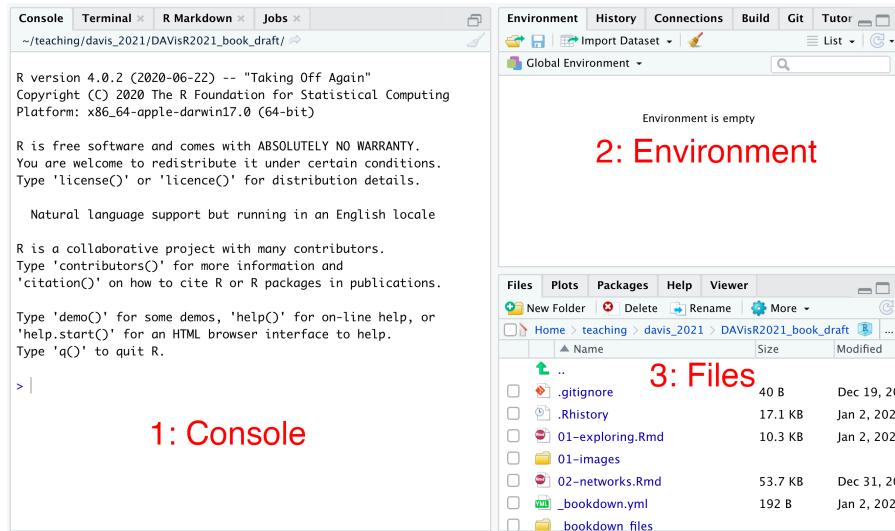


FIGURE 2.3: The RStudio window is by default divided into three panes: console, environment and files.

2.4 R console

Let us begin our exploration with the R console (i.e. the left pane in figure 2.3). A console in computer programming is a command-line interface. After a prompt (in R represented by the greater-than symbol >) at the start of the line, we type a command. When we press the return key, the command is executed.

For example, the R console can be used like a pocket calculator. When you type

```
2 * (9 + 12)
```

after the prompt, you get the following result.

```
## [1] 42
```

This is the console's way of telling us that $2 \cdot (9 + 12) = 42$. The two hash symbols ## do not actually appear in the output. I use ## to indicate that the following expression is output from R. The actual output starts with [1]. I explain the meaning of the number in square brackets in section 3.1.1.

Sometimes, the R console changes the command prompt symbol from > to +, for example on the second line of the following code chunk.

```
> 2 * (9 +
+   12)
## [1] 42
```

The `+` at the start of the second line appears because the open parenthesis `'('` in the first line does not have a matching closing parenthesis `')'` on the same line. Until the R interpreter encounters a closing parenthesis, it treats the input on the first line as an incomplete command. The R console indicates this fact by changing the command prompt from `>` to `+` until the parenthesis is closed. If you encounter the `+` prompt in error, you can simply press the escape key on your keyboard. The escape key terminates the previous command and sends you back to the usual command prompt `>`, where you can start typing a fresh command.

Similar to a simple graphing calculator, you can also make quick plots of functions. They appear to the right of the console (figure 2.4).

```
> plot(cos, -2 * pi, 2 * pi)
```

In this and the following examples, you do not need to type the `>` symbol at the start of the line. Instead `>` indicates that the following command appears after the command prompt.

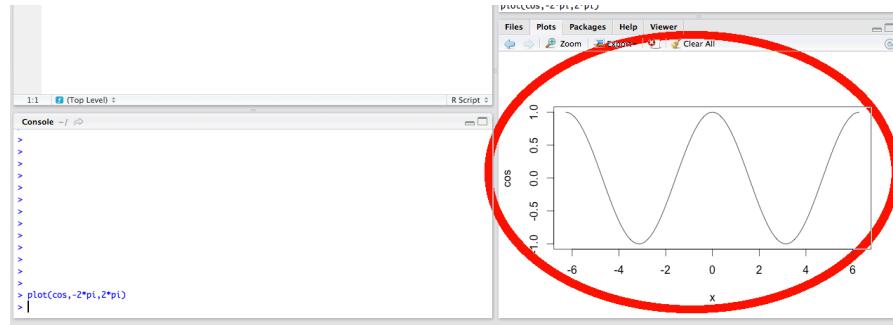


FIGURE 2.4: In RStudio, plots appear in the bottom right pane. By clicking on the tab names at the top of the pane (e.g. ‘Files’ or ‘Plots’), you can navigate between different tabs.

2.5 Working with RStudio projects

In later chapters of this book, you use R to import many different data sets, analyse their content and write reports. To stay organised, you need a way to place files on your computer so that it is easy to link data with code. The recommended practice is to work with RStudio projects. You can think of an RStudio project as a directory on your computer with additional features that help you organize your workflow. You start a new project by going to the menu item ‘File’ → ‘New Project’. A dialogue window with three options appears: ‘New Directory’, ‘Existing Directory’ and ‘Version Control’ (figure 2.5). If you are familiar with a version control system (e.g. Git or Subversion), you can test out the corresponding option in the dialogue box. Version control systems are highly useful, but we would have to veer too far away from the main topic of this book to cover them in depth. If you have not worked with version control systems before, please choose either ‘Existing Directory’ or ‘New Directory’ followed by ‘New Project’.

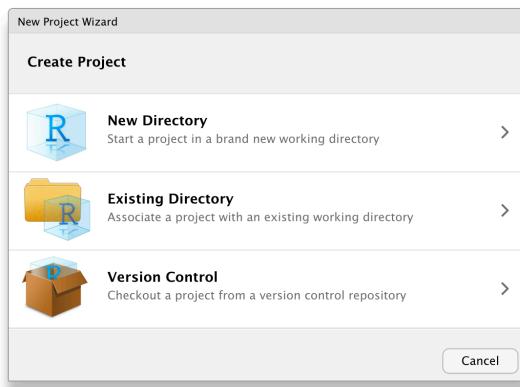


FIGURE 2.5: Pop-up window that appears when you start a new project.

After you created a project, RStudio changes the working directory to the project directory so that you can conveniently access data and R scripts in this directory without having to manually change directories. RStudio also placed a file with the extension `.Rproj` in the project directory. When you open this file (e.g. by searching for it with Spotlight on a Mac or similar tools on Windows or Linux), RStudio automatically starts a new session with the project directory as your working directory. If RStudio is already open, but you are in another directory, you can resume a previous project with the menu item ‘File’ → ‘Recent Projects’. If RStudio does not consider the project to be recent any longer, you can use the more general option ‘Open Project’.

I strongly recommend you make RStudio projects a regular part of your workflow. They help you stay organised. As a rule of thumb, whenever you work on a new data set, you create a new project dedicated to these data. The project folder is where you store the data files and save all R scripts that have to do with these data. Do not hesitate to create many small projects. Projects with only a few files are much easier to navigate than projects in which many unrelated files were dumped into one and the same directory.

2.6 Summary and outlook

In this chapter, you started exploring R and RStudio. The combination of both software products can create an efficient workflow for data analysis and visualisation. You took your first steps towards learning R and RStudio by typing commands into the R console. So far, R (with RStudio as its front end) may appear to be nothing but a luxury version of a graphing calculator. However, R's real strength is to automate more complicated tasks than those you have just seen. To take full advantage of R, you need to represent data in a way that R understands. In the next chapter, you learn about the most important tool for representing data in R: a data structure called 'vector'.

2.7 Just checking

Find the correct answer option for each of the following questions. You can confirm your answers by looking at appendix A.1.

- (I) What is one of the reasons for the popularity of R among statisticians and data scientists?
 - (a) R is quick and uses memory efficiently because it is a compiled language.
 - (b) R runs on all common operating systems (i.e. Windows, MacOS, Linux).
 - (c) R has so many pre-installed statistical functions that it is unnecessary and uncommon to load additional packages.
 - (d) There are three competing manufacturers who sell different versions of R: (i) The R Consortium, (ii) RStudio, (iii) RCommander. Because they compete against each other, they must produce an efficient and stable product.
- (II) What is RStudio?
 - (a) RStudio is one of three competing manufacturers of R.

- (b) RStudio is a special version of R that integrates features of VisualStudio.
- (c) RStudio is an integrated development environment for R.
- (d) RStudio is an R library for data visualisation.
- (III) What is the purpose of the left RStudio pane (highlighted in figure 2.6)?
- (a) Here we can type R commands and see their output.
- (b) This pane shows a list of all objects in our current working directory.
- (c) In this pane, RStudio lists all additional software packages that we can download from CRAN (the Comprehensive R Archive Network).
- (d) RStudio displays plots in this pane.

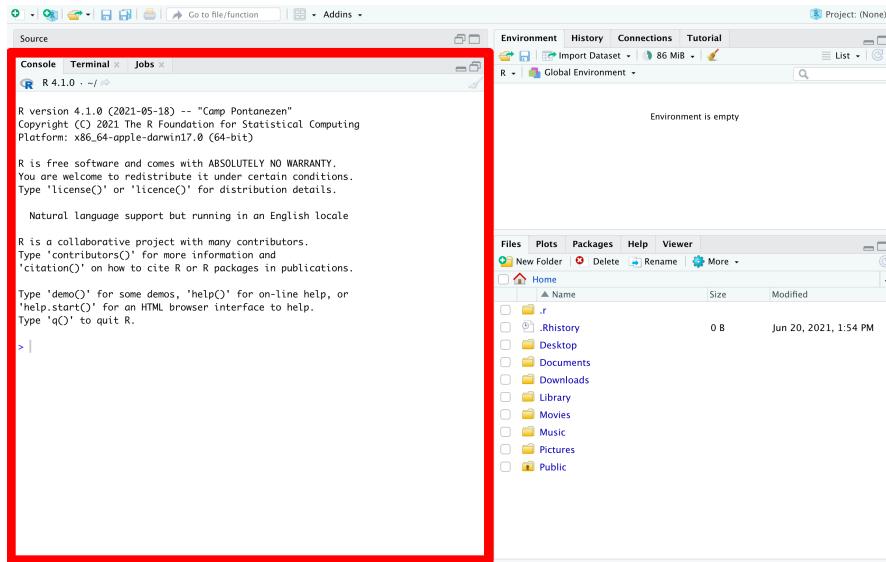


FIGURE 2.6: RStudio interface with left pane highlighted in red.

- (IV) What is the meaning of the + symbols in the following console input?

```
> plot(
+   cos,
+   -2 * pi,
+   2 * pi
+ )
```

- (a) The first + symbol indicates that -2π is added to the cosine

function. The second + symbol indicates that 2π is added afterwards.

- (b) The + symbols indicate that the command is still incomplete at the beginning of the second and third line of input.
- (c) The + symbol warns the user that there is an error on the previous line.
- (d) The + symbol converts the following input (e.g. `-2 * pi`) from characters to numbers.

3

Vectors

The most basic data structure in R is a *vector*, a combination of elements that are all of the same class. For example, all elements in a vector can be numbers, or all elements can be logical values (i.e. `TRUE` or `FALSE`). By the end of this chapter, you know how to create vectors and find out basic information about a vector (e.g. the number of its elements and their class). We also learn how to access and change elements in a vector.

3.1 Elementary operations with vectors

3.1.1 Combining elements with `c()`

We can create a vector with the function `c()`, which stands for ‘combine’. Here is an example of a *numeric* vector. (Please remember that `>` at the start of the line is the command prompt. You do not need to type `>`.)

```
> c(31, -50, 93, 29, -44, 93)
## [1] 31 -50 93 29 -44 93
```

The `[1]` at the start of the output line indicates that the next value is the *first* element in the vector. Later, when we work with longer vectors, it becomes apparent why the number in square brackets is useful (section 3.3).

3.1.2 Assignment operator `<-`

If we need the values in a vector several times during a calculation, it becomes tedious to repeat typing all elements every time. Instead, we can save a vector (and any of the other R objects we encounter in later chapters) in a *variable*. In the following example, R assigns the vector to the variable `v`.

```
> v <- c(31, -50, 93, 29, -44, 93)
```

The symbol `<-` (composed of a left angle bracket and a hyphen) is called the

assignment operator. Instead of literally typing `<-` on the keyboard, we can use the shortcut ‘Option and -’ (Mac) or ‘Alt and -’ (Windows and Linux). The assignment operator is often pronounced ‘gets’. In this example, `v` gets the combination of numbers on the right-hand side.

Many other programming languages use the equals sign `=` as assignment operator. R would also understand what we mean if we replaced `<-` by `=`, but the use of `=` as assignment operator in R is considered to be bad style. The most important style guide for R is the ‘tidyverse style guide’ at <https://style.tidyverse.org/>. It recommends using `<-` instead of `=` (<https://style.tidyverse.org/syntax.html#assignment-1>). In general, it is good practice to adopt a coding style that is consistent with common professional standards. Although the code may still work if we do not follow a consistent style, it will be more difficult to read. We can compare the situation to writing a business letter: the content may still be understandable if we ignore consistent spelling or punctuation styles, but it looks unprofessional.

Also a question of good style in R are the spaces before and after `<-` as well as spaces after the commas. In principle, the following three lines of code have the same effect.

```
> v <- c(31, -50, 93, 29, -44, 93)
> v<-c(31, -50 ,93, 29, -44,93)
> v<- c ( 31,-50,93,29,-44,93 )
```

Spaces in R do not influence the calculation as long as we do not insert spaces inside operators (e.g. `< -` with a space in the middle means something different from `<-`). However, the tidyverse style guide recommends only the first of the three options above (see <https://style.tidyverse.org/syntax.html#spacing>):

- one space before and another space after the assignment operator `<-`,
- one space *after* each comma,
- no spaces *before* a comma,
- no spaces before or after an opening parenthesis in a function call (here we call the function `c()`),
- no space before a closing parenthesis.

In section 6.6, I explain how to use the `styler` package that helps us to adhere to the tidyverse style.

3.1.3 The class of a vector

We can check the content of a variable by typing the variable name at the console.

```
> v
## [1] 31 -50 93 29 -44 93
```

Different vectors can be of different classes, depending on the data that are stored in the vector (e.g. numbers or character strings). We can find out the class of a vector with the function `class()`.

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

The class `numeric` is a general storage format for all numbers, whether positive (e.g. 31), negative (e.g. -50), integer or floating-point (e.g. 31.4).

We can confirm that `v` is numeric with `is.numeric()`.

```
> is.numeric(v)
## [1] TRUE
```

Another way to find out that `v` contains numbers is to look at the environment tab (see figure 2.3 for a reminder where to find the environment tab inside the RStudio window). To the left of the variable name `v`, RStudio displays the class in abbreviated form: `num` stands for `numeric` (figure 3.1).

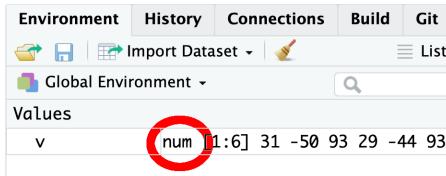


FIGURE 3.1: RStudio displays an abbreviation of a vector's class in the environment tab.

3.1.4 Integer vectors

Every element of `v` defined in section 3.1.2 has an integer value. From other programming languages, you might already know that computers can store integers in a special `integer` data type, saving some memory because computers can represent integers with fewer bits than floating-point numbers.

R's `numeric` class, however, does not make a distinction between integers and floating-point numbers. When we ask R whether `v` is an integer vector, R's answer is no because it assumes that our question is about the data type, not the value of the numbers.

```
> is.integer(v)
## [1] FALSE
```

In rare cases, it might be important to force R to treat numbers as integers. We tell R that numbers in a vector must be stored as integers by adding an `L` after each number.¹

```
> w <- c(31L, -50L, 93L, 29L, -44L, 93L)
> class(w)
## [1] "integer"
```

We can confirm that `w` is an integer vector with `is.integer()`.

```
> is.integer(w)
## [1] TRUE
```

We can see the difference between `numeric` and `integer` in the environment tab too (figure 3.2).

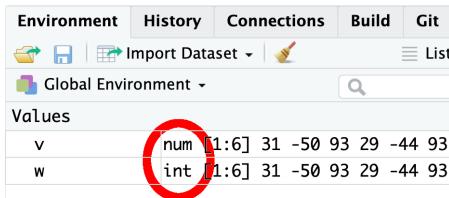


FIGURE 3.2: The abbreviations `num` and `int` in the environment tab show that `v` and `w` are of different classes: `numeric` versus `integer`.

When we run `is.numeric(w)`, the result is also TRUE.

```
> is.numeric(w)
## [1] TRUE
```

In conclusion, all `integer` vectors are `numeric`, but not all `numeric` vectors belong to the `integer` class.

3.1.5 Character vectors

So far, we have only seen examples in which a vector contains numbers, but vectors can also store non-numeric data. For example, the elements can be

¹The reason why R uses the suffix `L` for integers is shrouded in mystery (see <https://hypatia.math.ethz.ch/pipermail/r-devel/2017-June/074462.html>).

character strings. In computer jargon, a character string is a sequence of characters (i.e. combinations of letters, numbers and special symbols like & or \$). In R, character strings are entered between a pair of double quotes ". We can in principle enclose strings in single quotes ' too, but double quotes are stylistically preferred (<https://style.tidyverse.org/syntax.html#quotes>).

```
> mrt <- c("North South Line", "East West Line", "Circle Line", "Downtown Line")
```

For easier readability, we may want to spread such long commands over multiple lines and align the vector elements. At the end of each line, we hit the return key. As we learned in section 2.4, the R console automatically adds a + at the start of the line when it detects that the command on the preceding lines is not yet complete.

```
> mrt <- c(
+   "North South Line",
+   "East West Line",
+   "Circle Line",
+   "Downtown Line"
+ )
```

The class of a vector containing character strings is `character`.

```
> class(mrt)
## [1] "character"
```

We can confirm that `mrt` is a character vector with `is.character()`.

```
> is.character(mrt)
## [1] TRUE
```

In the environment tab, the class is abbreviated as `chr`.

There are two built-in character vectors that are frequently used: `letters` and `LETTERS`. They contain all lower-case and all upper-case letters, respectively, of the English alphabet.

```
> letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
> LETTERS
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

3.1.6 Logical vectors

A logical vector contains only elements that are `TRUE` or `FALSE`.² As before, we can use `c()` to combine multiple logical elements into a vector.

```
> lgcl <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
> class(lgcl)
## [1] "logical"
> is.logical(lgcl)
## [1] TRUE
```

In the environment tab, the class name `logical` is abbreviated as `logi`.

Because `TRUE` and `FALSE` are important keywords, R does not allow anyone to use them as variable names. If we try to assign any value to `TRUE` or `FALSE`, R spits out an error message.³

```
> TRUE <- 42
## Error in TRUE <- 42: invalid (do_set) left-hand side to assignment
```

When we start a fresh R session, `T` and `F` are synonyms of `TRUE` and `FALSE`.

```
> T
## [1] TRUE
> F
## [1] FALSE
```

Unlike `TRUE` and `FALSE`, neither `T` nor `F` are reserved keywords; thus, it is possible to use them as variable names.

```
> T <- 42
> T
## [1] 42
```

After assigning any value to `T` other than `TRUE`, R no longer treats `T` as a synonym of `TRUE`. This feature can be a source of errors that are difficult to detect. As a precaution and as a matter of good R coding style, let us

- always use the long forms `TRUE` and `FALSE` instead of `T` and `F` (<https://style.tidyverse.org/syntax.html#data>).
- only use `T` and `F` as variable names when there is no other sensible alternative.

²As we learn in section 11.1, logical vectors can also contain missing values in the form of `NA` (*Not Assigned*).

³`TRUE` and `FALSE` are not the only reserved words in R. We can find a comprehensive list by running the command `?Reserved` in the console.

One of the main applications of logical vectors is subsetting of other, not necessarily logical, vectors (section 3.3.2).⁴

For the sake of completeness, we mention in passing that—besides the classes `numeric`, `integer`, `character` and `logical`—there are two more classes in R: `complex` and `raw`. The class `complex` is used when we must perform arithmetic operations with complex numbers (i.e. numbers involving multiples of the imaginary unit i with $i^2 = -1$). Vectors of the type `raw` are used for storing bytes of data. In this book, we neither work with `complex` nor `raw` vectors; thus, we skip the details here.

3.1.7 All elements in a vector must belong to the same class

Although different vectors can be of different types, it is impossible to have different classes combined into one and the same vector.⁵ For example, if a vector belongs to the class `numeric`, then *all* of its elements are numbers. If another vector has the class `character`, then *all* of its elements are character strings. It is not possible to mix numeric and character objects within a single vector. Later, we learn about other data structures (data frames, tibbles and lists) that can combine objects of all classes, but vectors are not suitable for this task. We learn in chapter 10 how R coerces elements of different classes into a single class if we insist on combining them into one vector. For the time being, let us refrain from mixing classes. The world will not come to an end, but the result may not be what you expect.

3.1.8 The length of a vector

We can find out how many elements are in a vector with the function `length()`.

```
> length(v)
## [1] 6
```

The length of a vector is shown in the environment tab too (figure 3.3).

R treats even a single element as a vector, namely a vector that happens to have length 1.

⁴In programming jargon, ‘subset’ is frequently used as a verb (e.g. in <http://adv-r.had.co.nz/Subsetting.html>). I adopt this convention even if grammar purists may find it intolerable (see the discussion at <https://english.stackexchange.com/questions/297323/simple-past-of-the-verb-subset>).

⁵Technically, it is possible that all elements of a vector belong to more than one class (see <https://stackoverflow.com/questions/19335914/can-an-object-in-r-have-more-than-one-class>). However, it is still correct that if *any* element in a vector is in a certain class, then *all* elements must belong to this class.

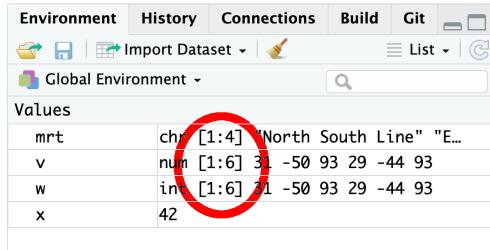


FIGURE 3.3: In the environment tab, `[1:4]` indicates that the elements have indices from 1 to 4 (i.e. the length of the vector is 4; note that indices in R start from 1, not 0). For vectors of length 1 (e.g. `x` in the depicted example), RStudio does not explicitly show the length, but it is obvious from the listed values that there is only one element in `x`.

```
> x <- 42
> length(x)
## [1] 1
```

Note that we do not need to (and stylistically should not) create a vector of length 1 with `c()`. Thus, the following code is bad style although it has the same effect as `x <- 42` above.

```
> # Bad style. c() is superfluous.
> x <- c(42)
```

In the previous code chunk, I used the hash symbol `#` to add a comment. In general, the R interpreter ignores everything that follows on the same line after a hash symbol.

3.2 Shortcuts for generating commonly needed vectors

The function `c()` is the most general way of creating vectors (section 3.1.1), but, for common special cases, R knows some shortcuts.

3.2.1 The colon operator :

If we want a sequence of consecutive integers, we can use the colon (`:`) operator.

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

The syntax also works for decreasing sequences.

```
> 4:-3
## [1] 4 3 2 1 0 -1 -2 -3
```

The colon operator is one of only few in-fix operators (i.e. operators with one operand on the left and another on the right) for which it is stylistically preferred not to insert spaces (<https://style.tidyverse.org/syntax.html#spacing>).

3.2.2 Use `seq()` for general sequences

For sequences with step sizes $\neq 1$ between consecutive elements, we can use `seq()`. It takes three arguments (i.e. values inside the parentheses that are separated by commas). The first two numbers are start and end points of the sequence. The third argument can be either the step size ...

```
> seq(1.0, 3.2, by = 0.4)
## [1] 1.0 1.4 1.8 2.2 2.6 3.0
```

... or the length of the sequence ...

```
> seq(2.4, 7.3, length.out = 6)
## [1] 2.40 3.38 4.36 5.34 6.32 7.30
```

... or another vector whose length we would like to imitate.

```
> seq(2.4, 7.3, along.with = 11:16)
## [1] 2.40 3.38 4.36 5.34 6.32 7.30
```

In these examples, we used the syntax ‘`variable.name = value`’ in the third arguments of `seq()` (e.g. ‘`along.with = 11:16`’). For right now, let us just accept the syntax above as a recipe to build sequences. The rules behind this syntax make more sense after we learned about function arguments in chapter 5.

3.2.3 Use `rep()` for vectors with repeating elements

If we want to create a vector with many repeating elements, we can use `rep()`. It takes two arguments. The first argument consists of the elements we would

like to repeat. The second argument indicates either how often we want to repeat the complete first argument ...

```
> rep(-3:-5, 4)
## [1] -3 -4 -5 -3 -4 -5 -3 -4 -5 -3 -4 -5
```

... or how often we want to repeat each element before moving on to the next.

```
> rep(-3:-5, each = 4)
## [1] -3 -3 -3 -3 -4 -4 -4 -4 -5 -5 -5 -5
```

3.3 Extracting, removing, replacing and inserting vector elements

R uses square brackets to identify the *index* of a vector element. For example, when we deal with a long vector, the index of the first element on a line of console output is shown in square brackets.

```
> letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

In this example, [1] on the first line implies that "a" is the first element. In R, the first element has index 1, not 0 as in many other languages (e.g. C, Python or JavaScript). [20] on the second line means that "t" is the 20th element.

As we learn next, square brackets are also often used when we try to extract, remove, replace or insert elements from a vector.

3.3.1 Extracting vector elements

If we want to find the 8th letter, we use the square bracket operator as follows.

```
> letters[8]
## [1] "h"
```

If we need more than one element, we can insert a numeric vector into the square brackets.

```
> letters[1:8]
## [1] "a" "b" "c" "d" "e" "f" "g" "h"
> letters[c(2, 8, 4, 4)]
## [1] "b" "h" "d" "d"
```

3.3.2 Subsetting with logical vectors

There is an alternative to extracting vector elements with numeric vectors in square brackets; we can subset with logical vectors too. For simplicity, let us take only the first 5 letters of the alphabet and define a logical vector with 5 elements.

```
> a2e <- letters[1:5]
> y <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
```

When we subset `a2e` by `y`, we only get those indices from `a2e` whose value in `y` is `TRUE`.

```
> a2e[y]
## [1] "a" "b" "e"
```

Subsetting with logical vectors may seem indirect at first glance, but it often turns out to be easier than subsetting with numeric indices.

3.3.3 Removing vector elements with a minus sign

In R, there is a convenient way to remove elements from a vector: prepend a minus sign in front of the indices that we want to drop. For example, the following object returns a vector that contains all except the 4th letter (i.e. `d`).

```
> a2e[-4]
## [1] "a" "b" "c" "e"
```

Note that `a2e` still contains `d` after running `a2e[-4]`. If you want to permanently remove the fourth element, you would have to assign the result of `a2e[-4]` back to `a2e` (i.e. `a2e <- a2e[-4]`).

If we want to remove more than one element, we use a minus sign in front of a vector that specifies the set of indices to be removed.

```
> a2e[-c(1, 4)]
## [1] "b" "c" "e"
```

3.3.4 Replacing vector elements

If we want to replace the 4th letter in `a2e` by "I am new!", we use the assignment operator `<-` as follows.

```
> a2e[4] <- "I am new!"
> a2e
## [1] "a"      "b"      "c"      "I am new!" "e"
```

We can also replace multiple elements at a time.

```
> a2e[1:3] <- c("First", "Second", "Third")
> a2e
## [1] "First"    "Second"   "Third"    "I am new!" "e"
```

3.3.5 Inserting vector elements

We can append an element to a vector by inserting it immediately after the last index.

```
> a2e[length(a2e) + 1] <- "f"
> a2e
## [1] "First"    "Second"   "Third"    "I am new!" "e"       "f"
```

If we insert a new element at a higher index, R inserts `NA` in all indices that have not been filled. `NA` stands for *Not Assigned*. We talk more about the meaning of `NA` in section 11.1.

```
> a2e[9] <- "g"
> a2e
## [1] "First"    "Second"   "Third"    "I am new!" "e"       "f"
## [7] NA        NA        "g"
```

A user-friendly alternative to juggling with indices is to combine additional elements with `c()`.

```
> a2e <- letters[1:5]
> c(a2e, "f", "g")
## [1] "a" "b" "c" "d" "e" "f" "g"
```

We learn from this example that the arguments in `c()` do not need to be individual elements; they can also be vectors of length greater than 1 (e.g. `a2e`).

In contrast to the two preceding code chunks, this latest code chunk does not

change `a2e`: it still consists of only five letters (i.e. `a2e` neither contains "`f`" nor "`g`").

```
> a2e
## [1] "a" "b" "c" "d" "e"
```

If we want to replace `a2e` by the combination `c(a2e, "f", "g")`, we must use the assignment operator `<-`.

```
> a2e <- c(a2e, "f", "g")
> a2e
## [1] "a" "b" "c" "d" "e" "f" "g"
```

A slightly more verbose alternative to `c()` is the function `append()`.

```
> a2e <- letters[1:5]
> append(a2e, c("f", "g"))
## [1] "a" "b" "c" "d" "e" "f" "g"
```

The advantage of `append()` is that it also allows us to insert new elements between existing elements.

```
> append(a2e, c("something", "in", "between"), after = 2)
## [1] "a" "b" "something" "in" "between" "c"
## [7] "d" "e"
```

3.4 Numeric operations with R

After we store numbers in a vector, we often want to perform some calculation with them.

3.4.1 Basic arithmetic operations

Table 3.1 lists the most common arithmetic operations.

Numeric vectors of equal length can be added and subtracted in an intuitive way. (We learn in section 3.5 what happens if the vectors are of unequal length.)

TABLE 3.1: Common arithmetic operators in R. Note that integer division $\%/\%$ has a forward slash / between the percent signs, whereas mod $\%\%$ does not contain a forward slash.

Operator	Description
$x + y$	sum of x and y
$x - y$	y subtracted from x
$x * y$	x multiplied by y
x / y	x divided by y
x^y or $x**y$	x raised to the power y
$x \%/\% y$	integer division (e.g. $7 \%/\% 3 = 2$)
$x \%\% y$	x mod y (i.e. remainder after integer division of x by y). Example: $7 \%\% 3 = 1$

```
> v1 <- c(9, 5, 4, 5)
> v2 <- c(4, 9, 3, 6)
> v1 + v2
## [1] 13 14  7 11
> v1 - v2
## [1] 5 -4  1 -1
```

Multiplication with $*$ is also elementwise.

```
> v1 * v2
## [1] 36 45 12 30
```

Similarly, when we apply any of the operators in table 3.1 to two equally long vectors, R will perform the operation elementwise.

These are our first examples of *vectorisation* in R. An operator or a function is vectorised if it can accept one or more vectors with multiple elements as input, performs the same operation on each element and returns an equally long vector as output. Taking advantage of vectorisation is a sign of well-written R code. We return to this point many times in later chapters.

3.4.2 Mathematical functions

Besides the basic arithmetic operations shown in section 3.4.1, R can carry out a large variety of mathematical functions. Table 3.2 lists a selection of the available functions.

TABLE 3.2: Selection of mathematical functions in R

	Description
<code>abs(x)</code>	Absolute value
<code>exp(x)</code>	Exponential function
<code>factorial(x)</code>	Factorial
<code>log(x)</code>	Natural logarithm
<code>log2(x)</code>	Binary logarithm
<code>log10(x)</code>	Logarithm with base 10
<code>log(x, base = y)</code>	Logarithm with base y
<code>sqrt(x)</code>	Square root
<code>cos(x), sin(x), tan(x)</code>	Trigonometric functions
<code>acos(x), asin(x), atan(x)</code>	Inverse trigonometric functions
<code>cosh(x), sinh(x), tanh(x)</code>	Hyperbolic functions
<code>acosh(x), asinh(x), atanh(x)</code>	Inverse hyperbolic functions

Like the basic arithmetic operators of section 3.4.1, all of these mathematical functions are vectorised: if we pass arguments with multiple elements to a function, the result is the function applied to each element.

```
> exp(v1)
## [1] 8103.08393 148.41316 54.59815 148.41316
> sqrt(v1)
## [1] 3.000000 2.236068 2.000000 2.236068
```

If a function accepts vectors with multiple elements in more than one argument, it carries out the calculation by going through the vector indices in parallel.

```
> log(v1, base = v2)
## [1] 1.5849625 0.7324868 1.2618595 0.8982444
```

The first element of the output is $\log_4(9)$ (i.e. the logarithm of the first element of $v1$ to the base given by the first element of $v2$). The second element is $\log_9(5)$, followed by $\log_3(4)$ etc.

3.4.3 Summary statistics

Summary statistics are numbers that contain important information about a data set.



"Autosum aside, these numbers just don't add up."

FIGURE 3.4: The R function `sum()` performs the function that is called Autosum in many spreadsheet programs. Cartoon by [Anderson \(2012\)](#).

3.4.3.1 Basic summary statistics

We have already seen an example of a summary statistic: the length of a vector.

```
> length(rep(c(3, 5, 2), 4))
## [1] 12
```

Another elementary summary statistic is the sum of all vector elements (figure 3.4).

```
> sum(1:7)
## [1] 28
```

A similar statistic is the product of all elements.

```
> prod(seq(0.5, 2, by = 0.5))
## [1] 1.5
```

The mean of a vector (v_1, \dots, v_n) , defined as $\bar{v} = (\sum_{i=1}^n v_i) / n$, can, in principle, be computed as `sum(v) / length(v)`, but it is easier and more readable to use `mean(v)`.

```
> mean(c(9, -15, -9, 3, 17))
## [1] 1
```

There are similar shortcuts for the variance $\text{var}(v_1, \dots, v_n) = (\sum_{i=1}^n (v_i - \bar{v})^2) / (n - 1)$ and the standard deviation $\sqrt{\text{var}}$.

```
> var(c(9, -15, -9, 3, 17))
## [1] 170
> sd(c(9, -15, -9, 3, 17))
## [1] 13.0384
```

The median of a vector is the number that divides the smaller half of the elements from the larger half. We can find the median with the R function `median()`.

```
> median(c(9, -15, -9, 3, 17))
## [1] 3
```

In this example, the median is 3 because there are two smaller (-15, -9) and two larger (9, 17) elements.

If the input vector is of even length $2n$, R returns the number halfway between the n -th and $(n + 1)$ -th largest elements.

```
> median(c(9, -15, -9, 3, 17, 6))
## [1] 4.5
```

3.4.3.2 Minimum and maximum

We obtain the minimum with `min()` and the maximum with `max()`.

```
> min(c(9, -15, -9, 3, 17))
## [1] -15
> max(c(9, -15, -9, 3, 17))
## [1] 17
```

If the argument is a character vector, then `min()` returns the first element in alphabetical order, and `max()` returns the alphabetically last element.

```
> min(c("mike", "papa", "yankee", "india"))
## [1] "india"
> max(c("mike", "papa", "yankee", "india"))
## [1] "yankee"
```

`range()` returns the minimum and maximum in a single vector.

```
> range(c(-10:-8, -5:5, 10:12))
## [1] -10 12
```

```
> range(c("r", "a", "n", "g", "e"))
## [1] "a"  "r"
```

3.4.3.3 Quantiles and related summary statistics

The q -quantile of a vector v is a number x such that a fraction q of v 's elements are less than x . R computes the q -quantile with `quantile(v, q)`.⁶

```
> quantile(c(9, -15, -9, 3, 17, 6), 0.2)
## 20%
## -9
```

The second argument can be a vector.

```
> quantile(c(9, -15, -9, 3, 17, 6), c(0.2, 0.6))
## 20% 60%
## -9    6
```

If we do not specify the second argument, R returns the 0.00-quantile, 0.25-quantile, 0.50-quantile, 0.75-quantile and 1-quantile (also known as minimum, lower quartile, median, upper quartile and maximum).

```
> quantile(c(9, -15, -9, 3, 17, 6))
##      0%     25%     50%     75%   100%
## -15.00  -6.00   4.50   8.25  17.00
```

The combination of minimum, lower quartile, median, upper quartile and maximum is called the *Tukey five-number summary*.⁷ An alternative to the Tukey five-number summary is the six-number summary we obtain from `summary()`. In addition to Tukey's five numbers, `summary()` also returns the mean.

```
> summary(c(9, -15, -9, 3, 17, 6))
##   Min. 1st Qu. Median Mean 3rd Qu. Max.
## -15.00 -6.00  4.500  1.833  8.250 17.00
```

⁶If q is not a multiple of $1 / (\text{length}(v) - 1)$, the quantile is not uniquely defined (see the section 'Types' at <https://www.rdocumentation.org/packages/stats/versions/3.5.1/topics/quantile>). The small differences between the various types of quantiles rarely matter in practice; thus, we do not dwell on the definitions here.

⁷There is a function `fivenum()` whose result is essentially the same as that of `quantile()` with one small difference: the Tukey five-number summary contains the lower and upper *hinges* instead of the lower and upper *quartiles*. See for example <http://statisticsbypeter.blogspot.sg/2014/05/quantiles-fractiles-quartiles-hinges.html>.

The interquartile range is the difference between the upper and lower quartile. R computes the interquartile range with `IQR()`.

```
> IQR(c(9, -15, -9, 3, 17, 6))
## [1] 14.25
```

Another useful way to summarise elements in a vector, especially when it consists of many repetitions of only a few distinct values, is `table()`.

```
> table(c(5, -8, -2, -2, 5, -2, 5, -2, -8, -2))
##
## -8 -2  5
##  2  5  3
```

The output means that -8 appears twice, -2 five times and 5 three times in the argument passed to `table()`.

3.5 Vector recycling

What happens when an arithmetic operator or a function receives two vectors of unequal length as input? Let us see whether we can spot a pattern.

```
> w1 <- c(2, 4)
> w2 <- 1:6
> w1 + w2
## [1]  3  6  5  8  7 10
> w1 * w2
## [1]  2  8  6 16 10 24
```

If input vectors are not equally long, the result of the calculation is a vector with the same length as the longest input vector. R ‘recycles’ shorter vectors until their length matches that of the longest one.

In our example, `w1` has length 2 and is, thus, shorter than `w2` whose length is 6. Table 3.3 shows how recycling proceeds in this case.

In the previous example, the length of the longer vector is an integer multiple of the shorter length. What happens if the vectors are of incommensurable length? R produces a result, but it comes with a warning attached.

```
> x1 <- c(2, 4)
> x2 <- 1:7
```

TABLE 3.3: Illustration of vector recycling rules

w1	w1 recycled	w2	w1 + w2	w1 * w2
2	2	1	3	2
4	4	2	6	8
	2	3	5	6
	4	4	8	16
	2	5	7	10
	4	6	10	24

TABLE 3.4: Illustration of fractional vector recycling

x1	x1 recycled	x2	x1 + x2
2	2	1	3
4	4	2	6
	2	3	5
	4	4	8
	2	5	7
	4	6	10
	2	7	9

```
> x1 + x2
## Warning in x1 + x2: longer object length is not a multiple of shorter object
## length
## [1] 3 6 5 8 7 10 9
```

The numbers of the previous addition are computed according to table 3.4. During the last cycle, `x1` is only partially repeated so that it is just long enough to match the length of `x2`. Such fractional recycling is usually a sign that something went wrong. Therefore, R issues a warning.

By the way, we can always retrieve the latest warning message with `warnings()`. This feature can be useful when debugging code.

```
> warnings()

## Warning in x1 + x2: longer object length is not a multiple of shorter object
## length
```

If we really understand the consequences, we can, in principle, turn off the warning.

```
> suppressWarnings(x1 + x2)
## [1] 3 6 5 8 7 10 9
```

However, it is almost always bad coding style to suppress a warning. Instead, our code should handle special cases so that they do not trigger warnings in the first place. In our example, if we really intend to do addition with vectors of incommensurable length, we should break down the addition into pieces that avoid fractional recycling. In practice, however, fractional vector recycling usually indicates a mistake in the data, so displaying the default warning message is probably what we really want here.

3.6 Unique and duplicated values

Often, we want to determine whether values in a vector appear more than once. In some cases, a repeated value may indicate an error that occurred during the data collection, thus, we may want to remove the repetition. In other cases, the frequency with which a value occurs might be an insightful summary statistic of our data.

Here are, for example, the Australian Open women's singles tennis champions between 2010 and 2018 in chronological order.

```
> winner <- c(
+   "Serena Williams",
+   "Kim Clijsters",
+   "Victoria Azarenka",
+   "Victoria Azarenka",
+   "Li Na",
+   "Serena Williams",
+   "Angelique Kerber",
+   "Serena Williams",
+   "Caroline Wozniacki"
+ )
```

We already learned in section 3.4.3.3 that we can use `table()` to find out how often a value appears in a vector.

```
> table(winner)
## winner
## Angelique Kerber Caroline Wozniacki      Kim Clijsters      Li Na
##                      1                      1                      1                      1
```

```
##   Serena Williams  Victoria Azarenka
##                 3                  2
```

The output of `table()` reveals which values in `winner` are unique or repeated. However, there are also more specialized functions for this purpose: `unique()` and `duplicated()`.

3.6.1 `unique()` and `duplicated()`

When we want to obtain a vector that excludes any duplicated values, we use `unique()`.

```
> unique(winner)
## [1] "Serena Williams"      "Kim Clijsters"       "Victoria Azarenka"
## [4] "Li Na"                  "Angelique Kerber"    "Caroline Wozniacki"
```

This vector contains, for example, `Serena Williams` only once, although she appears three times in `winner`. Thanks to `unique()`, we can easily determine how many different players won the tournament between 2010 and 2018.

```
> length(unique(winner))
## [1] 6
```

Which of the players won the Australian Open more than once between 2010 and 2018? We can find it out with the help of `duplicated()`.

```
> duplicated(winner)
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

This output means that the 4th, 6th and 8th element in `winner` (i.e. the values that are `TRUE`) are duplicates of earlier elements. We find multiple champions as follows.

```
> d <- duplicated(winner)
> winner[d]
## [1] "Victoria Azarenka" "Serena Williams"   "Serena Williams"
```

3.7 `any()` and `all()`

There are two strategies to find out whether a vector, say `v`, contains duplicated values.

- (1) We can ask whether the length of `v` matches the length of `unique(v)`. It is `TRUE` if and only if there are *no* duplicates.

```
> v <- c(6, 0, 2, 1)
> length(v) == length(unique(v))
## [1] TRUE
> length(winner) == length(unique(winner))
## [1] FALSE
```

- (2) We can ask whether `duplicated(v)` contains any elements that are `TRUE`. We can use the function `any()` to find the answer. `any()` accepts a logical vector, say `x`, as argument and returns `TRUE` if and only if at least one element in `x` is `TRUE`.

```
> any(duplicated(v))
## [1] FALSE
> any(duplicated(winner))
## [1] TRUE
```

Because the concatenation of `any()` and `duplicated()` is so common, R possesses a function to replace this combination: `anyDuplicated()`. In general, `anyDuplicated()` has slightly shorter run-times than `any(duplicated())`.

There is one small difference between `any(duplicated(x))` and `anyDuplicated(x)`.

- `any(duplicated(x))` returns `TRUE` or `FALSE`.
- If there are duplicated elements in `x`, `anyDuplicated(x)` returns the index `i` of the first duplication `x[i]`. If there is no duplication in `x`, `anyDuplicated(x)` returns 0.

```
> anyDuplicated(v)
## [1] 0
```

```
> anyDuplicated(winner)
## [1] 4
```

The counterpart to `any()` is `all()`, which returns `TRUE` if and only if all elements in the argument are `TRUE`.

```
> all(c(TRUE, TRUE, TRUE))
## [1] TRUE
> all(c(TRUE, TRUE, FALSE))
## [1] FALSE
```

The next code chunk uses `all()` to confirm that Serena Williams is not the only player who won more than once between 2010 and 2018.

```
> d <- duplicated(winner)
> all(winner[d] == "Serena Williams")
## [1] FALSE
```

However, it is true that the only players who won more than once are Serena Williams and Victoria Azarenka. We confirm this observation with the next command, where we use the `%in%` operator, which checks whether the value to its left is in the vector to its right.

```
> all(winner[d] %in% c("Serena Williams", "Victoria Azarenka"))
## [1] TRUE
```

3.8 Summary and outlook

In this chapter, we learned how to work with vectors (e.g. how to define them, how to access and change elements, and how to perform basic arithmetic operations). Vectors are the basic building blocks for more advanced data structures (e.g. tibbles and lists). Thus, we apply the knowledge gained in this chapter throughout the rest of this book. As you work more with R, vector operations will become second nature. If you feel overwhelmed, do not despair. This feeling is normal for newcomers to programming. Just read this chapter a second time to feel more confident in light of the upcoming challenges.

3.9 Just checking

Find the correct answer options for the following questions. First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.2.

- (I) What is the output at the end of the following R commands? (Here and elsewhere, the symbol > at the start of a line is the command prompt.)

```
> x <- seq(6, 2, length.out = 3)
> y <- 3:1
> x[x - y]
```

- (a) ## [1] 4 4 4
- (b) ## [1] 6 4 2
- (c) ## Error in x[y]: only 0's may be mixed with negative subscripts
- (d) ## [1] 2 4 6

- (II) What is the output at the end of the following R commands?

```
> u <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
> v <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
> u[v]
```

- (a) ## [1] TRUE FALSE
- (b) ## [1] TRUE TRUE TRUE TRUE FALSE
- (c) ## [1] TRUE FALSE FALSE FALSE FALSE
- (d) ## [1] TRUE FALSE FALSE

- (III) What is the output at the end of the following R commands?

```
> v <- 4
> w <- 3:4
> x <- 1:4
> v + w + x
```

- (a) ## [1] 8 10 10 12

- (b) `## Warning in v + w + x: longer object length is not a multiple of shorter object
length
[1] 8 10 10`
- (c) `## [1] 8`
- (d) `## [1] 8 10 10`
- (IV) Let us assume that `v` is a logical vector without any missing values (`NA`). Which of the following expressions must be `TRUE` regardless of the elements in `v`?
- (a) `any(v) != all(v)`
(b) `any(v) == all(!v)`
(c) `any(v) != all(!v)`
(d) `any(v) != any(!v)`

4

Getting help

Learning a programming language can feel daunting, and R is no exception. However, there is plenty of help available when working with R. In this chapter, we take a look at some of the most common resources that even experienced R programmers regularly consult for help.

4.1 R’s built-in documentation

The main source of information about R is its built-in documentation. For every R function and built-in data set, there is a help page. We access the help page either with `help()` or, even shorter, with `?` followed by the name of the function that we want to look up.

```
> # The next two lines are synonymous
> help(var)
> ?var
```

The help page opens in RStudio’s bottom right pane. Let us take a look at the information shown in this pane.

Each help page starts with a short *Description* of the function so that we can quickly decide whether the function might meet our needs (figure 4.1).

The second section on every help page is about *Usage* (figure 4.2). It shows which arguments the function accepts. For example, `var()` can take arguments named `x`, `y`, `na.rm` and `use`. We can obtain the same information with the `str()` function, where `str()` stands for ‘structure’.

```
> str(var)
## function (x, y = NULL, na.rm = FALSE, use)
```

The ‘Usage’ section also often lists related functions. In this example, `cor()`, `cov()` and `cov2cor()` appear in the list because they are all related to the concept of variance.

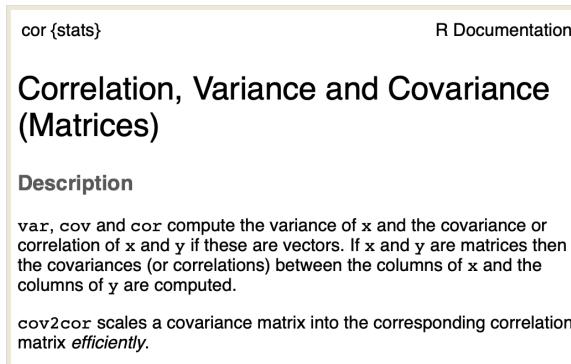


FIGURE 4.1: The top of R’s built-in documentation about `var()` gives a brief description of the function.

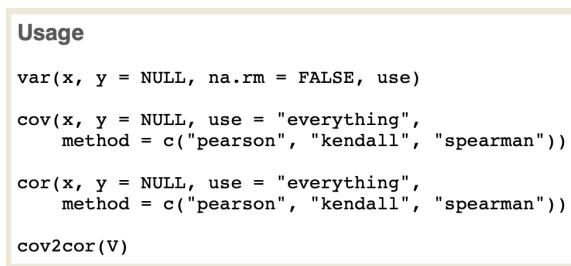


FIGURE 4.2: In the ‘Usage’ section of R’s built-in help pages, there is a summary of the arguments which we can pass to the functions.

The third section, *Arguments*, shows more information about the arguments (e.g. their data types and data structures). The text also indicates what the arguments mean (figure 4.3).

The fourth section, *Details*, gives additional explanation (figure 4.4). For example, we might have been wondering what `use` is supposed to mean. Here we can find an answer. The description might currently look cryptic, but it will soon make sense as we gain more experience with R.

The fifth section, *Value*, normally indicates the data type and structure of the function’s output. For example, the output might be a ‘length-one numeric vector’ (see `?max`) or a ‘list with class `htest`’ (see `?cor.test`).

The help page for `var()` is, in this respect, rather atypical because it does not state the data type and structure. Instead, the text informs readers about a change in a recent R update (figure 4.5).

An example often explains more than many words. For this reason, almost every built-in help document in R ends with examples. We can copy and

Arguments	
x	a numeric vector, matrix or data frame.
y	NULL (default) or a vector, matrix or data frame with compatible dimensions to x. The default is equivalent to <code>y = x</code> (but more efficient).
na.rm	logical. Should missing values be removed?
use	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings "everything", "all.obs", "complete.obs", "na.or.complete", or "pairwise.complete.obs".
method	a character string indicating which correlation coefficient (or covariance) is to be computed. One of "pearson" (default), "kendall", or "spearman": can be abbreviated.

FIGURE 4.3: The ‘Arguments’ section lists all arguments, their values and their meaning.

Details	
	...
	If <code>use</code> is "everything", <code>NAs</code> will propagate conceptually, i.e., a resulting value will be <code>NA</code> whenever one of its contributing observations is <code>NA</code> .
	If <code>use</code> is "all.obs", then the presence of missing observations will produce an error. If <code>use</code> is "complete.obs" then missing values are handled by casewise deletion (and if there are no complete cases, that gives an error).
	"na.or.complete" is the same unless there are no complete cases, that gives <code>NA</code> . Finally, if <code>use</code> has the value "pairwise.complete.obs" then the correlation or covariance between each pair of variables is computed using all complete pairs of observations on those variables. This can result in covariance or
	...

FIGURE 4.4: The ‘Details’ section goes into greater technical depth than previous sections.

Value	
	For <code>r <- cor(*, use = "all.obs")</code> , it is now guaranteed that <code>all(abs(r) <= 1)</code> .

FIGURE 4.5: Section titled ‘Values’.

paste individual lines to the console, or we can run all examples by typing `example(var)` into the console. Some examples can often be modified to suit our needs.

```
Examples
var(1:10) # 9.166667
var(1:5, 1:5) # 2.5
## Two simple vectors
cor(1:10, 2:11) # == 1
```

FIGURE 4.6: Examples at the end of a help page illuminate how a function can be used.

One disadvantage of `help()` and `?` is that we need to know the name of the function we are searching for. If we do not know it yet, we can use `help.search()` or, equivalently `??`. If we use `??` and the search term contains a space, we must enclose the search term in quotation marks.

```
> help.search("variance")
> ??variance
> ??"standard deviation"
```

After we type these commands, RStudio lists all pages in the built-in documentation that contain the phrase between the quotation marks.

4.2 Finding help on the World Wide Web

We can perform a search on the World Wide Web with `RSiteSearch()`.

```
> RSiteSearch("variance")
```

This command opens a browser window with search results found by the search engine <https://search.r-project.org/>.

In my opinion, a better search engine is <https://rseek.org/>. There is no pre-installed R command-line function for it, but we can, of course, directly type this URL into a web browser.

For more general searches, we can also resort to Google®.¹ This strategy is useful when our R code produces a cryptic error message. Copying and pasting

¹<https://www.google.com/>

the message text into Google often leads to pages that explain and solve the error.

If we cannot find a solution, it may help to post a question on public message boards. Among the relevant user forums are R-help² and Stack Overflow.³ R-help is a free, general mailing list about R that includes a message board for its subscribers. For questions about RStudio, you can post messages at <https://community.rstudio.com/>. Stack Overflow is a more general forum, not specifically about R or RStudio, but nevertheless with many active R users who share their knowledge. It takes a little bit of practice to ask good questions on public message boards, but the responses are often surprisingly quick.

4.3 Summary and outlook

No programmer knows everything about any computer language. R is no exception. Even experienced programmers search for help, using the methods I outlined in this chapter. Do not feel shy to search and ask for help. In fact, searching and asking for help is one of the best ways to learn a programming language.

4.4 Just checking

Here are four different R commands.

- (i) `help("linear model")`
- (ii) `?"linear model"`
- (iii) `??"linear model"`
- (iv) `help.search("linear model")`

Which of these commands can you use to find help about linear models?

- (a) (ii) and (iv), but neither (i) nor (iii).
- (b) (i) and (ii), but neither (iii) nor (iv).
- (c) (i) and (iii), but neither (ii) nor (iv).
- (d) (iii) and (iv), but neither (i) nor (ii).

First, try to answer the question without running the code on your computer.

²<https://stat.ethz.ch/mailman/listinfo/r-help>

³<https://stackoverflow.com/>

Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix [A.3](#).

5

How to pass arguments to R functions

In this chapter, we learn how to pass arguments to functions. We learn that there are two different options. We can either pass a value by matching the name of the argument or by the position of the argument. In practice, we often apply both options in one function call.

5.1 Arguments with default values

Let us take a look at the help document for the function `append()` that we already used in section 3.3.5. In the ‘Usage’ section, we find the line shown in figure 5.1.

```
append(x, values, after = length(x))
```

FIGURE 5.1: Usage information from `?append`.

We conclude that there are three arguments that can be passed to `append()`, named `x`, `values` and `after`. The last argument name is followed by `=` in the Usage information, whereas there is no `=` after the first and second argument names. Here is the difference.

- An argument whose name is followed by `=` has a default value (to the right of the `=` sign).
- All other arguments do not have an associated default value.

In this example, `after` is, by default, equal to `length(x)`. By contrast, `x` and `values` do not have default values. What exactly does this mean in practice?

5.2 How to match arguments by name

When we call a function, we can type the name of each argument (in our example `x`, `values` and `after`), followed by `=` and its value.

```
> a2e <- letters[1:5]
> append(x = a2e, values = c("F", "G"), after = length(a2e))
## [1] "a" "b" "c" "d" "e" "F" "G"
```

If we are happy with the default, we do not need to include the argument in our function call. For example, the previous command has the same effect as the next one.¹

```
> append(x = a2e, values = c("F", "G"))
## [1] "a" "b" "c" "d" "e" "F" "G"
```

However, we *must* specify a value for each argument that does *not* have a default. Otherwise R will complain.

```
> append(values = c("F", "G"), after = length(a2e))
```

```
## Error in append(values = c("F", "G"), after = length(a2e)) :
##   argument "x" is missing, with no default
```

When we include the name and `=` before the value of the argument, we say that we ‘match the argument by name’. When we match *all* arguments by name, we can change the order of the arguments. For example, the following commands are all equivalent.

```
> append(values = c("F", "G"), x = a2e, after = length(a2e))
> append(after = length(a2e), values = c("F", "G"), x = a2e)
> append(values = c("F", "G"), x = a2e)
```

¹We can run the next code chunk in the RStudio console with less typing if we take advantage of the following practical shortcut. By pressing the up-arrow key on the keyboard, we insert the previous command behind the command prompt. The up-arrow key can be pressed repeatedly to move to earlier commands. If we went too far back with the up-arrow key, we can press the down-arrow key to navigate to more recent commands.

5.3 How to match arguments by position

An alternative to matching by name is matching *by position*. In that case, we do not put the argument name in front of the argument value. For example, the following two commands are equivalent.

```
> append(x = a2e, values = c("F", "G"), after = length(a2e))
> append(a2e, c("F", "G"), length(a2e))
```

In the second command, R assumes that the arguments appear in the same order as in the ‘Usage’ section of the help page (figure 5.1): `x` comes first, `values` second and `after` last. We cannot change the order of the arguments when we match them by position.

If we are happy with the default values of the final arguments, we can omit them. For example, the following two commands have the same effect.

```
> append(a2e, c("F", "G"), length(a2e))
> append(a2e, c("F", "G"))
```

5.4 Mixing positional matching with matching by names

It is possible, and in fact quite common, to mix positional matching and matching by name, especially when we do not need to change all of the defaults. For example, the following code matches `x` and `values` by position, but `after` by name.

```
> append(a2e, c("something", "in", "between"), after = 2)
## [1] "a"      "b"      "something" "in"      "between"   "c"
## [7] "d"      "e"
```

Although it is in principle possible to continue with positional matching even after one of the arguments is matched by name, it becomes difficult to read such code. I recommend we match only the first few arguments by position and then continue to explicitly name the rest.

The tidyverse style guide suggests that we match ‘data arguments’ (in our example `x` and `values`) by position and omit their name (<https://style.tidyverse.org/syntax.html#argument-names>). Usually, we recognise a ‘data

argument' by the property that it does not have a default value. We should definitely match an argument by name when we override its default value.

5.5 Summary and outlook

We work with functions many times throughout this book. We often adopt the pattern that we match the first few arguments by position and the trailing arguments by name. As your R skills begin to mature, you will probably start to adopt a similar style in your own code.

5.6 Just checking

Figure 5.2 shows a snippet from R's built-in documentation page for the normal random number generator `rnorm()`.

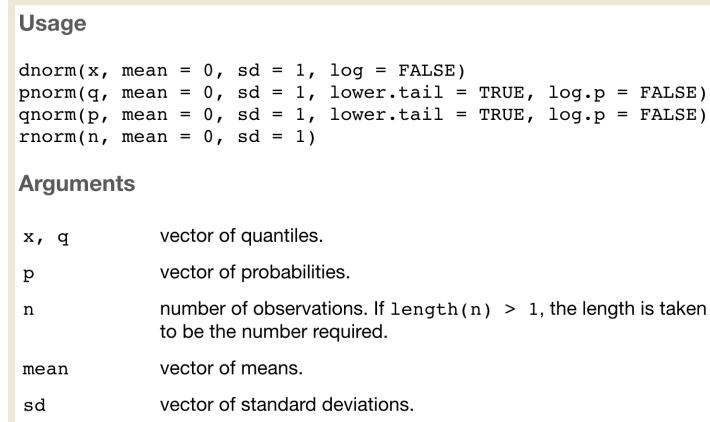


FIGURE 5.2: Excerpt of the documentation of the R function `rnorm()`.

Here are four different R commands.

- (i) `rnorm(mean = 0, sd = 1)`
- (ii) `rnorm(sd = 1, x = 10, mean = 0)`
- (iii) `rnorm(10, 0, 1)`
- (iv) `rnorm(10, 0, sd = 1)`

Which of these commands generate 10 normally distributed random numbers with mean 0 and standard deviation 1?

- (a) (i) and (iii), but neither (ii) nor (iv).
- (b) (iii) and (iv), but neither (i) nor (ii).
- (c) Only (iv).
- (d) (ii), (iii) and (iv), but not (i).

You can find the answer in appendix [A.4](#).



6

Writing R scripts and functions

Scripts are files that contain sequences of computer code. Saving code in scripts makes it easy to retrieve past work. Scripts also help to record the context in which each command should be run. R scripts are files that end with the file extension `.R`. In this chapter, you learn how to write simple R scripts that may include your own functions. You also learn about pipelines, which have become a common way to make complex scripts a little bit more readable and user-friendly.

6.1 Our first script

In chapter 5, we essentially ran several variations of this code chunk.

```
> a2e <- letters[1:5]
> append(x = a2e, values = c("F", "G"), after = length(a2e))
```

Suppose that, while we are developing some code, we frequently want to change the value of `a2e` on the first line. We then have to repeat two commands in the console (namely lines 1 and 2) to see the final result. If there were more than two lines, we would have to repeat even more commands in the console. Would it not be more convenient if we could accomplish the same result with fewer keystrokes?

For a more efficient workflow, we work with *scripts*. A script is a text file that contains a sequence of R commands. With very little effort, we can run all the commands in the script with a single click or keyboard shortcut.

Before we write our first script, I recommend that we start a new project as described in section 2.5 (e.g. with the title `my_first_script`) so that we become familiar with the recommended RStudio workflow.

In principle, we can write an R script with any text editor (e.g. TextEdit on a Mac or Notepad on Windows). However, it is more convenient to use the RStudio editor. We can start a new script either from the menu ('File' → 'New File' → 'R Script') or with the keyboard shortcut 'Command and Shift and N'

on a Mac or ‘Ctrl and Shift and N’ on Windows and Linux. Afterwards, the editor appears as a new pane in the top left of the RStudio window (figure 6.1).

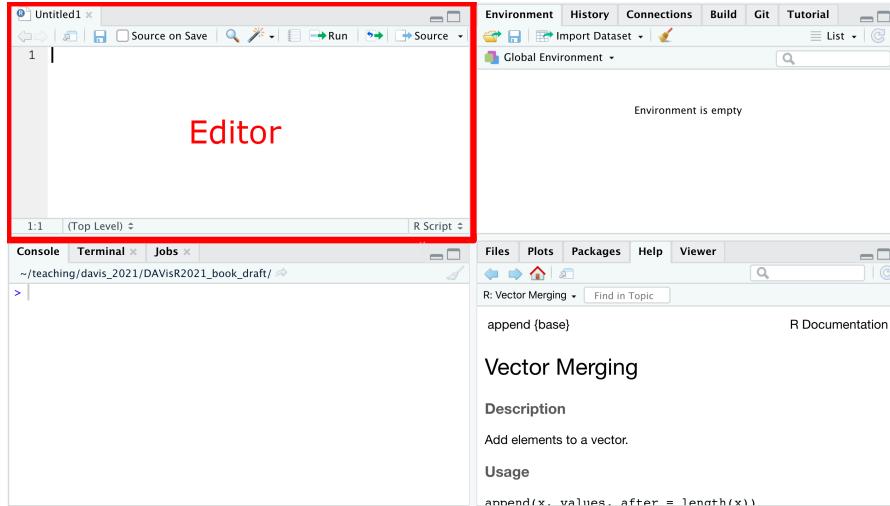


FIGURE 6.1: By default, the editor opens in the top left pane of the RStudio window.

As our first script, let us type the previous code chunk into RStudio’s editor (figure 6.2).



FIGURE 6.2: A simple script with only two commands.

Next, we should save the script. We open the ‘Save File’ dialogue box (figure 6.3) either from the menu (‘File’ → ‘Save’) or by clicking on the diskette symbol at the top of the editor pane. Alternatively, we can use the keyboard shortcut ‘Command and S’ on a Mac or ‘Ctrl and S’ on Windows and Linux.

Let us save the script under the name `append_arguments.R`. In general, we should choose file names that reveal the purpose of the script (see <https://style.tidyverse.org/files.html> for naming conventions). All R scripts must end with a `.R` extension. We can omit the extension in the dialogue box. RStudio automatically appends the extension.

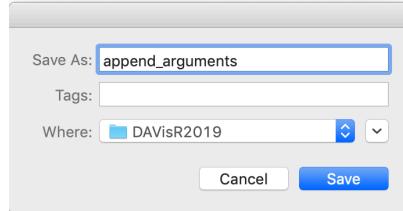


FIGURE 6.3: The ‘Save File’ dialogue box.

6.2 Running a script

There are essentially two different ways in which we can execute the commands in a script. In section 6.3, we learn how to *source* a script, which is usually the more efficient among the two options. The more pedestrian way is to *run* the commands. Running can be useful when developing code (e.g during debugging).

To run the script from section 6.1, move the text cursor to the first line. Then click the ‘Run’ button at the top of the Editor pane (figure 6.4).



FIGURE 6.4: The ‘Run’ button is highlighted by the red ellipse.

Instead of clicking the ‘Run’ button, we can alternatively run each line with the keyboard shortcut ‘Ctrl and Enter’.

Regardless of whether we ran the line via the menu or keyboard shortcut, the first command has been copied to the console and executed.

```
> a2e <- letters[1:5]
```

Meanwhile RStudio has automatically moved the text cursor in the editor pane to the second line. To run that line, we again click ‘Run’ or press ‘Ctrl and Enter’. If our script had still more lines to come, we could repeat this step over and over.

We can run multiple lines in one go by first highlighting them in the editor pane, and then we either click ‘Run’ or press ‘Ctrl and Enter’. We can run the entire script at one fell swoop with ‘Ctrl and Option and R’ (Mac) or ‘Ctrl and Alt and R’ (Windows and Linux).

6.3 Sourcing a script

When we *run* a script, each line is copied to the console. This procedure leaves a lot of output in the console, mostly of no real value to the reader. A cleaner solution is to *source* the script, either by clicking ‘Source’ at the top of the editor pane (figure 6.5) or by pressing ‘Command and Shift and S’ on a Mac or ‘Ctrl and Shift and S’ on Windows and Linux.



FIGURE 6.5: The ‘Source’ button is highlighted by the red ellipse.

As a result, RStudio automatically generates the console command ...

```
> source("~/our_directory/append_arguments.R")
```

... where `our_directory` is replaced by the directory where we saved the script.

In the console, there is nothing printed except the line with the `source()` command. We are able to confirm that R ran all commands in the script by checking the variables in the global environment. To cut a long story short, the global environment is where R stores all the variables created during the entire duration of an R session. In the environment tab, we can see a list of all variables in the global environment. Alternatively, we can run `ls()` in the console, which returns a character vector with the names of all objects in the global environment.

```
> ls()
## [1] "a2e"
```

Let us temporarily remove `a2e` from the global environment. We can remove variables with `rm()`.

```
> rm(a2e)
```

Looking at the environment tab reveals that `a2e` has indeed disappeared.

```
> "a2e" %in% ls()
## [1] FALSE
```

Here is how we can tell that sourcing the script really carries out the first command of the script: after we source `append_arguments.R`, we see that `a2e` appears again in the environment tab. Equivalently, repeating our previous console command now returns `TRUE`.

```
> source("~/our_directory/append_arguments.R")
> "a2e" %in% ls()
```

```
## [1] TRUE
```

We have just learned that, when sourcing rather than running, the values of the commands are not automatically printed in the console. If we source a script, but we still want to see output in the console, we can use the function `print()`. To indicate that the next three lines should be part of a script rather than typed in the console, the next code chunk does not contain any lines starting with the command prompt `>`. From here on, I only include the command prompt when I suggest that a command is to be typed in the console.

```
a2e <- letters[1:5]
append(x = a2e, values = c("something", "in", "between"), after = 2)
print(a2e)
```

```
## [1] "a" "b" "c" "d" "e"
```

We notice that the function `append()` did not change the value of `a2e`. If we want to see the value returned by `append()`, we can either wrap `print()` around the `append()` command ...

```
a2e <- letters[1:5]
print(append(x = a2e, values = c("something", "in", "between"), after = 2))
## [1] "a"           "b"           "something" "in"        "between"   "c"
## [7] "d"           "e"
```

... or, leading to more readable code, we can assign the result of `append()` to a variable, say `a2e_appended`, and print `a2e_appended`.

```
a2e <- letters[1:5]
a2e_appended <-
  append(x = a2e, values = c("something", "in", "between"), after = 2)
print(a2e_appended)
## [1] "a"           "b"           "something" "in"        "between"   "c"
## [7] "d"           "e"
```

As we learn next, a yet more elegant alternative is to write such code as a pipeline.

6.4 Pipes

The purpose of the code chunk at the end of the previous section is to take the first five letters of the alphabet and insert a few words between "b" and "c". Thinking more about this code chunk, we notice that the variable `a2e_appended` is only needed for temporary storage. In principle, we can eliminate `a2e_appended` if we pass the argument `append(x = a2e, values = c("something", "in", "between"), after = 2)` to `print()`.

```
a2e <- letters[1:5]
print(append(x = a2e, values = c("something", "in", "between"), after = 2))
```

We can even go one step further and eliminate `a2e` from the code chunk.

```
print(
  append(
    x = letters[1:5],
    values = c("something", "in", "between"),
    after = 2
  )
)
```

This version looks complicated. When we match the data arguments `x` and `values` by position instead of name, we have a little bit less to type.

```
print(
  append(
    letters[1:5],
    c("something", "in", "between"),
    after = 2
  )
)
## [1] "a"          "b"          "something" "in"        "between"   "c"
## [7] "d"          "e"
```

Still, the order of the function calls is unintuitive. The first function to appear when reading the code from top to bottom is `print()`, and only then we see the function call `append()`. However, the order in which the operations are carried out is the opposite; first R performs `append()`, afterwards it prints. If we had a longer chain of function calls (e.g. `print(log(length append(x, y)), base = 3, digits = 10)`) it becomes even more difficult to disentangle which argument is passed to which function. Although this example is rather

contrived, long chains of function calls arise naturally in data analysis; thus, it would be nice to have a tool that keeps such commands more readable.

Such a tool exists: the pipe operator `|>`. We can verbalise `|>` as ‘and then’. For example, in the command `print(log(length append(x , y)), base = 3, digits = 10)`, we perform these steps:

- We take `x`, and then
- we append `y`, and then
- we calculate the length, and then
- we take the logarithm with base 3, and then
- we print using the argument `digits = 10` (i.e. there are ten digits after the decimal point).

Here is how this statement translates into R code with the pipe operator.

```
x |>
  append(y) |>
  length() |>
  log(base = 3) |>
  print(digits = 10)
```

This code chunk looks much cleaner than the convoluted original version `print(log(length.append(x , y)), base = 3, digits = 10)`. A code chunk that consists of several lines of code that all end with `|>` is called a *pipeline*. For example, the code chunk

```
print(
  append(
    letters[1:5],
    c("something", "in", "between"),
    after = 2
  )
)
```

can be transformed into the following pipeline.

```
letters[1:5] |>
  append(c("something", "in", "between"), after = 2) |>
  print()
## [1] "a"          "b"          "something" "in"        "between"   "c"
## [7] "d"          "e"
```

The pipeline makes it obvious that the input is the object in the first line (i.e. `letters[1:5]`) and that we first append and then print, in contrast to

the order in which the functions appear in the non-pipeline version. The pipe operator does not change the calculations that R carries out under the hood, but it makes code more readable. For this reason, the pipe operator appears frequently in modern R code.

6.5 RStudio code suggestions

RStudio helps us to write R scripts by providing automatic code suggestions. After we type the first few letters of a function or variable name, RStudio opens a pop-up menu with suggested names (figure 6.6). We can select a name from this list with a mouse click. Alternatively, we can navigate to the correct name with the arrow keys before confirming our choice with the return key.

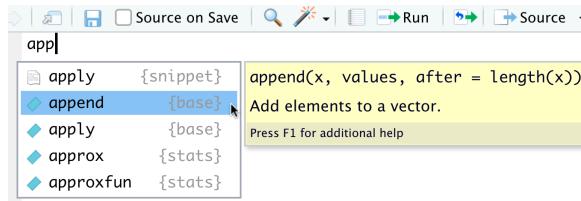


FIGURE 6.6: RStudio shows code suggestions when we type the first few letters of a function or variable name.

When we press the tab key inside the parentheses after the function name, we open another pop-up menu showing the function's arguments (figure 6.7). Each argument in the pop-up menu is accompanied by a brief description.

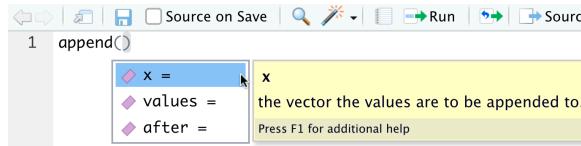


FIGURE 6.7: RStudio shows the arguments of a function in a popup menu.

I recommend to take advantage of RStudio's code suggestions because they reduce the risk of making a typo.

6.6 Writing a function

So far, our scripts have been sequences of commands that perform exactly the same action every time we call them. By writing a *function* we have greater flexibility; a function can perform a different action if we supply a different set of inputs. The purpose of a function is best explained by an example.

Let us start with the following script.

```
print("Hello, world", quote = FALSE)
```

The purpose of `quote = FALSE` is to suppress the quotation marks in the output, a feature that makes the output look a little bit prettier later on.

Let us save this script as `hello.R` (figure 6.8).

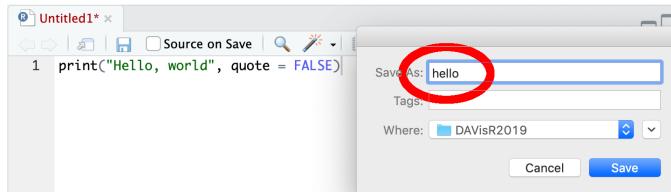


FIGURE 6.8: A ‘Hello, world’ program is a rite of passage when learning any programming language. Here is the R version.

Every time we run or source the script, we obtain the same output.

```
## [1] Hello, world
```

Suppose we want to greet specific people by name rather than the whole world. Instead of writing a script for every single addressee, we can modify the script `hello.R` as follows. (I explain in a moment how and why it works.)

```
say_hello <- function(name) {
  print(c("Hello,", name), quote = FALSE)
}
```

After we source `hello.R` again, there is a function `say_hello()` in the global environment (figure 6.9), a sign that R is ready to run this function.

Now we can enter an argument of our choice inside a pair of parentheses.

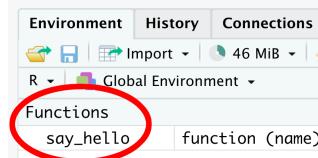


FIGURE 6.9: Besides vectors, the global environment can also contain functions.

```
say_hello("Kitty")
## [1] Hello, Kitty
```

Although this is a very simple example, it shows the main ingredients of an R function. The general syntax of an R function is as follows.

```
function_name <- function(arg1, arg2, ...) {
  # Body of the function
}
```

Here are the important features.

We begin with a function name that reveals the purpose of the function. After the function name, we insert an assignment operator `<-` followed by the keyword `function`.

In parentheses, we give a list of arguments. There can be more than one argument. It is also possible to have zero arguments inside the parentheses, which can be occasionally useful if we simply want to bundle several commands together that do not need any input. We can set a default value for an argument with the syntax `argument_name = default_value`. For example, let us set `name` by default equal to `world`.

```
say_hello <- function(name = "world") {
  print(c("Hello,", name), quote = FALSE)
}
```

When we source the script and then type `say_hello()` in the console with nothing inside the parentheses, we greet the whole world.

```
> say_hello()
## [1] Hello, world
```

Even if the parentheses are empty, we should not omit them. Otherwise, R prints the code of the function in the console without executing the function.

```
> say_hello
## function(name = "world") {
##   print(c("Hello,", name), quote = FALSE)
## }
```

The body of a function is the sequence of instructions that manipulate our input. The body is usually enclosed in braces {}. The commands between the braces should always be indented by two spaces at the start of the line. Indentation is not technically mandatory, but it improves readability enormously. In general, the RStudio text editor does a decent job at automatically indenting code while we are typing. RStudio also has an ‘Addin’ feature called ‘styler’, which formats R scripts according to the tidyverse style (e.g. by indenting lines properly). I explain in section 8.3 how to install the **styler** package.

6.7 Return value of a function

Often, we want functions to *return* a value. For example, the next function adds 1 to the argument named x and returns the incremented value.

```
increment <- function(x) {
  return(x + 1)
}
```

When a value is returned, it can be assigned to a variable (e.g. y). We can then reuse y at a later stage in our code.

```
y <- increment(5)
y
## [1] 6
```

A function immediately stops after executing **return()**. For example, when we source the following script, we do not see anything printed despite the **print()** command because R exited the function one line earlier.

```
increment <- function(x) {
  return(x + 1)
  print("We never get here.")
```

```
}
```

```
increment(5)
```

If there is no explicit `return()` in the body of a function, the return value is equal to the value of the last executed command. For example, the function below automatically returns the incremented value although there is no explicit `return()`.

```
increment <- function(x) {
  x + 1
}
```

The tidyverse style guide recommends to use `return()` only for early returns (<https://style.tidyverse.org/functions.html#return>). Otherwise, we should rely on R to return the value of the last evaluated command. Consequently, the latest code chunk is the stylistically preferred option for our function `increment()`.

Here is a common source of confusion: returning a value is not the same as printing this value. For example, if we source the following script, the value 6 is returned by `increment()` although it is never printed.

```
y <- increment(5)
y + 10
## [1] 16
```

Conversely, the function `return_null()` in the next code chunk prints text to the console as a side effect, but it returns nothing. In this example, we use the function `cat()` which stands for ‘concatenate and print’.

```
return_null <- function() {
  cat("I am printing a lot of stuff, but I will not return anything.")
}
z <- return_null()
## I am printing a lot of stuff, but I will not return anything.
```

We can confirm that `return_null` returns nothing by checking the content of `z`.

```
> z
## NULL
```

We learn more about the meaning of `NULL` in chapter 11. At this point, it suffices to view `NULL` as equivalent to nothing.

To add to the confusion, `cat()` behaves differently from `print()`, which does return its argument.

```
returned_by_print <- print("I print something and also return it")
## [1] "I print something and also return it"
```

Here is the proof that `print()` returned its argument.

```
> returned_by_print
## [1] "I print something and also return it"
```

In summary, returning and printing are generally, but not always, different actions. It is important to understand this distinction if we want to become proficient R programmers.

6.8 Summary and outlook

R scripts are the most common file format for saving and documenting R code. In this chapter, we learned how to work with scripts and writing our own function. In chapter 7, we learn about a related file type, R Markdown, which is useful for combining R code with text and figures (e.g. when writing a report). When working with R scripts, please make it a habit to place them in RStudio projects as outlined in section 2.5. It will help you stay more organised when we work with real-world data.

6.9 Just checking

Which of the following functions *return* the square of the argument `x`?

```
square_1 <- function(x) {
  print(x^2)
}
square_2 <- function(x) {
  x^2
}
square_3 <- function(x) {
  cat(x^2)
}
```

- (a) None of `square_1()`, `square_2()` or `square_3()`.
- (b) Only `square_1()`, but neither `square_2()` nor `square_3()`.
- (c) `square_1()` and `square_2()`, but not `square_3()`.
- (d) All of `square_1()`, `square_2()` and `square_3()`.

You can find the answer in appendix A.5.

7

R Markdown

R scripts are good for storing R code, but they are not great for documenting statistical analysis or reporting results. As we saw in section 3.1.7, it is possible to add comments about the code in R scripts by writing text preceded by a hash symbol `#`. However, such code comments in R scripts should be short so that they do not obstruct the actual code. Moreover, code comments are not ideal for reporting output from code because they are not automatically updated when the input changes. R Markdown files are an alternative to R scripts to overcome these issues. With R Markdown, we can produce nicely formatted reports that show R code together with its output. In this chapter, we learn how to create simple R Markdown files with RStudio.

7.1 Creating and knitting an R Markdown file

To create an R Markdown template, we first start a new project as outlined in section 2.5. Then, we go to the menu option ‘File’ → ‘New File’ → ‘R Markdown’. Afterwards, a dialogue window appears in which we can type a title and the author’s name (figure 7.1), which will later appear at the top of the document.

When we click ‘OK’, RStudio opens an R Markdown template in the editor pane. Before we look at the content of the file, let us learn how to generate output. The process of generating an output document (in HTML by default) from an R Markdown file is called ‘knitting’. We can knit either by clicking on the ‘Knit’ button in the editor pane (figure 7.2) or by using the keyboard shortcut ‘Command and Shift and K’ (Mac) or ‘Ctrl and Shift and K’ (Windows and Linux).

In the next dialogue window, we must enter a file name. R Markdown files have the extension `.Rmd`. In the dialogue window, we can leave out the file extension; RStudio will automatically append `.Rmd` for us. After we click ‘Save’, RStudio opens the knitted HTML document in a new window. RStudio also saves the HTML file in the project directory.

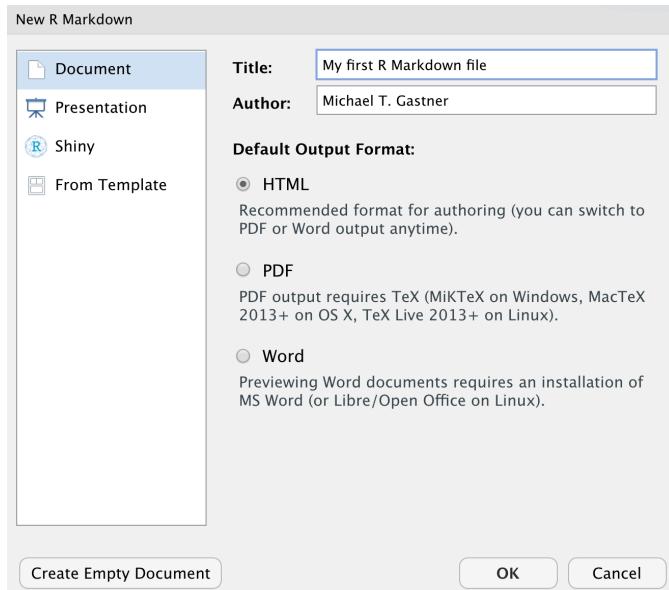


FIGURE 7.1: This dialogue window appears when we start a new R Markdown file.



FIGURE 7.2: The ‘Knit’ button is located at the top of the editor pane.

7.2 Basic building blocks of an R Markdown file

Let us have a look at the R Markdown template that RStudio automatically generated in the previous section. You can find the source code of the template in appendix B. It shows the basic building blocks of an R Markdown file:

- YAML header
- code chunk
- section headings
- marked-up text

At the start of the file is the so-called YAML header, whose start and end is signalled by three dashes each.

```
---
title: "My first R Markdown file"
author: "Michael T. Gastner"
date: "12/29/2021"
output: html_document
---
```

YAML is the recursive acronym for ‘YAML Ain’t Markup Language’. We do not need to learn YAML to get started with R Markdown. It suffices to know that we can change the title, author and date by editing the corresponding lines in the YAML header.

Below the YAML header is our first example of an R code chunk.

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

The telltale signs of an R code chunk are:

- three backticks ` ` at the start followed by a curly brace { and the letter r.
- three backticks at the end.

The first code chunk of an R Markdown file is also known as the ‘setup chunk’. For the time being, I recommend that you keep the setup chunk exactly as it is. There are two more R code chunks in the template:

```
```{r cars}
summary(cars)
```
```

and

```
```{r pressure, echo=FALSE}
plot(pressure)
```
```

Feel free to edit these chunks by replacing the R code on the line in the middle. Then knit the file to see the effect. We look at R code chunks in more detail in section [7.4.1](#).

After the setup chunk, we find an example of a section heading.

```
## R Markdown
```

Section headings always start with one or several hash symbols. The number of hash symbols (between 1 and 6) determines the level of the heading. Figure 7.3 shows how the following lines would be rendered in HTML.

```
# Top-level heading  
## Second level  
### Third level  
#### Fourth level  
##### Fifth level  
##### Sixth level
```

```
Top-level heading  
Second level  
Third level  
Fourth level  
Fifth level  
Sixth level
```

FIGURE 7.3: Section headings become smaller as more hash symbols are placed before the section title.

Between the section headings, we can find examples of text that is rendered verbatim, for example:

```
This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.
```

When you compare other text snippets with the HTML output, you notice that there are examples that are not rendered verbatim (e.g. ****Knit**** appears without asterisks but as boldface text: **Knit**). In the next section, we learn more about the rules that govern how text is rendered in the output.

7.3 Marked-up text

You are probably familiar with word processing software (e.g. Microsoft Word[®] or Google Docs[®]) that are based on the principle of ‘what you see is what you get’, also known as WYSIWYG. For example, if you want to display parts of the text in boldface, you would have to highlight the text, go through menu options (e.g. ‘Format’ → ‘Font’ in Word) and drop-down menus to select a bold font style. Afterwards, the text appears in boldface on your computer screen. The WYSIWYG method gives the user immediate visual feedback about the document. However, when opening the file with another word processing software, the text may not look exactly the same because of differences in the software implementation.

An alternative to WYSIWYG is provided by so-called markup languages. In a markup language, users insert symbols into the text that indicate how the user wants the text to appear. As a result, the source text does not immediately look like the rendered document because the symbols appear in the source file together with the text. The advantage of a markup language is that different software can consistently infer the user’s intention. An example of a markup language is HTML (HyperText Markup Language), which is used to write websites. When viewing an HTML document, the result is a consistently formatted website, regardless of which web browser is used.

Markdown is a markup language that was developed in 2004 with the intention of being simple to learn and easy to read. R Markdown uses the same rules as conventional Markdown for rendering plain text and adds a few additional features that allow users to include R code.

7.3.1 Paragraph breaks and forced line breaks

Most of the time, text in a Markdown file is rendered as the same text in the output file. However, there are some exceptions. For example, line breaks in a Markdown source file do not necessarily appear as line breaks in the output. To indicate a break between paragraphs, we must insert a blank line in the Markdown.

To indicate a line break inside a paragraph,
we can end the line with two spaces. Alternatively, we can end the line before
the line break
with a backslash.

Here is the previous text written in Markdown. There are two invisible spaces after ‘paragraph,’ on the fourth line.

To indicate a break between paragraphs, we must insert a blank line in the Markdown.

To indicate a line break inside a paragraph, we can end the line with two spaces.

Alternatively, we can end the line before the line break\\ with a backslash.

To prevent Markdown from inserting a line break at a space between words, we put a backslash before the non-breaking space (e.g. \ here) and continue with the next word on the same line in the Markdown file.

7.3.2 Highlighting text with italics and boldface

Text to be highlighted in italics is placed either between a pair of single *asterisks* or a pair of single *underscores*.

Text to be highlighted in italics is placed either between a pair of single *asterisks* or a pair of single _underscores_.

Text to be highlighted in boldface is placed either between a pair of double **asterisks** or a pair of double **underscores**.

Text to be highlighted in boldface is placed either between a pair of double **asterisks** or a pair of double __underscores__.

Text to be highlighted in bold italics is placed either between a pair of triple **asterisks** or a pair of triple **underscores**.

Text to be highlighted in bold italics is placed either between a pair of triple ***asterisks*** or a pair of triple ___underscores___.

7.3.3 Superscripts, footnotes and hyperlinks

Text in superscript is placed between a pair of carets.^{super} Text in subscript is placed between a pair of tildes._{sub}

It is also possible to insert footnotes, as shown here.¹

¹Here is a footnote.

```
Text in superscript is placed between a pair of carets.^super^
Text in subscript is placed between a pair of tildes.^sub~
```

```
It is also possible to insert footnotes, as shown
here.[^our_first_footnote]
```

```
[^our_first_footnote]: Here is a footnote.
```

If you need superscripts or subscripts in equations, it would be tedious to insert them with Markdown notation. Instead, I recommend to insert equations with L^AT_EX, a specialised typesetting language for mathematics. Apart from superscripts and subscripts, L^AT_EX has many special symbols (e.g. Greek letters) that frequently occur in mathematical expressions. L^AT_EX is outside the scope of this book. If you need L^AT_EX expressions in your R Markdown, you can find help online (e.g. at <https://www.stat.cmu.edu/~cshalizi/rmarkdown/#math-in-r-markdown>). By the way, URLs (such as the URL in the previous sentence) should be placed in angle brackets to render them as a clickable link.

```
If you need $\text{\LaTeX}{}$ expressions in your R Markdown, you can
find help online (e.g.\ at
<https://www.stat.cmu.edu/~cshalizi/rmarkdown/#math-in-r-markdown>).
```

7.3.4 Inline code

Computer code is conventionally printed in monospaced font. To include computer code as part of a sentence, place the code between a pair of backticks (e.g. `sum(1:10)`).

```
To include computer code as part of a sentence, place the code between a
pair of backticks (e.g.\ `sum(1:10)`).
```

In this example, the code is printed but not run. If you want to run the code and display the result in the text, we must place an `r` after the first backtick. For example, `r sum(1:10)` prints the number 55.

7.3.5 Unordered lists

- To produce an unordered list, place either a dash (-), asterisk (*) or plus sign (+) as delimiter before each list item.
- Leave a blank line before the first list item.
- Each item must start with a new line.

- There must be a space between the delimiter and the text.
 - For nested lists, indent the list item by four spaces.
 - As a matter of style, choose a different delimiter for the nested list (e.g. an asterisk if the outer list uses a dash).
- You can insert a paragraph break in a list item.
All you need to do is indent the new paragraph with four spaces.
- Afterwards, you can continue with the next list item.

- To produce an unordered list, place either a dash (`-`), asterisk (`*`) or plus sign (`+`) as delimiter before each list item.
 - Leave a blank line before the first list item.
 - Each item must start with a new line.
 - There must be a space between the delimiter and the text.

- * For nested lists, indent the list item by four spaces.
- * As a matter of style, choose a different delimiter for the nested list (e.g.\ an asterisk if the outer list uses a dash).

 - You can insert a paragraph break in a list item.

All you need to do is indent the new paragraph with four spaces.

- Afterwards, you can continue with the next list item.

If you followed my recommendation in section 2.2, you can see the level of indentation using the RStudio editor's indent guides, which are thin grey lines to the left of the code (figure 7.4).

- There must be a space between the delimiter and the text.
 * For nested lists, indent the list item by four spaces.
Indent guides → If you followed my recommendation in `\o{ref(sec:what-is-rstudio)}`, you can see the level of indentation using the RStudio editor's indent guides, which are thin grey lines to the left of the code.
 * As a matter of style, choose a different delimiter for the nested list (e.g.\ an asterisk if the outer list uses a dash).

FIGURE 7.4: Indent guides are thin grey lines to the left of the code that visualise the level of indentation. You can activate indent guides using the menu options ‘Tools’ → ‘Global Options’ (see section 2.2).

7.3.6 Ordered lists

1. Ordered lists can be produced by starting the line with either:
 - Arabic numerals followed by a full stop (e.g. 1.).
 - Roman numerals inside parentheses (e.g. (i) or (I)).
 - a letter inside parentheses (e.g. (a) or (A)).
2. Leave a blank line before the first list item.

3. It is possible to nest an ordered list with another unordered or ordered list.
 - (a) All you need to do is indent the nested list.
 - (b) Use four spaces for the indentation.
4. Markdown automatically continues the numbering of the list items.
Have a look at the source code for this list. The number appears as 1 in the source code and '4' in the output.
5. This feature is useful when you lose track of the count.

1. Ordered lists can be produced by starting the line with either:
 - Arabic numerals followed by a full stop (e.g.\ `1.`).
 - Roman numerals inside parentheses (e.g.\ `(i)` or `(I)`).
 - a letter inside parentheses (e.g.\ `(a)` or `(A)`).
2. Leave a blank line before the first list item.
3. It is possible to nest an ordered list with another unordered or ordered list.
 - (a) All you need to do is indent the nested list.
 - (b) Use four spaces for the indentation.
4. Markdown automatically continues the numbering of the list items.
Have a look at the source code for this list.
The number appears as `1` in the source code and '4' in the output.
5. This feature is useful when you lose track of the count.

7.3.7 Do not exceed 80 characters per line

Lines in a Markdown file should not exceed 80 characters because longer lines are difficult to review. Usually, staying within the 80-character limit is not a problem because line breaks in a Markdown file are invisible in the rendered document (see section 7.3.1). The only exceptions to the 80-character rule are long URLs.

If you followed my recommendation in section 2.2, you see a thin vertical grey line on the right of the editor pane. As long as your text is to the left of that line, you are within the 80-character limit.

7.4 R code chunks

7.4.1 Chunk options

As we saw in section 7.2, the start and end of an R code chunk is signalled by three backticks. Instead of manually typing the backticks, we can generate them automatically by clicking on the 'c' symbol at the top of the editor pane

(figure 7.5) and selecting ‘R’ from the drop-down menu. Alternatively, we can use the keyboard shortcut ‘Command and Option and I’ (Mac) or ‘Ctrl and Alt and I’ (Windows and Linux).



FIGURE 7.5: We can insert a new R code chunk by clicking on the button highlighted by the red ellipse.

After the first set of three backticks, we can insert a name for the code chunk. For example, the next code chunk is named `sequence`.

```
```{r sequence}
seq(0, 1, 0.2)
```
```

Chunk names are useful when we must refer to specific code chunks in the document (e.g. for adding a caption to a plot produced by a code chunk). However, chunk names are not strictly necessary.

After the `r` in the curly braces or after the optional chunk name, we can add ‘chunk options’ that determine how the chunk is run and displayed in the knitted document.

By default, the source code in a chunk and the output produced by the code are printed in the document. We can suppress the source code by adding the chunk option `echo=FALSE`. For example, the code chunk

```
```{r sequence, echo=FALSE}
seq(0, 1, 0.2)
```
```

is not shown in the knitted document, but it still produces output:

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

If we want to show the source code but not run it (and, consequently, not show any output), we use the chunk option `eval=FALSE`.

```
```{r sequence, eval=FALSE}
seq(0, 1, 0.2)
```

```

Occasionally, we neither want to show the source code nor its output, but we still want to run the code (e.g. in the setup chunk). We can achieve this effect by using the code chunk option `include=FALSE`.

There are many more chunk options. For a comprehensive list, see <https://yihui.org/knitr/options/>. It is possible to use more than one chunk option. In that case, the chunk options need to be separated by a comma. Instead of typing chunk options by hand, you can also use a simple graphical user interface that opens when clicking the cogwheel symbol to the right of the code chunk (figure 7.6).

7.4.2 Previewing output from code chunks with RStudio

RStudio allows users to preview the output of code chunks directly in the editor pane. To the right of every R code chunk, there is a green arrow pointing to the right (figure 7.6). When this arrow is clicked, RStudio executes the current chunk. The output is displayed below the chunk. When the code chunk is edited, the output is not automatically updated; thus, the user needs to click the green arrow again to view the effect of recent edits on the current chunk. Before running the current chunk, it may also be necessary to run all previous chunks (e.g. to have the latest values of the input variables). We can run all chunks above the current one by clicking on the grey downward arrow, which is just to the left of the green arrow.



FIGURE 7.6: To the right of every R code chunk, RStudio displays three buttons: (i) the cogwheel allows modifying chunk options, (ii) the downward arrow runs all chunks above the current one, (iii) the right arrow runs the current chunk.

TABLE 7.1: Selection of R Markdown rules

| R Markdown | Rendered |
|---|--|
| #, ##, ..., ##### | Top-level, 2 nd -level, ..., 6 th -level heading |
| Blank line | Paragraph break |
| Two spaces or one backslash at end of line | Line break |
| \ followed by space not at end of line | Non-breaking space |
| *italics*, _italics_ | <i>italics</i> |
| **bold**, __bold__ | bold |
| ***bold italics***, ___bold italics___ | <i>bold italics</i> |
| super ^{script} | super ^{script} |
| sub ^{script} | sub _{script} |
| < https://www.google.com/ > | https://www.google.com/ |
| - First item | |
| - Second item | Unordered list |
| 1. First item | |
| 2. Second item | Ordered list |
| `c(1, 7, -3)` | Inline code: <code>c(1, 7, -3)</code> |
| `r 1 + 1` | 2 |
| ```{r}
c(1, 7, -3)
``` | R code chunk |

7.5 Summary and outlook

R Markdown gives users many options to combine text, R source code and output from the code into a single document. Table 7.1 summarises what we learned in this chapter. So far, we have only scratched the surface of R Markdown. We revisit R Markdown in chapter 16 when we discuss the display of plots in knitted documents. You can find more details about R Markdown in the cheat sheet available from RStudio’s help menu (‘Help’ → ‘Cheat Sheets’ → ‘R Markdown Cheat Sheet’) or:

https://www.rstudio.org/links/r_markdown_cheat_sheet

7.6 Just checking

First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.6.

Which Markdown generates the output shown in figure 7.7?

```
Some of the greatest discoveries were made by accident.
• Chemistry:
  ◦ 1879: Vulcanized rubber
  ◦ 1859: Vaseline
• Physics:
  ◦ 1895: X-rays
  ◦ 1945: Microwave oven
```

FIGURE 7.7: Which Markdown generates this HTML output?

- (a) Some of the `__greatest__` discoveries were made by `*accident*`.
 - * Chemistry:
 - 1879: Vulcanized rubber
 - 1859: Vaseline
 - * Physics:
 - 1895: X-rays
 - 1945: Microwave oven
- (b) Some of the `*greatest*` discoveries were made by `__accident__`.
 - Chemistry:
 - + 1879: Vulcanized rubber
 - + 1859: Vaseline
 - Physics:
 - + 1895: X-rays
 - + 1945: Microwave oven
- (c) Some of the `_greatest_` discoveries were made by `**accident**`.
 - * Chemistry:
 - 1879: Vulcanized rubber
 - 1859: Vaseline
 - * Physics:
 - 1895: X-rays
 - 1945: Microwave oven
- (d) Some of the `**greatest**` discoveries were made by `_accident_`.

- Chemistry:
 - + 1879: Vulcanized rubber
 - + 1859: Vaseline
- Physics:
 - + 1895: X-rays
 - + 1945: Microwave oven

8

Installing and loading packages

The base installation of R contains a good selection of built-in functions and data sets. In practice, however, we still often encounter challenges that are difficult to solve only with the functions and data sets in the base installation. When you encounter such a challenge, it may console you that the R community is big and diverse; therefore, somebody else has probably already faced a similar challenge. Furthermore, some members of the R community are clever and altruistic developers who contribute additional functions and data sets to solve problems outside the scope of R's base installation. Such additional collections of code are called *packages*.

R's official package repository is known as the Comprehensive R Archive Network (CRAN). At the time of writing, there are almost 18,000 packages available on CRAN. When we face a difficult task, it is worth checking whether one of these packages can do the job for us.

8.1 Installing a package

We can find out which packages are already loaded in our R session as follows.

```
> print(.packages())
## [1] "stats"      "graphics"    "grDevices"   "utils"       "datasets"    "methods"
## [7] "base"
```

The output above shows the preloaded packages at the start of an R session. These packages contain most of the functions that we encounter in everyday use. After we load a new package, we see additional elements in the output.

Instead of running `print(.packages())`, we can also look at the 'Packages' tab in the lower right RStudio pane (figure 8.1). Every package that has a tick next to it is currently loaded in this R session.

Those packages that appear in the packages tab without a tick are installed but not loaded. The distinction between *installed* and *loaded* is important, as we see shortly.

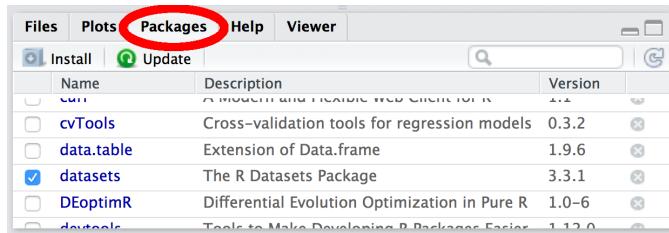


FIGURE 8.1: The packages tab in the lower right RStudio pane.

If we want to know whether a particular package is installed, we can either scroll through the list in the packages tab or look through the output of the following command.

```
> print(.packages(all.available = TRUE))
```

In chapter 19, we work extensively with the **dplyr** package. Let us check whether **dplyr** is listed in the packages tab.

```
> "dplyr" %in% .packages()
```

If the output is **TRUE** in your current R session, please remove the **dplyr** package temporarily so that we all have the same starting point. There are two ways to remove a package.

- We can run the function `remove.packages()`.

```
> remove.packages("dplyr")
```

- We click the \otimes button to the right of ‘**dplyr**’ in the packages tab (figure 8.2).

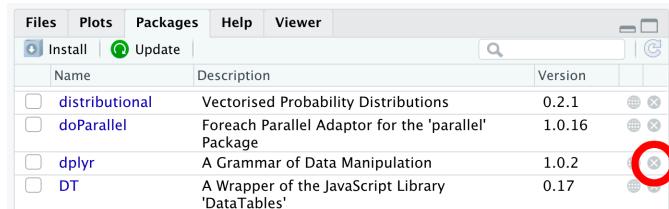


FIGURE 8.2: We can remove a package by clicking the \otimes button.

To install a package, we need an Internet connection and the function `install.packages()`.

```
> install.packages("dplyr")
```

Alternatively, we can click on the ‘Install’ button in the packages tab. It opens a dialogue box where we can type the name of the package we want to install (figure 8.3). While we are typing, RStudio tries to automatically complete the package name, which conveniently reduces the probability of typos.

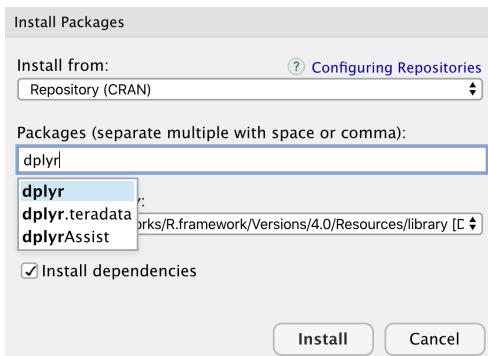


FIGURE 8.3: This dialogue box opens after clicking ‘Install’ in the packages tab.

The download and installation should only take a few seconds.

8.2 Loading a package

The **dplyr** package contains a variety of functions and data sets. We can find the package documentation with the function `help()` and the argument named `package`.

```
> help(package = "dplyr")
```

One of the data sets in the **dplyr** package is `starwars`, which contains information about the characters in the Star Wars film series. Let us take a look at it by typing `starwars` in the console.

```
> starwars
## Error in eval(expr, envir, enclos): object 'starwars' not found
```

Why can R not find `starwars`? Have we not just installed the necessary package?

Yes, but *installing* a package is not the same as *loading* it. When we installed the package, we saved the data on our computer, but the data are not automatically loaded as part of the current R session.

We can load the content of a package in two ways.

- We can use the double colon operator `:::`. We type the package name to the left of the operator and the variable name to the right.

```
dplyr::starwars
## # A tibble: 87 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>
## 1 Luke Sk~    172    77 blond      fair        blue         19 male   masculin
## 2 C-3PO       167    75 <NA>       gold        yellow      112 none   masculin
## 3 R2-D2        96    32 <NA>       white, bl~ red        white, yello~ 33 none   masculin
## 4 Darth V~     202   136 none       white       yellow      41.9 male   masculin
## 5 Leia Or~     150    49 brown      light       brown       19 fema~ feminin
## 6 Owen La~     178   120 brown, gr~ light       blue        52 male   masculin
## 7 Beru Wh~     165    75 brown      light       blue        47 fema~ feminin
## 8 R5-D4        97    32 <NA>       white, red~ red        white, yello~ NA none   masculin
## 9 Biggs D~     183    84 black      light       brown       24 male   masculin
## 10 Obi-Wan~    182    77 auburn, w~ fair        blue-gray    57 male   masculin
## # ... with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Specifying the variable with the double colon operator has the advantage that it is guaranteed to retrieve the correct variable if there is a naming conflict (i.e. if another package happens to have another variable with the same name). The disadvantage of the double colon operator is that we would have to repeatedly type `dplyr::` if our code relies on many different variables from **dplyr**.

- We can load a package with the `library()` function. Note that we need quotes around the argument of `install.packages()`, but not around the argument of `library()`.

```
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

After loading a package with `library()`, we can refer to the variables in this package without the double colon operator.

```
starwars
## # A tibble: 87 x 14
##   name    height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>        <dbl> <chr> <chr>
## 1 Luke Sk~    172    77 blond     fair       blue          19 male  masculin
## 2 C-3PO       167    75 <NA>      gold       yellow        112 none  masculin
## 3 R2-D2        96    32 <NA>      white, bl~ red        33 none  masculin
## 4 Darth V~    202   136 none     white       yellow        41.9 male  masculin
## 5 Leia Or~    150     49 brown    light       brown         19 fema~ feminin
## 6 Owen La~    178   120 brown, gr~ light       blue          52 male  masculin
## 7 Beru Wh~    165     75 brown    light       blue          47 fema~ feminin
## 8 R5-D4       97    32 <NA>      white, red red        NA none  masculin
## 9 Biggs D~    183     84 black    light       brown         24 male  masculin
## 10 Obi-Wan~   182    77 auburn, w~ fair      blue-gray       57 male  masculin
## # ... with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

The messages in the output above (after running `library(dplyr)`) alert us to naming conflicts between `dplyr` and other packages. For example, there is already a function called `filter()` in the preloaded `stats` package, but from now on any call of `filter()` in our current R session will call another function, namely the function `filter()` in the `dplyr` package. If we need to call `filter()` in the `stats` package, we must use the double colon operator, `stats::filter()`. If the start-up messages are unimportant to the reader, we may suppress them in output generated by R Markdown with the code chunk option `message=FALSE`.

We only need to load a package once during an R session. However, we cannot rely on other users having loaded the same packages that we loaded in our session. Therefore, it is good practice to include the relevant `library()` command in any script or R Markdown file that uses an add-on package. The recommended style is to place all `library()` commands at the beginning of the file (<https://style.tidyverse.org/files.html#internal-structure>).

8.3 styler package

In section 3.1.2, I pointed out the importance of a professional, consistent code style. The `styler` package installs an RStudio menu option that automatically edits R code so that it adheres to the tidyverse style guide. Although the package is not a perfect substitute for reading the style guide at <https://style.tidyverse.org/>, it can implement many aspects of the tidyverse style (e.g. proper indentation and using the operator `<-` instead of an equals sign `=` for assignments).

You can install `styler` from the RStudio Packages tab or with:

```
install.packages("styler")
```

Afterwards, various options for styling the code can be selected from the ‘Addins’ drop-down menu (figure 8.4). I recommend selecting the option ‘Style active file’ from the ‘Addins’ menu before sharing code with others.

8.4 Summary and outlook

User-contributed packages have been a major reason for R’s popularity. Searching the Internet often points towards one or several packages that can simplify our code. A popular suite of packages is the tidyverse. For example, **dplyr** is one of the tidyverse packages. Because we repeatedly use various tidyverse packages, I recommend you install all of them at this point.

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

8.5 Just checking

Suppose we want to inspect the data set `band_members` in the **dplyr** package by running

```
> band_members
```

in the console.

We receive this error message:

```
## Error in eval(expr, envir, enclos): object 'band_members' not found
```

Trying to find the reason for the error, we find that RStudio’s packages tab displays the information shown in figure 8.5.

What action will ensure that we can successfully run the command `band_members`?

- (a) We must restart R.
- (b) We must run `library(dplyr)`.
- (c) We must run `install.packages("dplyr")`.
- (d) We must run `update.packages("dplyr")`.

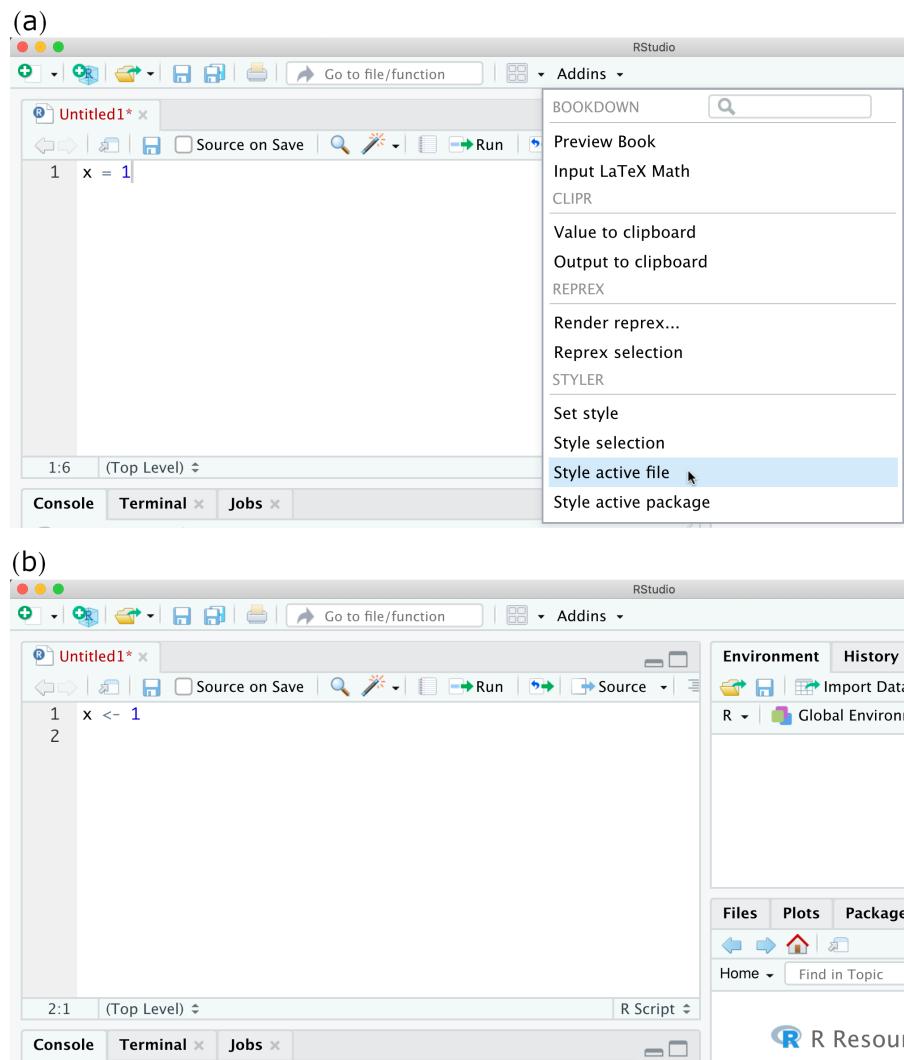


FIGURE 8.4: We can use the **styler** package to implement elements of the tidyverse style. The code shown in panel (a) uses the equals sign `=`, which is not recommended by the style guide. When we choose the menu option ‘Style active file’ from the drop-down menu, `=` is replaced by the assignment operator `<-`, as shown in panel (b).



FIGURE 8.5: In section 8.5, you are asked to interpret this information from RStudio’s packages tab.

You can find the answer in appendix A.7.

Exercise: How profitable is your online shop?

Prerequisite: chapters 1–6

Approximate duration: 120 minutes

Submission format: R script

(I) Creating vectors

Suppose you started a small online shop last week. Table 8.1 shows the number of daily customers and the daily sales. You are trying to figure out your operational strategy based on these data.

- (a) Create two vectors to represent the two columns: `customers` and `sales`. Implement `customers` as an integer vector.
- (b) How can you confirm the class and number of elements in each vector?

TABLE 8.1: Daily number of customers and sales during the first week of operations.

| Day | Customers | Sales (in \$) |
|-----------|-----------|---------------|
| Saturday | 33 | 65.05 |
| Sunday | 22 | 80.17 |
| Monday | 41 | 326.53 |
| Tuesday | 39 | 137.88 |
| Wednesday | 38 | 374.34 |
| Thursday | 46 | 329.04 |
| Friday | 47 | 251.29 |

(II) Vector operations and subsetting

- (a) Generate a new vector called `sales_per_customer` to store the average sales per customer on each day.
- (b) You just double-checked your accounts, and you discovered that you incorrectly entered the sales income for Saturday as \$137.88 instead of \$317.88. Oops! Please correct this value in the `sales` vector with vector subsetting (i.e. do not generate the whole vector again). Afterwards, re-calculate the `sales_per_customer` vector.
- (c) Calculate the total number of customers during this week and the total weekly sales.
- (d) Calculate the mean daily sales on the weekend (i.e. Saturday and Sunday) only.
- (e) Calculate the mean daily sales on weekdays (i.e. Monday to Friday) in two different ways:
 - Select ‘positively’ those indices that correspond to weekdays.
 - Select ‘negatively’ those indices that correspond to weekends.
- (f) Sales during the first two days were low. You are wondering whether it would have been better to pay a freelance web designer for work on the weekend. She would have charged you \$50, and you would not have sold anything that weekend. However, you expect that sales on weekdays would have been up by 10%. Would the additional sales have been bigger than the cost for the web designer?

(III) Writing our own function

In this section, we write a simple function to calculate the profit when implementing different advertising strategies. We begin by defining the daily number of customers and daily sales as vectors, given in the code chunk below.

```
daily_customers <- c(33L, 22L, 41L, 39L, 38L, 46L, 47L)
daily_sales <- c(65.05, 80.17, 326.53, 317.88, 374.34, 329.04, 251.29)
```

- (a) Using these vectors, calculate the total number of customers dur-

- ing the week and the total weekly sales. These numbers should be identical to those in (II)-c.
- (b) Your business consultants recommend paying for online advertisement. The advertising company offers a basic package that costs \$50 per week. Your consultants expect that the ads will attract two additional customers to your website each day. They also predict that these customers spend on average the same amount as other customers on the same day. Calculate the predicted weekly profit (i.e. weekly sales minus advertising cost) with this strategy.
- (c) Write an R function `weekly_profit()` to calculate the weekly profit. This function should accept the following arguments:
- `d_customers`: daily number of customers in the no-advertisement scenario (vector of length 7).
 - `d_sales`: daily sales income in the no-advertisement scenario (vector of length 7).
 - `w_ad_cost`: advertising cost per week.
 - `addl_d_customers`: predicted additional number of daily customers.
- (d) Use `weekly_profit()` to calculate the weekly profit if you do not pay for advertisement. Confirm that the result matches the answer to (II)-c.
- (e) Use `weekly_profit()` to calculate the weekly profit if you purchase the basic package from the advertising company. Confirm that the result matches the answer to (III)-b.
-

(IV) Working with sequences

It appears as if purchasing the basic advertisement package is profitable. You are now considering upgrading. The advertising company offers several packages. Your business consultants expect that, for every additional customer, you need to pay an increasing amount for advertisement because some ads may be shown to the same user multiple times. The costs for advertisement and the number of additional customers are listed in table 8.2.

- (a) Store the number of additional customers in table 8.2 as a vector. Use `seq()`.
- (b) The costs of the advertising packages in table 8.2 follow a regular pattern (50, 50 + 60, ..., 50 + 60 + 70 + 80 + 90 + 100). How can you generate this pattern as a vector with relatively little typing using `cumsum()` and `seq()`? See `?cumsum` or search the World Wide Web for help.
- (c) Using the elements in the vectors from (IV)-a and (IV)-b, calculate

TABLE 8.2: Predicted additional customers per day compared to the no-advertisement scenario and the corresponding costs of advertising packages.

| Package | Additional customers per day | Weekly cost (in \$) |
|---------|------------------------------|---------------------|
| 1 | 2 | 50 |
| 2 | 4 | 110 |
| 3 | 6 | 180 |
| 4 | 8 | 260 |
| 5 | 10 | 350 |
| 6 | 12 | 450 |

the weekly profit under each scenario (i.e. each row in table 8.2) with the function `weekly_profit()` from (III)-c.

(V) Finding the maximum of a vector

- (a) In (IV)-c, you probably ran `weekly_profit()` six times following the pattern:

```

weekly_profit(
  daily_customers,
  daily_sales,
  weekly_ad_costs[1],
  addl_customers_per_day[1]
)
...
weekly_profit(
  daily_customers,
  daily_sales,
  weekly_ad_costs[6],
  addl_customers_per_day[6]
)

```

R code with many almost identical lines may do the job in a quick and dirty manner, but there are almost always better options. In this case, functional programming (i.e. a coding style in which functions are passed as arguments) can shorten the code. We learn about the functional programming tools provided by the `purrr` package in chapter 23. Base R also has adequate tools that

do not require third-party packages, for example the `mapply()` function. The following code chunk uses `mapply()` to store the weekly profit in a vector `wp`. You do not need to memorize this function and its arguments. Here is a brief explanation. The first argument `weekly_profit` is the function that is applied to the remaining arguments. The first two arguments, named `weekly_profit` and `weekly_ad_costs`, are passed to `weekly_profit` in parallel (i.e. `weekly_profit[1]` and `weekly_ad_costs[1]` first, `weekly_profit[2]` and `weekly_ad_costs[2]` second etc.). The final argument `MoreArgs` lists the arguments that are to be held constant in each run of `weekly_profit()`.

```
wp <- mapply(  
  weekly_profit,  
  weekly_ad_costs,  
  addl_customers_per_day,  
  MoreArgs = list(  
    d_customers = daily_customers,  
    d_sales = daily_sales  
  )  
)
```

- (b) Write R code to find the maximum profit from `wp`.
- (c) Write R code to find the advertising package (1-6) that maximises profit. You may find `which.max()` useful.



Part III

Fundamentals of R programming



9

Conditional element selection

When we create a variable with `c()`, we usually insert specific values that do not depend on other parts of the script. Here is an example where we hard-code specific numbers.

```
v <- c(31, -50, 93, 29, -44, 93)
```

However, most of the time, we do not want to hard-code vectors. In many applications, vectors should have elements that depend on other vectors that have already been assigned. For example, we may want to create a vector `w` whose i -th element is the character string "even" if `v[i]` is even and "odd" otherwise.

```
## [1] "odd"  "even" "odd"  "odd"  "even" "odd"
```

In this chapter, we learn about two functions in the `dplyr` package that are useful to create vectors whose elements depend on elements in another vector: `if_else()` and `case_when()`. We also learn about operators that create a logical vector on the basis of one or two logical input vectors.

9.1 Comparison operators

In the example above, the elements of `w` depend on the values of `v %% 2`. (See table 3.1 for the definition of the modulo operator `%%`). If the i -th element of `v %% 2` equals zero, we assign "even" to `w[i]`. Otherwise, the i -th element of `w` is "odd".¹ To make the correct assignment, we need a way to compare the values in `v %% 2` with 0 and 1. The tool for this purpose is the equality operator `==`. The comparison `v %% 2 == 0` returns `TRUE` for all elements for which the left-hand side and right-hand side of the `==` operator are equal. For all other elements, `v %% 2 == 0` returns `FALSE`. The equality operator is vectorised; thus, the result of `v %% 2 == 0` has as many elements as `v`. (See section 3.4.1 for the definition of vectorisation.) Note that we use *two* equals signs (`==`) to make a

¹This method is in fact too simplistic. (Do you see why?) We address this issue in section 9.3.

TABLE 9.1: Comparison operators in R

| Operator | Description |
|---|--|
| <code>x == y</code> | TRUE if and only if $x = y$ |
| <code>x != y</code> | TRUE if and only if $x \neq y$ |
| <code>x < y</code> | TRUE if and only if $x < y$ |
| <code>x > y</code> | TRUE if and only if $x > y$ |
| <code>x <= y</code> | TRUE if and only if $x \leq y$ |
| <code>x >= y</code> | TRUE if and only if $x \geq y$ |
| <code>dplyr::between(x, left, right)</code> | TRUE if and only if $x \geq \text{left}$ and $x \leq \text{right}$ |
| <code>x %in% y</code> | TRUE if and only if x is an element in y |

comparison, whereas we use a single equals sign when we match arguments by name (section 5.2).

```
v %% 2 == 0
## [1] FALSE TRUE FALSE FALSE TRUE FALSE
```

The equality operator is one of several comparison operators in R. We list other common comparison operators in table 9.1.

We can apply the `==` and `!=` operators from table 9.1 to numbers, logical values and character strings.

```
FALSE == FALSE
## [1] TRUE
"farewell" == "welcome"
## [1] FALSE
```

For character strings, we can use the inequality operators `<`, `>`, `<=` and `>=` to check their alphabetical order.²

```
"farewell" < "welcome"
## [1] TRUE
```

²Alphabetical order depends on the user's 'locale', a set of parameters from which R infers the user's location. In Danish, for example, "z" is between "y" and "aa", whereas in Estonian "z" comes between "s" and "t". In this book, I use standard English alphabetical order.

9.2 `if_else()`

Let us return to our earlier example: we want to turn

```
v <- c(31, -50, 93, 29, -44, 93)
```

into a character string whose elements are "even" and "odd".

```
## [1] "odd"  "even" "odd"  "odd"  "even" "odd"
```

So far, we managed to turn `v` into a logical vector whose elements are `TRUE` and `FALSE`.

```
v %% 2 == 0
## [1] FALSE  TRUE FALSE FALSE  TRUE FALSE
```

We still need a way to translate `TRUE` into "even" and `FALSE` into "odd". The `dplyr` package offers a solution in form of the function `if_else()`.³ The basic syntax of `if_else()` is as follows.

```
if_else(condition, true, false)
```

The first argument, named `condition`, is a logical vector. It is often the result of one of the comparison operators in table 9.1. The remaining two arguments, named `true` and `false`, must be vectors of the same class (e.g. both are `numeric` or both are `character`). The function call `if_else(condition, true, false)` returns a vector of the same length as `condition` with elements

- `true[i]` if `condition[i]` is `TRUE`,
- `false[i]` if `condition[i]` is `FALSE`.

Here's an example.

```
library(dplyr)
if_else(c(TRUE, TRUE, FALSE, FALSE), 1:4, -1:-4)
## [1] 1 2 -3 -4
```

The vectors `true` and `false` must be either as long as `condition` or of length 1. In the latter case, R performs its usual vector recycling (section 3.5).

³An alternative to `dplyr`'s `if_else()` is the built-in function `ifelse()`. However, `if_else()` gives faster and more intuitive results. See `?if_else()` for details.

```
if_else(c(TRUE, TRUE, FALSE, FALSE), 1, -1)
## [1] 1 1 -1 -1
```

Returning to our exercise, here is how we turn `v` into a character string with elements "even" and "odd".

```
if_else(v %% 2 == 0, "even", "odd")
## [1] "odd"  "even" "odd"  "odd"  "even" "odd"
```

9.3 case_when()

The solution we just proposed in the previous code chunk is not completely correct because we have, so far, assumed that `v` contains only integers. When we insert a floating-point value into `v`, we may get a wrong answer.

```
x <- c(31, -50, 93, 29, -44, 93, -91.5)
if_else(x %% 2 == 0, "even", "odd")
## [1] "odd"  "even" "odd"  "odd"  "even" "odd"  "odd"
```

The last element in the output is not meaningful because non-integer numbers are neither even nor odd. Suppose we want to set the corresponding character string to "neither even nor odd". In principle, we can insert `if_else()` inside another `if_else()`.

```
if_else(
  x %% 2 == 0,
  "even",
  if_else(
    x %% 2 == 1,
    "odd",
    "neither even nor odd"
  )
)
## [1] "odd"           "even"          "odd"
## [4] "odd"           "even"          "odd"
## [7] "neither even nor odd"
```

Nesting one `if_else()` inside another `if_else()` may be appropriate when we check two conditions, but the code becomes difficult to read when we have

even more conditions to check. A better alternative is the **dplyr** function `case_when()`.

```
case_when(
  x %% 2 == 0 ~ "even",
  x %% 2 == 1 ~ "odd",
  TRUE ~ "neither even nor odd"
)
## [1] "odd"           "even"          "odd"
## [4] "odd"           "even"          "odd"
## [7] "neither even nor odd"
```

If there are more than two conditions to check, we can add more arguments using the pattern `condition ~ value` in `case_when()`. R checks the conditions on the left side of the tilde `~` starting from the top. As soon as a condition evaluates to `TRUE`, `case_when()` returns the corresponding value to the right of the tilde and ignores any of the remaining conditions. The last condition, `TRUE ~ "neither even nor odd"`, is a catch-all condition that takes effect if none of the conditions in the previous lines are true. The syntax may look unusual because of the tilde. However, compared to nested `if_else()` statements, `case_when()` allows us to line up all conditions with equal indentation, which is semantically preferable. I recommend that we use `if_else()` if we only check one condition. Otherwise `case_when()` is the better option.

9.4 Logical operators

So far, we have always used a single logical vector as the condition (i.e. first argument) in `if_else()`. In each argument of `case_when()`, we also always have had only one condition on the left-hand side of the `~` operator until now. However, in many applications, conditions depend on more than one vector. Suppose we operate a website that only permits users to log in if they provide a PIN code and a password. The next code chunk shows hypothetical data about the login attempts of four users. The i -th element in the vector `pin` is `TRUE` if the user entered the correct PIN. Otherwise, `pin[i]` is `FALSE`. We also create a logical vector `password` that indicates whether the user entered the correct password.

```
pin <- c(TRUE, TRUE, FALSE, FALSE)
password <- c(FALSE, TRUE, FALSE, TRUE)
```

We want to find out which users entered the PIN *and* the password correctly.

In the jargon of computer programming, we need an AND operator to create the effect of the word ‘*and*’ in the previous sentence. The AND operator in R is the ampersand &. It returns TRUE if and only if both input elements are TRUE. The output of `pin & password` shows that only the second user had two valid credentials.

```
pin & password
## [1] FALSE TRUE FALSE FALSE
```

After your customers complained that they find it too tedious to look up their PIN and password every time, you change the security settings of your website so that it suffices to provide only the PIN *or* only the password *or* both. To represent the condition described in the previous sentence, we need an OR operator. In R, the OR operator is a vertical bar |.

```
pin | password
## [1] TRUE TRUE FALSE TRUE
```

We also often encounter situations where we need a NOT operator. Its purpose is to return a vector in which TRUE in the input is turned into FALSE in the output and vice versa. The NOT operator in R is the exclamation mark !. For example, `!pin` indicates whether a user did *not* submit a correct PIN.

```
!pin
## [1] FALSE FALSE TRUE TRUE
```

With the three logical operators &, | and !, we can construct complex queries about data. For example, to find users of our website who are female, younger than 40 and neither married nor divorced, we would schematically create the following search term.

```
sex == "female" &
age < 40 &
!(status == "married" | status == "divorced")
```

When working with expressions like these, it is important to be aware of the order in which R carries out the operators. The comparison operators of table 9.1 (e.g. == and <) are always carried out before the logical operators &, | and !. We can find the complete rules of ‘operator precedence’ under ?Syntax.

9.5 Summary and outlook

In this chapter, we learned how to express conditions in R. We can now determine whether an element in one vector is larger than an element in another vector with the comparison operators in table 9.1. We now also know how to combine two logical vectors with AND, OR and NOT operators. In R,

- the AND operator is `&`. It operates elementwise and returns `TRUE` if and only if the corresponding elements in the vectors on either side of the `&` are both `TRUE`.
- the OR operator is `|`. It also operates elementwise and returns `FALSE` if and only if the corresponding elements are both `FALSE`.
- the NOT operator is `!`. It changes all `TRUE` elements to `FALSE` and vice versa.

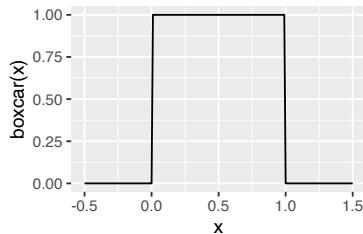
We also learned how to express conditional statements with `if_else()` and `case_when()`. We encounter conditional expressions in many later chapters (e.g. when we filter data in data frames in chapter 19). The operators and functions we learned in this chapter will turn out to be handy in these contexts.

9.6 Just checking

Suppose we want to implement the ‘boxcar function’

$$\text{boxcar}(x) = \begin{cases} 1 & \text{if } 0 < x < 1, \\ 0 & \text{otherwise.} \end{cases}$$

The graph of the function is sketched below.



Which of the following four R functions correctly implements the boxcar function? First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.8.

- (a) `boxcar_a <- function(x) {`

```
dplyr::case_when(  
  (x < 1) ~ 1,  
  (x <= 0) ~ 0,  
  TRUE ~ 0  
)  
}  
  
(b) boxcar_b <- function(x) {  
  dplyr::case_when(  
    (x <= 0) ~ 0,  
    (x >= 1) ~ 0,  
    TRUE ~ 1  
)  
}  
  
(c) boxcar_c <- function(x) {  
  dplyr::case_when(  
    (x > 0) ~ 1,  
    (x >= 1) ~ 0,  
    TRUE ~ 0  
)  
}  
  
(d) boxcar_d <- function(x) {  
  dplyr::case_when(  
    (x > 0) ~ 1,  
    (x < 1) ~ 1,  
    TRUE ~ 0  
)  
}
```

10

Data type conversion

We stated in section 3.1.7 that all elements in a vector must be in the same class. In this chapter, we learn what happens when we try to break this rule with brute force. We find out that R automatically converts one data type to another to ensure that all elements in a vector belong to the same class. This data type conversion follows simple rules. In some cases, we may want to deliberately take advantage of R’s implicit data type conversion. In many other cases, data type conversion can be the source of bugs and unexpected behaviour. Therefore, we should be aware of the conversion rules to work efficiently with R.

10.1 Mixing numeric and character elements produces a character vector

Let us combine one numeric and one character vector.

```
# Numeric vector
num <- seq(-0.25, 0.75, 0.5)

# Character vector
chr <- c("breakfast", "lunch", "dinner")
num_and_chr <- c(num, chr)
num_and_chr
## [1] "-0.25"      "0.25"       "0.75"       "breakfast"   "lunch"      "dinner"
```

In the output, the quotes around each element are a telltale sign of a character vector. Checking the class of `num_and_chr` confirms our suspicion.

```
class(num_and_chr)
## [1] "character"
```

The order in which we combine `num` and `chr` affects the order of the elements, but it does not affect the class of the combined vector.

```
chr_and_num <- c(chr, num)
chr_and_num
## [1] "breakfast" "lunch"      "dinner"     "-0.25"      "0.25"      "0.75"
class(chr_and_num)
## [1] "character"
```

There is a good reason why R chooses both combinations (i.e. `num_and_chr` and `chr_and_num`) to be character vectors. Every number can easily be turned into a character string by putting quotation marks around it. The opposite direction would not be clear. For example, how should we turn "`lunch`" into a number? We conclude that the `character` class can store more general information than the `numeric` class.

However, there is a price to pay for using the more general `character` class: we cannot perform arithmetic operations with characters, even if they contain only numbers between the quotation marks. Note the different output after the following two code chunks.

```
42 + 8
## [1] 50
```

```
"42" + 8
## Error in "42" + 8: non-numeric argument to binary operator
```

Inequality operators may also return confusing output after a number has been converted into a character string. Again note the different output after the next two code chunks.

```
42 > 8
## [1] TRUE
```

```
"42" > 8
## [1] FALSE
```

The logic behind the latest output is that the letter 4 comes alphabetically before the letter 8.

10.2 General rules of coercion

When merging vectors of two different classes, R always ‘coerces’ the result into one of the two classes involved. It chooses the class that can store more general information. We can sort the four main classes in R from the least to the most general type as follows:

$$\text{logical} < \text{integer} < \text{numeric} < \text{character}. \quad (10.1)$$

When we merge two different classes, the result is always in the class farther to the right in this hierarchy.

Let us set up some examples.

```
# Logical vector
logi <- c(TRUE, FALSE)

# Integer vector
int <- 1:3

# Numeric vector
num <- seq(-0.25, 0.75, 0.5)

# Character vector
chr <- c("breakfast", "lunch", "dinner")
```

Combining a logical and a character vector produces a character vector. TRUE and FALSE are turned into the character strings "TRUE" and "FALSE", respectively.

```
c(logi, chr)
## [1] "TRUE"      "FALSE"      "breakfast"   "lunch"      "dinner"
class(c(logi, chr))
## [1] "character"
```

Combining a logical and an integer vector yields an integer vector. TRUE and FALSE are turned into the integers 1 and 0, respectively.

```
c(logi, int)
## [1] 1 0 1 2 3
class(c(logi, int))
## [1] "integer"
```

Similarly, when logical values are coerced into the `numeric` class, `TRUE` and `FALSE` turn into 1 and 0, but now the numbers are of the `numeric` rather than `integer` class.

```
c(logi, num)
## [1] 1.00 0.00 -0.25 0.25 0.75
class(c(logi, num))
## [1] "numeric"
```

10.3 Data type conversion with the `as.-`family of functions

Coercion is the data type conversion that happens when R *implicitly* forces data of different classes into one and the same class. Sometimes, we would like to deliberately change data types ourselves. For *explicit* conversion between different classes, we can use the `as.-`family of functions. For example, `as.logical()` converts its argument to a logical vector. There are also functions called `as.integer()`, `as.numeric()` and `as.character()` to convert the argument into an integer, numeric and character vector, respectively.

We can predict most of the results of the `as.-`functions from our observations in section 10.2. For example, `as.numeric()` applied to a logical vector turns `TRUE` into 1 and `FALSE` into 0.

```
as.numeric(c(TRUE, FALSE))
## [1] 1 0
```

`as.logical()` applied to a number (regardless of whether `integer` or `numeric`) results in

- `FALSE` if the number is exactly 0,
- `TRUE` otherwise.

```
as.logical(seq(-1.5, 1.5, 0.5))
## [1] TRUE TRUE TRUE FALSE TRUE TRUE TRUE
```

`as.logical()` applied to a character string results in

- `FALSE` if the string is "FALSE", "False", "false" or "F",
- `TRUE` if the string is "TRUE", "True", "true" or "T",
- a missing value `NA` (for 'Not Available') in all other cases. We learn more about `NA` in section 11.1.

```
as.logical(c("FALSE", "true", "lunch"))
## [1] FALSE TRUE NA
```

When we apply `as.integer()` to a number, R truncates this number at the decimal point. If the number is negative, `as.integer()` rounds up. If it is positive, `as.integer()` rounds down.¹

```
as.integer(c(-512.50, -41.49, -0.81, 0, 0.81, 41.49, 512.50))
## [1] -512 -41 0 0 0 41 512
```

`as.numeric()` converts those character strings to `numeric` that can be interpreted as numbers. Everything else is converted to `NA`. When `NA` appears in the result, R issues a warning.

```
as.numeric(c("-2.6", "dinner", "TRUE"))
## Warning: NAs introduced by coercion
## [1] -2.6 NA NA
```

`as.integer()` and `as.numeric()` behave similarly when they face a character-valued argument. However, `as.integer()` also truncates floating-point numbers at the decimal point.

```
as.integer(c("-2.6", "dinner", "TRUE"))
## Warning: NAs introduced by coercion
## [1] -2 NA NA
```

10.4 Taking advantage of data type conversion in R programming

In many cases, we can rely on R to generate sensible output by converting arguments to a more general data type [i.e. by moving to the right in equation (10.1)]. For example, R converts logical values to numbers before performing arithmetic operators or mathematical functions.

¹There is another similar function `trunc()`, which also rounds towards 0, but `trunc()` returns a `numeric` vector instead of an `integer` vector. `trunc()` is part of a larger family of functions—also containing `ceiling()`, `floor()`, `round()` and `signif()`—that perform rounding of numbers in various ways and return `numeric` output.

```
TRUE + TRUE
## [1] 2
cos(FALSE)
## [1] 1
```

In other cases, R automatically converts logical or numeric arguments to character strings if it is appropriate. One such example is `nchar()`, a function that counts the number of characters in a string. `nchar()` takes a character vector as argument. If the argument is not a character vector, then `nchar()` implicitly coerces the argument.

```
nchar(c("welcome", "farewell"))
## [1] 7 8
nchar(c(42, -3.6047))
## [1] 2 7
nchar(c(TRUE, FALSE))
## [1] 4 5
```

We can sometimes take advantage of R's data type conversion to achieve our objectives with shorter and faster code. For example, it is common practice to determine the number of `TRUE` elements in a logical vector with `sum()`.

```
logical_vec <- rep(c(TRUE, FALSE), c(500, 1500))
sum(logical_vec)
## [1] 500
```

It is also common to determine the fraction of `TRUE` elements with `mean()`.

```
mean(logical_vec)
## [1] 0.25
```

However, in many other cases, it leads to clearer code when data types are explicitly converted with one of the `as.-functions`.

10.5 Summary and outlook

Unanticipated data coercion can be a source of mistakes. However, as long as one is aware of the consequences, the rules of coercion, given by equation (10.1), are straightforward. As I pointed out in section 10.4, summing the elements

of logical vectors as if they were numbers is, in fact, a common shortcut that we apply in later chapters.

10.6 Just checking

Find the correct answer option for the following questions. First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.9.

- (I) We run the following commands.

```
t <- c(1, 2)
u <- c(TRUE, FALSE)
v <- c("apple", "banana")
w <- c(t, u)
x <- c(v, w)
```

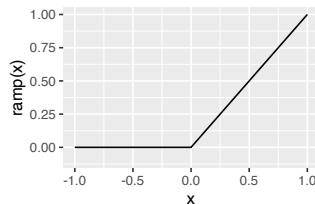
What is the output of `print(x)`?

- (a) ## [1] NA NA 1 2 1 0
- (b) ## [1] "apple" "banana" "1" "2" "1" "0"
- (c) ## [1] "apple" "banana" "1" "2" "TRUE" "FALSE"
- (d) ## [1] NA NA "1" "2" "TRUE" "FALSE"

- (II) Which of the following four R functions correctly implements the ‘ramp function’?

$$\text{ramp}(x) = \begin{cases} x & \text{if } x \text{ is positive,} \\ 0 & \text{otherwise.} \end{cases}$$

The function values are plotted below.



(a) `ramp_a <- function(x) {
 (x < 0) * x
}`

(b) `ramp_b <- function(x) {
 -(x > 0) * x
}`

(c) `ramp_c <- function(x) {
 -(x < 0) * x
}`

(d) `ramp_d <- function(x) {
 (x > 0) * x
}`

11

Missing and undefined values

In section 6.7, we learned that some R functions (e.g. `cat()`) return the special value `NULL` as a signal that there are no data to be returned.

```
x <- cat("I do not return anything.")  
## I do not return anything.  
x  
## NULL
```

`NULL` is not the only value R uses for missing or undefined data. In section 3.3.5, we saw that R inserts `NA` (Not Assigned) when we insert an element into a vector `v` at an index that is greater than `length(v) + 1`.

```
v <- 1:5  
v[10] <- -1  
v  
## [1]  1  2  3  4  5 NA NA NA NA -1
```

The purpose of this chapter is to shed light onto the meaning of these different values. We learn that there are four main reasons why we may encounter missing or undefined values in R:

- (1) Our statistical data have missing entries.
- (2) We apply a mathematical formula whose result is infinite ...
- (3) ... or not unique.
- (4) An R object does not exist.

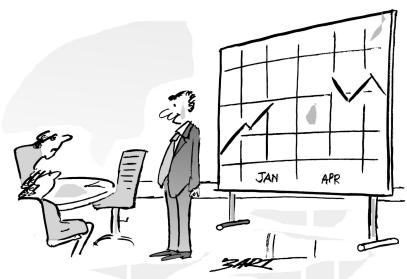
R uses distinct values for these four cases:

- (1) `NA`: Not Available
- (2) `Inf` and `-Inf`: positive and negative infinity
- (3) `NaN`: Not a Number
- (4) `NULL`

This list is a good general guideline for the purpose of `NA`, `Inf`, `NaN` and `NULL`.

However, R has many quirks and idiomatic applications for these objects; thus, it makes sense to discuss them in more detail.

11.1 NA for missing data



*"There are no data between January and April,
as my department was abducted by aliens."*

CartoonStock.com

FIGURE 11.1: Missing values can arise for a variety of reasons. Cartoon by BART (Search ID: bron1954), reproduced from CartoonStock.com with permission.

Statistical samples often have missing data points (e.g. because the measurement equipment failed, a participant in a survey gave no response; also see figure 11.1). The purpose of the value `NA` is to represent such missing data. We can insert `NA` into vectors of all data types (e.g. `numeric` or `character`).

```
x <- c(1, 2, 3, NA, 5, 6)
y <- c("papa", NA, "bravo", "echo", NA)
```

We do not put `NA` inside quotation marks unless we truly mean the character string "`NA`" rather than a missing value. R does not print `NA` in quotes either.

```
y
## [1] "papa"   NA        "bravo"  "echo"   NA
```

By default, `NA` is a `logical` value.

```
class(NA)
## [1] "logical"
```

However, `NA` can change its class like a chameleon can change its colour. When `NA` is combined with non-logical data (e.g. `numeric` or `character` vectors), R turns `NA` into the most general class [i.e. the class farthest to the right in equation (10.1)] involved in this operation. R prints out the same symbol `NA` (without quotes!), no matter what the class of `NA` is.

```
x[4]
## [1] NA
class(x[4])
## [1] "numeric"
y[2]
## [1] NA
class(y[2])
## [1] "character"
```

If we want to be explicit about the class of a missing value, we can use the constants `NA_integer_` and `NA_character_`.

```
class(NA_integer_)
## [1] "integer"
```

For example, we need to specify the type of `NA` in `if_else()` conditions to match the classes of the second and third arguments.

```
dplyr::if_else(c(TRUE, FALSE), 1:2, NA)
## Error in `dplyr::if_else()`:
## ! `false` must be an integer vector, not a logical vector.
dplyr::if_else(c(TRUE, FALSE), 1:2, NA_integer_)
## [1] 1 NA
```

One might think that the explicit version of `NA` for the `numeric` class would be `NA_numeric_`. Alas, this constant is named differently: `NA_real_`.

```
class(NA_real_)
## [1] "numeric"
```

Most operations and functions return `NA` when at least one of their arguments is `NA`.

```
# NA + 2 is equal to NA
x + 2
## [1] 3 4 5 NA 7 8
```

```
# The number of characters in NA is equal to NA
nchar(y)
## [1] 4 NA 5 4 NA
```

When we want to find out whether an element equals a given value, we usually apply the `==` operator. For example:

```
x == 2
## [1] FALSE TRUE FALSE NA FALSE FALSE
```

By analogy, we might, at first glance, believe that we find the `NA` values in `x` with `x == NA`. However, the output from `x == NA` does not answer whether an element is `NA`. Instead, all elements of `x == NA` are themselves `NA`.

```
x == NA
## [1] NA NA NA NA NA NA
```

On second thought, the previous output makes sense. `NA` is a vector of length 1. Thus, `NA` is recycled six times to match the length of `x`. The vectorised operation `==` then carries out the comparisons `x[1] == NA, ... x[6] == NA`. As we have just learned, operations with `NA` as an argument usually return `NA`. The `==` operator follows this rule and, consequently, returns `NA` six times.

The proper way to determine whether an element equals `NA` is the function `is.na()`.

```
is.na(x)
## [1] FALSE FALSE FALSE TRUE FALSE FALSE
```

If we want find out whether a vector contains an `NA` element, we could use the `any()` function from section 3.7.

```
any(is.na(x))
## [1] TRUE
```

A shortcut for `any(is.na())` is `anyNA()`, which tends to run slightly faster.

```
anyNA(x)
## [1] TRUE
```

Consistent with the rule that operations involving `NA` result in `NA`, the mean of a vector with `NA` equals `NA`.

```
mean(x)
## [1] NA
```

On the one hand, this return value makes sense: we cannot determine the mean if one of the values is missing. On the other hand, it is also sensible to compute the mean for the remaining (i.e. non-NA) values. In principle, we can use the `is.na()` function to remove the NA elements before computing the mean.

```
mean(x[!is.na(x)])
## [1] 3.4
```

The result is correct, but there is a more elegant solution. We can add the argument `na.rm = TRUE` in `mean()` to achieve the same effect while keeping the command more intuitively readable.

```
mean(x, na.rm = TRUE)
## [1] 3.4
```

In section 3.4.3, we saw many more R functions for summary statistics besides `mean()`. Most of them accept the optional argument `na.rm = TRUE`. It is worth checking the R documentation of the respective function before attempting any complicated subsetting strategy to remove NA from our data.

11.2 *Inf* and *-Inf* for infinities

Besides missing data, a calculation may also run aground because it yields infinite results (i.e. a mathematical function is evaluated at a singularity). Typical cases are the functions $1/x$ and $\log(x)$ evaluated at 0 (figure 11.2).

When R evaluates $1/x$ at $x = 0$, it returns `Inf` to signal positive infinity. For $\log(0)$, R returns `-Inf`, which stands for negative infinity. The sign is consistent with the behaviour of the functions sketched in figure 11.2.

```
x <- seq(0, 1, 0.25)
1/x
## [1]      Inf 4.000000 2.000000 1.333333 1.000000
log(x)
## [1]      -Inf -1.3862944 -0.6931472 -0.2876821  0.0000000
```

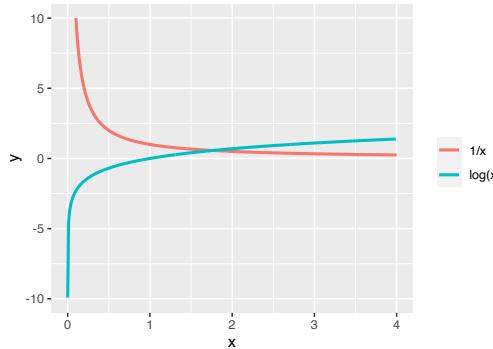


FIGURE 11.2: The mathematical functions $1/x$ and $\log(x)$ are examples of functions that have singularities (i.e. the function value tends to positive or negative infinity as we approach a critical value of x , here $x = 0$).

When R produces `NA` because of data type coercion (see section 10.3), it issues a warning. However, R does not warn when it produces *infinite* values.

The class of `Inf` and `-Inf` is `numeric`.

```
class(Inf)
## [1] "numeric"
```

We can insert `Inf` and `-Inf` in arithmetic operations and obtain sensible results.

```
Inf + 1
## [1] Inf
1 / Inf
## [1] 0
```

We can identify elements that are either `Inf` or `-Inf` with the `==` operator.

```
x <- c(1/0, log(0), 1234.56)
x == Inf
## [1] TRUE FALSE FALSE
x == -Inf
## [1] FALSE TRUE FALSE
```

The inequality operators from table 9.1 also work as expected.

```
x > 0
## [1] TRUE FALSE TRUE
```

To find infinite values, regardless of whether they are positive or negative, we use `is.infinite()`.

```
is.infinite(x)
## [1] TRUE TRUE FALSE
```

Finite values can be identified by `is.finite()`.

```
is.finite(x)
## [1] FALSE FALSE TRUE
```

A return value `Inf` does not necessarily mean infinity in a strict mathematical sense. We may also encounter `Inf` when we work with large, but mathematically finite, numbers.

```
x <- 305:310
10^x
## [1] 1e+305 1e+306 1e+307 1e+308      Inf      Inf
```

Although 10^{309} is finite, R cannot distinguish between `Inf` and a number exceeding the limit that is set by the 64-bit double-precision format ($\approx 1.8 \cdot 10^{308}$) that R uses for numeric elements (see https://en.wikipedia.org/wiki/Double-precision_floating-point_format for the technical specification). We can look up the maximum double number as follows.

```
.Machine$double.xmax
## [1] 1.797693e+308
```

A similar phenomenon happens for very small numbers.

```
x <- -320 : -325
10^x
## [1] 9.999889e-321 9.980126e-322 9.881313e-323 9.881313e-324 0.000000e+00
## [6] 0.000000e+00
```

The numbers start deviating from exact integer powers of 10 and even become at some point indistinguishable from zero. We can find the smallest number that R can accurately represent as follows.

```
.Machine$double.xmin
## [1] 2.225074e-308
```

The problem that very large and very small numbers are not accurately represented is not specific to R. It happens in all standard programming languages running on standard hardware.

Errors caused by accidentally breaching the upper limit ($\approx 1.8 \cdot 10^{308}$) are called *numeric overflow*. Errors caused by breaching the lower limit ($\approx 2.2 \cdot 10^{-308}$) are called *numeric underflow*.

There are special packages (e.g. the **gmp** package) to widen these limits, but they are beyond the scope of this book. These packages slow down runtimes tremendously and require some knowledge about the way a computer internally represents numbers by bits. For most data analysis problems, the standard 64-bit double-precision floating-point format is sufficient.

11.3 **NaN** for undefined numeric results

When a mathematical expression is not uniquely defined, R returns the value **NaN**, which stands for “Not a Number.” For example, the fractions $0/0$ and ∞/∞ can take any finite or infinite values, depending on what limits the numerators and denominators represent. For this reason, it would not be appropriate to assign any particular number to $0/0$ and ∞/∞ , regardless of whether the number would be finite, **Inf** or **-Inf**. The special value **NaN** is R’s way to communicate this situation to us.

```
0 / 0
## [1] NaN
Inf / Inf
## [1] NaN
```

The class of **NaN** is **numeric**.

```
class(NaN)
## [1] "numeric"
```

We can insert **NaN** in arithmetic operations. The result is then also **NaN**.

```
10 - NaN
## [1] NaN
```

In the previous expressions, R did not issue a warning. However, under certain circumstances, R does warn about generating `NaN`.

```
log(-1)
## Warning in log(-1): NaNs produced
## [1] NaN
```

We cannot use the `==` operator to identify `NaN` elements. It produces `NA` instead of `TRUE` or `FALSE`.

```
x <- c(1, Inf, NaN, NA)
x == NaN
## [1] NA NA NA NA
```

Instead we must use `is.nan()`.

```
is.nan(x)
## [1] FALSE FALSE TRUE FALSE
```

11.4 NULL

While `NA`, `Inf` and `NaN` serve clearly defined purposes, there is no simple explanation for the meaning of `NULL`. It is so special that it is even literally a class of its own.

```
class(NULL)
## [1] "NULL"
```

Another difference is that `NULL` has length 0 whereas the other missing and undefined values have length 1.

```
length(NULL)
## [1] 0
```

We can set a vector to `NULL`.

```
v <- NULL
v
## NULL
```

We can test with `is.null()` whether a vector is `NULL`.

```
is.null(v)
## [1] TRUE
```

When we insert `NULL` into a vector, it is omitted from the final result.

```
v <- c(1, Inf, NULL, NA, NaN)
v
## [1] 1 Inf NA NaN
```

When we attempt to replace an existing element by `NULL`, R throws an error.

```
w <- 11:14
w[1] <- NULL
## Error in w[1] <- NULL: replacement has length zero
```

Some functions return `NULL` as a signal that they do not return any value. We have already seen that `cat()` returns `NULL`. Another function that sometimes returns `NULL` is `attributes()`. Attributes are, loosely speaking, pieces of information we can add to an R object. Simple vectors, such as those we have worked with so far, do not have attributes.

```
attributes(1:10)
## NULL
```

Examples where the `attributes()` function does not return `NULL` are factors, matrices and data frames, which we encounter in sections 12–14.

Besides indicating the lack of a return value, the other main use of `NULL` is to remove information by assigning `NULL` to an existing value (e.g. an attribute of an R object or an element in a list; see section 13.2 and chapter 22).

11.5 Summary and outlook

In this chapter, we learned about R’s symbolic expressions for missing and undefined values. The most common among these expressions is `NA`, which stands for missing information in the input (e.g. because a participant in an experiment gave no response or because a survey excluded some geographic regions). In later chapters, we often have to remove `NA` values during data analysis.

`Inf`, `-Inf` and `NaN` usually arise because of mathematical operations in the analysis. For example, when working with quantitative data from a skewed distribution, it is common to apply logarithmic transformations. If any of the original data points is equal to zero, the logarithm becomes `-Inf`.

`NULL` is less common in practice than `NA`, `Inf`, `-Inf` and `NaN`, but we still encounter it in some contexts (e.g. when we want to remove elements or attributes); thus, it is good to be aware of its connotation.

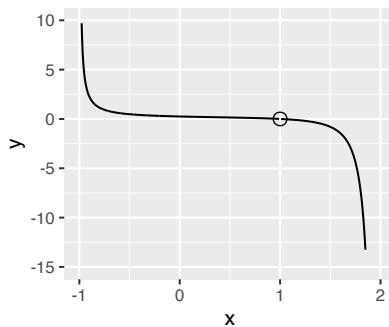
11.6 Just checking

Try to answer the following questions. You can find answers in appendix A.10.

- (I) Compare the results of `letters[c(1, NA)]` and `letters[c(TRUE, NA)]`. Can you explain the difference?
- (II) Consider the following function.

```
ratio <- function(x) {
  numerator <- (1-x)^2
  denominator <- (1-x) * (1+x) * (x-2)^2
  numerator / denominator
}
```

The function values are plotted below.



What is the output of `print(ratio(-1:2))`?

- (a) [1] `Inf` 0.25 0.00 `-Inf`
- (b) [1] `Inf` 0.25 `NaN` `-Inf`

- (c) [1] Inf 0.25 -Inf -Inf
- (d) [1] Inf 0.25 Inf -Inf

12

Factors

So far, we have worked with vectors that belonged to the classes `numeric`, `integer`, `logical` and `character`. In this chapter, we encounter a new class, `factor`, which is R's data structure for representing categorical data. Factors are more complicated than the simple vectors we have worked with until now. However, when working with categorical data, there are many tasks that can be accomplished more easily with factors than with simple vectors.

12.1 Example: US states and US regions

R ships with several built-in data sets about US states. For example, there is a built-in character vector `state.name`.

```
state.name
## [1] "Alabama"      "Alaska"       "Arizona"      "Arkansas"
## [5] "California"   "Colorado"     "Connecticut"   "Delaware"
## [9] "Florida"       "Georgia"      "Hawaii"       "Idaho"
## [13] "Illinois"      "Indiana"      "Iowa"         "Kansas"
## [17] "Kentucky"      "Louisiana"    "Maine"        "Maryland"
## [21] "Massachusetts" "Michigan"     "Minnesota"    "Mississippi"
## [25] "Missouri"      "Montana"      "Nebraska"     "Nevada"
## [29] "New Hampshire" "New Jersey"   "New Mexico"   "New York"
## [33] "North Carolina" "North Dakota" "Ohio"         "Oklahoma"
## [37] "Oregon"        "Pennsylvania" "Rhode Island" "South Carolina"
## [41] "South Dakota"   "Tennessee"   "Texas"        "Utah"
## [45] "Vermont"        "Virginia"    "Washington"  "West Virginia"
## [49] "Wisconsin"      "Wyoming"
class(state.name)
## [1] "character"
```

There is also a data set `state.region` that indicates the region (Northeast, South, North Central or West) that each state belongs to.

```
state.region
## [1] South      West       West       South      West
## [6] West       Northeast  South      South      South
## [11] West      West       North Central North Central North Central
## [16] North Central South     South      Northeast   South
## [21] Northeast North Central North Central South     North Central
## [26] West       North Central West     Northeast  Northeast
## [31] West      Northeast  South      North Central North Central
## [36] South     West       Northeast  Northeast  South
## [41] North Central South     South      West      Northeast
## [46] South     West       South      North Central West
## Levels: Northeast South North Central West
```

At first glance, it may appear as if `state.region` is a character vector. However, after careful inspection, we notice two differences between the output of the character vector `state.name` and the output of `state.region`.

- The character strings in the output of `state.region` are not inside quotes.
- The output of `state.region` has an additional line at the end that starts with the word `Levels`.

Why do we get this unfamiliar output format? The answer lies in the class of `state.region`: it is a factor, not a character vector.

```
class(state.region)
## [1] "factor"
```

We can confirm that `state.region()` is a factor with `is.factor()`.

```
is.factor(state.region)
## [1] TRUE
```

Why do we need a `factor` class? Let us briefly think about the types of data that we may encounter in data analysis.

12.2 Categorical data

Statistical data fall into three main classes:

- **Identifiers:** unique values associated with individual observations. Here are some examples:
 - names of US states
 - passport numbers

- Bayer names that identify stars in the sky
- **Quantitative data:** numbers with which we can perform arithmetic operations such as addition or multiplication. Here is a list of examples:
 - US state area (available in R as numerical vector `state.area`)
 - height of a person
 - diameter of a star
- **Categorical data:** a variable that splits the observations into a small number of groups or categories, for example:
 - the region of a US state (Northeast, South, North Central, West).
 - sex (male, female, other).
 - the classification of stars by surface temperature with the Morgan–Keenan system (O, B, A, F, G, K and M).

We often use words or individual characters to identify groups of categorical data. In other cases, we may use numbers. For example, an opinion poll may represent ‘I agree’ by +1, ‘no opinion’ by 0 and ‘I disagree’ by –1. However, these numbers do not have any arithmetic connotation. It would not make sense to compute, for instance, the standard deviation of the opinions.

Although categories are represented by words or numbers, there are semantic differences between data represented by numeric or character vectors on the one hand and categorical data on the other hand. For example, when working with character vectors, it is a perfectly sensible task to count the number of characters in each element with `nchar()` (see section 10.4). By contrast, it does not matter how many characters we use to name a category. The essence of categorical data is the meaning attributed to each category, not the category name. For this reason, we would like to have a data structure that allows us to easily rename a category without changing the meaning of the represented data. Factors in R are designed to perform this and many other tasks specific to categorical data.

12.3 Creating a factor

A globally operating coffeehouse chain sells beverages in the sizes ‘Short’, ‘Tall’, ‘Grande’ and ‘Venti’ (figure 12.1). Suppose the chain wants to collect information about the sizes ordered by their customers. We can treat the sizes as categorical data and store them in a factor. Let us assume that the first four customers ordered the sizes ‘Grande’, ‘Venti’, ‘Grande’ and ‘Tall’. We create a factor from these data in two steps. First we enter the data as a character vector.



CartoonStock.com

FIGURE 12.1: Cartoon by Bill Whitehead, reproduced from CartoonStock.com with permission.

```
size <- c("Grande", "Venti", "Grande", "Tall")
```

Then we create a factor from these data with the function `factor()`.

```
factor(size)
## [1] Grande Venti Grande Tall
## Levels: Grande Tall Venti
```

We observe that there are no quotes around the character strings in the output, and there is a final line starting with the word `Levels`.

In R jargon, *levels* are the names of the categories. The character strings printed after `Levels`: are the different levels in the factor. By default, they are listed in alphabetical order. In the example above, there are three categories: ‘Grande’, ‘Tall’, ‘Venti’.

Wait! Did we not say that there were *four* categories? What happened to the category ‘Short’? We can answer the question with another question: how should R have known about the category ‘Short’? ‘Short’ did not appear in the vector `size`.

We can make R aware of the names of *all* categories by passing an argument named `levels` to `factor()`.

```
f1 <- factor(size, levels = c("Short", "Tall", "Grande", "Venti"))
f1
## [1] Grande Venti Grande Tall
## Levels: Short Tall Grande Venti
```

‘Short’ now appears among the levels. In the final line of output, the levels are listed in the same order that we used in our `levels` argument. This feature allows us to keep the levels in non-alphabetical order.

We obtain `NA` when we use `factor()` with the `levels` argument and do not specify the level that appears in the corresponding element of the first argument.

```
f2 <- factor(
  c("Venti", "Tall", "Short", "Piccolo"),
  levels = c("Short", "Tall", "Grande", "Venti")
)
f2
## [1] Venti Tall Short <NA>
## Levels: Short Tall Grande Venti
```

In the output, `<NA>` is surrounded by angle brackets. If there is a factor called “`NA`”, it appears in the output as `NA` without angle brackets.

```
f3 <- factor(
  c("Venti", "Tall", "Short", "Piccolo", "NA"),
  levels = c("Short", "Tall", "Grande", "Venti", "NA")
)
f3
## [1] Venti Tall Short <NA> NA
## Levels: Short Tall Grande Venti NA
```

12.4 Subsetting of factors

For subsetting factors, we can use the same methods that we learned for vectors in section 3.3. For example, we can extract subsets with a numeric vector.

```
f1[1:3]
## [1] Grande Venti Grande
## Levels: Short Tall Grande Venti
```

We can remove elements with a minus sign.

```
f1[-4]
## [1] Grande Venti Grande
## Levels: Short Tall Grande Venti
```

And we can subset with a logical vector.

```
f1[c(TRUE, TRUE, TRUE, FALSE)]
## [1] Grande Venti Grande
## Levels: Short Tall Grande Venti
```

We can create a logical vector from a factor by using == or !=. It is important to put the names of the levels in quotation marks even though the R output does not show the quotes.

```
# We need quotes
f1 == "Grande"
## [1] TRUE FALSE TRUE FALSE

# The output will not have quotes
f1[f1 == "Grande"]
## [1] Grande Grande
## Levels: Short Tall Grande Venti
```

The assignment operator <- works for factors in the same manner as for vectors. For example, we can make a copy of f1.

```
test <- f1
test
## [1] Grande Venti Grande Tall
## Levels: Short Tall Grande Venti
```

We can also replace subsets with <-.

```
test[1:2] <- c("Tall", "Short")
test
## [1] Tall Short Grande Tall
## Levels: Short Tall Grande Venti
```

When we assign an element to a factor that does not match any of the levels, R replaces the value by `NA` and issues a warning.

```
test[1] <- "Piccolo"
## Warning in `[<-.factor`(`*tmp*`, 1, value = "Piccolo"): invalid factor level, NA
## generated
test
## [1] <NA>  Short Grande Tall
## Levels: Short Tall Grande Venti
```

12.5 Retrieving and changing the levels

Given a factor `f1`, the command `levels(f1)` returns the set of levels in `f1`.

```
levels(f1)
## [1] "Short"  "Tall"    "Grande"  "Venti"
```

The function `nlevels()` returns the number of levels.

```
nlevels(f1)
## [1] 4
```

Suppose that, in the interest of shorter output, we wish to abbreviate the levels as `S`, `T`, `G` and `V`. We can change the levels with `fct_recode()` from the `forcats` package. The name ‘forcats’ is a play on words: it is simultaneously an anagram of the word ‘factors’ and the short form of ‘*for categorical variables*’. We encounter the `forcats` package several times in this chapter because it contains utility functions that are more straightforward to use than the functions in R’s base installation.

```
library(forcats)
f1 <-
  fct_recode(f1, S = "Short", T = "Tall", G = "Grande", V = "Venti")
```

The first argument in `fct_recode()` can be either a factor or a character vector. In the latter case, the character vector, say `v` is treated as if the input were `factor(v)`.

We can retrieve the elements in a factor as a character vector by using `as.character()`.

```
as.character(f1)
## [1] "G"  "V"  "G"  "T"
```

If our input contains one and the same category under different names, we can merge the categories by assigning arguments with the same name in `fct_recode()` more than once.

```
f4 <- factor(c("Grande", "Big", "T"))

# Here are the current levels
levels(f4)
## [1] "Big"    "Grande"  "T"

# Combine "Grande" and "Big" into a new category "G"
fct_recode(f4, G = "Grande", G = "Big")
## [1] G G T
## Levels: G T
```

Another option is to use the **forcats** function `fct_collapse()`.

```
fct_collapse(f4, G = c("Grande", "Big"))
## [1] G G T
## Levels: G T
```

Similar to `fct_recode`, `fct_collapse()` accepts either a factor or a character vector as its first argument.

12.6 R's internal representation of factors

Let us generate the factor `f1` again and have a look at its structure. We can display the internal structure of any R object with the `str()` function.

```
f1 <- factor(
  c("Grande", "Venti", "Grande", "Tall"),
  levels = c("Short", "Tall", "Grande", "Venti")
)
str(f1)
## Factor w/ 4 levels "Short","Tall",...: 3 4 3 2
```

The final four numbers (3, 4, 3, 2) in the output reveal that factors are in

fact ‘embellished’ integer vectors. We can confirm this observation with the `typeof()` function, which returns R’s internal storage type of its argument.

```
typeof(f1)
## [1] "integer"
```

Factors add two attributes as embellishment to an integer vector: `levels` and `class`. The output of `attributes(f1)` below may currently look unfamiliar because `attributes()` returns a list, and we only cover lists in chapter 22.

```
attributes(f1)
## $levels
## [1] "Short"   "Tall"    "Grande"   "Venti"
##
## $class
## [1] "factor"
```

Here is a sneak preview of lists: the names of the attributes are on the lines starting with the `$` signs. We can obtain a character vector containing the names as follows.

```
names(attributes(f1))
## [1] "levels" "class"
```

If we think of the levels and the class attributes as embellishments, then the integers (in our case 3, 4, 3, 2) are the essence of `f1`. We can view the integers hiding behind `f1` by using `as.integer()`.

```
as.integer(f1)
## [1] 3 4 3 2
```

We did not insert any integers as input when we created `f1`. How does R make the conversion from `c("Grande", "Venti", "Grande", "Tall")` to 3, 4, 3, 2 in our example?

The levels are encoded as integers in the same order in which they appear in the `levels` argument:

- Short = 1,
- Tall = 2,
- Grande = 3,
- Venti = 4.

When R goes through the names of the first argument `c("Grande", "Venti",`

"Grande", "Tall"), it matches the first element "Grande" with 3, the second element "Venti" with 4 etc.

Suppose we change the name of each level so that it corresponds to the size in ounces:

- 8 = Short
- 12 = Tall
- 16 = Grande
- 20 = Venti

Because argument names in R cannot be numbers, we must surround the argument names in `fct_recode()` with quotation marks.

```
f5 <- fct_recode(
  f1,
  "8" = "Short",
  "12" = "Tall",
  "16" = "Grande",
  "20" = "Venti"
)
```

When we print `f5`, the labels appear as if they were integers.

```
f5
## [1] 16 20 16 12
## Levels: 8 12 16 20
```

However, `as.integer(f5)` does not return the labels (i.e. 16, 20, 16, 12). Instead, the returned integers are still the same values previously returned by `as.integer(f1)`.

```
as.integer(f5)
## [1] 3 4 3 2
```

If we want to retrieve the labels (i.e. 16, 20, 16, 12) as an integer vector, we can either use `as.integer(as.character(f5))`, or we follow the recommended technique in the R documentation (see `?factor`):

```
as.integer(levels(f5))[f5]
## [1] 16 20 16 12
```

12.7 Dropping unused levels

Out of the four possible levels (Short, Tall, Grande, Venti), `f1` only contains three because there is no element equal to Short, as we can see by applying `unique()`.

```
unique(f1)
## [1] Grande Venti  Tall
## Levels: Short Tall Grande Venti
```

Sometimes, it is useful to drop those levels that are not part of the data. If we want to drop unused levels from a factor, we can use `fct_drop()` from the `forcats` package.

```
# Removing level "Short"
f_drop <- fct_drop(f1)
f_drop
## [1] Grande Venti  Grande Tall
## Levels: Tall Grande Venti
```

We now have indeed only three levels left instead of four.

```
nlevels(f_drop)
## [1] 3
```

12.8 Arranging the order of levels according to their frequency

When we summarise categorical data, it is sometimes useful to sort the levels in the order of frequency with which they appear in the data. Let us assume that a coffeehouse asked 100 customers about their favourite item in the shop. On the questionnaire, the respondents could choose from four options: coffee, tea, food and merchandise. Here are the results stored as a factor.

```
favourite <- factor(c(
  rep("coffee", 61),
  rep("tea", 13),
  rep("merchandise", 1),
```

```
rep("food", 25)
))
```

By default, the levels are the unique values in `favourite` in alphabetical order.

```
levels(favourite)
## [1] "coffee"      "food"        "merchandise" "tea"
```

When we use `table()` to summarise the data, the columns in the output are also in alphabetical order.

```
table(favourite)
## favourite
##   coffee      food merchandise      tea
##   61          25            1         13
```

From R's output, it is not immediately obvious how the answers rank in terms of their frequency. It would be more reader-friendly if the columns were sorted from the most frequent answer on the left to the least frequent answer on the right. We can accomplish this task with the **forcats** function `fct_infreq()`.

```
favourite_sorted <- fct_infreq(favourite)
```

Now the levels are sorted from the most frequent to the least frequent answer.

```
levels(favourite_sorted)
## [1] "coffee"      "food"        "tea"        "merchandise"
```

The columns in the output of `table(favourite_sorted)` are now also sorted by frequency.

```
table(favourite_sorted)
## favourite_sorted
##   coffee      food      tea merchandise
##   61          25         13            1
```

12.9 Ordered factors

Categorical data come in two distinct flavours: nominal and ordinal. Nominal data consist of categories that have no inherent order, whereas ordinal data can be meaningfully sorted. Examples of nominal data are:

- gender.
- nationality.
- computer operating system.

Examples of ordinal data are:

- opinion polls with answers on a Likert scale (e.g. ‘Disagree’, ‘Neutral’, ‘Agree’).
- military rank.
- drink sizes ‘Short’, ‘Tall’, ‘Grande’, ‘Venti’.

For ordinal data, it makes sense to compare categories with inequalities. For example, it is reasonable to regard "Tall" > "Short" as a TRUE statement.

When using R’s plain-vanilla factors, any comparison of levels with >, >=, < or <= produces NAs.

```
size <- c("Grande", "Venti", "Grande", "Tall")
f1 <- factor(size, levels = c("Short", "Tall", "Grande", "Venti"))
f1 > "Tall"
## Warning in Ops.factor(f1, "Tall"): '>' not meaningful for factors
## [1] NA NA NA NA
```

However, there is another R data structure that we can use for ordinal data: an *ordered* factor. There are two ways to make a factor ordered.

- (1) We add the argument `ordered = TRUE` inside `factor()`. We can tell that the levels are ordered because there are < signs on the last line of output below.

```
f_ordered <- factor(
  size,
  levels = c("Short", "Tall", "Grande", "Venti"),
  ordered = TRUE
)
```

```
f_ordered
## [1] Grande Venti Grande Tall
## Levels: Short < Tall < Grande < Venti
```

- (2) Alternatively, we can use the function `ordered()` instead of `factor()`.

```
f_ordered <- ordered(
  size,
  levels = c("Short", "Tall", "Grande", "Venti")
)
```

```
f_ordered
## [1] Grande Venti Grande Tall
## Levels: Short < Tall < Grande < Venti
```

Unlike for unordered factors, we can now meaningfully use all inequality operators, for example:

```
f_ordered > "Tall"
## [1] TRUE TRUE TRUE FALSE
```

Ordered factors are members of two classes: "ordered" and "factor".

```
class(f_ordered)
## [1] "ordered" "factor"
is.ordered(f_ordered)
## [1] TRUE
is.factor(f_ordered)
## [1] TRUE
```

Because ordered factors inherit the methods of conventional factors, we can apply the same techniques for changing or dropping levels. However, when applying these methods to an ordered factor, the results are also ordered factors.

12.10 Summary and outlook

When working with categorical data in R, we should represent them as factors. R has many bespoke methods for data wrangling with factors. In this chapter, we only covered some of the possibilities that R offers. The **forcats** package has many additional utility functions. Here is a link to a cheat sheet with the most important **forcats** functions: <https://github.com/rstudio/cheatsheets/raw/master/factors.pdf>. You may want to consult this cheat sheet while you are coding.

12.11 Just checking

First, try to answer the following questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.2.

- (I) Suppose we create the character vector `gender_chr` as follows.

```
gender_chr <- c("M", "F", "m", "f")
```

How can we obtain the following factor `gender_fct`?

```
## [1] male   female male   female  
## Levels: female male
```

(a) `gender_fct <-
 factor(gender_chr, levels = c("female", "male"))`

(b) `gender_fct <- forcats::fct_collapse(
 gender_chr,
 female = c("F", "f"),
 male = c("M", "m")
)`

```
(c) gender_fct <- forcats::fct_recode(
  gender_chr,
  female = c("F", "f"),
  male = c("M", "m")
)
```

```
(d) gender_fct <- factor(
  gender_chr,
  levels = c("male", "female", "male", "female")
)
```

- (II) The built-in factor `state.region` gives the region (Northeast, South, North Central, West) of each state in the USA (in the order given by `state.name`). Create a new factor where the names of the levels are changed to NE, S, NC and W.

- (III) What is the output at the end of the following code chunk?

```
opinion <- factor(c(
  "Disagree",
  "Agree",
  "Disagree",
  "Neutral",
  "Agree",
  "Agree"
))
str(opinion)

(a) ## Factor w/ 3 levels "Disagree","Agree",...: 1 2 1 3 2 2
(b) ## Factor w/ 3 levels "Disagree","Neutral",...: 1 3 1 2 3 3
(c) ## Factor w/ 3 levels "Agree","Disagree",...: 2 1 2 3 1 1
(d) ## Factor w/ 3 levels "Disagree","Neutral",...: 1 2 1 3 2 2
```

- (IV) What is the output of the following code chunk?

```
size <- c("Grande", "Venti", "Grande", "Tall")
f <- ordered(size, levels = c("Short", "Tall", "Grande", "Venti"))
f > "Tall"
```

```
(a) ## Warning in Ops.factor(f, "Tall"): '>' not meaningful for factors
## [1] NA NA NA NA
```

- (b) ## [1] FALSE TRUE FALSE FALSE
- (c) ## [1] TRUE TRUE TRUE FALSE
- (d) ## [1] FALSE FALSE FALSE TRUE



13

Matrices

At this point in our learning curve, we have become well versed in R vectors and factors. We can think of vectors and factors as one-dimensional objects in the sense that we can identify positions of elements in these objects with an index that is a single number. However, sometimes, we would like to work with two-dimensional objects with rows and columns (e.g. spreadsheets used by software such as Microsoft Excel® or Google Sheets®). In data analysis, the most common R data structures for such two-dimensional objects are data frames and tibbles. Before we learn about data frames and tibbles in chapter 14, let us first explore another two-dimensional data structure called ‘matrix’. Matrices used to be more common R data structures in the past than they are at present, but we still encounter them in some contexts. Some functions in the `stringr` package (e.g. `str_match()`, which we use in section 27.3.12) return matrices to identify character strings contained in other character strings. The `sf` package, which contains utility functions for working with geospatial data, also uses matrices to store data (e.g. to encode the two-dimensional coordinates along a polygon). We work with the `sf` package in chapter 31. It is worth learning the main features of matrices so that we can do some basic matrix operations when the need arises.

13.1 Creating matrices

We may have encountered matrices in high-school maths as rectangular tables of numbers. For example,

$$\mathbf{M} = \begin{pmatrix} 5 & 4 & 8 \\ 6 & 1 & 4 \end{pmatrix}$$

is a 2×3 matrix (i.e. \mathbf{M} has 2 rows and 3 columns). Numeric matrices occur in geometry and linear algebra. In R, we can enter a matrix in several ways. One possibility is the function `matrix()`, which typically needs two arguments. First, we enter the matrix elements in column-major order:

- leftmost column from top to bottom,
- second leftmost column from top to bottom

- etc.

Then we specify the number of rows.

```
matrix(c(5, 6, 4, 1, 8, 4), nrow = 2)
##      [,1] [,2] [,3]
## [1,]    5    4    8
## [2,]    6    1    4
```

Alternatively, we can specify the number of columns.

```
matrix(c(5, 6, 4, 1, 8, 4), ncol = 3)
##      [,1] [,2] [,3]
## [1,]    5    4    8
## [2,]    6    1    4
```

If we prefer to enter the values in row-major order (i.e. going first from left to right and then from top to bottom) instead of column-major order, we can pass the argument `byrow = TRUE`.

```
matrix(c(5, 4, 8, 6, 1, 4), ncol = 3, byrow = TRUE)
##      [,1] [,2] [,3]
## [1,]    5    4    8
## [2,]    6    1    4
```

An alternative to `matrix()` is `cbind()`. While `matrix()` requires us to enter all elements as one long vector, `cbind()` needs the columns as separate vectors. It then ‘binds’ the columns together.

```
cbind(c(5, 6), c(4, 1), c(8, 4))
##      [,1] [,2] [,3]
## [1,]    5    4    8
## [2,]    6    1    4
```

The function `rbind()` is similar to `cbind()`, but `rbind()` takes the rows as input instead of the columns.

```
m <- rbind(c(5, 4, 8), c(6, 1, 4))
```

Matrices are members of two classes: `matrix` and `array`.

```
class(m)
## [1] "matrix" "array"
```

Arrays are n -dimensional generalisations of matrices with $n \geq 2$. Three-dimensional arrays are useful for storing colour images, where the numbers on different slices through the array represent the intensities of the primary colours (i.e. red, green and blue) on a two-dimensional picture. Arrays with even more dimensions can sometimes be useful too (e.g. for storing multi-way tables or time series of satellite images). However, in this book, we only need arrays that are two-dimensional (i.e. they are matrices).

13.2 Getting and setting matrix dimensions

In the previous section, we learned how to create matrices with `matrix()`, `cbind()` and `rbind()`. Another option for creating a matrix is to store the matrix elements as a vector in column-major order. Afterwards, we set the number of rows and columns with the function `dim()`, which stands for ‘dimensions’.

```
m <- c(5, 6, 4, 1, 8, 4)
dim(m) <- c(2, 3)
m
##      [,1] [,2] [,3]
## [1,]     5     4     8
## [2,]     6     1     4
```

We can subsequently change the dimensions again by assigning new row and column numbers.

```
dim(m) <- c(3, 2)
m
##      [,1] [,2]
## [1,]     5     1
## [2,]     6     8
## [3,]     4     4
```

So far, we have used `dim()` to set new matrix dimensions, but it can also be used to retrieve the current numbers of rows and columns.

```
dim(m)
## [1] 3 2
```

Alternatively, we can get the number of rows and columns with `nrow()` and `ncol()`.

```
nrow(m)
## [1] 3
ncol(m)
## [1] 2
```

The function `length()` returns the number of all elements in a matrix (i.e. the number of rows multiplied by the number of columns).

```
length(m)
## [1] 6
```

Matrix dimensions are stored as an attribute.

```
attributes(m)
## $dim
## [1] 3 2
```

If we remove the `dim` attribute from a matrix, the object becomes a vector with the same elements in column-major order.

```
# Right now, m belongs to the "matrix" class
class(m)
## [1] "matrix" "array"

# Let us remove the "dim" attribute
attr(m, "dim") <- NULL

# Alternative: dim(m) <- NULL

# At this point, m is a humble numeric vector
class(m)
## [1] "numeric"

# Numbers are in column-major order
m
## [1] 5 6 4 1 8 4
```

Conversely, adding a `dim` attribute to a vector creates a matrix.

```
attr(m, "dim") <- c(2, 3)

# Or, shorter, "dim(m) <- c(2, 3)"
```

```
class(m)
## [1] "matrix" "array"
```

With `is.matrix()`, we find out whether an object is a matrix.

```
is.matrix(m)
## [1] TRUE
```

Interestingly, although a matrix is essentially a vector with a `dim` attribute, `is.vector(m)` returns `FALSE`.

```
is.vector(m)
## [1] FALSE
```

When we want to turn `m` into a vector, we can remove the `dim` attribute either with `attr(m, "dim") <- NULL` or `dim(m) <- NULL`, as we saw above. Yet another option is `as.vector(m)`.

```
as.vector(m)
## [1] 5 6 4 1 8 4
```

13.3 Subsets of matrices

There are many methods to form subsets of matrices in R. The most common subsetting method is to specify the row and column numbers inside a pair of square brackets. The row and column numbers must be separated by a comma.

```
# Full matrix m
m
##      [,1] [,2] [,3]
## [1,]     5     4     8
## [2,]     6     1     4

# Element in first row, third column
m[1, 3]
## [1] 8
```

The row and column numbers can be vectors of length > 1 .

```
m[1:2, 2:3]
##      [,1] [,2]
## [1,]    4    8
## [2,]    1    4
```

We can also drop rows or columns with a minus sign.

```
m[1:2, -1]
##      [,1] [,2]
## [1,]    4    8
## [2,]    1    4
```

If we want to keep all rows or columns, we can leave the corresponding entry in the brackets empty.

```
m[, -1]
##      [,1] [,2]
## [1,]    4    8
## [2,]    1    4
m[1:2, ]
##      [,1] [,2] [,3]
## [1,]    5    4    8
## [2,]    6    1    4
```

It is also possible to use logical vectors for matrix subsetting.

```
m[, c(TRUE, TRUE, FALSE)]
##      [,1] [,2]
## [1,]    5    4
## [2,]    6    1
```

Usually, the subset of a matrix is again a matrix.

```
m1 <- m[, c(TRUE, TRUE, FALSE)]
is.matrix(m1)
## [1] TRUE
```

However, if the result of matrix subsetting is a single row or a single column, the default return value is no longer a matrix. Instead, R automatically simplifies the object to a vector.

```
# Keeping only one row
v1 <- m[1, c(1, 3)]

# The result is no longer a matrix but a numeric vector
is.matrix(v1)
## [1] FALSE
class(v1)
## [1] "numeric"

# Keeping only one column
v2 <- m[, 3]

# The result is coerced to a numeric vector
class(v2)
## [1] "numeric"
```

Note that R prints `v2`, just like any vector, as a row (i.e. from left to right) although it was originally a column in a matrix (i.e. going from top to bottom).

```
v2
## [1] 8 4
```

R's default is sometimes convenient. However, simplifying single-column or single-row matrices to vectors is often the source of bugs. If we prefer to keep the subset as a matrix, we can use the argument `drop = FALSE` inside the square brackets.

```
m2 <- m[, 3, drop = FALSE]

# "drop = FALSE" prevents R from simplifying m2 to a vector
class(m2)
## [1] "matrix" "array"
```

Because `m2` is a matrix, R prints its elements, unlike those of `v2` above, as a column.

```
m2
##      [,1]
## [1,]    8
## [2,]    4
```

Another subsetting strategy is to use a single vector inside square brackets (i.e. there is no comma separating two vectors in the brackets). In that case,

R uses vector-style indexing: `m[5]` is the fifth element in the vector `m` that we would have obtained if we had stripped `m` of its `dim` attribute (i.e. `m[5]` is the first element in the third column).

```
m[5]
## [1] 8
```

It is important to understand the distinction between matrix-style and vector-style subsetting: `m[2,]` is different from `m[2]`. The comma makes a big difference.

```
m[2, ]
## [1] 6 1 4
m[2]
## [1] 6
```

13.4 Replacing elements in a matrix

If we want to replace some of the values in an existing matrix, we first use one of the subsetting methods described above. Then we assign the new values as a vector.

```
# Here is what is currently in m
m
##      [,1] [,2] [,3]
## [1,]     5     4     8
## [2,]     6     1     4
m[, 2] <- c(-1, 7)
m
##      [,1] [,2] [,3]
## [1,]     5    -1     8
## [2,]     6     7     4
```

If the assigned vector is shorter than the number of elements it is assigned to replace, the vector will be recycled.

```
m[2, ] <- 3
m
##      [,1] [,2] [,3]
```

```
## [1,]    5   -1    8
## [2,]    3    3    3
```

It is also possible to replace a submatrix by a matrix, say `n`, rather than a vector.

```
n <- matrix(rep(-100, 4), nrow = 2)
m[, 2:3] <- n
m
##      [,1] [,2] [,3]
## [1,]    5 -100 -100
## [2,]    3 -100 -100
```

When replacing matrix elements, R does not allow fractional vector recycling.

```
m[1, ] <- c(-2, -3)
```

```
## Error in m[1, ] <- c(-2, -3) :
##   number of items to replace is not a multiple of replacement length
```

13.5 Sums of rows and columns

The functions `sum()` and `mean()` calculate the sum and mean of *all* elements in a matrix.

```
m <- rbind(
  c(5, 4, 8),
  c(6, 1, 4)
)
sum(m)
## [1] 28
mean(m)
## [1] 4.666667
```

However, more frequently, we would like to know the sums or means of individual rows or columns. We can obtain these values with the functions `rowSums()`, `colSums()`, `rowMeans()` and `colMeans()`. Their results are vectors whose length matches the number of rows or columns, respectively.

```
rowSums(m)
## [1] 17 11
colMeans(m)
## [1] 5.5 2.5 6.0
```

If there are `NA`s in the matrix, the sum and mean of the corresponding row and column is `NA` unless we set `na.rm = TRUE`.

```
m[2, 2] <- NA
rowSums(m)
## [1] 17 NA
rowSums(m, na.rm = TRUE)
## [1] 17 10
```

13.6 Matrices do not need to be numeric

Throughout this chapter, we have worked exclusively with numeric matrices. However, just like vectors, matrices can also contain `character`, `integer` and `logical` elements. Here is an example of a character matrix.

```
m <- matrix(LETTERS[1:6], nrow = 2)
m
##      [,1] [,2] [,3]
## [1,] "A"  "C"  "E"
## [2,] "B"  "D"  "F"
typeof(m)
## [1] "character"
```

Also just like vectors, we cannot mix the data types in a matrix: if one of the elements is of type `character`, then all other elements are also of type `character`. If necessary, R coerces the data type according to the rules we learned in section 10.2.

```
v1 <- c(5, 3, 4)
v2 <- c("A", "0", "C")
m <- rbind(v1, v2)
m
##      [,1] [,2] [,3]
## v1 "5"  "3"  "4"
```

```
## v2 "A"  "O"  "C"
typeof(m)
## [1] "character"
```

In chapter 14, we learn how we can combine numerical and character elements into a data frame without coercing numbers into character strings.

13.7 Summary and outlook

Although matrices are, at present, less common than they were in the early days of R, they are still sometimes used when all data points are of the same class and when they can be represented as rows and columns of a rectangular structure. We encounter matrices in the context of string handling in section 27.3.12 and in the context of geographic data in chapter 31. Next, we turn our attention to more common data structures in modern R: data frames and tibbles. Many of the methods we have just learned for matrices turn out to be directly applicable to data frames and tibbles with a few subtle differences.

13.8 Just checking

What is the output when we run the following code chunk?

```
m <- matrix(letters[1:6], nrow = 3)
attr(m, "dim") <- c(1, 6)
m
```

- (a) ## [,1] [,2] [,3] [,4] [,5] [,6]
 ## [1,] "a" "b" "c" "d" "e" "f"
- (b) ## [1] "a" "b" "c" "d" "e" "f"
- (c) ## [1] 1
 ## [1] 2
 ## [1] 3
 ## [1] 4
 ## [1] 5
 ## [1] 6
- (d) ## [,1]

```
## [1,] "a"  
## [2,] "b"  
## [3,] "c"  
## [4,] "d"  
## [5,] "e"  
## [6,] "f"
```

First, try to answer the question without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix [A.12](#).

14

Data frames and tibbles

Many data sets of interest have a spreadsheet-like format. That is, we can arrange them into a rectangular shape with rows and columns. For example, consider the data in table 14.1. It summarises the team scores in group A of the 2019 women’s world cup.

Each column has four elements; thus, we can fill the data into a rectangular spreadsheet. At first glance, the rectangular shape of table 14.1 is reminiscent of a matrix. However, at least in terms of R, a matrix cannot do the job for us because not all columns are of the same class. The ‘Team’ column consists of character strings, ‘Points’ are numeric, and ‘Qualified’ is logical data (i.e. TRUE or FALSE). In a matrix, all data would be coerced into character strings (see section 13.6); thus, we would lose, for example, the ability to do arithmetic with the numeric data. Instead, we would like a data structure with the following properties:

- rectangular in shape (i.e. all columns are equally long).
- different columns can be of different type (e.g. one column may contain numeric data, whereas another column may consist of character strings).
- every column can be expressed as a vector, factor or list. (So far, we have not yet seen lists; they are the topic of chapter 22.)

R grants our wish in the form of *data frames*. A modern subclass of data frames are *tibbles*, which have some nice additional properties (e.g. for printing formatted output to the console). Tibbles are defined in the **tibble** package; thus, we can work with tibbles after executing `library(tibble)`. Because **tibble** is part of the tidyverse, we can alternatively execute `library(tidyverse)`, which loads **tibble** together with many other packages that we use through-

TABLE 14.1: Group A in the 2019 women’s football world cup.

| Team | Continent | Points | Qualified |
|-------------|-----------|--------|-----------|
| France | Europe | 9 | TRUE |
| Norway | Europe | 6 | TRUE |
| Nigeria | Africa | 3 | TRUE |
| South Korea | Asia | 0 | FALSE |

out this book (e.g. **dplyr**, **forcats** and **ggplot2**). To keep code readable, I will use `library(tidyverse)` from here on instead of loading every package individually.

We can coerce a traditional data frame (e.g. the built-in data frame `iris`) into a tibble with `as_tibble()`. While traditional data frames only belong to the class "data.frame", tibbles are also in the classes "tbl_df" and "tbl".

```
library(tidyverse)
class(iris)
## [1] "data.frame"
iris_tbl <- as_tibble(iris)
class(iris_tbl)
## [1] "tbl_df"     "tbl"        "data.frame"
```

In practice, most methods we apply to tibbles also work for general data frames; thus, we rarely need to use `as_tibble()`.

14.1 Creating tibbles

Most of the time, we create tibbles by importing spreadsheet data from a file. We cover that topic in chapter 15. Alternatively, we can generate tibbles directly in an R script. In that case, we have essentially three options.

- We first define the columns as individual vectors or factors. Afterwards we combine them with `tibble()`.

```
team <- c("France", "Norway", "Nigeria", "South Korea")
continent <- factor(c("Europe", "Europe", "Africa", "Asia"))
points <- c(9, 6, 3, 0)
qualified <- c(TRUE, TRUE, TRUE, FALSE)
group_a <- tibble(team, continent, points, qualified)
group_a
## # A tibble: 4 x 4
##   team      continent points qualified
##   <chr>     <fct>    <dbl>  <lgl>
## 1 France    Europe      9  TRUE
## 2 Norway    Europe      6  TRUE
## 3 Nigeria   Africa      3  TRUE
## 4 South Korea Asia       0 FALSE
```

- We initialize all columns directly inside `tibble()`.

```
group_a <-
tibble(
  team = c("France", "Norway", "Nigeria", "South Korea"),
  continent = factor(c("Europe", "Europe", "Africa", "Asia")),
  points = c(9, 6, 3, 0),
  qualified = c(TRUE, TRUE, TRUE, FALSE)
)
```

- We insert the data in row-wise fashion with `tibble()`.

```
group_a_from_tibble <-
tibble(
  ~team, ~continent, ~points, ~qualified,
  "France", "Europe", 9, TRUE,
  "Norway", "Europe", 6, TRUE,
  "Nigeria", "Africa", 3, TRUE,
  "South Korea", "Asia", 0, FALSE
)
```

```
group_a_from_tibble
## # A tibble: 4 x 4
##   team      continent points qualified
##   <chr>     <chr>     <dbl> <lgl>
## 1 France    Europe        9  TRUE
## 2 Norway    Europe        6  TRUE
## 3 Nigeria   Africa        3  TRUE
## 4 South Korea Asia        0  FALSE
```

If the input is nicely aligned, the rows and columns in the code look like those in the generated tibble as long as we align the columns. The trouble with `tibble()`¹ is that it does not allow us to turn the `continent` column directly into a factor. You can see it in the output by the type specification `<chr>` instead of `<fct>` below the column name. We would have to turn columns into factors with a separate function call. In section 14.4, we see one out of many options for going about this task.

R also has a built-in function `data.frame()`, which creates data frames that are not tibbles. It can be used in a similar way as `tibble()`, but there is no harm done in giving a data frame the additional properties of a tibble. Therefore, I prefer to use `tibble()` instead of `data.frame()`.

¹Trekkies may recognise the pun (https://en.wikipedia.org/wiki/The_Trouble_with_Tibbles).

14.2 Getting information about the structure of a data frame

We learned in section 12.6 that we can display the structure of any R object with `str()`. This technique also works for data frames.

Alternatively, we can obtain information about the structure of a data frame with `glimpse()`.

```
glimpse(group_a)
## #> Rows: 4
## #> Columns: 4
## #> $ team      <chr> "France", "Norway", "Nigeria", "South Korea"
## #> $ continent <fct> Europe, Europe, Africa, Asia
## #> $ points    <dbl> 9, 6, 3, 0
## #> $ qualified <lgl> TRUE, TRUE, TRUE, FALSE
```

The printed information of `str()` and `glimpse()` only differ in detail. For example, `str()` uses `num` as abbreviation for ‘numeric’, whereas `glimpse()` calls this data type `dbl` as abbreviation for ‘double-precision floating-point number’. There is also a difference between the return values of `str()` and `glimpse()`. While `str()` returns `NULL`, `glimpse()` returns the argument (here `group_a`), which can be convenient in pipelines. Therefore, `glimpse()` is the slightly better alternative in my opinion.

Similar to matrices, we can find the number of rows in a data frame with `nrow()` and the number of columns with `ncol()`. The function `dim()` returns both numbers as a vector of length 2.

```
nrow(group_a)  
## [1] 4  
ncol(group_a)  
## [1] 4  
dim(group_a)  
## [1] 4 4
```

The output of `length()` is different for matrices and data frames. On the one hand, the length of a matrix is the product of `nrow()` and `ncol()`, as we learned in section 13.2. On the other hand, the length of a data frame is simply equal to `ncol()`.

```
length(group_a)
## [1] 4
```

When we learn about lists in chapter 22, this behaviour makes sense. It turns out that data frames are lists whose elements are the column vectors. Right now, it is sufficient to know that `length()` behaves differently for matrices and data frames.

Besides the number of columns, we often would like to know the names of the columns during exploratory data analysis. We can obtain this output with `names()`.

```
names(group_a)
## [1] "team"      "continent"   "points"     "qualified"
```

14.3 Matrix-style subsetting

Data frames permit many of the same subsetting operations that we learned for matrices in section 13.3.

```
# First and second row, third column
group_a[1:2, 3]
## # A tibble: 2 x 1
##   points
##   <dbl>
## 1     9
## 2     6

# All elements in the third column
group_a[, 3]
## # A tibble: 4 x 1
##   points
##   <dbl>
## 1     9
## 2     6
```

```
## 3      3
## 4      0
```

However, unlike for matrices, if we use a single number `n` without comma inside the square brackets, we obtain the `n`-th column, not the `n`-th element in column-major order.

```
group_a[3]
## # A tibble: 4 x 1
##   points
##   <dbl>
## 1     9
## 2     6
## 3     3
## 4     0
```

Let us accept this behaviour as a quirk of data frames. It will make sense after we learned more about lists in chapter 22.

We learned in section 13.1 that we can append columns to matrices with `cbind()`. The same method also works for data frames.

```
cbind(group_a, goals_scored = c(7, 6, 2, 1))
##           team continent points qualified goals_scored
## 1     France    Europe     9     TRUE          7
## 2   Norway    Europe     6     TRUE          6
## 3  Nigeria    Africa     3     TRUE          2
## 4 South Korea    Asia     0    FALSE          1
```

The second argument passed to `cbind()` does not need to be a vector; it can also be a data frame.

```
goals <- tibble(
  goals_scored = c(7, 6, 2, 1),
  goals_received = c(1, 3, 4, 8)
)
cbind(group_a, goals)
##           team continent points qualified goals_scored goals_received
## 1     France    Europe     9     TRUE          7            1
## 2   Norway    Europe     6     TRUE          6            3
## 3  Nigeria    Africa     3     TRUE          2            4
## 4 South Korea    Asia     0    FALSE          1            8
```

Similarly, `rbind()` combines the rows of two data frames, provided that the

column names are exactly the same in both data frames. Otherwise, we get an error.

```
group_b <-
  tibble(
    team = c("Germany", "Spain", "China", "South Africa"),
    continent = factor(c("Europe", "Europe", "Asia", "Africa")),
    points = c(9, 4, 4, 0),
    qualified = c(TRUE, TRUE, TRUE, FALSE)
  )
rbind(group_a, group_b)
## # A tibble: 8 x 4
##   team      continent points qualified
##   <chr>     <fct>    <dbl> <lgl>
## 1 France    Europe      9 TRUE
## 2 Norway    Europe      6 TRUE
## 3 Nigeria   Africa      3 TRUE
## 4 South Korea Asia       0 FALSE
## 5 Germany   Europe      9 TRUE
## 6 Spain     Europe      4 TRUE
## 7 China     Asia       4 TRUE
## 8 South Africa Africa     0 FALSE
```

Often, the data in two data frames are not so neatly aligned that we can directly bind rows or columns together. We learn in chapter 21 how we join such data into one data frame.

14.4 Extracting a single column from a data frame

Often we want to extract a single column from a data frame (e.g. to calculate summary statistics for the data in this column). We may be tempted to use matrix-style subsetting (e.g. `group_a[, 3]`) or the alternative subsetting with a single number that we saw in section 14.3 (e.g. `group_a[3]`).

```
group_a[3]
## # A tibble: 4 x 1
##   points
##   <dbl>
## 1      9
## 2      6
```

```
## 3      3
## 4      0
```

Although the output contains the numbers we are interested in, `mean()` applied to the output of `group_a[3]` does not return the mean of these numbers.

```
mean(group_a[3])
## Warning in mean.default(group_a[3]): argument is not numeric or logical:
## returning NA
## [1] NA
```

The problem is that `group_a[3]` is not a numeric vector. Instead, it is a data frame that contains a numeric vector.

```
class(group_a[3])
## [1] "tbl_df"     "tbl"        "data.frame"
```

A useful metaphor is to think of a data frame as a container and the column vectors as the content. The result of `group_a[3]` is a container. Admittedly, `group_a[3]` is a smaller container than `group_a`, but it is still a container. We are interested in the content (i.e. the vector that contains the points), not the container. If we think of the container as a metal tin and the content as soup, we need a tin opener so that we can scoop the soup. It turns out that we have a choice between two different tin openers: the double-square-bracket operator `[[...]]` and the `$`-operator. The double-square-bracket operator accepts either the column number or the column name as input.

```
group_a[[3]]
## [1] 9 6 3 0
group_a[["points"]]
## [1] 9 6 3 0
```

The next code chunk confirms that the return value of the double-square-bracket operator is a vector, not a data frame or a tibble.

```
class(group_a[[3]])
## [1] "numeric"
is_vector(group_a[[3]])
## [1] TRUE
is.data.frame(group_a[[3]])
## [1] FALSE
is_tibble(group_a[[3]])
## [1] FALSE
```

The \$-operator is a convenient shortcut for the double-square-bracket operator. Instead of `group_a[["points"]]`, we can equivalently write `group_a$points`.

```
group_a$points
## [1] 9 6 3 0
```

Subsetting with the \$-operator requires specifying the column by name, not by number (i.e. `group_a$3` produces an error). In practice, this behaviour is not a problem because it is usually more intuitive to work with column names instead of column numbers. For this reason, the \$-operator is ubiquitous in R code. For example, here is a way to convert the `continent` column in `group_a_from_tibble()` from a character vector into a factor.²

```
group_a_from_tibble$continent <-
  as.factor(group_a_from_tibble$continent)
```

Here is output that confirms that the `continent` column is now indeed a factor.

```
glimpse(group_a_from_tibble)
## Rows: 4
## Columns: 4
## $ team      <chr> "France", "Norway", "Nigeria", "South Korea"
## $ continent <fct> Europe, Europe, Africa, Asia
## $ points    <dbl> 9, 6, 3, 0
## $ qualified <lgl> TRUE, TRUE, TRUE, FALSE
is.factor(group_a_from_tibble$continent)
## [1] TRUE
```

14.5 Summary and outlook

Data frames are the most common data structure when performing data analysis with R. A data frame is a rectangular spreadsheet-like data structure that mimics the way most real-world data look like. Tibbles are a modern subclass of data frames. The methods we presented here work on both tibbles and traditional data frames. Because tibbles are a little bit more user-friendly, we generally work with tibbles in this book. Next, we learn how to import spreadsheet data from a file into R so that the result is a tibble.

²We learn more elegant ways of ‘mutating’ a data frame column in chapter 19.

14.6 Just checking

First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.13.

- (I) Suppose we have created the following tibble named `sales`.

```
sales
## # A tibble: 3 x 3
##   customer gender location
##   <chr>     <chr>   <chr>
## 1 Antonette Female Sembawang
## 2 Chloe      Female Bedok
## 3 Leonard    Male   Kallang
```

What is the output when we run the following command?

```
sales[c(1, 3)]
```

- (a)

```
## # A tibble: 2 x 3
##   customer gender location
##   <chr>     <chr>   <chr>
## 1 Antonette Female Sembawang
## 2 Leonard    Male   Kallang
```
- (b)

```
## # A tibble: 1 x 1
##   location
##   <chr>
## 1 Sembawang
```
- (c)

```
## # A tibble: 3 x 2
##   customer location
##   <chr>     <chr>
## 1 Antonette Sembawang
## 2 Chloe      Bedok
## 3 Leonard    Kallang
```
- (d)

```
[1] "Antonette" "Leonard"
```

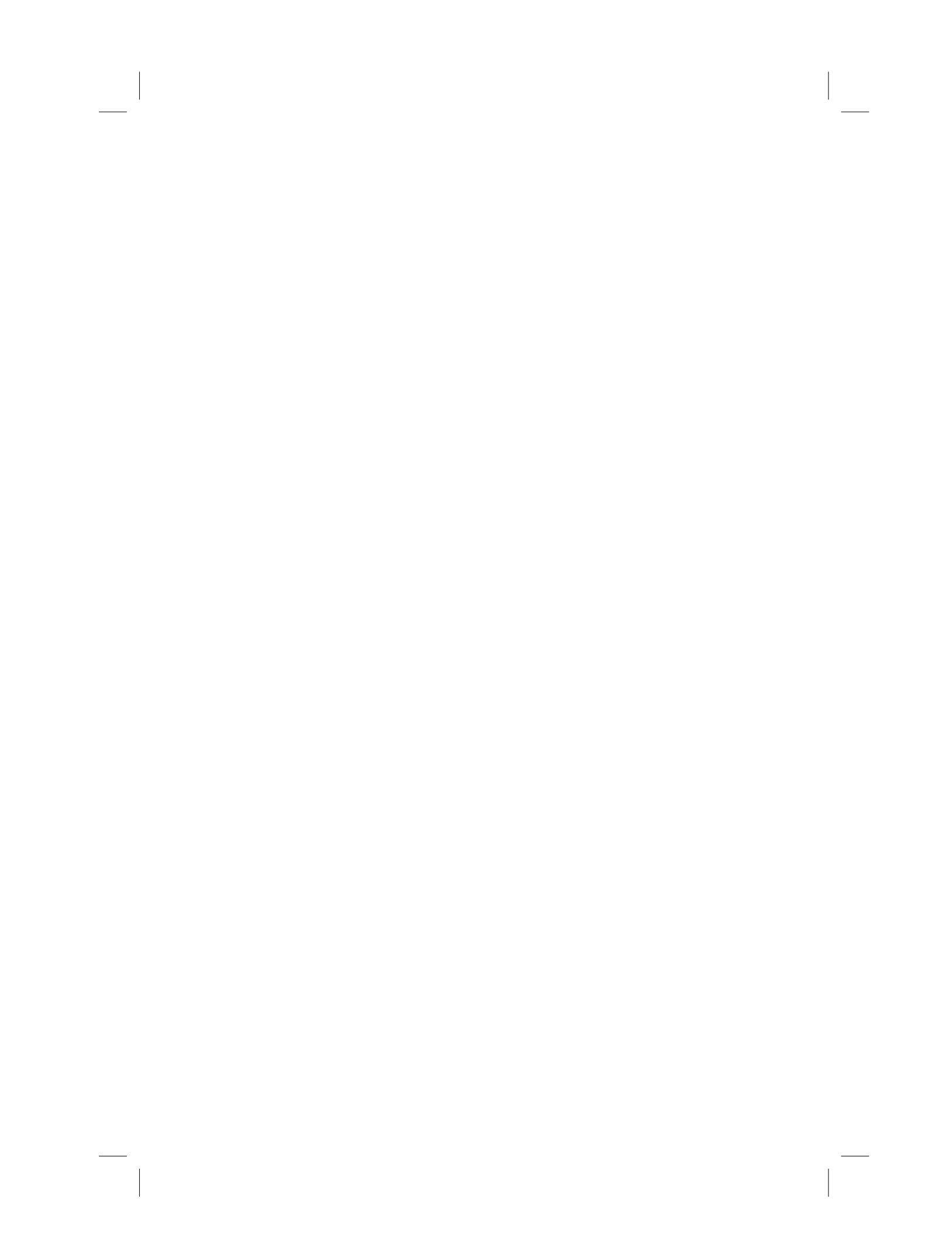
- (II) Consider again the tibble called `sales` from question (I). How can we turn `gender` into a factor?

(a) `sales[[gender]] <- as.factor(sales[[gender]])`

(b) `sales["gender"] <- as.factor(sales["gender"])`

(c) `sales[gender] <- as.factor(sales[gender])`

(d) `sales[["gender"]] <- as.factor(sales[["gender"]])`



15

Importing spreadsheet data as a tibble

For all our previous examples, we have either manually typed all data values or relied on data sets in base R (e.g. `iris` at the beginning of chapter 14) or in third-party packages (e.g. `dplyr::starwars` in section 8.2). However, in practice many interesting data are stored in spreadsheet files that are not directly part of R. In this chapter, we learn how to import spreadsheet data from two common file formats (CSV and Microsoft Excel's® XLSX).

15.1 Excel versus CSV files

There are some R packages that can help us import data directly from the native file formats of common spreadsheet programs. At present, by far the most common file type is XLSX for Microsoft Excel®. In section 15.3, we take a look at the `readxl` package, which can import data from XLSX files.

Until recently, importing directly from XLSX was not considered to be the best R practice. Here is a quote from the official R manual at <https://cran.r-project.org/doc/manuals/r-release/R-data.html#Reading-Excel-spreadsheets>:

The most common R data import/export question seems to be ‘how do I read an Excel spreadsheet’... The first piece of advice is to avoid doing so if possible!

Instead of directly importing XLSX files, the recommended strategy was to use a much simpler file format: *comma-separated values*, also known as CSV. Before we learn what the features of a CSV file are, here are the reasons why it is better to save data in CSV rather than XLSX files.

- CSV files have a simple format that can be written and read even by basic text editors.

- The CSV format has existed for a long time and is unlikely to go out of fashion soon.
- Excel is proprietary software. If Microsoft decides to change the file format, R packages may have to depend on third-party software or libraries that have technical or licensing limitations.

Working with CSV instead of XLSX is for most applications not a big handicap: Excel and all other common spreadsheet programs can export data as a CSV file. Figure 15.1 shows the traditionally recommended workflow. Nowadays, the `readxl` package offers a sensible alternative, as we see in section 15.3. Still, it is a good idea to become familiar with CSV files at this stage because, as a file type, CSV has many advantages compared to XLSX.

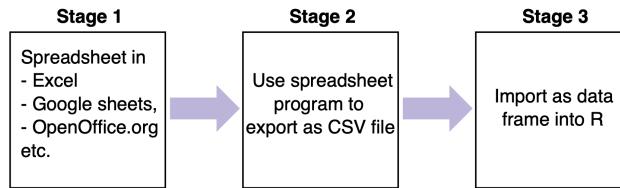


FIGURE 15.1: Traditionally recommended workflow when working with spreadsheet files.

Let us have a look at a concrete example. If you do not have Excel on your computer, you can still follow the example by using freely available software (e.g. OpenOffice or Google Sheets[®]), for which the procedure is similar.

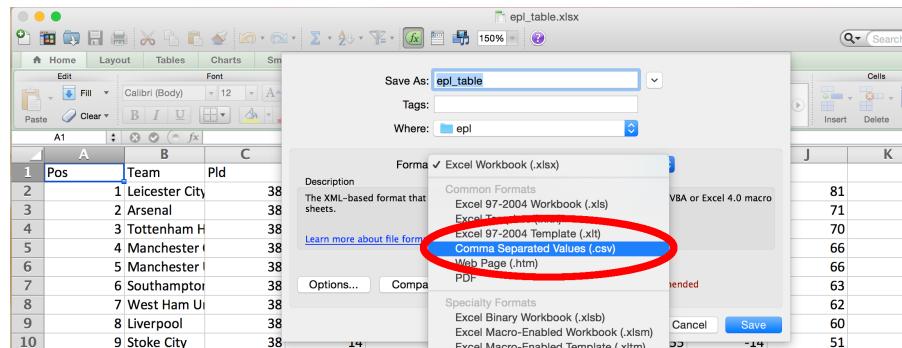


FIGURE 15.2: We can export data from Excel's native XLSX format to CSV with only a few clicks in Excel's user interface.

- Download the spreadsheet https://michaelgastner.com/DAVisR_data/epl_table.xlsx. It contains the overall results of football teams in the English Premier League in 2015-2016. Move the XLSX file to your RStudio project folder. If you need a reminder how to work with RStudio projects, please read section 2.5.

- Open Excel and navigate to the downloaded file with the menu File → Open. Double-click to load `epl_table.xlsx`.
- Use the menu File → Save As... and select ‘Comma Separated Values (.csv)’ from the drop down list (figure 15.2).
- Save the CSV file in your RStudio project directory. You can ignore Excel’s warning that ‘features in your workbook might be lost’.

Let us briefly inspect the content of `epl_table.csv`. Open the file with a simple text editor (e.g.TextEdit on Mac or Notepad on Windows). The appearance of this CSV file in the text editor reveals typical features of the CSV file format (figure 15.3).

| Pos | Team | Pld | W | D | L | GF | GA | Gd | Pts |
|-----|-------------------|-----|----|----|----|----|----|----|-----|
| 1 | Leicester City | 38 | 23 | 12 | 3 | 68 | 36 | 32 | 81 |
| 2 | Arsenal | 38 | 20 | 11 | 7 | 65 | 36 | 29 | 71 |
| 3 | Tottenham Hotspur | 38 | 19 | 13 | 6 | 69 | 35 | 34 | 70 |
| 4 | Manchester City | 38 | 19 | 9 | 10 | 71 | 41 | 30 | 66 |
| 5 | Manchester United | 38 | 19 | 9 | 10 | 49 | 35 | 14 | 66 |
| 6 | Southampton | 38 | 18 | 9 | 11 | 59 | 41 | 18 | 63 |
| 7 | West Ham United | 38 | 16 | 11 | 11 | 65 | 51 | 14 | 62 |

FIGURE 15.3: A CSV file is a plain-text document that we can view with any basic text editor.

- Every row in the spreadsheet corresponds to one line of text in the CSV file.
- Entries in neighbouring columns are separated by a comma.
- CSV files are much more basic than XLSX files. There is only one sheet of data, and the cells cannot contain formulas that automatically update the value of one cell in the spreadsheet if we change another cell. For some applications, these limitations may be inconvenient. However, the simple CSV format is almost always sufficient.
- For small data sets, we do not need a spreadsheet program. We can, in principle, write CSV files directly with a simple text editor.

15.2 Importing CSV into R

We can import CSV files into R

- either with the RStudio Graphical User Interface (GUI)
- or by typing an R command at the console or in a script.

We first explore how to use the GUI. When we import the file for the first time, it is often quicker to resort to the GUI. The GUI automatically generates an R command that we can copy and paste into a script for future use.

In the menu of RStudio’s environment tab in the top right pane, there is a drop down menu ‘Import Dataset’ (figure 15.4).

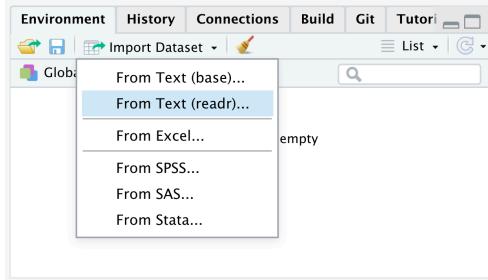


FIGURE 15.4: Because CSV is a plain-text format, we choose ‘From Text’ from RStudio’s GUI. There are ‘base’ and ‘readr’ options, referring to different R packages. I recommend choosing ‘readr’ because of its computational speed.

Let us choose ‘From Text (readr)’ from the menu. Compared to ‘From Text (base)’, importing with ‘From Text (readr)’ is noticeably faster when the imported data file is large. In the next dialogue window, click on the ‘Browse’ button, find `epl_table.csv` in your RStudio project directory and double-click on the file name. RStudio now shows a preview of the data (figure 15.5).

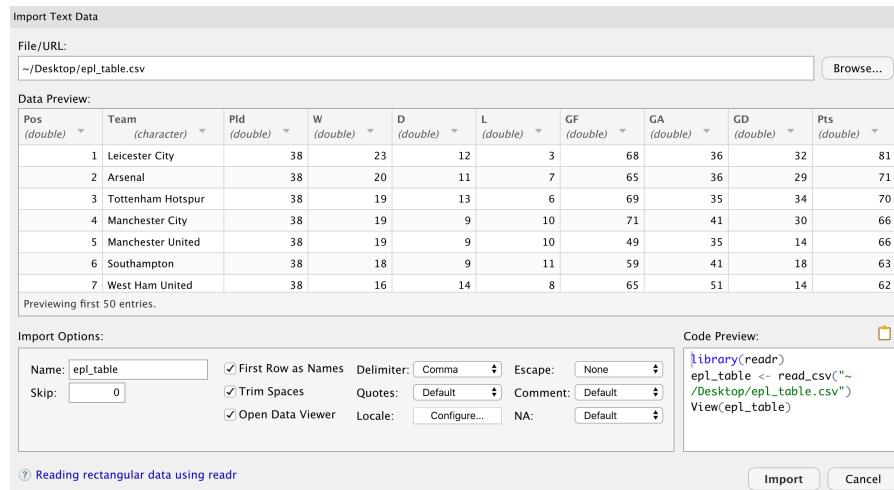


FIGURE 15.5: RStudio shows a preview of the data in the ‘Import Text Data’ GUI.

Sometimes, we need to tweak some of the options in the bottom of the ‘Import Text Data’ GUI. If the preview appears to be wrong, I recommend playing with the options until there are no visible issues any longer. In the present case, everything looks fine; thus, we can click ‘Import’. This action causes several changes in the RStudio window.

- By looking at the environment tab in the top right pane (figure 15.6), we can tell that there is a data frame `epl_table` in the global environment.
- The data appear in a spreadsheet in RStudio's top left pane (figure 15.7). If we cannot see the spreadsheet, we can click on the variable name `epl_table` in the environment tab. In fact, by clicking on the variable name in the environment tab, we can open a spreadsheet view of *any* data frame or matrix, not only those imported from files.
- RStudio automatically ran some commands in the console.

```
> library(readr)
> epl_table <- read_csv("your_directory/epl_table.csv")
> View(epl_table)
```

In the actual console commands, `your_directory` is replaced by the directory where you saved `epl_table.csv`.



FIGURE 15.6: From the environment tab, we can tell that `epl_table` is a data frame with 20 rows and 10 columns.

| | Pos | Team | Pld | W | D | L | GF | GA | GD | Pts |
|---|-----|-------------------|-----|----|----|----|----|----|----|-----|
| 1 | 1 | Leicester City | 38 | 23 | 12 | 3 | 68 | 36 | 32 | 81 |
| 2 | 2 | Arsenal | 38 | 20 | 11 | 7 | 65 | 36 | 29 | 71 |
| 3 | 3 | Tottenham Hotspur | 38 | 19 | 13 | 6 | 69 | 35 | 34 | 70 |
| 4 | 4 | Manchester City | 38 | 19 | 9 | 10 | 71 | 41 | 30 | 66 |

FIGURE 15.7: We can view the content of the data frame `epl_table` in RStudio's top left pane.

Let us try to understand the console output.

- The first line `library(readr)` loads the `readr` package that contains many utility functions for importing spreadsheet data. It is part of the tidyverse.
- The second line runs `read_csv()`, which is one of the functions in `readr`. The output is saved to a tibble called `epl_table`.
- The function `View()` on the final line opens a spreadsheet view of any data frame or matrix. This function can be helpful when we want to get a quick overview of the data.

It would have been possible to enter the commands ourselves in the console. We can speed up typing the `read_csv()` command by using RStudio’s automatic code completion to find the file name. However, I recommend using the GUI when importing a file for the first time. There is some variety among plain-text spreadsheet formats. For example, not all files use a comma as a column separator. Tabs and semicolons are also commonly used. The preview in the GUI makes it easy to detect whether we must change some of the defaults. After clicking the ‘Import’ button, we can copy any useful bits and pieces from the automatic console output into an R script so that we will not have to go through the GUI again next time we want to import the spreadsheet:

- If you have not loaded the `readr` package individually or as part of the `tidyverse` (`library(tidyverse)`), copy `library(readr)` to the top of your R Markdown file.
- Copy `epl_table <- read_csv("your_directory/epl_table.csv")` to an appropriate position in your R Markdown file.

Please do not copy any calls to the `View()` function to your R Markdown file because your readers are usually not interested in seeing all elements in the data frame. Instead, if you want to present a snippet of the data, use `head()`, `str()` or `tibble::glimpse()`.

Calls to `read_csv()` automatically generate control output, showing the inferred data type of each column. Because this information is not of interest to readers of the knitted R Markdown, I recommend turning off the control output with the chunk option `message = FALSE`. Instead of adding this option to each chunk individually, it can be set as a global option in the setup chunk as follows.

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(message = FALSE)
```
```

15.3 Importing Excel spreadsheets

Although a CSV file is often the better option when storing spreadsheet data for the reasons given in section 15.1, it is sometimes convenient to import Excel spreadsheets directly into R without taking the detour via the CSV format.

In RStudio’s environment tab, please click on ‘Import Dataset’. Then choose ‘From Excel...’ (figure 15.8).

After you click on `epl_table.xlsx` in your RStudio project folder, you can see

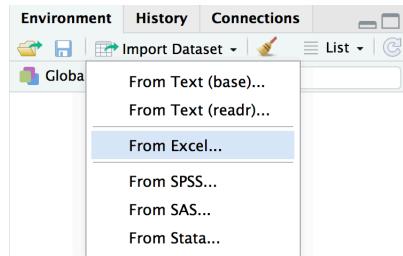


FIGURE 15.8: It is possible to directly import data from XLSX with the RStudio GUI.

a preview of the data in a dialogue window (figure 15.9). In this example, the default settings are fine; thus, you can simply click ‘Import’.

The screenshot shows the 'Import Options' dialogue window. It includes a 'Data Preview' table with 6 rows of EPL team data, an 'Import Options' section with fields for Name, Sheet, Range, Max Rows, Skip, NA, First Row as Names, and Open Data Viewer, and a 'Code Preview' section with R code for reading the file. At the bottom are 'Import' and 'Cancel' buttons.

| Pos
(double) | Team
(character) | Pld
(double) | W
(double) | D
(double) | L
(double) | GF
(double) | GA
(double) | GD
(double) |
|-----------------|---------------------|-----------------|---------------|---------------|---------------|----------------|----------------|----------------|
| 1 | Leicester City | 38 | | 23 | 12 | 3 | 68 | 36 |
| 2 | Arsenal | 38 | | 20 | 11 | 7 | 65 | 36 |
| 3 | Tottenham Hotspur | 38 | | 19 | 13 | 6 | 69 | 35 |
| 4 | Manchester City | 38 | | 19 | 9 | 10 | 71 | 41 |
| 5 | Manchester United | 38 | | 19 | 9 | 10 | 49 | 35 |
| 6 | Southampton | 38 | | 18 | 9 | 11 | 59 | 41 |

Previewing first 50 entries.

Import Options:

- Name: epl_table
- Sheet: Default
- Range: A1:D10
- Max Rows:
- Skip: 0
- NA:
- First Row as Names
- Open Data Viewer

Code Preview:

```
library(readxl)
epl_table <- read_excel("teaching/davis_2018/lesson10/slides/epl_table.xlsx")
View(epl_table)
```

Import Cancel

FIGURE 15.9: A dialogue window appears when we import data from XLSX with RStudio’s GUI.

RStudio generates the following console output.

```
> library(readxl)
> epl_table <- read_excel("~/your_directory/epl_table.xlsx")
> View(epl_table)
```

We notice that, compared to importing a CSV file, we call a different function to import the XLSX file: `read_excel()` instead of `read_csv()`. The function `readxl::read_excel()` has many useful features. For example, it can handle XLSX files with multiple sheets. You can find more information about the `readxl` package by running `help(package = "readxl")`.

15.4 Summary and outlook

Importing spreadsheet data from external files is a standard procedure in data analysis. In this chapter, we learned how to import data from CSV and XLSX files. CSV is the recommended format for most applications. The `read_csv()` function from the `readr` package can import most CSV files that we encounter in the wild. The `readr` package also contains additional functions that can be useful in some cases. If the input data are in XLSX format, the recommended practice used to be to convert the XLSX file to CSV format with third-party software (figure 15.1). Thanks to the `readxl` package, we now have R functions that simplify the workflow by directly importing XLSX files into R.

15.5 Just checking

Which of the following statements about CSV files is correct?

- (a) We must use bold font to indicate that the first line in the CSV contains column headers.
- (b) Unlike XLSX files, CSV files cannot contain formulas that automatically update the value of one cell in the spreadsheet if we change another cell.
- (c) Unlike XLSX files, CSV files cannot be opened by Excel. We must first export the CSV to XLSX with another piece of software (e.g. `read_excel()` in R).
- (d) Just like XLSX files, CSV files can have multiple sheets.

You can confirm your answer by looking at appendix [A.14](#).

Exercises: Practising R programming

Days of the week in the Shire calendar

Prerequisite: chapters 8 and 9

Approximate duration: 45 minutes

In J. R. R. Tolkien's novels, the hobbits of the Shire use a calendar that differs from the Gregorian calendar. Table 15.1 shows the relationship between the days of the week in these two calendars.¹

- (I) Write functions that converts a Gregorian day of the week into the name of the corresponding Shire day using
- (I) `if_else()`.
 - (II) `case_when()`.
 - (III) `recode()`. You may need to look up the documentation at `?recode`.

Call these functions `shire_day_from_gregorian_a()`, `shire_day_from_gregorian_b()` and `shire_day_from_gregorian_c()`, respectively.

¹Data from http://tolkiengateway.net/wiki/Shire_Calendar, accessed on 27 January 2022.

TABLE 15.1: Days of the week

| Shire day | Day in Gregorian calendar |
|-----------|---------------------------|
| Sterday | Monday |
| Sunday | Tuesday |
| Monday | Wednesday |
| Trewsday | Thursday |
| Hevensday | Friday |
| Mersday | Saturday |
| Highday | Sunday |

If the argument passed to any of these functions is not the name of a Gregorian weekday, the function should return `NA`. Which of these three functions would you stylistically prefer?

- (II) The `bsts` package contains a vector called `weekday.names`. Install the package and test whether `shire_day_from_gregorian(weekday.names)` returns the correct result.

Measuring run-times with the `microbenchmark` package

Prerequisite: chapters 8–10

Approximate duration: 60 minutes

We learned in section 2.4.2 that R has a built-in vectorised function `abs()` that calculates the absolute values of elements in a numeric input vector. Remember that the absolute value of a number x is defined as

$$|x| = \begin{cases} x & \text{if } x \geq 0, \\ -x & \text{otherwise.} \end{cases}$$

Below are four alternatives to `abs()` that produce the same numeric output.

- `abs_with_if_else <- function(x) {
 dplyr::if_else(x >= 0, x, -x)
}`
- `abs_with_subsetting <- function(x) {
 neg <- (x < 0)
 x[neg] <- -x[neg]
 x
}`
- `abs_with_data_type_conversion <- function(x) {
 ((x > 0) - (x < 0)) * x
}`
- `abs_with_for_loop <- function(x) {
 for (i in seq_along(x)) {
 if (x[i] < 0) {
 x[i] <- -x[i]
 }
 }
 x
}`

Go through the code of `abs_with_if_else()`, `abs_with_subsetting()` and `abs_with_data_type_conversion()`. Make sure you understand why these functions return the absolute values of elements in `x`.² We want to compare the performance of these functions in terms of their run-times.

- (a) Load the **microbenchmark** package. Look through the documentation of the `microbenchmark()` function. You may want to test a few examples in the documentation and, if necessary, search the World Wide web for more information.
 - (b) Compare the run-times of
 - `abs()`.
 - `abs_with_if_else()`.
 - `abs_with_subsetting()`.
 - `abs_with_data_type_conversion()`.
 - `abs_with_for_loop()`.
 For a fair comparison, run all five functions with identical input (e.g. one million standard normal random numbers, generated with `rnorm(1e6)`).
 - (c) Comment on your results in (b). Is it worth writing our own function to replace `abs()`? Suppose we need to write our own function for something less common than the absolute value. Judging from your results in (b), which programming strategy should we choose (e.g. conditional element selection, subsetting, data type conversion or `for`-loops)?
-

Comparing two functions for calculating Pythagorean sums

Prerequisite: chapters 8 and 11

Approximate duration: 60 minutes

Below is the code for two functions, called `pythag_1()` and `pythag_2()`, that both try to calculate the ‘Pythagorean sum’ $\sqrt{a^2 + b^2}$ for numeric input vectors `a` and `b`. In this problem, we assess their advantages and disadvantages.

```
pythag_1 <- function(a, b) {
  sqrt(a^2 + b^2)
}
pythag_2 <- function(a, b) {
```

²There is no need to memorise the syntax of `abs_with_for_loop()`. In this book, we are not going to work with `for`-loops. One of the reasons for this decision will be evident by the end of this exercise.

```

absa <- abs(a)
absb <- abs(b)
p <- pmax(absa, absb)
q <- pmin(absa, absb)
ratio <- q / p
ratio[is.nan(ratio)] <- 1
p * sqrt(1.0 + ratio^2)
}

```

- (a) What is the purpose of `is.nan(ratio)` in the second-to-last line of `pythag_2()`'s function body?
- (b) Compare the run-times of `pythag_1()` and `pythag_2()` with the **microbenchmark** package. Use identical input consisting of long numeric vectors. Summarise your observation in a few sentences.
- (c) By performing numerical tests, find out under which conditions the functions numerically overflow. When do the functions underflow? Comment on the observed differences between `pythag_1()` and `pythag_2()`.
- (d) Is `pythag_1()` or `pythag_2()` better as a general-purpose method? There is no simple right-or-wrong answer. I am interested in your reasoning.

Floating-point accuracy

Prerequisite: chapter 11

Approximate duration: 45 minutes

- (a) Explain the following output. Keep your explanation shorter than ten sentences. Feel free to search the World Wide Web for help.

```

0.1 + 0.2
## [1] 0.3
0.1 + 0.2 == 0.3
## [1] FALSE

```

- (b) Which R function should we use instead of `==` to test for equality of two floating-point numbers?

Categorical data in the General Social Survey

Prerequisite: chapters 12 and 14

Approximate duration: 120 minutes

Submission format: R Markdown

The General Social Survey is a biannual survey conducted by the National Opinion Research Center at the University of Chicago. The survey collects demographic information from representative respondents of the US population. It also records information about their opinions, attitudes and behaviours.

The **forcats** package contains a data frame `gss_cat` with responses to some of the survey questions. Most columns in `gss_cat` contain categorical data encoded as factors. In this exercise, we use some of these data to practise working with factors. For background information about the data, please have a look at the documentation (`?gss_cat`) and the web site of the General Social Survey (<https://gss.norc.org/About-The-GSS>).

For this exercise, we need the packages **formattable**, **kableExtra** and various tidyverse packages; thus, please put the following code immediately below your setup chunk.

```
library(formattable)
library(kableExtra)
library(tidyverse)
```

(I) ‘Race’ of respondents

The ‘race’ of the respondents is the factor `gss_cat$race`.

| Race | Frequency |
|----------------|-----------|
| Other | 1959 |
| Black | 3129 |
| White | 16395 |
| Not applicable | 0 |

FIGURE 15.10: We can produce attractive tables with the packages **formattable** and **kableExtra**. The code for this table is given in the exercise.

- (a) What are the levels of `gss_cat$race`?
- (b) Tabulate the frequency of the ‘races’ with `table()`.

- (c) We can obtain the table in a more readable format with the help of **formattable** and **kableExtra**. You do not need to understand the following code chunk in detail. Just accept it as a recipe for making nice tables. The table produced by this code is shown in figure 15.10.

```
table(gss_cat$race) |>
  as.data.frame() |>
  mutate(Freq = color_bar("lightgreen")(Freq)) |>
  kbl(
    col.names = c("Race", "Frequency"),
    align = c("l", "r"),
    escape = FALSE
  ) |>
  kable_styling(c("striped", "condensed"), full_width = FALSE)
```

Because we also want to apply the same typesetting technique to most of the other tables in this exercise, let us write a function `nice_table()` that takes two arguments:

- (1) the name of the column in `gss_cat` ("race" in the example above)
 - (2) the column name to be printed in the table ("Race" in this example)
- (d) From the table in (c), we can see that one level is not present in the data. Create a new column `gss_cat$race_drop_unused` that has the same data as `gss_cat$race`, but drop the unused level.
- (e) Make a ‘nice’ frequency table of `race_drop_unused`. Confirm that the unused level was dropped.
- (f) Sort the levels in the order of frequency. Put the most frequent ‘race’ at the top of the ‘nice’ table.

(II) Reported income

- (a) The reported income is the factor `gss_cat$rincome`. What are the levels?
- (b) Without editing labels and levels, tabulate the frequency of the income levels with the `nice_table()` function from (I)-c. Explain why this table would not be suitable for a published report.
- (c) Improve the table. You may want to look at the **forcats** cheat sheet for help: <https://raw.githubusercontent.com/rstudio/cheatsheets/main/factors.pdf>.

(III) Party affiliation

- (a) What are the levels of `gss_cat$partyid`?
- (b) Add a factor column `gss_cat$partyid_aggregated` in which (in this order):
 - (1) "Strong republican" and "Not str republican" become "Republican".
 - (2) "Ind,near rep", "Independent" and "Ind,near dem" become "Independent".
 - (3) "Not str democrat" and "Strong democrat" become "Democrat".
 - (4) "Other party" becomes "Other".
 - (5) "No answer" and "Don't know" become "Missing".
- (c) Make a frequency table of `gss_cat$partyid_aggregated` with the `nice_table()` function from (I)-c.

(IV) Religion and denomination

In this section, we want to demonstrate that ‘denomination’ is predominantly a Protestant concept.

- (a) Add a factor column `gss_cat$is_protestant` with levels "Protestant" and "Other".
- (b) Add a factor column `gss_cat$has_denom` with levels (in this order):
 - (1) "Has a denomination"
 - (2) "No denomination"
 - (3) "Not applicable"
 - (4) "Don't know"
 - (5) "No answer"
- (c) Make a two-way table as indicated in figure 15.11. The function `nice_table()` makes one-way table; thus, the function is not directly applicable. Be creative and search for help online!

| | Protestant | Other |
|--------------------|------------|-------|
| Has a denomination | | |
| No denomination | | |
| Not applicable | | |
| No answer | | |
| Don't know | | |

FIGURE 15.11: In part (IV) of the exercise, you are asked to fill in the numbers in this two-way table.



Part IV

Data visualisation with ggplot2



16

From data frame to plot



FIGURE 16.1: This chapter is under construction.



17

Grammar of graphics



FIGURE 17.1: This chapter is under construction.



18

Fine-tuning the appearance of plots



FIGURE 18.1: This chapter is under construction.



Exercises: Preston curve

Prerequisite: chapters 16–18

Approximate duration: 120 minutes

Submission format: R Markdown

In a classic paper, [Preston \(1975\)](#) discussed scatter plots of life expectancy versus national income per capita (see figure 18.2), where each point represents one country. The term ‘Preston curve’ has since then become a synonym for curves fitted to similar data, usually with the per-capita gross domestic product (GDP), instead of national income, as x-value. [Preston \(1975\)](#) and many others have used untransformed x-values and y-values. For a different take on plotting the data, the Swedish foundation [Gapminder \(2016\)](#) uses a logarithmic scale for income (figure 18.3). A logarithmic scale makes sense because most economic indicators are right-skewed. I recommend to adopt Gapminder’s approach.

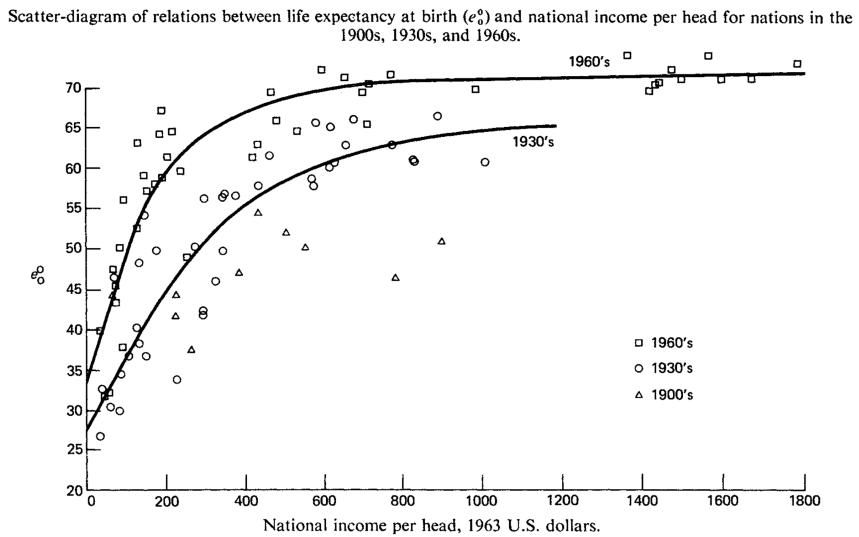


FIGURE 18.2: Diagram from [Preston \(1975\)](#).

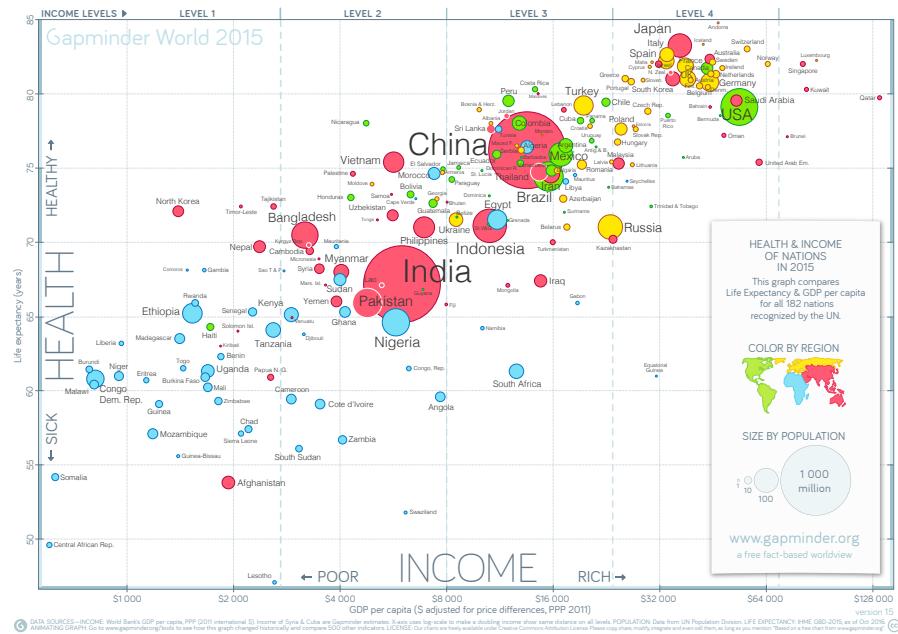


FIGURE 18.3: Diagram from Gapminder (2016).

Learning objectives

We practise our **ggplot2** skills by making a plot that is comparable to the plot of life expectancy as a function of GDP by the Gapminder Foundation (figure 18.4).

Tasks

- (1) Download https://michaelgastner.com/DAVisR_data/life_quality.csv.¹ This CSV file contains the columns:

¹These data are based on information available from the World Bank (accessed on 14 February 2022).

- GDP per capita (current US\$): <https://data.worldbank.org/indicator/NY.GDP.PCAP.CD>
- Life expectancy at birth, total (years): <https://data.worldbank.org/indicator/SP.DYN.LE00.IN>
- Population: <https://data.worldbank.org/indicator/SP.POP.TOTL>

- `country_name`
- `country_code`: standardised 3-letter code (ISO 3166-1 alpha-3)
- `gdp_per_capita` (US\$, PPP 2015)
- `life_expectancy` (in years)
- `pop`: population
- `continent`

These numbers are not exactly the same as those used by [Gapminder \(2016\)](#). Thus, please do not worry if your final plot does not look identical.

Import the CSV as a tibble.

- (2) Make a bubble chart with `ggplot2` where:

- the x-coordinate is the GDP per capita.
- the y-coordinate is the life expectancy.
- the colour indicates the continent.
- the size of the bubble indicates the population.

Change the axis labels and give the plot a title. Give credit to the World Bank as data source in the form of a caption. Make the bubbles semitransparent. (An improvement compared to Gapminder!)

Do not worry about the scales for the coordinates and the bubble areas yet. We will fix them shortly.

- (3) Change the x-coordinates from a linear to a logarithmic scale. Change the minor breaks as shown in figure 18.4, where they appear as thin white lines. See section 10.1.5 in [Wickham et al. \(2021\)](#) for related examples. Change the default tick mark labels ('1e+03', '1e+04', '1e+05') to reader-friendlier labels ('\$1,000', '\$10,000', '\$100,000'). Why do you think `ggplot2` puts axis ticks, by default, at integer powers of 10 compared to Gapminder's tick positions (e.g. '\$8,000', '\$16,000', '\$32,000')?
- (4) Use `scale_size_area()` so that the areas of the bubbles in the legend represent populations of 1 million, 10 million, 100 million and 1 billion. Change the numbers in the legends from '1e+06', '1e+07', '1e+08', '1e+09' to the reader-friendlier strings '1 million', '10 million', '100 million' and '1 billion'. Increase `max_size` so that the bubble areas are approximately the same as in the Gapminder figure.
- (5) Change the colour scale to the ColorBrewer palette 'Set1'. See section 11.3.1 in [Wickham et al. \(2021\)](#) for related examples. In my opinion, Set1 provides clearer contrasts than `ggplot2`'s default colours. These are not the same colours as in the Gapminder figure, but let us not worry about it.

- (6) Semi-transparent colours are great at dealing with overplotting in the bubble plot. However, they are not optimal for the legend, where we would like to see clear contrasts between the colours. Override the alpha aesthetic in the legend. Also increase the sizes of the circles in the colour legend so that the colours are easier to read. See section 11.3.6 in [Wickham et al. \(2021\)](#) for related examples.
- (7) Fit a single LOESS curve to all data points. Use the countries' population sizes as weighting variable. Use a neutral colour for the curve to indicate that the curve is not specific to any continent. Do not show the geom for the LOESS curve in the legend.
- (8) Using `ggrepel::geom_text_repel()`, add the country names as labels to:
 - the 5 countries with the highest GDP.
 - the 5 countries with the lowest GDP.
 - the 5 countries with the longest life expectancy.
 - the 5 countries with the shortest life expectancy.
 - the 5 most populous countries.

(Some countries are in more than one of these categories.)

Make the text colour equal to the continent's colour. Choose the font sizes and text positions so that the labels are easily legible.

- (9) Feel free to make more adjustments if you think they improve the quality of the plot. Then adjust the figure dimensions in the knitted file. Labels should be clearly legible without appearing disproportionately large. Figure 18.4 shows my attempt.
- (10) Write a few sentences about the data. What does the plot reveal about the data? If you refer to specific countries, make sure to add the corresponding labels in the plot if necessary.

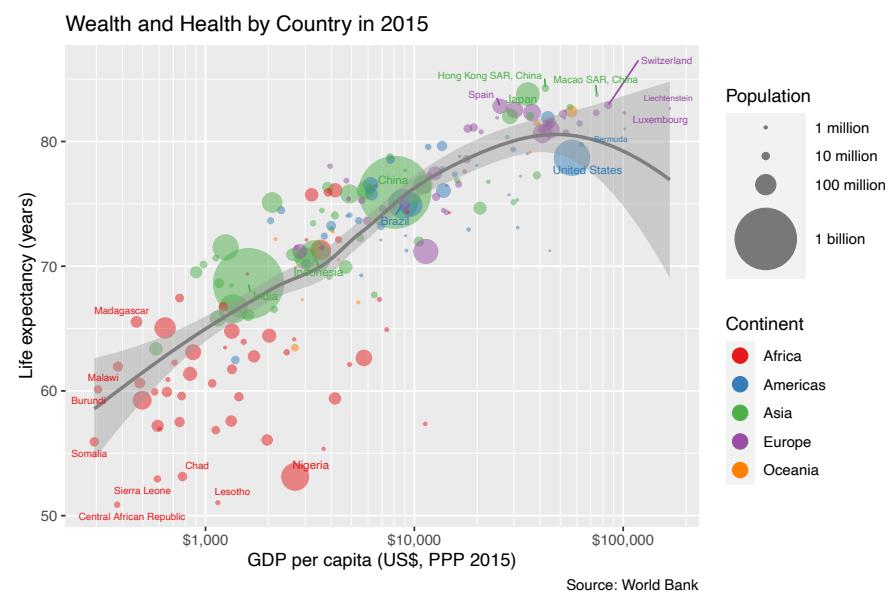


FIGURE 18.4: Data from [Gapminder \(2016\)](#) plotted with **ggplot2**.



— | — | —

Part V

Refining raw data



19

*Data transformation with **dplyr***



FIGURE 19.1: This chapter is under construction.

In chapter 14, we learned techniques for subsetting data frames using functions in R’s base installation. In practice, there are many more operations besides subsetting that we would like to apply to data frames (e.g. sorting or obtaining summaries of the data). Although R’s built-in functions provide adequate solutions for most data-frame operations, their use tends to be cumbersome and often leads to repetitive code. The **dplyr** package, which is part of the tidyverse, offers greater convenience thanks to functions that are intuitive to use and allow easy constructions of pipelines. When working with large data sets, **dplyr** functions also tend to be computationally more efficient than their base-R counterparts.

In this chapter, we learn **dplyr**’s essential vocabulary and grammar. We use the built-in data frame `iris` in the examples. We briefly encountered `iris` in chapter 14. This data frame contains measurements of iris flowers by the American botanist Edgar Anderson. There are five columns in `iris`: sepal

length, sepal width, petal length, petal width and the species. The first four columns are numeric, whereas species is encoded as a factor.

```
library(tidyverse)
glimpse(iris)
## #> #> Rows: 150
## #> #> Columns: 5
## #> #> $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.~  
## #> #> $ Sepal.Width <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.~  
## #> #> $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.~  
## #> #> $ Petal.Width <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.~  
## #> #> $ Species <fct> setosa, setosa, setosa, setosa, setosa, setosa, s~
```

In the next sections, we transform `iris` by applying the following **dplyr** functions:

- `filter()` in section ...
- `select()` in section ...
- `summarise()` in section ...
- `group_by()` in section ...
- `arrange()` in section ...
- `mutate()` in section ...

The names of all these functions are verbs, which makes them easy to memorise and easy to use, as we see shortly.

19.1 Create a subset of rows with `filter()`

20

*Tidy data with **tidyverse***



FIGURE 20.1: This chapter is under construction.



21

Merging data in different data frames



FIGURE 21.1: This chapter is under construction.



Exercises: Relational data for country-level statistics

Prerequisite: chapters 19–21

Approximate duration: 120 minutes

Submission format: R Markdown

Prerequisite: chapters 19–21

Approximate duration: 120 minutes

Submission format: R Markdown

In an earlier exercise, we created figure 18.4, which shows GDP per capita (x-axis), life expectancy (y-axis) and population (size) by country. In this exercise, we take a closer look at publicly available data for these variables.

Learning objectives

We will practice our data wrangling skills. We will work with different kinds of joins and other functions from the **dplyr** package.

Data

We need three data sets from the World Bank:¹

- GDP per capita (current US\$): https://michaelgastner.com/DAVisR_data/API_NY.GDP.PCAP.CD_DS2_en_excel_v2_2105330.xls
- Life expectancy at birth, total (years): https://michaelgastner.com/DAVisR_data/API_SP.DYN.LE00.IN_DS2_en_excel_v2_2056863.xls

¹These files were downloaded from <https://data.worldbank.org/indicator/> on 21 March 2021. We work with XLS files because there is a minor formatting issue with the CSV files provided by the World Bank.

- Population: https://michaelgastner.com/DAVisR_data/API_SP.POP.TOTL_DS2_en_excel_v2_2106205.xls

At the end of this exercise, we compare these data with data from R's **gapminder** package.

Tasks

- (1) Import the World Bank data for GDP per capita, life expectancy and population.
- (2) Is the column with three-letter country codes (second column from the left) the same in all three spreadsheets?
- (3) Merge the three spreadsheets into a single tibble `wb` (for World Bank) with columns for
 - country name.
 - country code.
 - year.
 - GDP per capita.
 - life expectancy.
 - population.
- (4) Some rows in the World Bank spreadsheets do not represent a (single) country, for example 'East Asia & Pacific (excluding high income)'. We want to remove the corresponding columns from `wb`. We are going to automate this task by using the tibble `codelist` in the **countrycode** package.

The purpose of the **countrycode** package is to simplify the task of merging country-level data in different data bases. The same country often appears under a variety of names in official documents. For example, 'United States of America', 'U.S.A.' and 'US' all refer to the same country. The recommended practice when joining country-level data in different data bases is to use a standardised set of codes that uniquely identify each country. One option is to work with ISO 3166-1 alpha-3 codes.² These codes are in the column `iso3c` of `codelist`.

Perform an anti-join to find out which three-letter country codes in the World Bank spreadsheets do not have a matching code in `codelist`. What are the corresponding 'country names'? Do the results make sense?

²For background information, see https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3.

- (5) Use a join function from **dplyr** to remove all rows from **wb** that do not match any country code in **codelist**.
- (6) Given the input data from the World Bank, a country can only be added to the scatter plot shown in figure 18.4 if all of the following three pieces of information about the country are known:
 - GDP per capita.
 - life expectancy.
 - population.

Summarise the number of countries per year that cannot be plotted on the basis of the World Bank data. Here are the first few rows of a tibble that shows the number of missing countries for each year.

```
head(missing_values)
## # A tibble: 6 x 2
##   year  na_countries
##   <chr>     <int>
## 1 1960        120
## 2 1961        118
## 3 1962        115
## 4 1963        116
## 5 1964        116
## 6 1965        106
```

- (7) Plot the number of missing countries per year. Comment on the result.
- (8) R's **gapminder** package contains a tibble **gapminder** that also has information about GDP, life expectancy and population.

Make a copy of **gapminder** and append a column with three-letter country codes. Here is the code snippet for this task.

```
gapminder_copy <-
  gapminder |>
  mutate(country_code = countrycode(country, "country.name", "iso3c"))
```

After running this command, are there countries in **gapminder_copy** without a country code?

- (9) Which countries are in **gapminder** but do not appear in **wb**? Which countries are in **wb** but do not appear in **gapminder**?

- (10) Let us compare the GDP data in `wb` and `gapminder` for 2007. Remove all unrelated rows and columns. Merge the information from `wb` and `gapminder` into a tibble `wb_gap` so that only those countries are included that appear in both tibbles.
- (11) Append a column to `wb_gap` that shows the percentage difference of Gapminder's GDP estimate compared to the World Bank estimates. (For example, if the World Bank's estimate is \$5000 and Gapminder's estimate is \$2500, the percentage difference is -50%).
- (12) For which five countries is the percentage difference largest? For which five countries is it smallest (i.e. most strongly negative).

Part VI

Functional programming



22

Lists

At this stage along our R learning curve, we are familiar with matrices and data frames. These data structures are excellent tools in many situations, but there is a limitation: the shapes of matrices and data frames must be rectangular (i.e. every column must be equally long). However, real data often do not naturally fit into a rectangular format.

Table 22.1 shows, as an example, the track listings of the soundtrack of the film ‘Magical Mystery Tour’ by the Beatles. The soundtrack was released as a double extended-play record; thus, it appeared on four sides (two on each vinyl record). Some of the information is numeric (e.g. the track number), whereas other information is in the form of character strings (e.g. the song titles). Another complication is that the number of writers is not the same for all songs. Most songs have two authors (John Lennon and Paul McCartney), but ‘Flying’ was written by all four Beatles, and ‘Blue Jay Way’ is a single-authored song by George Harrison. We would not be able to express the information in the ‘Writer(s)’ column naturally within the confines of a vector. Moreover, the data are hierarchical: the record has four sides, each side has a variable number of songs, and songs have a variable number of writers. Representing these data as a rectangular data frame would not be straightforward.

A better solution to store such data is to save them as a *list*. Lists in R can combine data of different types (e.g. one numeric and one character vector) without coercing the elements into a single type. Lists can also store objects of different lengths (in our example, the writers of a song) in a single data structure. Furthermore, lists can contain other lists; thus, lists can be used to represent hierarchically nested data. Because of their flexibility, lists are a great tool when working with real-world data.

22.1 Creating lists

In R, we generate a list with the function `list()`. The arguments passed to `list()` are the objects we want to combine into a single list. For example, here is a list with the data for side D of ‘Magical Mystery Tour’, comprising the track number, title, writer and duration.

TABLE 22.1: Track listing of the soundtrack of 'Magical Mystery Tour.'

| Track | Title | Writer(s) | Duration (in sec) |
|---------------|-------------------------|--------------------------------------|-------------------|
| Side A | | | |
| 1 | Magical Mystery Tour | Lennon, McCartney | 168 |
| 2 | Your Mother Should Know | Lennon, McCartney | 153 |
| Side B | | | |
| 1 | I Am the Walrus | Lennon, McCartney | 275 |
| Side C | | | |
| 1 | The Fool on the Hill | Lennon, McCartney | 180 |
| 2 | Flying | Harrison, Lennon, McCartney, Starkey | 136 |
| Side D | | | |
| 1 | Blue Jay Way | Harrison | 230 |

```
side_d <- list(1, "Blue Jay Way", "Harrison", 230)
side_d
## [[1]]
## [1] 1
##
## [[2]]
## [1] "Blue Jay Way"
##
## [[3]]
## [1] "Harrison"
##
## [[4]]
## [1] 230
```

We discuss the output in more detail in section 22.2. However, we can already see that there are no quotation marks around the track number 1 or the duration 230, which indicates that R did not coerce the numbers into character strings.

Vectors in a list do not need to be equally long. For example, here is a list with the song writers on side C. The first vector in `writers_c` has two elements and the second vector four.

```
writers_c <- list(
  c("Lennon", "McCartney"),
```

```
c("Harrison", "Lennon", "McCartney", "Starkey")
)
writers_c
## [[1]]
## [1] "Lennon"      "McCartney"
##
## [[2]]
## [1] "Harrison"   "Lennon"      "McCartney" "Starkey"
```

In `side_d` and `writers_c`, all list elements are vectors. However, lists can be used much more generally. They have no restrictions regarding the number of elements, their data types and lengths. Apart from vectors, lists can also contain factors, matrices and other lists. In the next code chunk, we demonstrate how we create a list (`side_c`) that contains another list (`writers_c`).

```
side_c <- list(
  1:2,
  c("The Fool on the Hill", "Flying"),
  writers_c,
  c(180, 136)
)
side_c
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] "The Fool on the Hill" "Flying"
##
## [[3]]
## [[3]][[1]]
## [1] "Lennon"      "McCartney"
##
## [[3]][[2]]
## [1] "Harrison"   "Lennon"      "McCartney" "Starkey"
##
## [[4]]
## [1] 180 136
```

22.2 Accessing a single list element

The output after `side_c` in the previous code chunk looks different from the output we have seen for vectors, factors, matrices or data frames. Some indices are, as we are used to, inside a single pair of square brackets (e.g. `[1]`). Other indices are inside *two* pairs of square brackets (e.g. `[[1]]`). What does this notation mean?

In the output above, double square brackets contain the indices of the *list* elements:

- `[[1]]` is the index of the list's first element, (i.e. a numeric vector with the track numbers).
- `[[2]]` is the index of the list's second element (i.e. a character vector with the song titles).
- `[[3]]` is the index of the list's third element (i.e. a list with the writers' names). This sub-list has two elements with indices `[[3]][[1]]` and `[[3]][[2]]`, each of which is a character vector.
- `[[4]]` is the index of the list's fourth element, (i.e. a numeric vector with the durations in seconds).

We can confirm with the function `length()` that the list has exactly four elements.

```
length(side_c)
## [1] 4
```

We can use a single number inside double square brackets to obtain the corresponding list element.

```
side_c[[2]]
## [1] "The Fool on the Hill" "Flying"
```

The result of `side_c[[2]]` is a character vector. If we want to extract, for example, the first element from this vector, we can continue with the subsetting method we learned for vectors in section 3.3.1—*single* square brackets.

```
side_c[[2]][1]
## [1] "The Fool on the Hill"
```

It is important to understand the distinction between double and single square

brackets in this example. Double square brackets extract an element from a list, whereas single square brackets extract an element from a vector.¹

As if the situation were not confusing enough already, it is in fact possible to subset list elements with either double or single square brackets. However, the result is different. Compare the output from the next two commands.

```
side_c[[2]]
## [1] "The Fool on the Hill" "Flying"
side_c[2]
## [[1]]
## [1] "The Fool on the Hill" "Flying"
```

In the first case (i.e. `side_c[[2]]`), there are no double square brackets in the output. However, in the second case (i.e. `side_c[2]`), the output starts with a number in double square brackets: `[[1]]`. We can solve the mystery behind the different output by looking at the classes of the returned objects. When we extract an element with double square brackets, the result is the content of this list element (in our case, a character vector).

```
class(side_c[[2]])
## [1] "character"
```

By contrast, when we subset a list with single square brackets, the result is again a list.

```
class(side_c[2])
## [1] "list"
is.list(side_c[2])
## [1] TRUE
```

This distinction is important. While we can extract the first song title from the second list element with `side_c[[2]][1]`, it will not work if we replace the double square brackets by single square brackets. In that case, the output contains both song titles together because they form the character vector in the first list element of `side_c[2]`.

```
side_c[2][1]
## [[1]]
## [1] "The Fool on the Hill" "Flying"
```

¹In the technical literature about R, vectors are sometimes referred to as ‘atomic vectors’ and lists as ‘recursive vectors’. The intention of the adjective ‘recursive’ is to emphasise that lists can contain other lists. I find this nomenclature verbose. In this book, I use the term ‘vector’; only to refer to ‘atomic vectors’, not to lists.

Although the different types of subsetting are confusing at first sight, it is possible to make sense of R's behaviour if we think of lists as containers rather than content. In our example, the song titles are the content. The list `side_c` is a container that stores the song titles together with the track numbers, the writers and the durations. When we ask R for the subset containing the song titles, we have to be clear about whether we want

- either the content
- or a container with this content

to be returned. If we are interested in the content, we use double square brackets. If, instead, we want a container with the content, we use single square brackets. Metaphorically, if we think of the container as a metal can and the content as soup, the double square brackets act like a soup spoon that helps us to consume the soup. By contrast, single square brackets pour the soup from one large container into another, possibly smaller, container.

22.3 Subsets with multiple list elements

It is possible to form subsets with multiple elements of a list by supplying a numeric vector of length > 1 in *single* square brackets. The result is another list. For example, here is how we build a list that contains only the track numbers and writers from `side_c`.

```
side_c[c(1, 3)]
## [[1]]
## [1] 1 2
##
## [[2]]
## [[2]][[1]]
## [1] "Lennon"      "McCartney"
##
## [[2]][[2]]
## [1] "Harrison"    "Lennon"      "McCartney" "Starkey"
```

As for vectors, we can exclude list elements with a minus sign (e.g. `side_c[c(1, 3)]` is the same as `side_c[-c(2, 4)]`). We can also supply a logical vector for subsetting (e.g. `side_c[c(1, 3)]` is the same as `side_c[c(TRUE, FALSE, TRUE, FALSE)]`).

When we form subsets with multiple elements of a list, it is important that we use *single* square brackets. When we use *double* square brackets, R applies 'recursive indexing':

- R interprets `side_c[[c(3, 1)]]` as `side_c[[3]][[1]]`, which happens to be the same as `side_c[[3]][1]` (i.e. the writers of the first song).²
- R interprets `side_c[[c(3, 1, 2)]]` as `side_c[[3]][[1]][[2]]` (i.e. the second writer of the first song).

```
side_c[[c(3, 1)]]
## [1] "Lennon"    "McCartney"
side_c[[c(3, 1, 2)]]
## [1] "McCartney"
```

Recursive indexing is confusing and rarely needed in practice. It is better to stay away from it.

22.4 Named list elements

We can add names as attributes of list elements. Either we use the `names()` function:

```
names(side_c) <- c("track", "title", "writers", "duration")
```

Or we pass the names of the list elements to `list()` as argument names:

```
side_c <- list(
  track = 1:2,
  title = c("The Fool on the Hill", "Flying"),
  writers = writers_c,
  duration = c(180, 136)
)
side_c
## $track
## [1] 1 2
##
## $title
## [1] "The Fool on the Hill" "Flying"
##
## $writers
```

²For technical reasons beyond the scope of this course, it is possible to subset vectors with both single and double square brackets. So `side_c[[3]][1]` and `side_c[[3]][[1]]` are in fact equivalent. Because single and double square brackets are synonymous for vectors, it is better style to always use the more concise single-bracket notation.

```
## $writers[[1]]
## [1] "Lennon"    "McCartney"
##
## $writers[[2]]
## [1] "Harrison"  "Lennon"     "McCartney" "Starkey"
##
##
## $duration
## [1] 180 136
```

The output in the console looks different from before. R replaced the numbers in double square brackets by dollar signs \$ followed by the element names.

We can use the names to extract elements from the list in four different ways.

- We enter the name in quotes and inside *single* square brackets. In this case, the result is a list.

```
side_c["title"]
## $title
## [1] "The Fool on the Hill" "Flying"
class(side_c["title"])
## [1] "list"
```

- We enter the names in quotes and inside *double* square brackets. In this case, the result is the content of the list element (e.g. a character vector).

```
side_c[["title"]]
## [1] "The Fool on the Hill" "Flying"
class(side_c[["title"]])
## [1] "character"
```

- We append a dollar sign after the variable name of the list (here `side_c`). Then, we type the name of the list element without quotes. The result is the same as for the double-square-bracket method: the content of the list element.

```
side_c$title
## [1] "The Fool on the Hill" "Flying"
class(side_c$title)
## [1] "character"
```

- The tidyverse package `purrr` contains the pipe-friendly function `pluck()`.

```
library(purrr)
pluck(side_c, "title")
## [1] "The Fool on the Hill" "Flying"
side_c |>
  pluck("title") |>
  class()
## [1] "character"
```

As we have seen, if `names()` is followed by the assignment operator `<-`, then `names()` sets new names for the elements. If, instead, we want to look up the existing names, we use `names()` without an assignment operator.

```
names(side_c)
## [1] "track"     "title"      "writers"    "duration"
```

If we want to form a subset with more than one list element, we can use a character vector of length > 1 inside *single* square brackets. The result is a list.

```
side_c[c("title", "writers")]
## $title
## [1] "The Fool on the Hill" "Flying"
##
## $writers
## $writers[[1]]
## [1] "Lennon"    "McCartney"
##
## $writers[[2]]
## [1] "Harrison"  "Lennon"    "McCartney" "Starkey"
```

Using multiple names in double square brackets or as arguments of `pluck()` is technically possible; however, it sends us into the quagmire of recursive indexing. We cannot use multiple names at all when using the $\$$ -subsetting method.

We remove names from all list elements by assigning `NULL`.

```
side_c_copy <- side_c
names(side_c_copy) <- NULL
```

Alternatively, the function `unname()` returns a list without the names. For example, the return value of `unname(side_c)` does not have names, which we can tell from the output below because the $\$$ -signs are gone and the double

square brackets are back. However, `uname()` did not overwrite `side_c`; it still has names.

```
uname(side_c)
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] "The Fool on the Hill" "Flying"
##
## [[3]]
## [[3]][[1]]
## [1] "Lennon"      "McCartney"
##
## [[3]][[2]]
## [1] "Harrison"    "Lennon"      "McCartney" "Starkey"
##
## [[4]]
## [1] 180 136
names(side_c)
## [1] "track"     "title"      "writers"    "duration"
```

22.5 Replacing list elements

To change list elements, we first specify the subset we want to replace. Then, we use the assignment operator `<-` and enter the new value(s). For example, here is how we add the writers' first initials to the information about the first song.

```
side_c$writers <- list(
  c("J. Lennon", "P. McCartney"),
  c("G. Harrison", "J. Lennon", "P. McCartney", "R. Starkey")
)
side_c
## $track
## [1] 1 2
##
## $title
## [1] "The Fool on the Hill" "Flying"
```

```

## 
## $writers
## $writers[[1]]
## [1] "J. Lennon"      "P. McCartney"
##
## $writers[[2]]
## [1] "G. Harrison"   "J. Lennon"      "P. McCartney" "R. Starkey"
##
## 
## $duration
## [1] 180 136

```

22.6 Adding a new element to a list

We can add a new element at the end of an existing list by typing a \$ after the list's variable name, followed by the name of the new list element. For example, here is how we add a list element with the name `lead_vocals`. The second element in the vector to the right of the assignment operator `<-` is an empty string because 'Flying' is a purely instrumental song.

```

side_c$lead_vocals <- c("P. McCartney", "")
side_c
## $track
## [1] 1 2
##
## $title
## [1] "The Fool on the Hill" "Flying"
##
## $writers
## $writers[[1]]
## [1] "J. Lennon"      "P. McCartney"
##
## $writers[[2]]
## [1] "G. Harrison"   "J. Lennon"      "P. McCartney" "R. Starkey"
##
## 
## $duration
## [1] 180 136
## 
```

```
## $lead_vocals
## [1] "P. McCartney" ""
```

We could have equivalently used the double square bracket notation.

```
side_c[["lead_vocals"]] <- c("P. McCartney", "")
```

If the printed output from a list becomes too long to read at one glance, we can alternatively look at a more compact display with `str()`. We can apply `str()` also to any other R object, but it is particularly handy when working with lists.

```
str(side_c)
## List of 5
## $ track      : int [1:2] 1 2
## $ title      : chr [1:2] "The Fool on the Hill" "Flying"
## $ writers    :List of 2
##   ..$ : chr [1:2] "J. Lennon" "P. McCartney"
##   ..$ : chr [1:4] "G. Harrison" "J. Lennon" "P. McCartney" "R. Starkey"
## $ duration   : num [1:2] 180 136
## $ lead_vocals: chr [1:2] "P. McCartney" ""
```

22.7 Removing an element from a list

We can remove a list element by assigning `NULL`.

```
side_c$lead_vocals <- NULL
str(side_c)
## List of 4
## $ track      : int [1:2] 1 2
## $ title      : chr [1:2] "The Fool on the Hill" "Flying"
## $ writers    :List of 2
##   ..$ : chr [1:2] "J. Lennon" "P. McCartney"
##   ..$ : chr [1:4] "G. Harrison" "J. Lennon" "P. McCartney" "R. Starkey"
## $ duration   : num [1:2] 180 136
```

22.8 Tibbles can have list columns

All tibbles we have encountered so far were composed of one vector per column. It is also possible that a tibble column is a list instead of a vector. Here is side C of table 22.1 stored as a tibble with a list column for the writers.

```
library(tibble)
side_c_tibble <-
  tibble(
    track = 1:2,
    title = c("The Fool on the Hill", "Flying"),
    writers = writers_c,
    duration = c(180, 136)
  )
side_c_tibble
## # A tibble: 2 x 4
##   track     title       writers   duration
##   <int> <chr>      <list>     <dbl>
## 1     1 The Fool on the Hill <chr [2]>     180
## 2     2 Flying           <chr [4]>     136
```

We can subset list columns in the same way as any other list. For example, the next code chunk returns the second writer in the first element of the `writers` column.

```
side_c_tibble$writers[[1]][2]
## [1] "McCartney"
```

Tibbles with list columns should be used sparingly, but they arise naturally in some contexts. For example, we encounter list columns again when we discuss networks in chapter 24.

22.9 Summary and outlook

Lists are powerful data structures because they impose hardly any restrictions on the data to be stored. Lists can mix different data of different classes and structures. Lists can contain other lists; thus, they are an excellent data structure to store hierarchically nested data. We encounter lists again when we discuss network analysis in chapter 24 and regression in chapter 34. Before

we get there, we still have to learn in the next chapter how to apply functions to list elements.

22.10 Just checking

First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix [A.15](#).

- (I) What are the lengths and classes of `side_c[4][1]`, `side_c[[4]][1]` and `side_c[4][[1]]`?
- (II) Here are some results from the 2016 Olympic women's long jump qualifying round. Each athlete had a maximum of three jumps. Athletes automatically qualified for the final round if they reached or exceeded 6.75 metres. If they satisfied this criterion before the third jump, they did not need to continue with more jumps in the qualification. A foul jump is denoted by `NA`.

```
longjump <- list(  
  spanovic = 6.87,  
  mihambo = c(6.63, 6.82),  
  reese = 6.78,  
  bartoletta = c(6.44, 6.70, 6.61),  
  klishina = c(6.64, NA, 6.64)  
)
```

Which of the following three code chunks extracts Mihambo's results **as a numeric vector**?

Chunk (i):

```
longjump[["mihambo"]]
```

Chunk (ii):

```
longjump["mihambo"]
```

Chunk (iii):

```
longjump$mihambo
```

- (a) Chunks (i) and (iii), but not (ii).
 (b) Only chunk (iii) is correct.
 (c) All of (i), (ii) and (iii).
 (d) Chunks (i) and (ii), but not (iii).
- (III) Consider again the list `longjump` from question (II). How can the same data be represented as a tibble with two columns: the athlete's names as first column and the lengths of their jumps as second column?

```
(a) longjump_tibble <- tibble(  
  spanovic = 6.87,  
  mihambo = c(6.63, 6.82),  
  reese = 6.78,  
  bartoletta = c(6.44, 6.70, 6.61),  
  klishina = c(6.64, NA, 6.64)  
)
```

```
(b) longjump_tibble <- as_tibble(longjump)
```

```
(c) longjump_tibble <- tibble(  
  athlete = names(longjump),  
  result = unname(longjump)  
)
```

- (d) The data in the list cannot be represented as a tibble. The number of jumps is not the same for all athletes. Thus, it is impossible to store the data in a rectangular format.
- (IV) Create a list `mmt` that contains the information in table 22.1 for sides A to D of 'Magical Mystery Tour'. The structure of `mmt` should be as follows.

```
str(mmt)
## List of 4
## $ side_a:List of 4
##   ..$ track : int [1:2] 1 2
##   ..$ title : chr [1:2] "Magical Mystery Tour" "Your Mother Should Know"
##   ..$ writers :List of 2
```

```
## ... .$. : chr [1:2] "Lennon" "McCartney"
## ... .$. : chr [1:2] "Lennon" "McCartney"
## ... $. duration: num [1:2] 168 153
## $ side_b:List of 4
## ... $. track   : num 1
## ... $. title   : chr "I Am the Walrus"
## ... $. writers : chr [1:2] "Lennon" "McCartney"
## ... $. duration: num 275
## $ side_c:List of 4
## ... $. track   : int [1:2] 1 2
## ... $. title   : chr [1:2] "The Fool on the Hill" "Flying"
## ... $. writers :List of 2
## ... ... $. : chr [1:2] "Lennon" "McCartney"
## ... ... $. : chr [1:4] "Harrison" "Lennon" "McCartney" "Starkey"
## ... $. duration: num [1:2] 180 136
## $ side_d:List of 4
## ... $. track   : num 1
## ... $. title   : chr "Blue Jay Way"
## ... $. writers : chr "Harrison"
## ... $. duration: num 230
```

23

Applying functions to list elements

A common feature of many programming languages are loop structures (e.g. for-loops or while-loops) that carry out repeated operations on all elements in a data structure. R also has for-loops and while-loops, but they are used less frequently than in other languages. In this book, I do not introduce for-loops and while-loops because they require relatively much boilerplate code and, thus, often obscure the users intention. Instead, we now learn an alternative to for-loops that leads to more readable code: the `map`-family of functions. These functions are in the `purrr` package, which is part of the tidyverse. Thus, `purrr` can be loaded with `library(purrr)` or:

```
library(tidyverse)
```

23.1 `map()` and related functions

The general feature of the `map`-family is that its functions take two types of arguments: objects (vectors or lists) to loop over and a function to apply to each element in these objects. In the context of this book, we mostly need `map`-type functions if the input objects are lists. Therefore, I limit the discussion in this chapter to this special case.

Let us consider again the data about the soundtrack of ‘Magical Mystery Tour’ in table 22.1. Suppose we store the titles in a list in which each element is a character vector with songs on each side.

```
titles <- list(  
  c("Magical Mystery Tour", "Your Mother Should Know"),  
  "I Am the Walrus",  
  c("The Fool on the Hill", "Flying"),  
  "Blue Jay Way"  
)
```

If we want to find out the number of songs on each side, we call `map()` with

two arguments: `titles` and `length`. Here `length` is the name of the function `length()` that we want to apply to each list element. Note that we do not put parentheses after `length` if it is an argument in `map()`.

```
map(titles, length)
## [[1]]
## [1] 2
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 1
```

The result contains the lengths (2, 1, 2 and 1) as list elements.

Next, let us calculate the total duration of each side from a list in which numeric vectors contain the durations of each song.

```
durations <- list(
  c(168, 153),
  275,
  c(180, 136),
  230
)
```

In this case, we apply `sum()` to each element of `durations`.

```
map(durations, sum)
## [[1]]
## [1] 321
##
## [[2]]
## [1] 275
##
## [[3]]
## [1] 316
##
## [[4]]
## [1] 230
```

Thus, the lengths of the four sides are 321, 275, 316 and 230 seconds.

Let us consider yet another example. Suppose we have a list `side_a` with elements of different classes (integer, character, list and numeric).

```
side_a <- list(
  track = 1:2,
  title = c("Magical Mystery Tour", "Your Mother Should Know"),
  writers = list(
    c("Lennon", "McCartney"),
    c("Lennon", "McCartney")
  ),
  duration = c(168, 153)
)
```

To find out the classes of elements in `side_a`, we call `map()` with `side_a` as first argument and `class` as second argument.

```
map(side_a, class)
## $track
## [1] "integer"
##
## $title
## [1] "character"
##
## $writers
## [1] "list"
##
## $duration
## [1] "numeric"
```

These examples demonstrate that `map()` is simple to use, but its output can sometimes be difficult to read. In our first example, all list elements were single integers (the number of songs on each side). It would be more concise to combine these numbers into an integer vector. We can accomplish this task with the function `map_int()`.¹

```
titles_per_side <- map_int(titles, length)
titles_per_side
## [1] 2 1 2 1
class(titles_per_side)
## [1] "integer"
```

¹This example is mainly for didactic reasons. There is an easier alternative: `lengths(titles)`.

In our second example, the elements from `map(durations, sum)` were numeric (i.e. represented as double-precision floating-point numbers). We obtain the result as a numeric vector with `map_dbl()`.

```
duration_per_side <- map_dbl(durations, sum)
duration_per_side
## [1] 321 275 316 230
class(duration_per_side)
## [1] "numeric"
```

Similarly, the output of `map(side_a, class)` is a list whose elements were single character strings. We can coerce the result into a character vector with `map_chr()`.

```
map_chr(side_a, class)
##      track      title     writers   duration
##  "integer" "character"    "list"    "numeric"
```

In the output, R printed the names of the list elements (track, title, writers and duration) above the values to make the output more informative. The returned object is an example of a ‘named vector’ (i.e. a vector with a `names` attribute; you can see the attribute by piping the previous code chunk into the `attributes()` function).²

Apart from `map_int()`, `map_dbl()` and `map_chr()`, there is also a function `map_lgl()` which returns a logical vector if the applied function returns exactly one logical element per list element.

Sometimes, the default version of a function does not quite work for us in a `map()` context, and we would need to pass additional arguments. Here are some results from the 2016 Olympic women’s long jump qualifying round (see [22.10](#)). A foul jump is denoted by NA.

```
longjump <- list(
  spanovic = 6.87,
  mihambo = c(6.63, 6.82),
  reese = 6.78,
  bartoletta = c(6.44, 6.70, 6.61),
  klishina = c(6.64, NA, 6.64)
)
```

Qualification for the next round only depended on the length of the longest

²Other examples of named vectors are the return values of `quantile()` or `summary()` in section [3.4.3.3](#).

jump. Suppose we want to calculate the length of the longest jump for each athlete. By default, `max()` returns `NA` whenever one of its arguments is `NA`. For this reason, `map_dbl(list_with_na, max)` returns `NA` as its last value.

```
map_dbl(longjump, mean)
##   spanovic    mihambo      reese bartoletta  klishina
## 6.870000  6.725000  6.780000  6.583333        NA
```

However, we are interested in the maximum length of the non-`NA` jumps. We saw in section 11.1 that functions for summary statistics (e.g. `max()`) ignore missing values if we pass the argument `na.rm = TRUE`. If we call `max()` inside a function of the `map`-family, we can pass `na.rm = TRUE` as additional argument to the `map...()` function.

```
map_dbl(longjump, max, na.rm = TRUE)
##   spanovic    mihambo      reese bartoletta  klishina
##       6.87       6.82       6.78       6.70       6.64
```

This trick works for any function `f()` that is passed as second argument to `map...()`: we pass additional arguments to `f()` by appending them to the argument list of `map...()`.

23.2 Passing our own function as argument to `map()`

The examples in the previous section applied built-in functions (e.g. `class()` or `max()`) to the list items. However, sometimes we want to apply custom-made functions. Let us consider a simple example: suppose we want to convert the durations of the songs on ‘Magical Mystery Tour’ from seconds to minutes. In this case, we may want to write our own function that divides the duration in seconds by 60 to obtain the duration in minutes.³

```
convert_sec_to_min <- function(x) {
  x / 60
}
```

We can now pass the argument `convert_sec_to_min` to `map()`.

³It is possible to avoid writing our own function. We can instead use the `'/()'` function, which performs division, and pass 60 as an additional argument: `map(durations, '/`, 60)`.

```
map(durations, convert_sec_to_min)
## [[1]]
## [1] 2.80 2.55
##
## [[2]]
## [1] 4.583333
##
## [[3]]
## [1] 3.000000 2.266667
##
## [[4]]
## [1] 3.833333
```

For simple one-line functions, it is relatively much boilerplate code to define a function outside `map()`. Alternatively, we can use a so-called anonymous function. That is, we insert our own function directly as second argument into `map()` without giving the function a name (such as `convert_sec_to_min()`).

```
map(durations, function(x) x / 60)
## [[1]]
## [1] 2.80 2.55
##
## [[2]]
## [1] 4.583333
##
## [[3]]
## [1] 3.000000 2.266667
##
## [[4]]
## [1] 3.833333
```

We can shorten the second argument further. Instead of `function(x)` we can simply use a tilde ~ and replace the argument (`x` in this example) in the function body by a full stop (.) .

```
map(durations, ~ . / 60)
## [[1]]
## [1] 2.80 2.55
##
## [[2]]
## [1] 4.583333
##
## [[3]]
```

```
## [1] 3.000000 2.266667
##
## [[4]]
## [1] 3.833333
```

The return value is the same regardless of whether we use the named function `convert_sec_to_min()`, the anonymous function with the `function` keyword or the shortcut notation with `~` and `..`. We would not be able to convert the return value to a numeric vector with `map_dbl()` because some list elements are vectors of length > 1 .

```
map_dbl(durations, ~ . / 60)
## Error in `stop_bad_type()`:
## ! Result 1 must be a single double, not a double vector of length 2
```

However, we can ‘flatten’ the list into a vector with `flatten_dbl()`.

```
map(durations, ~ . / 60) |> flatten_dbl()
## [1] 2.800000 2.550000 4.583333 3.000000 2.266667 3.833333
```

Flattening the output has the disadvantage that it does not reveal the original hierarchy (e.g. that the first two numbers came from the same list element). It depends on the application whether flattening may be appropriate.

23.3 Summary and outlook

In this chapter, we only skimmed the surface of the possibilities offered by the `purrr` package. You can find more options in the cheat sheet available from RStudio’s help menu (“Help” → “Cheat Sheets” → “List manipulation with purrr”) or:

https://www.rstudio.org/links/purrr_cheat_sheet

If you are familiar with for-loops in R or another programming language, you are probably used to a lot more boilerplate code. Thanks to the `purrr` package, we can iterate over list items with succinct code, leading to more readable and maintainable R programs.

23.4 Just checking

First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix [A.16](#).

- (I) Consider the list `longjump` defined in section [23.1](#). How can we obtain the number of jumps per athlete **as an integer vector** (including foul jumps, represented as `NA`, in the count)?

(a) `map_int(longjump, length)`

(b) `map(longjump, length, as.integer)`

(c) `map(longjump, function(x) as.integer(length(x)))`

(d) `map_int(longjump, length())`

- (II) The numbers in the list `longjump` (defined in section [23.1](#)) are lengths in metres. How can we convert the numbers into inches? 1 metre equals 39.3701 inches.

(a) `map(longjump, function(x) ~ . * 39.3701)`

(b) `map(longjump, function(x) x * 39.3701)`

(c) `map(longjump, ~ x * 39.3701)`

(d) `map(longjump, function(x) . * 39.3701)`

Exercise: Relation between tibbles, data frames and lists

Prerequisite: chapter 22

Approximate duration: 30 minutes

Submission format: R Markdown

In this exercise, we look behind the scenes of tibbles, data frames and lists. We look at their attributes and find out what happens when we remove some of the attributes.

- (a) Create a tibble `musicians_tbl` that represents the following data:

```
musicians_tbl
## # A tibble: 3 x 3
##   name    band   instrument
##   <chr>   <chr>  <chr>
## 1 Keith  Stones  guitar
## 2 John   Beatles  guitar
## 3 Paul   Beatles  bass
```

- (b) Use `data.frame()` to create a variable `musicians_dfr` that represents the same data as `musicians_tbl`. Make sure that every column is a character vector.
- (c) Use `list()` to create a variable `musicians_list` with the following structure.

```
str(musicians_list)
## List of 3
## $ name      : chr [1:3] "Keith" "John" "Paul"
## $ band      : chr [1:3] "Stones" "Beatles" "Beatles"
## $ instrument: chr [1:3] "guitar" "guitar" "bass"
```

- (d) Apply the following functions to `musicians_tbl`, `musicians_dfr` and `musicians_list`:

- `class()`
- `is_tibble()`
- `is.data.frame()`
- `is.list()`.

Do you find any of the return values surprising?

- (e) Compare the return values of `attributes()` when applied to `musicians_tbl`, `musicians_dfr` and `musicians_list`.
- (f) Are there subsetting operations we can perform on `musicians_tbl` and `musicians_dfr`, but not on `musicians_list`?
- (g) Remove the `class` and `row.names` attributes from `musicians_tbl` and `musicians_dfr`. Then, use `identical()` to compare `musicians_tbl`, `musicians_dfr` and `musicians_list`. Briefly write what you conclude from the result.

Part VII

Network data



24

*Introduction to **igraph** and **tidygraph***

A network is a collection of *nodes* and *edges*. Metaphorically, we can think of nodes as points and edges as lines. Each edge connects two nodes. Here is a simple example.

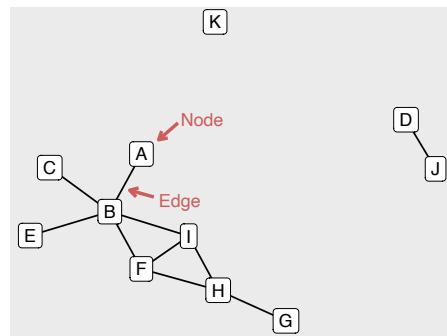


FIGURE 24.1: A small example network.

Nodes are also sometimes called vertices (singular: vertex), and networks are often called graphs. It is usually clear from the context whether the word ‘graph’ is meant in this sense, or whether ‘graph’ is used as a synonym for a plot that represents a data set, which may not necessarily be a network.

Networks are abundant in our daily life. Here are some examples:

- Relationships between humans can be represented by social networks. Nodes are individuals, and there is an edge between two nodes if there is a tie (e.g. friendship).
- In a metabolic network, nodes are metabolites, and edges represent their chemical interactions.
- Food webs are networks in which the nodes are biological species. There is an edge between two species if one species eats the other species.
- The Internet is a technological network in which nodes are computers, and edges are data connections.

The examples in this chapter are motivated by social networks, but the techniques can be easily transferred to other applications.

24.1 Importing network data

R can import network data from various file types. For the sake of brevity, we only cover the case of data given in two CSV files:

- one file with node identifiers and possibly additional node attributes (e.g. name or gender).
- another file with an edge list. In this context an ‘edge list’ is a technical term from the network literature, not to be confused with the R data structure that is also called a list. In its simplest form, an edge list consists of two elements per row that encode the nodes at both ends of the corresponding edge.

Here is the content of two CSV files that encode the example network in figure 24.1. The files are available at these URLs:

- https://michaelgastner.com/DAVisR_data/network_data/demo_nodes.csv
- https://michaelgastner.com/DAVisR_data/network_data/demo_edges.csv

| <code>demo_nodes.csv:</code> | <code>demo_edges.csv:</code> |
|------------------------------|------------------------------|
| <code>name</code> | <code>from,to</code> |
| A | A,B |
| B | B,C |
| C | B,E |
| D | B,F |
| E | B,I |
| F | D,J |
| G | F,H |
| H | F,I |
| I | G,H |
| J | H,I |
| K | |

The column for the node identifiers does not need to be called `name`, and the columns in the edge list do not need to be called `from` and `to`. The **tidygraph** package, which we are going to use in a moment, automatically changes the column names in the edge list to `from` and `to`, regardless of how they are called in the CSV. To avoid confusion, I recommend to use column names `from` and `to` in the CSV containing the edge list.

We import the CSV files with two calls to `read_csv()`:

```
library(tidyverse)
demo_nodes <- read_csv("networks_data/demo_nodes.csv")
demo_edges <- read_csv("networks_data/demo_edges.csv")
```

Afterwards, we combine nodes and edges into a network with `tbl_graph()` from the **tidygraph** package. The demo network in figure 24.1 is meant to be undirected. For example, an edge from A to B is equivalent to an edge from B to A. Later on, we encounter examples of directed networks. However, in the present example, we pass the argument `directed = FALSE` to `tbl_graph()`:

```
library(tidygraph)
demo_undirected_netw <- tbl_graph(
  nodes = demo_nodes,
  edges = demo_edges,
  directed = FALSE
)
demo_undirected_netw
## # A tbl_graph: 11 nodes and 10 edges
## #
## # An undirected simple graph with 3 components
## #
## # Node Data: 11 x 1 (active)
##   name
##   <chr>
## 1 A
## 2 B
## 3 C
## 4 D
## 5 E
## 6 F
## # ... with 5 more rows
## #
## # Edge Data: 10 x 2
##   from     to
##   <int> <int>
## 1 1       2
## 2 2       3
## 3 2       5
## # ... with 7 more rows
```

In the output of the edge data, **tidygraph** has replaced the node labels A, B, ..., K by integers 1, 2, ..., 11.

A network that is created by the `tbl_graph()` function belongs to the class `tbl_graph`, which is a subclass of `igraph`.

```
class(demo_undirected_netw)
## [1] "tbl_graph" "igraph"
```

The **igraph** class is a data structure used by the **igraph** package, which contains efficient implementations of many graph algorithms. However, **igraph** does not integrate seamlessly into a tidyverse workflow. For example, most of **igraph**'s functions are not pipe-friendly, and its data structures are substantially different from data frames or tibbles. The **tidygraph** package provides the **tbl_graph** class to give users the option to work with tibbles while running **igraph** functions behind the scenes. We can think of a **tbl_graph** object as a combination of two tibbles: one node tibble and one edge tibble. If we want to operate on one of these two member tibbles, we must first ‘activate’ it. Immediately after a **tbl_graph** is created, the active ‘context’ is the node tibble. We can find out with the **active()** function whether “nodes” or “edges” are the currently active context.

```
active(demo_undirected_netw)
## [1] "nodes"
```

If we want to operate on the edge tibble, we must first activate the edge context with the **activate()** function. For example, here is how we obtain the edge list from **demo_undirected_netw** as a tibble.

```
demo_undirected_netw |>
  activate(edges) |>
  as_tibble()
## # A tibble: 10 x 2
##       from     to
##   <int> <int>
## 1     1     2
## 2     2     3
## 3     3     5
## 4     4     6
## 5     5     9
## 6     6    10
## 7     7     8
## 8     8     9
## 9     9     8
## 10    10    9
```

In our next example, we work with a hypothetical directed network that shows email exchanges between people working for a company with offices in London, Los Angeles and Singapore. Unlike in **demo_edges.csv**, the column names **from** and **to** carry a directional meaning in the email network. The person in the **from**-column is the sender. The person in the **to**-column is the recipient. This example also demonstrates how we can add attributes to nodes and edges

by adding columns in the corresponding CSV (here: node locations and the number of emails sent over an edge). The files are available at these URLs:

- https://michaelgastner.com/DAVisR_data/network_data/email_nodes.csv
- https://michaelgastner.com/DAVisR_data/network_data/email_edges.csv
- email_nodes.csv:
 name,location
 Bernadette,Los Angeles
 Fatimah,Singapore
 Giovanni,London
 Henrik,London
 Hossam,London
 Jonathan,Los Angeles
 Lakshmi,Singapore
 Pauline,London
 Simon,Los Angeles
- email_edges.csv:
 from,to,emails
 Bernadette,Jonathan,9
 Bernadette,Simon,3
 Fatimah,Bernadette,2
 Fatimah,Giovanni,5
 Fatimah,Henrik,6
 Fatimah,Lakshmi,2
 Fatimah,Pauline,3
 Giovanni,Fatimah,8
 Giovanni,Henrik,1
 Giovanni,Pauline,2
 Henrik,Fatimah,4
 Henrik,Pauline,1
 Hossam,Hossam,1
 Jonathan,Simon,3
 Lakshmi,Fatimah,1
 Lakshmi,Pauline,1
 Pauline,Lakshmi,2
 Pauline,Henrik,3
 Simon,Jonathan,5

To import a directed network, we can set the argument `directed = TRUE` in `tbl_graph()`. However, because `TRUE` is the default argument, we can also simply omit the `directed`-argument.

```
email_nodes <- read_csv("networks_data/email_nodes.csv")
email_edges <- read_csv("networks_data/email_edges.csv")
email_directed_netw <- tbl_graph(
  nodes = email_nodes,
  edges = email_edges
)
```

24.2 Obtaining basic information about the network

We often want to know some basic properties about a network stored in a `tbl_graph`:

- Is the network directed?
- How many nodes are in the network?
- How many edges are in the network?

The **igraph** package contains functions to answer these questions.

24.2.1 Directedness

We can find out with `is_directed()` whether a network is directed or undirected.

```
library(igraph)
is_directed(email_directed_netw)
## [1] TRUE
is_directed(demo_undirected_netw)
## [1] FALSE
```

24.2.2 Graph order and graph size

The number of nodes in a network is called the *graph order*. The number of edges is called the *graph size*. We can find out the order with `gorder()` and the size with `gsize()`. These functions work for undirected and directed networks.

```
gorder(demo_undirected_netw)
## [1] 11
gsize(demo_undirected_netw)
## [1] 10
gorder(email_directed_netw)
## [1] 9
gsize(email_directed_netw)
## [1] 19
```

24.3 Summary and outlook

In this chapter, we learned how to import network data with the `tbl_graph()` function from the **tidyverse** package. Information about the network is read from two CSV files: a node list and an edge list. After creating a `tbl_graph`, we can extract basic information about the network with the functions `is_directed()`, `gorder()` and `gsize()` from the **igraph** package. There are many more functions that provide insights into the structure of a network. We learn about some of these functions in chapter 26. However, before em-

barking on quantitative network analysis, it is often a good idea to visualise the network. Therefore, network visualisation is the topic of the next chapter.



25

Visualising networks

Exploratory data analysis usually starts with visualising the data, and networks are no exception. In this chapter, we learn how to produce plots with the **ggraph** package, which uses syntax similar to **ggplot2** and adds useful functions for visualising networks.

25.1 Plotting with ggraph

Unlike for other kinds of data, the positions of nodes and edges in visual representations of networks are usually not determined by attributes of the data themselves. The only exception are spatial networks (e.g. roads or shipping routes, where we can use the longitude and latitude of the nodes to determine unique positions in the plot). If you are interested in working with geospatial networks, I recommend to take a look at the **sfnetworks** package. However, in this chapter, we focus on non-spatial networks. In this case, we are free to place nodes at convenient positions in the plot. Ideally, we would like to avoid that nodes overlap and that edges cross each other. It may not be possible to achieve these objectives, especially when working with large networks. However, for networks with up to a few hundred nodes, **ggraph** often produces good solutions with little effort.

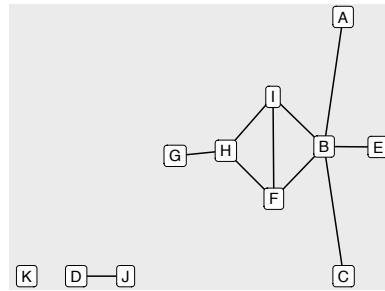
25.1.1 Choosing a network layout

Many algorithms have been developed to automate the layout of a network in a plot, see for example Tamassia (2013). The **ggraph** package contains implementations of some of these algorithms. The next few code chunks use the `ggraph()` function to illustrate **ggraph**'s layout options, using the demo network of section 24.1. Please do not worry about the syntax of `ggraph()` yet; we discuss it in detail in section 25.1.2.

There are three options for creating layouts with **ggraph**.

- (1) We leave it to the `ggraph()` function to pick the layout algorithm for us.

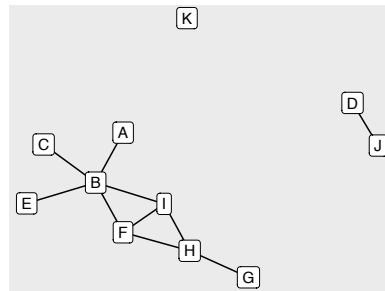
```
library(ggraph)
ggraph(demo_undirected_netw) +
  geom_edge_link() +
  geom_node_label(aes(label = name))
## Using `stress` as default layout
```



The automated message reveals that `ggraph()` chose the so-called stress-majorisation algorithm ([Gansner et al., 2005](#)).

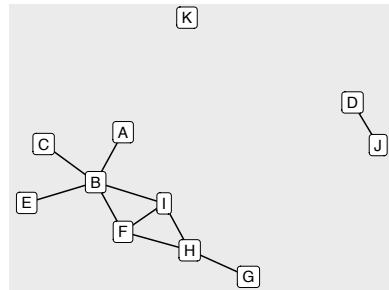
- (2) We pass an argument named `layout` to `ggraph()`. We can find the abbreviations of layout algorithms known to `ggraph` under `?layout_tbl_graph_igraph`. In the next example, we ask `ggraph()` to use the algorithm "kk" by [Kamada and Kawai \(1989\)](#).

```
ggraph(demo_undirected_netw, layout = "kk") +
  geom_edge_link() +
  geom_node_label(aes(label = name))
```



- (3) We create a `layout_ggraph` object with `create_layout()` before calling `ggraph()`. In this case, we should pass the `layout_ggraph` object as argument to `ggraph()`.

```
layout <- create_layout(demo_undirected_netw, layout = "kk")
ggraph(layout) +
  geom_edge_link() +
  geom_node_label(aes(label = name))
```

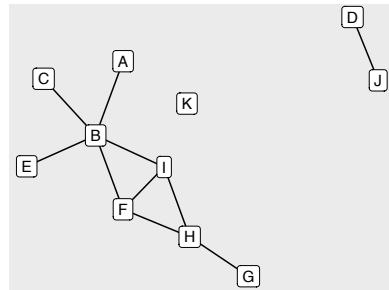


An advantage of using `create_layout()` is that we can inspect the generated `layout_ggraph` object.

| layout | x | y | name | .ggraph.orig_index | circular | .ggraph.index |
|--------|-------------|------------|------|--------------------|----------|---------------|
| ## 1 | -0.42837762 | 0.9882247 | A | 1 | FALSE | 1 |
| ## 2 | -0.75580277 | 0.1325053 | B | 2 | FALSE | 2 |
| ## 3 | -1.37675760 | 0.7859809 | C | 3 | FALSE | 3 |
| ## 4 | 2.30849252 | 1.4976341 | D | 4 | FALSE | 4 |
| ## 5 | -1.57844827 | -0.2228458 | E | 5 | FALSE | 5 |
| ## 6 | -0.42946496 | -0.7182009 | F | 6 | FALSE | 6 |
| ## 7 | 1.06681368 | -1.4893764 | G | 7 | FALSE | 7 |
| ## 8 | 0.36546418 | -1.0348485 | H | 8 | FALSE | 8 |
| ## 9 | 0.06088109 | -0.2309851 | I | 9 | FALSE | 9 |
| ## 10 | 2.61454815 | 0.7675067 | J | 10 | FALSE | 10 |
| ## 11 | 0.33361855 | 2.9502202 | K | 11 | FALSE | 11 |

We can also edit the `layout_ggraph` object. For example, we can move node K to a lower position:

```
layout$y[layout$name == "K"] <- 0.5
ggraph(layout) +
  geom_edge_link() +
  geom_node_label(aes(label = name))
```



Besides the layout algorithms that we encountered in the code chunks above (stress majorisation algorithm and Kamada-Kawai), there are many more algorithms that `ggraph()` and `create_layout()` can apply. Some of the options are presented at the following web sites:

- <https://www.data-imaginist.com/2017/ggraph-introduction-layouts/>
- <https://cran.r-project.org/web/packages/ggraph/vignettes/Layouts.html>

For small networks (i.e. less than ≈ 1000 nodes), the algorithm by [Fruchterman and Reingold \(1991\)](#) often produces attractive layouts. The corresponding argument passed to `ggraph()` or `create_layout()` is `layout = "fr"`. The Fruchterman-Reingold algorithm and many other network layout algorithms are based on random numbers; thus, we must use `set.seed()` to obtain reproducible output.

In practice, I recommend to first let `ggraph()` choose the layout for us while we start exploring the data. Afterwards, we should try a variety of possible layout algorithms, and, if the algorithm is based on random numbers, we should try a few different seeds for the random number generator. If we aim to produce a publication-quality visualisation of a small network, we should eventually tweak the layout manually until the result is polished and legible.

25.1.2 ggraph's geoms

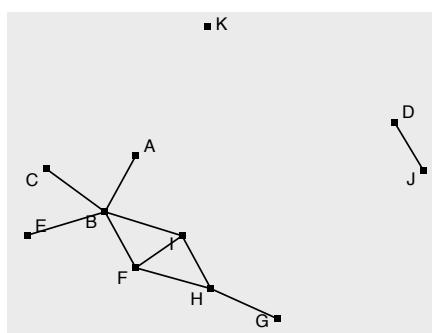
From the examples above, we already saw that the `ggraph` package uses syntax that is similar to `ggplot2`. Instead of `ggplot()`, we initiate the plot with `ggraph()`, and then we add geoms to show edges and nodes. Afterwards, we can adjust scales, legends and other details of the plot, if necessary, with conventional `ggplot2` functions (e.g. `coord_equal()` or `scale_fill_brewer()`).

Here are a few options for the edge and node geoms:

- The first example uses `geom_edge_link()` to draw straight lines between nodes. The function `geom_node_point()` draws nodes using point symbols. By default, the shapes are solid circles, but they can be changed with the `shape` argument (e.g. `shape = 15` for solid squares). It is also possible to use

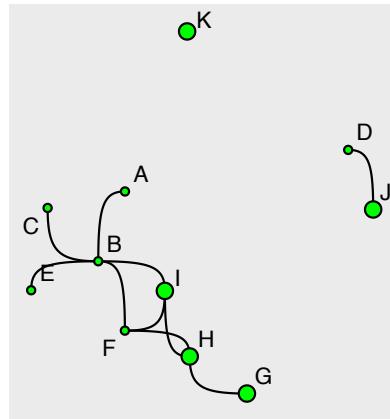
different shapes for different nodes, depending on their attributes, by using `shape` as an aesthetic (e.g. `aes(shape = col_name)`, where `col_name` is the name of a column in the node tibble of the `tbl_graph` object). The function `geom_node_text()` adds labels to nodes. The argument `repel = TRUE` moves the labels away from the nodes, but unfortunately not necessarily away from the edges (see <https://stackoverflow.com/questions/55464447/repel-text-from-edges-in-network>).

```
ggraph(demo_undirected_netw, layout = "kk") +
  geom_edge_link() +
  geom_node_point(shape = 15) +
  geom_node_text(aes(label = name), repel = TRUE)
```



- The next example uses `geom_edge_bend()` to draw curved edges. The argument named `strength` determines how much the curves bend. Its value can be chosen between 0 and 1. A strength of 0 results in a straight line between nodes, whereas 1 produces strong curvature. The function `geom_node_circle()` draws nodes as circles. The `r` aesthetic determines the radii. When we use `geom_node_circle()`, we usually also use `coord_equal()` so that circles do not become elongated ellipses because of different length scales along the x-axis and y-axis.

```
ggraph(demo_undirected_netw, layout = "kk") +
  geom_edge_bend(strength = 0.8) +
  geom_node_circle(
    aes(r = if_else(name < "G", 0.05, 0.1)),
    fill = "green"
  ) +
  geom_node_text(aes(label = name), repel = TRUE) +
  coord_equal()
```



- The final example is more elaborate. First, a layout is created with the Fruchterman-Reingold algorithm. Because this algorithm relies on random numbers, we must use `set.seed()` to make the layout reproducible. In this example, the node with the name "Hossam" is moved to the bottom right. The function `geom_edge_fan()` draws parallel edges as non-overlapping curves, which is advantageous in directed networks in which edges between two nodes point in opposite directions. The `end_cap()` aesthetic with a value of `label_rect(node2.name)` ensures that the end points of the edges attach exactly on the boundaries of the node labels. Passing the argument named `arrow` in `geom_edge_fan()` allows specifying the shapes of arrow heads. The argument named `strength` determines the width of the fans; values greater than 1 create wider fans. The argument named `angle_calc = "along"` aligns the text of the edge labels along the directions of the edges. Finally, `label_dodge` shifts the label away from the edge.

For loops in the network (e.g. Hossam's email to himself), we need a special geom called `geom_edge_loop()`.

```
set.seed(-688508752)
email_layout <- create_layout(email_directed_netw, layout = "fr")
email_layout$x[email_layout$name == "Hossam"] <- min(email_layout$x)
email_layout$y[email_layout$name == "Hossam"] <- min(email_layout$y)
ggraph(email_layout) +
  geom_edge_fan(
    aes(
      label = emails,
      end_cap = label_rect(node2.name)
    ),
    arrow = arrow(length = unit(2, "mm")),
    strength = 1.5,
    angle_calc = "along",
    loop = TRUE
  )
```

```

    label_dodge = unit(2, "mm")
) +
geom_edge_loop(
  aes(label = emails, end_cap = label_rect(node2.name)),
  arrow = arrow(length = unit(2, "mm")),
  angle_calc = "along",
  label_dodge = unit(2, "mm")
) +
geom_node_label(aes(label = name, colour = location)) +
scale_colour_brewer(name = "Location", palette = "Dark2") +
theme(legend.position = c(1, 1), legend.justification = c(1, 1))

```

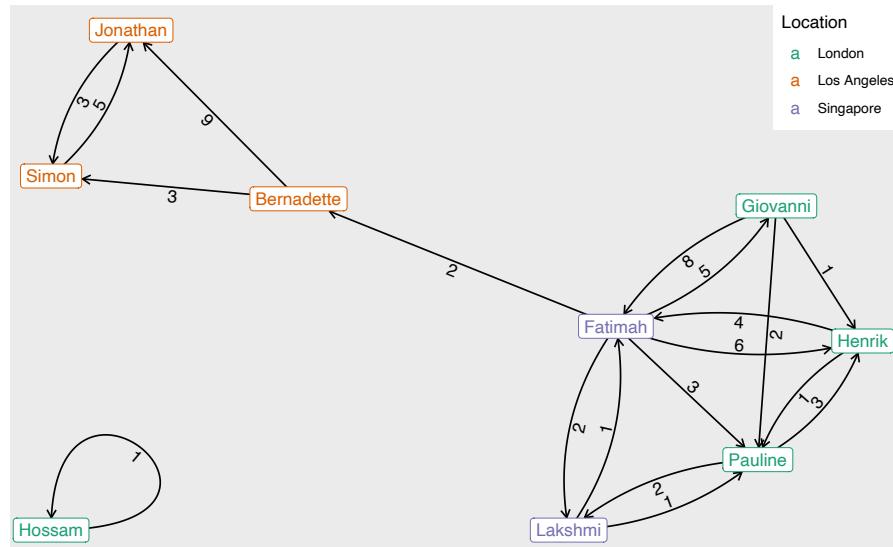


FIGURE 25.1: Example of a directed network with edge weights.

25.2 Summary and outlook

The abundance of **ggraph**'s network layout algorithms and geoms appears daunting at first glance, and this chapter only showed a small portion of the available options. The best way to learn **ggraph** is to work on examples and read the **ggraph** package documentation at <https://cran.r-project.org/web/packages/ggraph/ggraph.pdf>.

After exploring network data with a plot, the next task is to quantify the

network's structure. We learn essential techniques for network analysis in the next chapter.

26

Essentials of network analysis

In this chapter, we learn essential skills for extracting quantitative information from network data. First, we learn in section 26.1 that many **dplyr** verbs can be applied to **tbl_graph** objects. Then, we learn in section 26.2 some standard techniques using **tidygraph**'s `convert()` function so that the data become easier to analyse and interpret. Section 26.3 shows how to extract information about the positions of nodes in a network. Afterwards, we learn in section 26.4 how to characterise the overall structure of a network. Finally, section 26.5 shows techniques to detect whether and how a network is composed of modules known as communities.

26.1 dplyr syntax for **tbl_graph** objects

We can apply many **dplyr** verbs (e.g. `mutate()` and `rename()`) to **tbl_graph** objects. The verb is applied to the tibble corresponding to the active context (i.e. either "nodes" or "edges"). The next example uses the email network of section 24.1 and shows how to use the **dplyr** verb `mutate()` in a node-context and `rename()` in an edge-context.

```
library(tidygraph)
email_directed_netw |>
  activate(nodes) |>
  mutate(
    continent = case_when(
      location == "London" ~ "Europe",
      location == "Los Angeles" ~ "North America",
      location == "Singapore" ~ "Asia"
    )
  ) |>
  activate(edges) |>
  rename(electronic_mail = emails)
## # A tbl_graph: 9 nodes and 19 edges
## #
```

```

## # A directed multigraph with 2 components
## #
## # Edge Data: 19 x 3 (active)
##   from      to electronic_mail
##   <int> <int>      <dbl>
## 1     1       6         9
## 2     1       9         3
## 3     2       1         2
## 4     2       3         5
## 5     2       4         6
## 6     2       7         2
## # ... with 13 more rows
## #
## # Node Data: 9 x 3
##   name    location   continent
##   <chr>    <chr>     <chr>
## 1 Bernadette Los Angeles North America
## 2 Fatimah  Singapore   Asia
## 3 Giovanni London     Europe
## # ... with 6 more rows

```

Sometimes we want to refer to information that is contained in the node data while making changes to the edge data. In the next example, we want to add the names of the senders and recipients as two new columns to the edge tibble, but the names are in the node tibble. In such situations, we first activate the edges and then access the node data with the `.N()` function.

```

email_directed_netw <-
  email_directed_netw |>
  activate(edges) |>
  mutate(
    from_id = .N()$name[from],
    to_id = .N()$name[to]
  )
email_directed_netw
## # A tbl_graph: 9 nodes and 19 edges
## #
## # A directed multigraph with 2 components
## #
## # Edge Data: 19 x 5 (active)
##   from      to emails from_id   to_id
##   <int> <int> <dbl> <chr>     <chr>
## 1     1       6       9 Bernadette Jonathan

```

```

## 2     1     9      3 Bernadette Simon
## 3     2     1      2 Fatimah   Bernadette
## 4     2     3      5 Fatimah   Giovanni
## 5     2     4      6 Fatimah   Henrik
## 6     2     7      2 Fatimah   Lakshmi
## # ... with 13 more rows
## #
## # Node Data: 9 x 2
##   name      location
##   <chr>    <chr>
## 1 Bernadette Los Angeles
## 2 Fatimah   Singapore
## 3 Giovanni  London
## # ... with 6 more rows

```

If we remove a node with `filter()`, all edges connected to that node are automatically removed too. For example, in the next code chunk, we filter out Fatimah.

```

email_directed_netw_without_fatimah <-
  email_directed_netw |>
  activate(nodes) |>
  filter(name != "Fatimah")

```

We plot this graph in figure 26.1. (Try it yourself!) The plot confirms that Fatimah is no longer part of the network.

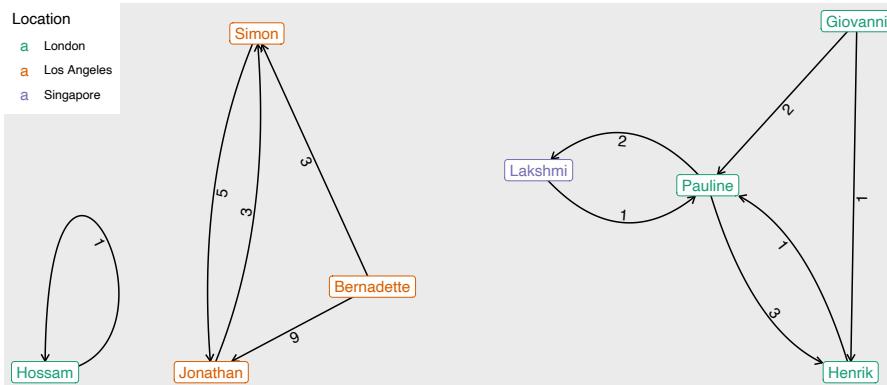


FIGURE 26.1: Network after filtering out Fatimah.

One of the few *dplyr* verbs that do not work on a *tbl_graph* is `summarise()`. This feature makes sense. For example, it would be unclear how the node

data is supposed to be changed if we carry out a summary of the edge data. However, there are several graph manipulations with which **tidygraph** can effectively summarise network data. We can

- turn a directed network into an undirected network.
- collapse parallel edges and remove loops.
- contract a network by combining multiple nodes into a single node.

These tasks are accomplished with a new verb: `convert()`. The next section gives examples of how to use `convert()`.

26.2 Transforming networks with `convert()`

26.2.1 Turning a directed network into an undirected network

We can remove the edge directions with `convert(to_undirected)`.

```
email_undirected_netw <-
  email_directed_netw |>
  convert(to_undirected)
```

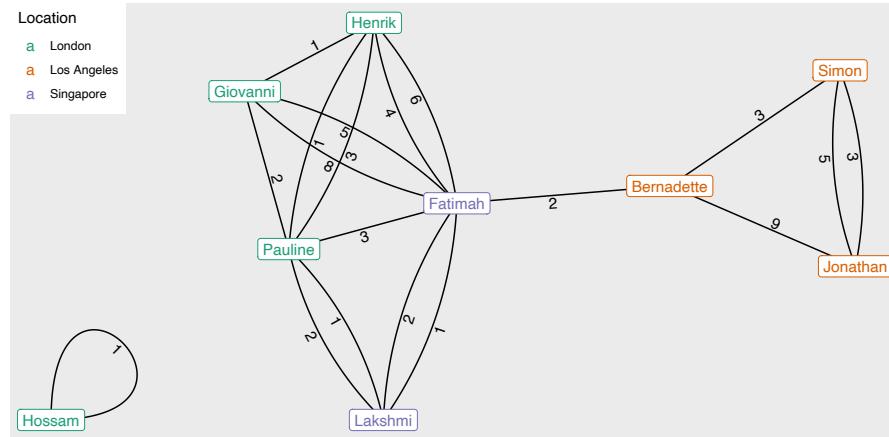


FIGURE 26.2: Undirected version of the email network.

When we compare the edge lists of the directed and undirected versions of this network, we notice that, in the undirected version, the number in the `from`-column is never larger than the number in the `to`-column, even if this was the case in the directed version. The column `.tidygraph_edge_index` contains the indices of the corresponding edges in the original (i.e. directed) network.

In this example, the numbers in `.tidygraph_edge_index` are simply the row indices. For other applications of `convert()`, `.tidygraph_edge_index` can be more interesting, as we see shortly.

```
email_undirected_netw |>
  activate(edges) |>
  as_tibble()

## # A tibble: 19 x 6
##   from      to emails from_id    to_id    .tidygraph_edge_index
##   <int> <int> <dbl> <chr>     <chr>                <int>
## 1 1       1       6  Bernadette Jonathan            1
## 2 2       1       9  Bernadette Simon               2
## 3 3       1       2  Fatimah    Bernadette            3
## 4 4       2       3  Fatimah    Giovanni             4
## 5 5       2       4  Fatimah    Henrik              5
## 6 6       2       7  Fatimah    Lakshmi             6
## 7 7       2       8  Fatimah    Pauline             7
## 8 8       2       3  Giovanni   Fatimah             8
## 9 9       3       4  Giovanni   Henrik              9
## 10 10      3       8  Giovanni   Pauline            10
## 11 11      2       4  Henrik     Fatimah            11
## 12 12      4       8  Henrik     Pauline            12
## 13 13      5       5  Hossam     Hossam              13
## 14 14      6       9  Jonathan   Simon               14
## 15 15      2       7  Lakshmi   Fatimah            15
## 16 16      7       8  Lakshmi   Pauline            16
## 17 17      7       8  Pauline   Lakshmi             17
## 18 18      4       8  Pauline   Henrik              18
## 19 19      6       9  Simon     Jonathan            19
```

26.2.2 Collapsing parallel edges and removing loops with `to_simple()`

Reading through the edge list of `email_undirected_netw`, we notice that there are two edges from node 2 to node 3. They correspond to the two edges between Fatimah and Giovanni in figure 26.2. Such parallel edges (also known as multi-edges) can sometimes cause problems during data analysis; thus, we would like to have a method to collapse parallel edges into a single edge. Another feature that is often undesirable are ‘loops’, which are defined as edges from a node to itself (e.g. the loop caused by an email that Hossam sent to himself). A network without parallel edges and loops is called ‘simple’. We can check whether a network is simple with `is_simple()` in the **igraph** package.

```
library(igraph)
is_simple(email_undirected_netw)
## [1] FALSE
```

We convert a network into a simple network with `convert(to_simple)`.

```
email_simple_netw <-
  email_undirected_netw |>
  convert(to_simple)
email_simple_edges <-
  email_simple_netw |>
  activate(edges) |>
  as_tibble()
email_simple_edges
## # A tibble: 12 x 4
##       from     to .tidygraph_edge_index .orig_data
##   <int> <int> <list>           <list>
## 1     1     2 <int [1]>      <tibble [1 x 5]>
## 2     1     6 <int [1]>      <tibble [1 x 5]>
## 3     1     9 <int [1]>      <tibble [1 x 5]>
## 4     2     3 <int [2]>      <tibble [2 x 5]>
## 5     2     4 <int [2]>      <tibble [2 x 5]>
## 6     2     7 <int [2]>      <tibble [2 x 5]>
## 7     2     8 <int [1]>      <tibble [1 x 5]>
## 8     3     4 <int [1]>      <tibble [1 x 5]>
## 9     3     8 <int [1]>      <tibble [1 x 5]>
## 10    4     8 <int [2]>      <tibble [2 x 5]>
## 11    6     9 <int [2]>      <tibble [2 x 5]>
## 12    7     8 <int [2]>      <tibble [2 x 5]>
```

Besides the usual columns ‘from’ and ‘to’, the edge list of `email_simple_netw` has two additional columns: `.tidygraph_edge_index` and `.orig_data`. The list column `.tidygraph_edge_index` contains the indices of the corresponding edges in the original network (i.e. the network before applying `convert(to_simple)`). For example, we know from our earlier output that the edges between nodes 2 (Fatimah) and 3 (Giovanni) had the indices 4 and 8 in `email_undirected_netw`. We can confirm this statement as follows.

```
email_simple_edges |>
  filter(from == 2, to == 3) |>
  pull(.tidygraph_edge_index)
## [[1]]
## [1] 4 8
```

The attributes of the original edges are stored in the list column `orig.data`.

```
email_simple_edges |>
  filter(from == 2, to == 3) |>
  pull(.orig_data)
## # [1]
## # A tibble: 2 x 5
##   from   to emails from_id to_id
##   <int> <int> <dbl> <chr>   <chr>
## 1     2     3     5 Fatimah Giovanni
## 2     2     3     8 Giovanni Fatimah
```

These data allow us to assign a weight to each undirected edge that is equal to the sum of the emails in both directions.

```
email_simple_netw <- 
  email_simple_netw |>
  activate(edges) |>
  mutate(weight = map_dbl(.orig_data, ~ sum(pull(., emails))))
```

Here is a plot of the network, which shows the cumulative weights as labels along the edges. Note that, although we use `geom_edge_loop()`, the plot does not contain the loop from Hossam to himself any longer because otherwise the network would not be simple.

```
ggraph(email_simple_netw, layout = "stress") +
  geom_edge_link(
    aes(label = weight),
    angle_calc = "along",
    label_dodge = unit(2, "mm"),
    label_push = unit(-2, "mm")
  ) +
  geom_edge_loop(
    aes(label = weight),
    angle_calc = "along",
    label_dodge = unit(2, "mm")
  ) +
  geom_node_label(aes(label = name, colour = location)) +
  scale_colour_brewer(name = "Location", palette = "Dark2") +
  theme(legend.position = c(0, 1), legend.justification = c(0, 1))
```

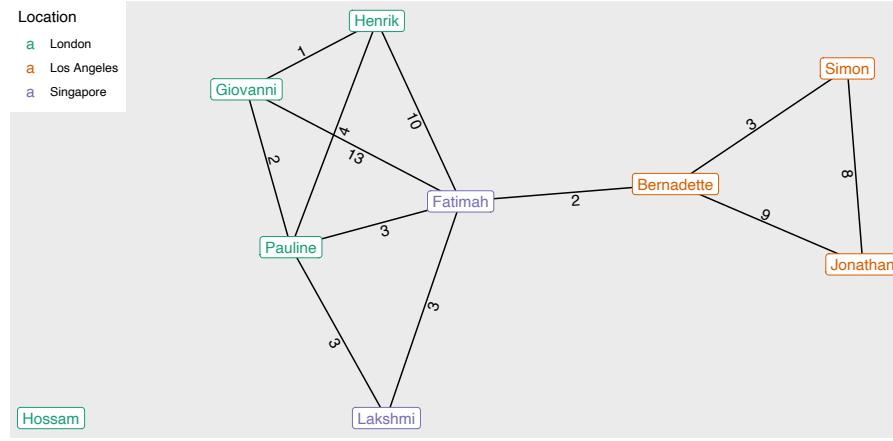


FIGURE 26.3: Undirected email network after collapsing parallel edges and removing loops.

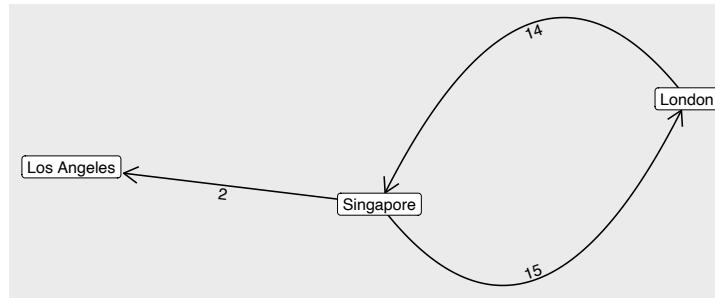
26.2.3 Contracting a network

When working with large networks, it is often uninformative to show all nodes and all edges because such plots tend to look like fuzzy hairballs that do not reveal how the nodes are connected. A better strategy is to aggregate nodes into larger groups. For example, in our email network, we may wish to combine all staff working in the same location into a single node, removing all internal emails. In principle, the strategy to use is `convert(to_contracted)`. Unfortunately, there is a bug as of 2020-12-23; thus we temporarily need to resort to a workaround (see <https://github.com/thomasp85/tidygraph/issues/65>). You do not need to understand the details.

```
factor_location <-
  email_directed_netw |>
  activate(nodes) |>
  pull(location) |>
  factor()

email_contracted_netw <-
  email_directed_netw |>
  activate(nodes) |>
  mutate(level_as_integer = as.integer(factor_location)) |>
  convert(to_contracted, level_as_integer) |>
  mutate(location = levels(factor_location)) |>
  activate(edges) |>
  mutate(emails = map_dbl(.orig_data, ~ sum(pull(., emails)))) |
ggraph(email_contracted_netw, layout = "stress") +
  geom_edge_fan()
```

```
aes(label = emails, end_cap = label_rect(node2.location)),
arrow = arrow(length = unit(4, "mm")),
angle_calc = "along",
label_dodge = unit(2, "mm")
) +
geom_node_label(aes(label = location)) +
scale_x_continuous(expand = expansion(c(0.1, 0.06)))
```



We can tell from this plot, that the staff in London and Singapore send emails to each other frequently, whereas Los Angeles did not contact the other two locations.

26.3 Extracting information about nodes

26.3.1 Finding nodes and edges that are connected to a node

We can find out whether an edge is connected to a node or a set of nodes with `edge_is_to()`. The answer may depend on the direction of the edges. Applied to the directed network in figure 25.1, there is an edge from Giovanni to Henrik, but not from Henrik to Giovanni. There is an analogous function `edge_is_from()` if we want to find edges in the opposite direction. These functions can be passed as arguments to `mutate()`.

```
email_directed_netw |>
  activate(edges) |>
  filter(edge_is_to(.N$name == "Fatimah")) |>
  select(ends_with("_id")) |>
  as_tibble()
## # A tibble: 3 x 4
##   from    to from_id to_id
##   <int> <int> <chr>   <chr>
```

```
## 1      3    2 Giovanni Fatimah
## 2      4    2 Henrik    Fatimah
## 3      7    2 Lakshmi  Fatimah
```

From the output, we can see that there is still a `from`-column and a `to`-column in the `tbl_graph` although these columns were excluded from the argument of `select()`. The **tidygraph** package demands that these two columns remain part of a `tbl_graph` so that it is always possible to reconstruct the network.

If we want to find all edges that are incident to a node (i.e. either pointing towards or away from a node), we use `edge_is_incident()`.

```
email_directed_netw |>
  activate(edges) |>
  filter(edge_is_incident(.N()$name == "Fatimah")) |>
  select(ends_with("_id")) |>
  as_tibble()

## # A tibble: 8 x 4
##       from     to from_id   to_id
##   <int> <int> <chr>     <chr>
## 1      2      1 Fatimah Bernadette
## 2      2      3 Fatimah Giovanni
## 3      2      4 Fatimah Henrik
## 4      2      7 Fatimah Lakshmi
## 5      2      8 Fatimah Pauline
## 6      3      2 Giovanni Fatimah
## 7      4      2 Henrik    Fatimah
## 8      7      2 Lakshmi  Fatimah
```

From this output, we can, in principle, find out whether a given node is connected to Fatimah. However, there is a more direct approach. The condition `node_is_adjacent(name == "Fatimah")` is TRUE if and only if a node is connected to Fatimah.

```
email_directed_netw |>
  activate(nodes) |>
  filter(node_is_adjacent(name == "Fatimah")) |>
  pull(name)

## [1] "Bernadette" "Fatimah"      "Giovanni"     "Henrik"       "Lakshmi"
## [6] "Pauline"
```

By default, **tidygraph** considers each node to be adjacent to itself. This is why Fatimah is part of the output above although there is no edge from Fatimah to herself. The way **tidygraph** defines adjacency is unusual in this respect (see

<https://github.com/thomasp85/tidygraph/issues/133>). We can retrieve the more conventional definition by passing the additional argument `include_to = FALSE` to `node_is_adjacent()`.

```
email_directed_netw |>
  activate(nodes) |>
  filter(node_is_adjacent(name == "Fatimah", include_to = FALSE)) |>
  pull(name)
## [1] "Bernadette" "Giovanni"    "Henrik"      "Lakshmi"     "Pauline"
```

This output no longer includes Fatimah, but it still includes all nodes that are linked to Fatimah, regardless of whether the link is pointing towards or away from her. If we want to include only nodes that sent emails to Fatimah, we add the argument `mode = "in"` to `node_is_adjacent()`.

```
email_directed_netw |>
  activate(nodes) |>
  filter(node_is_adjacent(
    name == "Fatimah",
    mode = "in",
    include_to = FALSE
  )) |>
  pull(name)
## [1] "Giovanni" "Henrik"    "Lakshmi"
```

There is an analogous argument `mode = "out"` that we can pass to `node_is_adjacent()` if we want to include only those nodes that received emails from Fatimah.

26.3.2 Degree and strength of a node

Often we want to quantify the importance of a node in a network. In this section, we learn how to compute two measures of importance that are commonly used in network analysis: the *degree* and the *strength*. We will learn about other measures of importance in section 26.3.5.

In an undirected network, the degree is the number of incident edges. For example, in the network shown in figure 24.1, the degree of node A is 1, and the degree of node B is 5. Therefore, judging by the degree, B is a more important node than A. The `tidygraph` package calculates the degree with `centrality_degree()`. The next code chunk shows how to apply `centrality_degree()` to the network shown in figure 24.1.

```
demo_undirected_netw |>
  activate(nodes) |>
  mutate(degree = centrality_degree()) |>
  as_tibble()
## # A tibble: 11 x 2
##   name   degree
##   <chr>  <dbl>
## 1 A       1
## 2 B       5
## 3 C       1
## 4 D       1
## 5 E       1
## 6 F       3
## 7 G       1
## 8 H       3
## 9 I       3
## 10 J      1
## 11 K      0
```

In directed networks, we must distinguish between three different types of degrees.

- In-degree: number of edges pointing towards the node.
- Out-degree: number of edges pointing away from the node.
- Degree: sum of in-degree and out-degree.

For example, in the network shown in figure 25.1, Pauline has in-degree 4, out-degree 2 and degree 6. We can find these values by passing the arguments `mode = "in"`, `mode = "out"` and `mode = "all"` to `centrality_degree()`.

```
email_directed_netw |>
  activate(nodes) |>
  mutate(
    in_degree = centrality_degree(mode = "in"),
    out_degree = centrality_degree(mode = "out"),
    degree = centrality_degree(mode = "all")
  ) |>
  as_tibble()
## # A tibble: 9 x 5
##   name     location   in_degree  out_degree  degree
##   <chr>    <chr>        <dbl>        <dbl>    <dbl>
## 1 Bernadette Los Angeles      1          2        3
## 2 Fatimah   Singapore       3          5        8
## 3 Giovanni  London         1          3        4
```

```
## 4 Henrik    London      3      2      5
## 5 Hossam   London      1      1      2
## 6 Jonathan Los Angeles 2      1      3
## 7 Lakshmi  Singapore  2      2      4
## 8 Pauline  London      4      2      6
## 9 Simon    Los Angeles 2      1      3
```

When measuring node importance in terms of the degree, we attribute equal importance to every edge, but sometimes it is more appropriate to give different weights to different edges. In the email network shown in figure 25.1, it makes sense to weigh the importance of an edge by the number of emails sent across this edge. A measure of node importance in such a weighted network is the *strength*, defined as the sum of the weights of all edges incident to a node. We calculate node strengths by passing an argument named `weights` to `centrality_degree()`. If the network is directed, we must distinguish between in-strength, out-strength and strength (i.e. the sum of in-strength and out-strength).

```
email_directed_netw |>
  activate(nodes) |>
  mutate(
    in_strength = centrality_degree(weights = emails, mode = "in"),
    out_strength = centrality_degree(weights = emails, mode = "out"),
    strength = centrality_degree(weights = emails, mode = "all")
  ) |>
  as_tibble()
## # A tibble: 9 x 5
##   name      location  in_strength out_strength strength
##   <chr>     <chr>          <dbl>        <dbl>      <dbl>
## 1 Bernadette Los Angeles     2           12         14
## 2 Fatimah   Singapore      13          18         31
## 3 Giovanni  London         5           11         16
## 4 Henrik    London         10          5          15
## 5 Hossam    London         1           1          2
## 6 Jonathan  Los Angeles   14          3          17
## 7 Lakshmi   Singapore     4           2          6
## 8 Pauline   London         7           5          12
## 9 Simon     Los Angeles   6           5          11
```

26.3.3 Geodesic distances

The *geodesic distance* (or *distance* for short) from a node x to a node y is the minimum number of edges one must traverse to get from x to y . Sometimes,

the term “shortest distance” is used; however, the combination of “shortest” and “distance” is tautological because distances are always the minimum over all possible path lengths. When calculating the distance in a directed network, it is important to remember that the only permitted steps are those along the direction of the edge. For example, in the directed email network in figure 25.1, the distance from Giovanni to Pauline is 1 because there is an edge from Giovanni to Pauline. However, the distance from Pauline to Giovanni is 3 because the shortest paths traverse three edges (either Pauline → Lakshmi → Fatimah → Giovanni or Pauline → Henrik → Fatimah → Giovanni).

We calculate the distance from each node to a given destination, which can be either a single node or a set of nodes, with `node_distance_to()`. Similarly, we calculate the distance from a given origin with `node_distance_from()`. If there is no path between two nodes, `tidygraph` assigns a distance `Inf` to the corresponding pair of nodes.

```
email_directed_netw |>
  activate(nodes) |>
  mutate(
    distance_to_pauline = node_distance_to(name == "Pauline"),
    distance_from_pauline = node_distance_from(name == "Pauline")
  ) |>
  as_tibble()
## # A tibble: 9 x 4
##   name      location  distance_to_pauline distance_from_pauline
##   <chr>     <chr>                <dbl>                  <dbl>
## 1 Bernadette Los Angeles            Inf                   3
## 2 Fatimah   Singapore              1                    2
## 3 Giovanni  London                 1                    3
## 4 Henrik    London                 1                    1
## 5 Hossam    London                Inf                   Inf
## 6 Jonathan  Los Angeles            Inf                   4
## 7 Lakshmi   Singapore              1                    1
## 8 Pauline   London                 0                    0
## 9 Simon     Los Angeles            Inf                   4
```

With `local_members()`, we can find out which nodes are within a given distance of a focal node. The next example shows how to find all nodes that are no more than two steps away from Lakshmi:

```
email_directed_netw |>
  activate(nodes) |>
  mutate(
    neighbours = map(local_members(order = 2), ~ name[.])
```

```
) |>
filter(name == "Lakshmi") |>
pull(neighbours)
## [[1]]
## [1] "Lakshmi"    "Fatimah"     "Pauline"      "Bernadette" "Giovanni"
## [6] "Henrik"
```

The *diameter* of a network is the longest finite distance between any pair of nodes. Thus, the diameter is the maximum of the shortest path lengths between any two nodes. We calculate the diameter with the function `diameter()`.

```
diameter(email_directed_netw)
## [1] 4
```

26.3.4 Shortest paths

We determine a shortest path between two nodes with `convert(to_shortest_path)`.

```
pauline_to_simon <-
email_directed_netw |>
convert(to_shortest_path, name == "Pauline", name == "Simon")
```

Here are the nodes on a shortest path from Pauline to Simon:

```
pauline_to_simon |>
activate(nodes) |>
as_tibble()
## # A tibble: 5 x 3
##   name      location  .tidygraph_node_index
##   <chr>     <chr>                <int>
## 1 Bernadette Los Angeles           1
## 2 Fatimah    Singapore            2
## 3 Henrik     London               4
## 4 Pauline    London               8
## 5 Simon      Los Angeles          9
```

And here are the edges on the shortest path.

```
pauline_to_simon |>
activate(edges) |>
```

```
as_tibble()
## # A tibble: 4 x 6
##   from      to emails from_id    to_id    .tidygraph_edge_index
##   <int>  <int>  <dbl> <chr>     <chr>                <int>
## 1     1      5     3 Bernadette Simon                  2
## 2     2      1     2 Fatimah    Bernadette               3
## 3     3      2     4 Henrik     Fatimah                11
## 4     4      3     3 Pauline    Henrik                 18
```

The output only shows which nodes and edges are on the shortest path, but not the sequence in which they appear along the path (i.e. Pauline → Henrik → Fatimah → Bernadette → Simon). If we need this information, we must resort to the **igraph** function `shortest_paths()`.

```
shortest_paths(email_directed_netw, "Pauline", "Simon")$vpath
## [[1]]
## + 5/9 vertices, named, from 5273a59:
## [1] Pauline      Henrik       Fatimah      Bernadette Simon
```

Inspecting figure 25.1, we notice that there is another equally short path (Pauline → Lakshmi → Fatimah → Bernadette → Simon), which is not part of the output above. We can obtain both shortest paths in this example by using the **igraph** function `all_shortest_paths()`.

```
all_shortest_paths(email_directed_netw, "Pauline", "Simon")$res
## [[1]]
## + 5/9 vertices, named, from 5273a59:
## [1] Pauline      Lakshmi      Fatimah      Bernadette Simon
##
## [[2]]
## + 5/9 vertices, named, from 5273a59:
## [1] Pauline      Henrik      Fatimah      Bernadette Simon
```

So far, we assumed that each traversed edge contributes with equal weight towards the path length. If we want to assign different weights (e.g. the number of emails sent) to different edges, we can calculate a weighted shortest path by passing a `weights` argument to `convert()`. For example, the next code chunk returns information about the path from Pauline to Simon that has the smallest sum of emails sent along the edges in the path.

```
email_directed_netw |>
  convert(
    to_shortest_path,
```

```

name == "Pauline",
name == "Simon",
weights = emails
)
## # A tbl_graph: 5 nodes and 4 edges
## #
## # A rooted tree
## #
## # Edge Data: 4 x 6 (active)
##   from      to emails from_id    to_id    .tidygraph_edge_index
##   <int> <int> <dbl> <chr>     <chr>             <int>
## 1     1      5     3 Bernadette Simon                2
## 2     2      1     2 Fatimah    Bernadette            3
## 3     3      2     1 Lakshmi    Fatimah              15
## 4     4      3     2 Pauline    Lakshmi              17
## #
## # Node Data: 5 x 3
##   name      location    .tidygraph_node_index
##   <chr>      <chr>             <int>
## 1 Bernadette Los Angeles                  1
## 2 Fatimah    Singapore                 2
## 3 Lakshmi    Singapore              7
## # ... with 2 more rows

```

If we want to highlight the shortest path on a network plot (e.g. by marking edges on the shortest path with a darker line), we must keep the original edges in the `tbl_graph`. We can achieve this objective by replacing `convert()` with `morph()`. We can apply the same data transformations on a morphed and on a converted network. However, the result of `morph()` can still be ‘unmorphed’ so that the network returns to its original topology (i.e. with the original nodes and edges). The unmorphed network keeps the information that was added during the morphed stage (e.g. whether an edge is on the shortest path). If a node or edge was temporarily not present during the morphed stage, `tidygraph` uses NA to indicate that no information is available. The next code chunk shows how to highlight the shortest weighted path from Pauline to Simon.

```

pauline_to_simon_weighted <-
  email_directed_netw |>
  morph(
    to_shortest_path,
    name == "Pauline",
    name == "Simon",

```

```
    weight = emails
) |>
activate(nodes) |>
mutate(node_from_pauline_to_simon = TRUE) |>
activate(edges) |>
mutate(edge_from_pauline_to_simon = TRUE) |>
unmorph() |>
activate(nodes) |>
mutate(node_from_pauline_to_simon = !is.na(node_from_pauline_to_simon)) |>
activate(edges) |>
mutate(edge_from_pauline_to_simon = !is.na(edge_from_pauline_to_simon))
ggraph(
  pauline_to_simon_weighted,
  layout = "stress"
) +
  geom_edge_fan(
    aes(
      label = emails,
      end_cap = label_rect(node2.name),
      alpha = edge_from_pauline_to_simon
    ),
    arrow = arrow(length = unit(2, "mm")),
    strength = 1.5,
    angle_calc = "along",
    label_dodge = unit(2, "mm"),
    label_push = unit(-2, "mm"),
    show.legend = FALSE
  ) +
  geom_edge_loop(
    aes(
      label = emails,
      end_cap = label_rect(node2.name),
      alpha = edge_from_pauline_to_simon
    ),
    arrow = arrow(length = unit(2, "mm")),
    angle_calc = "along",
    label_dodge = unit(2, "mm"),
    show.legend = FALSE
  ) +
  geom_node_label(
    aes(
      label = name,
      colour = location,
      fontface = if_else(
```

```

    node_from_pauline_to_simon,
    "bold",
    "plain"
)
)
) +
scale_colour_brewer(name = "Location", palette = "Dark2") +
theme(legend.position = c(1, 0), legend.justification = c(1, 0))

```

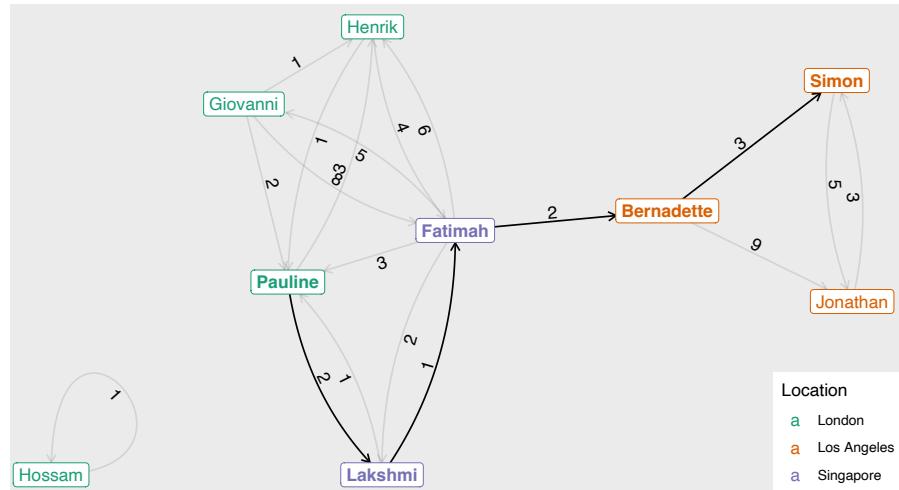


FIGURE 26.4: Email network with highlighted shortest path from Pauline to Simon.

26.3.5 Finding central nodes

The objective of many network studies is to find out which nodes are important or ‘central’. There are a variety of measures for node centrality. We already encountered two such measures in section 26.3.2: the degree and the strength. In this section, we discuss two additional centrality measures: *betweenness* and *eigenvector centrality*. There are also many other centrality measures (e.g. closeness centrality or PageRank), which can be calculated using **tidygraph**, but we do not cover them here for the sake of brevity.

The betweenness of a node is the number of shortest paths that go through this node (excluding the start and end nodes). A node with a high betweenness is important for the flow of information in the network because this node is on many of the most efficient communication channels between different parts of the network. If there is more than one shortest path between two nodes, we

split the contribution of all paths to the betweenness equally. Edges can, in principle, be weighted; however, here we only consider the unweighted case.

Let us take a look at two example nodes from the email network shown in figure 25.1.

- Bernadette is on all shortest paths with a start node in the set $S_1 = \{\text{Fatimah}, \text{Giovanni}, \text{Henrik}, \text{Lakshmi}, \text{Pauline}\}$ and an end node in the set $S_2 = \{\text{Jonathan}, \text{Simon}\}$. Because S_1 contains 5 nodes and S_2 comprises 2 nodes, we can form $5 \cdot 2 = 10$ pairs. Therefore, Bernadette's betweenness is 10.
- Pauline is on the shortest paths from Giovanni to Lakshmi, from Henrik to Lakshmi and from Lakshmi to Henrik. In all three cases, there is an alternative shortest path via Fatimah; thus, each path through Pauline only contributes 1/2 to Pauline's score. Consequently, Pauline's betweenness is 3/2.

The betweenness can be calculated with the **tidygraph** function `centrality_betweenness()`.

```
email_directed_netw |>
  activate(nodes) |>
  mutate(betw = centrality_betweenness()) |>
  arrange(desc(betw)) |>
  as_tibble()

## # A tibble: 9 x 3
##   name      location     betw
##   <chr>    <chr>       <dbl>
## 1 Fatimah  Singapore   16.5
## 2 Bernadette Los Angeles 10
## 3 Henrik    London      2.5
## 4 Lakshmi   Singapore   2.5
## 5 Pauline   London      1.5
## 6 Giovanni  London      0
## 7 Hossam    London      0
## 8 Jonathan  Los Angeles 0
## 9 Simon     Los Angeles 0
```

The output above shows that, in terms of betweenness, Fatimah is the most central node followed by Bernadette, who is on many shortest paths despite her relatively low degree.

Another measure of importance is eigenvector centrality. A node has a high eigenvector centrality if it is connected to many other nodes that have themselves many connections to central nodes.¹ Eigenvector centrality can be calculated with the **tidygraph** function `centrality_eigen()`. For example, the

¹Mathematically, the eigenvector centrality x_i of a node i follows from the equation

next code chunk applies `centrality_eigen()` to find eigenvector centralities of all nodes in the directed email network shown in figure 25.1. In this calculation, we assume that all edge weights are equal; however, it is, in principle, possible to pass an argument called `weights` to `centrality_betweenness()`.

```
email_directed_netw |>
  activate(nodes) |>
  mutate(
    eigen = centrality_eigen(),
    betw = centrality_betweenness()
  ) |>
  arrange(desc(eigen)) |>
  as_tibble()
## # A tibble: 9 x 4
##   name      location     eigen    betw
##   <chr>     <chr>        <dbl>    <dbl>
## 1 Fatimah  Singapore    1e+0    16.5
## 2 Pauline   London      8.62e-1   1.5
## 3 Henrik    London      8.12e-1   2.5
## 4 Lakshmi   Singapore   6.87e-1   2.5
## 5 Giovanni  London      6.78e-1   0
## 6 Bernadette Los Angeles 2.07e-1  10
## 7 Jonathan  Los Angeles 6.04e-2   0
## 8 Simon     Los Angeles 6.04e-2   0
## 9 Hossam    London      1.51e-17  0
```

The output shows that Fatimah is the most highly ranked node in terms of both eigenvector centrality and betweenness. Bernadette, who has the second-highest betweenness, only ranks sixth in terms of eigenvector centrality because two out of her three connections (Jonathan and Simon) only play a peripheral role in the network. We conclude that there is no simple answer to the question: is a node central to the network? Different centrality measures give different results, and it is good practice to compare a variety of measures when assessing the role that a node plays for the network.

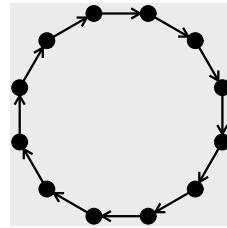
26.4 Characterising the overall structure of a network

So far, we have learned how to characterise the importance of a node to the network. Sometimes we also want to characterise the overall structure of a

$x_i = \frac{1}{\lambda} \sum_j x_j$. The summation is over all nearest neighbours of i , and the so-called eigenvalue λ must be adjusted so that the equation can be solved self-consistently (Newman, 2018).

network. For example, what are the differences between `email_directed_netw` and the following ring network?

```
ring_netw <- create_ring(12, directed = TRUE)
ggraph(ring_netw, layout = "linear", circular = TRUE) +
  geom_edge_link(
    edge_width = 1.5,
    arrow = arrow(length = unit(5, "mm")),
    end_cap = circle(4, "mm")
  ) +
  geom_node_point(size = 10) +
  coord_equal() # Needed so that circles do not become elongated ellipses
```



We already saw two basic measures for the overall network structure in section [24.2](#): the graph order and the graph size.

```
gorder(email_directed_netw)
## [1] 9
gorder(ring_netw)
## [1] 12
gszie(email_directed_netw)
## [1] 19
gszie(ring_netw)
## [1] 12
```

We conclude that `email_directed_netw` has fewer nodes but more edges than `ring_netw`.

Apart from the graph order and graph size, some of the most useful network summary statistics are

- the network's *density*.
- the *mean distance*.
- *transitivity*.
- *reciprocity* (in directed networks).

We now learn how these measures are defined and how to calculate them.

26.4.1 Density

The density of a network is the proportion of edges in a network that actually do exist out of all edges that could potentially exist if all node pairs were connected. When we count the number of theoretically possible edges, we exclude the possibility of parallel edges and loops. For example, in `email_directed_netw`, there are 9 nodes, which could be connected by as many as $9 \cdot 8 = 72$ directed edges. In reality, there are 19 edges, so the density is $19 / 72 \approx 0.26$. The density can be calculated by using the **igraph** function `edge_density()`.

```
edge_density(email_directed_netw)
## [1] 0.2638889
edge_density(ring_netw)
## [1] 0.09090909
```

The output shows that `email_directed_netw` is more densely connected than `ring_netw`.

26.4.2 Mean distance

Another measure for the overall connectedness of a network is the mean distance. It is calculated by first determining the lengths of the shortest paths between all pairs of nodes. Then we take the mean of these lengths. If there is no shortest path between two nodes (e.g. between Hossam and Lakshmi in figure 25.1), we exclude this node pair before taking the average. The mean distance can be calculated by using the **igraph** function `mean_distance()`.

```
mean_distance(email_directed_netw)
## [1] 1.846154
mean_distance(ring_netw)
## [1] 6
```

We conclude that shortest paths in `email_directed_netw` tend to be shorter than in `ring_netw`. However, we should bear in mind that, in contrast to `ring_netw`, not all nodes in `email_directed_netw` can be connected by a path.

26.4.3 Triangles and transitivity

We can obtain a different view on connectivity by investigating whether two nodes with a common neighbour tend to be neighbours themselves. If the answer is yes, the three nodes form a *triangle*. For simplicity's sake, we consider nodes to be mutual neighbours if they are connected by an edge, regardless of the direction of the edge. For our email network, we thus count triangles in

the simple, undirected network shown in figure 26.3.² For example, Henrik is part of 3 triangles: Henrik–Giovanni–Fatimah, Henrik–Giovanni–Pauline and Henrik–Fatimah–Pauline. With `local_triangles()`, we can find the number of triangles that each node is part of.

```
triangles <-
  email_directed_netw |>
  activate(nodes) |>
  mutate(n_tri = local_triangles()) |>
  as_tibble()
triangles
## # A tibble: 9 x 3
##   name      location     n_tri
##   <chr>    <chr>        <dbl>
## 1 Bernadette Los Angeles     1
## 2 Fatimah   Singapore      4
## 3 Giovanni  London         3
## 4 Henrik    London         3
## 5 Hossam    London         0
## 6 Jonathan  Los Angeles    1
## 7 Lakshmi   Singapore      1
## 8 Pauline   London         4
## 9 Simon     Los Angeles    1
```

We can find the total number of triangles in the network by summing the column `n` and then dividing by 3 because, for every triangle, `n` increases in three different rows (i.e. for each of the nodes on the triangle).

```
sum(triangles$n_tri) / 3 # Number of triangles in the network
## [1] 6
```

The total number of triangles is a useful piece of information, but it is even more informative to know what proportion t of all triplets (i.e. sub-networks of three nodes x , y and z with at least two edges) are triangles. Because every triangle contains three triplets, we must have

$$t = \frac{3 \cdot \text{number of triangles}}{\text{number of triplets}} .$$

The quantity t is called the *transitivity* or *global clustering coefficient*. The network in figure 26.3 contains 28 triplets (happy counting!); thus, $t = 18/28 \approx 0.64$. If we prefer to let **igraph** do the counting for us, we use the function `transitivity()`. It does not matter whether we pass the directed or undirected

²If we want to compute the statistics of *directed* triangles, we can use `triad_census()`.

network as argument to `transitivity()`; the direction is ignored if the input is directed.

```
transitivity(email_directed_netw)
## [1] 0.6428571
```

The transitivity of `email_directed_netw` is much higher than that of `ring_netw`, which has lots of triplets, but none of them forms a triangle.

```
transitivity(ring_netw)
## [1] 0
```

The return value of `transitivity()` makes it possible to compare different networks globally, but sometimes we also would like to compare nodes in the same network in terms of their tendency to form local triangles. The *local transitivity* (also known as *local clustering coefficient*) of a node x is defined as

$$t_x = \frac{\text{number of undirected triangles that contain } x}{\text{number of undirected triplets with } x \text{ as midpoint}} .$$

For example, Bernadette is part of 3 triplets (Fatimah–Bernadette–Jonathan, Fatimah–Bernadette–Simon and Jonathan–Bernadette–Simon), but only Jonathan–Bernadette–Simon forms a triangle. Hence, Bernadette's local transitivity is 1/3. We can confirm this result and also compute the local transitivity for all other nodes with `local_transitivity()`.

```
email_directed_netw |>
  activate(nodes) |>
  mutate(cluster_coef = local_transitivity()) |>
  as_tibble()

## # A tibble: 9 x 3
##   name      location    cluster_coef
##   <chr>     <chr>          <dbl>
## 1 Bernadette Los Angeles     0.333
## 2 Fatimah    Singapore      0.4
## 3 Giovanni   London         1
## 4 Henrik     London         1
## 5 Hossam     London         0
## 6 Jonathan   Los Angeles    1
## 7 Lakshmi    Singapore      1
## 8 Pauline    London         0.667
## 9 Simon      Los Angeles    1
```

26.4.4 Reciprocity

As we have just learned, transitivity is a tendency for triplets to form triangles in undirected networks. In directed networks, there can be an even shorter cycle than a triangle: going from a node x to another node y and then back to x . The *reciprocity* is the fraction of outgoing edges (from x to y) for which there is also an incoming edge (from y to x). Loops are conventionally excluded from the count. For example, out of the 19 edges in figure 25.1, one is a loop and twelve are partnered with an edge in the opposite direction. Hence, the network's reciprocity is $12/18 \approx 0.67$. The reciprocity can be calculated by using the **igraph** function `reciprocity()`.

```
reciprocity(email_directed_netw)
## [1] 0.6666667
```

By contrast, none of the edges in `ring_netw` has a partner pointing in the opposite direction; thus, the network's reciprocity is zero.

```
reciprocity(ring_netw)
## [1] 0
```

26.5 Determining communities in a network

Often, the objective of network analysis is to cluster nodes into groups. Three common definitions of groups are:

- maximal cliques.
- connected components.
- communities.

In this section, we learn the definitions and the corresponding R functions.

26.5.1 Cliques

Let us assume we are working with a simple, undirected network. As we learned in section 26.4.3, triangles are subgraphs of three nodes that possess all possible edges. We can generalise this idea to sub-networks with more than three nodes. An n -*clique* is a sub-network of n nodes that contains all $n(n-1)/2$ possible edges. Two examples of 3-cliques in figure 26.3 are Bernard–Jonathan–Simon and Fatimah–Henrik–Pauline.

A clique is called ‘maximal’ if we cannot add another node to the sub-network so that the enlarged sub-network would still be a clique. The clique Bernard–

Jonathan–Simon is maximal. An example for a non-maximal clique is Fatimah–Henrik–Pauline because Giovanni is connected to all three clique members, so Fatimah–Henrik–Pauline–Giovanni is also a clique.

As of 2020-12-28, there is no straightforward way to find maximal connected cliques using **tidygraph**. The best solution I can come up with is a detour via the **igraph** function `max_cliques()`. Here is a rather roundabout way to obtain the information from `max_cliques()` as a tibble:

```
max_clique_tibble <-
  email_simple_netw |>
  max_cliques() |>
  map(names) |>
  as_tibble_col(column_name = "name") |>
  rowid_to_column("clique_id") |>
  unnest(col = name)
max_clique_tibble
## # A tibble: 13 x 2
##   clique_id name
##       <int> <chr>
## 1 1         Hossam
## 2 2         Simon
## 3 2         Bernadette
## 4 2         Jonathan
## 5 3         Lakshmi
## 6 3         Fatimah
## 7 3         Pauline
## 8 4         Bernadette
## 9 4         Fatimah
## 10 5         Pauline
## 11 5         Fatimah
## 12 5         Henrik
## 13 5         Giovanni
```

Next, we want to make a plot that shows the cliques. We first create the network layout and then join the information about the node coordinates with the data in `max_clique_tibble`.

```
email_simple_layout <-
  create_layout(email_simple_netw, layout = "stress")
clique_coordinates <-
  max_clique_tibble |>
  left_join(email_simple_layout, by = "name")
```

Finally, we draw the network and highlight the edges with `geom_mark_hull()` from the `ggforce` package.

```
library(ggforce)
ggraph(email_simple_layout) +
  geom_mark_hull(
    aes(
      x,
      y,
      fill = as.factor(clique_id),
      label = str_c("clique ", clique_id)
    ),
    data = clique_coordinates,
    colour = NA, # Do not draw an outline around the cliques
    con.colour = "grey",
    show.legend = FALSE
  ) +
  geom_edge_fan(
    aes(label = weight),
    angle_calc = "along",
    label_dodge = unit(2, "mm"),
    label_push = unit(-2, "mm")
  ) +
  geom_node_label(aes(label = name, colour = location)) +
  scale_color_brewer(name = "Location", palette = "Dark2") +
  scale_fill_brewer(palette = "Set2") +
  theme(legend.position = c(0, 1), legend.justification = c(0, 1))
```

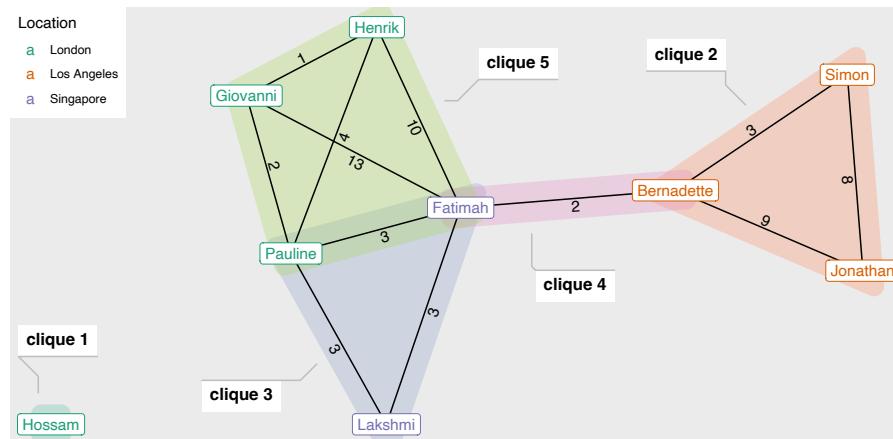


FIGURE 26.5: Maximal cliques in the email network.

26.5.2 Connected components

Cliques form the most tightly knit groups possible: every node in a clique is connected with every other node in the clique. Another way to split nodes into groups is to assign them into *connected components*. For *undirected* networks the definition is straightforward: two nodes x and y are in the same connected component if there is a path from x to y . The undirected network shown in figure 24.1 has three components:

- A, B, C, E, F, G, H, I.
- D, J.
- K.

We find connected components with the **tidygraph** function `group_components()`. If we pass `group_components()` as an argument to `mutate()`, we get a new column with component identifiers, which are arbitrary integers ranging from 1 to the number of connected components.

```
component_info_undirected <-
  demo_undirected_netw |>
  activate(nodes) |>
  mutate(component_id = group_components()) |>
  as_tibble()
component_info_undirected
## # A tibble: 11 x 2
##   name  component_id
##   <chr>     <int>
## 1 A          1
## 2 B          1
## 3 C          1
## 4 D          2
## 5 E          1
## 6 F          1
## 7 G          1
## 8 H          1
## 9 I          1
## 10 J         2
## 11 K         3
```

If we want the output to look more similar to the bullet list above (with one component per bullet point), we can use the following tidyverse workflow to transform the data into a list.

```
component_info_undirected |>
  group_by(component_id) |>
  group_split() |>
```

```

map(pluck("name"))
## [[1]]
## [1] "A" "B" "C" "E" "F" "G" "H" "I"
##
## [[2]]
## [1] "D" "J"
##
## [[3]]
## [1] "K"

```

In *directed* networks, the definition of a connected component is more complicated because the existence of a path from node x to node y does not necessarily imply that there is also a path from y to x . We must, therefore, distinguish between *weakly* and *strongly connected components*.

- The nodes x and y are in the same *weakly* connected component if and only if there is a path from x to y or from y to x , but we do not necessarily require both paths to exist.
- Two nodes are in the same *strongly* connected component if and only if there are paths between them in both directions.

For example, in the directed network shown in figure 25.1, Lakshmi and Bernadette are in the same weakly connected component because there is a path from Lakshmi to Bernadette (e.g. Lakshmi–Fatimah–Bernadette). However, there is no path in the opposite direction; thus, Lakshmi and Bernadette are in different strongly connected components. The function `group_components()` distinguishes between weakly and strongly connected components with an argument named `type`. Its default value is `type = weak`, and the alternative is `type = strong`.

```

email_directed_netw |>
  activate(nodes) |>
  mutate(
    weak_id = group_components(),
    strong_id = group_components(type = "strong")
  ) |>
  as_tibble()
## # A tibble: 9 x 4
##   name      location  weak_id strong_id
##   <chr>     <chr>        <int>     <int>
## 1 Bernadette Los Angeles     1         4
## 2 Fatimah   Singapore      1         1
## 3 Giovanni  London         1         1
## 4 Henrik    London         1         1

```

```
## 5 Hossam    London      2      3
## 6 Jonathan  Los Angeles 1      2
## 7 Lakshmi   Singapore  1      1
## 8 Pauline   London      1      1
## 9 Simon     Los Angeles 1      2
```

The output shows that the largest weakly connected component encompasses all nodes except Hossam. However, there are additional splits if the nodes are grouped into strongly connected components.

26.5.3 Communities

In network analysis, a *community* is defined as a set of nodes that are highly connected with other nodes in the same community but have relatively few connections to nodes outside their community. A clique is an extreme case of a community because there is an edge for *every* node pair in the clique. A connected component is another extreme case of a community because a node has no path to nodes outside its own connected component.

For practical work, the definitions of cliques and connected components are often too restrictive to be meaningful definitions of a community. In real networks, we often find groups of nodes that we can intuitively regard as a ‘community’ because the group has many but not all possible internal edges. It would also be too restrictive to demand that a community cannot have any paths to other communities. Consequently, we need a ‘softer’ criterion for classifying a group of nodes as a community. Many different definitions have been suggested in the literature, and many different algorithms have been developed that can detect communities according to different definitions. The **tidygraph** package contains implementations of some of these algorithms. One of the possibilities is `group_louvain()`, which is based on the so-called ‘Louvain algorithm’ by [Blondel et al. \(2008\)](#). It has the advantage of being relatively fast even if the network is large. The help at `?group_louvain` contains references to several other algorithms too. Their results can be substantially different, and some of these algorithms are impractically slow for large networks.

Like most other community detection algorithms, `group_louvain()` gives more easily interpretable results when applied to undirected rather than directed networks. Consequently, we apply `group_louvain()` to the undirected version of our email network (figure 26.2). The Louvain algorithm detects three communities, as can be seen from the following output.

```
email_undirected_netw <-
  email_undirected_netw |>
```

```

activate(nodes) |>
  mutate(community_id = group_louvain())
email_undirected_netw |>
  as_tibble() |> # Extract node information as tibble
  group_by(community_id) |>
  group_split() |>
  map(pluck("name"))
## [[1]]
## [1] "Fatimah"  "Giovanni" "Henrik"   "Lakshmi"  "Pauline"
##
## [[2]]
## [1] "Bernadette" "Jonathan"  "Simon"
##
## [[3]]
## [1] "Hossam"

```

As we did for the cliques in figure 26.5, we can highlight the communities with `geom_mark_hull()`.

```

ggraph(email_undirected_netw, layout = "stress") +
  geom_mark_hull(
    aes(
      x,
      y,
      fill = as.factor(community_id),
      label = str_c("community ", community_id)
    ),
    colour = NA,
    con.colour = "grey",
    show.legend = FALSE
  ) +
  geom_edge_fan(
    aes(label = emails),
    strength = 1.2,
    angle_calc = "along",
    label_dodge = unit(2, "mm"),
    label_push = unit(2.5, "mm")
  ) +
  geom_edge_loop(
    aes(label = emails),
    angle_calc = "along",
    label_dodge = unit(2, "mm")
  )

```

```
geom_node_label(aes(label = name, colour = location)) +
  scale_colour_brewer(name = "Location", palette = "Dark2") +
  scale_fill_brewer(palette = "Set2") +
  theme(legend.position = c(0, 1), legend.justification = c(0, 1))
```

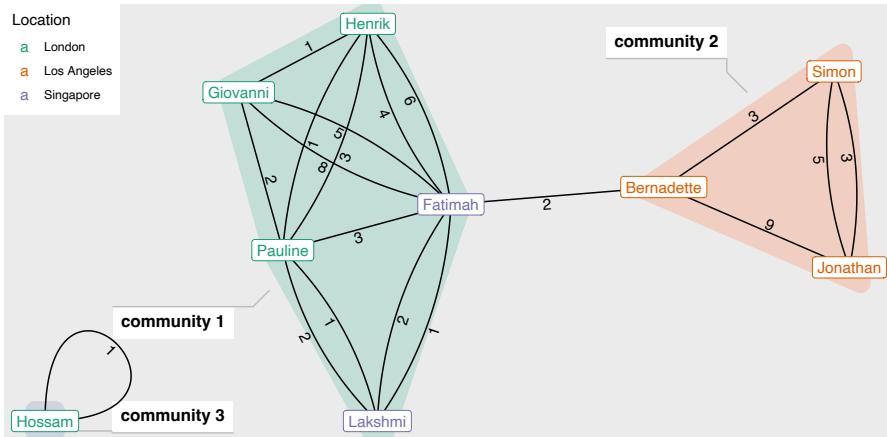


FIGURE 26.6: Communities determined by the Louvain algorithm.

26.6 Conclusion

Networks appear frequently in data analysis. In this chapter, we focused on tools from the **tidygraph** package. They are powerful and tie in nicely with **ggraph** to produce attractive visualisations. There are still some gaps that **tidygraph** needs to fill (e.g. calculating maximal cliques), but the package definitely holds a lot of promise.



Exercises: Visualising network communities

Prerequisite: chapters 24–26

Approximate duration: 120 minutes

Submission format: R Markdown

A classic study in social network science was conducted by [Zachary \(1977\)](#). He observed the interactions between members of a karate club at a US university. Each member is represented by a node. An edge between two nodes indicates that these two members also engaged in friendly relations outside the club's lessons and meetings.

By the end of Zachary's study, a conflict arose between the club president and the karate instructor. As a consequence, supporters of the instructor resigned from the club and started a new organisation. Zachary's observational data are often used in models that try to predict how networks split into groups if there is a conflict between members.

Zachary's network is undirected (i.e. an edge from i to j is equivalent to an edge from j to i).

Objectives

Using Zachary's data as example, we learn how to visualise communities (figure 26.7). We compute communities with **tidygraph**. We also develop more familiarity with **ggplot2** and the add-on packages **ggraph** and **ggforce**.

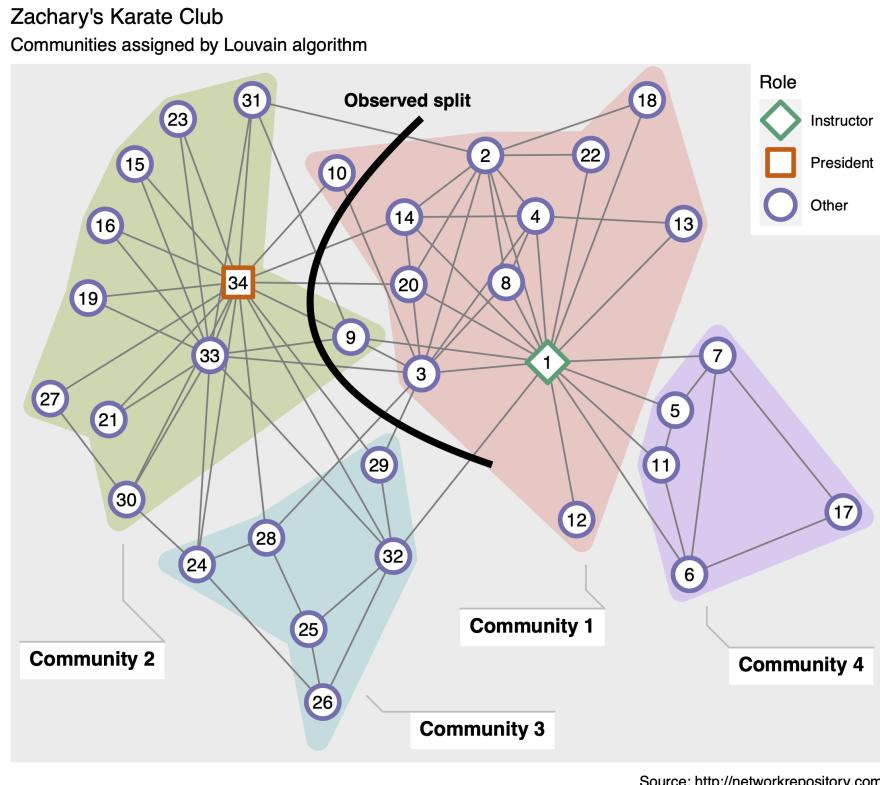


FIGURE 26.7: By the end of this exercise, we want to produce a plot similar to the plot shown above.

Data

The data for this exercise are available from https://michaelgastner.com/DAVisR_data/soc-karate mtx.³ The first few lines of soc-karate.mtx are metadata. Afterwards, there are lines with exactly two integers. This portion of the file is the edge list. All nodes in the network appear at least once in the edge list.

³The original data were obtained from <http://networkrepository.com/soc-karate.php> on 16 June 2017 (Rossi and Ahmed, 2015).

Tasks

- (1) Import the data as `tbl_graph`. Node 1 is the karate instructor. Node 34 is the club president. Add an attribute `role` to the `tbl_graph` with the values `Instructor`, `President` and `Other`.
- (2) What is the density of the network?
- (3) What is the mean distance in the network?
- (4) How many triangles are in the network? What is the global transitivity?
- (5) What is the local clustering coefficient of the instructor? What is the local clustering coefficient of the president?
- (6) What is the largest clique size in the network? Which nodes are in these cliques?
- (7) Confirm that the network is connected (i.e. there is only one component).
- (8) Which communities does the Louvain algorithm detect? Use `tidygraph` to add information about the communities to the `tbl_graph`.
- (9) Make a plot similar to figure 26.7. Here are a few properties that the final figure should have.
 - Use `geom_mark_hull()` to show the communities detected by the Louvain algorithm.
 - The nodes that stayed in the old club were 10, 15, 16, 19, 21, 23–34. Indicate the observed split with a Bezier curve (see `?geom_bezier` in the `ggforce` package).
 - Add the annotation ‘Observed split’ near the Bezier curve.
 - Draw the edges as thin grey lines and the Bezier curve as thick black line.
 - Use colour and shape to identify the node roles. Use a combined legend for colour and shape. Make sure that the legend does not overlap with any nodes or edges.
 - Add a title.
 - Acknowledge the source in a caption.

Feel free to make more adjustments if you think they improve the quality of the plot.

- (10) Briefly explain to a reader what the figure reveals about the karate-club data.



Part VIII

Extracting data from natural-language text



27

Working with character strings

There are many situations where character strings are a rich source of information. In section 10.4, we learned that we can find the number of characters in each element of a vector with `nchar()`. However, there is a lot more information we can extract from character vectors than only the number of characters in each element. For example, we may want to know how often "Trump" and "haircut" appear in the same news story. Or, for deciphering encrypted text, we may want to determine the most frequently used character combinations in different languages.

R can help us with many problems in text analysis. Some basic string handling features are preinstalled. In this chapter, we take advantage of functions in the `stringr` package, which is included in the tidyverse suite of packages. The functions in the `stringr` package have more intuitive names than the built-in functions. The order of their arguments is also more consistent so that they are more suitable when working with pipes. For more complex problems in natural language processing, there are specialised packages for text mining (e.g. `tidytext`) that are outside the scope of this chapter.

We begin with a concrete example where string parsing is necessary.¹ Let us download the file at https://michaelgastner.com/DAVisR_data/homicides.txt to our working directory. The file contains information about homicides in Baltimore between 1 January 2007 and 14 May 2012. When we look at the file with a basic text editor, we notice that the text is not in a standard delimited format (e.g. CSV where commas would separate different columns). Here are the first three lines. (I have inserted line breaks to fit the text inside the page width.)

```
39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon  
Nelson</dt><dd class="address">3400 Clifton Ave.<br />Baltimore, MD  
21216</dd><dd>black male, 17 years old</dd><dd>Found on January 1,  
2007</dd><dd>Victim died at Shock Trauma</dd><dd>Cause: shooting  
</dd></dl>'  
  
39.312641, -76.698948, iconHomicideShooting, 'p3', '<dl><dt>Eddie  
Golf</dt><dd class="address">4900 Challedon Road<br />Baltimore, MD
```

¹The example and the data are inspired by <https://www.youtube.com/watch?v=q8SzNKib5-4>.

```

21207</dd><dd>black male, 26 years old</dd><dd>Found on January 2,
2007</dd><dd>Victim died at scene</dd><dd>Cause: shooting</dd></dl>'  

39.309781, -76.649882, iconHomicideBluntForce, 'p4', '<dl><dt>  

Nelsene Burnette</dt><dd class="address">2000 West North Ave<br />  

Baltimore, MD 21217</dd><dd>black female, 44 years old</dd><dd>  

Found on January 2, 2007</dd><dd>Victim died at scene</dd><dd>  

Cause: blunt force</dd></dl>'

```

Suppose we want to find out how the number of homicides varies over time. We can use the date that follows after the string "Found on" as a proxy for the homicide date. How can we extract the date as a substring from each line? Those of us with HTML knowledge may recognise that `homicides.txt` must have once been part of an HTML website. If we had valid HTML code, a sensible strategy would be to use HTML parsers (e.g. in the `rvest` package) to obtain the desired information. Unfortunately, `homicides.txt` is not valid HTML (e.g. the `<body>` tag has been removed). Even if we had access to the original website, we would not be able to easily scrape the data. We could break the text into smaller pieces, but we would still be left with text chunks such as "Found on January 1, 2007" that would need to be dissected with other methods to extract only the date.

In this chapter, we learn how to import text data into R, split the text into pieces and extract useful information. A powerful tool for string operations are *regular expressions*, which are specially formatted character strings that identify patterns in text.

27.1 Importing non-spreadsheet data into R

In chapter 15, we learned how to import spreadsheet data with functions from the `readr` package. The package also has functions for importing non-spreadsheet data such as `homicides.txt`. We can import an entire file into a single character string (i.e. a character vector with only one element, which may contain a large number of characters) with `read_file()`. This strategy is suitable for natural language text that has almost no formatting (e.g. newspaper articles or novels). The text in `homicides.txt` is still a little bit more formatted than natural language text. For example, every line contains information about exactly one homicide. In this case, `read_lines()` is a better choice than `read_file()`. The function `read_lines()` returns a character vector with one element for each line in the file.

```

library(tidyverse)
homicides <- read_lines("homicides.txt")

```

```
class(homicides)
## [1] "character"
length(homicides)
## [1] 1249
```

Next, we must learn how to parse the elements in the `homicides` vector to extract the desired information from the text.

27.2 Functions in the **stringr** package

27.2.1 Splitting strings with `str_split()`

A frequent task when working with character strings is to split them into smaller pieces (e.g. into individual words or characters). The **stringr** package has a function `str_split()` for this purpose. We usually call `str_split()` with two arguments:

- `string`: the character vector whose elements we wish to split.
- `pattern`: the character where we want to split the `string`. Instead of a character, `pattern` can also be a ‘regular expression’. We learn more about regular expressions in section 27.3.

If we want to split a string into words, we pass a single space " " as second argument.²

```
x <- c("Here is one sentence.", "Here is another.")
str_split(x, " ")
## [[1]]
## [1] "Here"      "is"       "one"      "sentence."
## 
## [[2]]
## [1] "Here"      "is"       "another."
```

As we can tell from the output, `str_split()` returns a list. Each element of the list corresponds to one element in the vector `x`.

If we want to split `x` into single characters, we set `pattern = ""` (i.e. a character of length zero).

²We learn in section 27.3.8 that, if we want to split text into words, an even better value for the `pattern` argument is "\s+", which matches not only a single space, but also multiple spaces, tabs and line breaks.

```
str_split(x, "")  
## [[1]]  
## [1] "H" "e" "r" "e" " " "i" "s" " " "o" "n" "e" " " "s" "e" "n" "t" "e" "n" "c"  
## [20] "e" ".."  
##  
## [[2]]  
## [1] "H" "e" "r" "e" " " "i" "s" " " "a" "n" "o" "t" "h" "e" "r" ".."
```

Thanks to `str_split()`, we can easily extract the latitude and longitude from `homicides.txt`. The geographic coordinates of the crime scene are the two numbers at the start of each line (highlighted in red).

```
39.311024, -76.674227, iconHomicideShooting, 'p2', ...  
39.312641, -76.698948, iconHomicideShooting, 'p3', ...
```

The next code chunk shows how to store all 1249 coordinates in a tibble called `homicide_coord` that contains one column for the latitude and another column for the longitude.

```
homicide_split <- str_split(homicides, ", ")  
toNumber <- function(x, i) {  
  as.numeric(x[i])  
}  
homicide_coord <- tibble(  
  lat = map_dbl(homicide_split, toNumber, 1),  
  lon = map_dbl(homicide_split, toNumber, 2)  
)  
head(homicide_coord)  
## # A tibble: 6 x 2  
##       lat     lon  
##   <dbl> <dbl>  
## 1 39.3 -76.7  
## 2 39.3 -76.7  
## 3 39.3 -76.6  
## 4 39.4 -76.6  
## 5 39.2 -76.6  
## 6 39.4 -76.6
```

27.2.2 Combining strings with `str_c()`

While `str_split()` splits strings, `str_c()` combines them.

```
str_c("co", "m", "bin", "e")  
## [1] "combine"
```

We can pass separators (e.g. spaces or commas) as an argument named `sep`.

```
str_c("This", "is", "a", "sentence.", sep = " ")
## [1] "This is a sentence."
str_c("a", "b", "c", sep = "; ")
## [1] "a; b; c"
```

Like many other R function, `str_c()` is vectorised.

```
str_c("Postal code: ", 129800:129804)
## [1] "Postal code: 129800" "Postal code: 129801" "Postal code: 129802"
## [4] "Postal code: 129803" "Postal code: 129804"
```

27.2.3 Extracting substrings with `str_sub()`

We can extract substrings of given lengths with `str_sub()`. The function takes three arguments:

- `string`: character vector.
- `start`: integer vector with the position(s) of the first character. The default value is 1.
- `end`: integer vector with the position(s) of the last character. Negative values count the distance from the end. The default value is -1, which is the last character in the `string`.

```
x <- c("Here is one sentence.", "Here is another.")
str_sub(x, 9, -2)
## [1] "one sentence" "another"
```

We can also assign new substrings.

```
str_sub(x, 9, -2) <- str_c("sentence ", c("A", "B"))
x
## [1] "Here is sentence A." "Here is sentence B."
```

27.2.4 Searching for strings that match a pattern

Besides extracting and replacing substrings, we often would like to find out whether a string contains a given substring. For example, which US state names contain "ia" as a substring? We can find it out with `str_subset()`.

```
str_subset(state.name, "ia")
## [1] "California"    "Georgia"       "Indiana"        "Louisiana"
## [5] "Pennsylvania"  "Virginia"      "West Virginia"
```

With `str_which()`, we can find out which indices these elements have in `state.name()`.

```
str_which(state.name, "ia")
## [1] 5 10 14 18 38 46 48
```

With `str_detect()`, we obtain a logical vector that shows whether the substring "ia" is in an element of `state.name`.

```
str_detect(state.name, "ia")
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
## [13] FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE
## [37] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
## [49] FALSE FALSE
```

These functions are case sensitive.

```
str_subset(state.name, "al")
## [1] "California"
str_subset(state.name, "Al")
## [1] "Alabama" "Alaska"
```

So far, we have only used literal substrings as second argument in `str_subset()`, `str_which()` and `str_detect()`. We can also use ‘regular expressions’ for more complex pattern matching. We learn more about regular expressions in section 27.3.

27.2.5 Changing case with `str_to_upper()` and `str_to_lower()`

It is sometimes convenient to work with strings that are either all in upper case or all in lower case. The functions for this purpose are `str_to_upper()` and `str_to_lower()`.

```
x <- c("Here is sentence A.", "Here is sentence B.")
str_to_upper(x)
## [1] "HERE IS SENTENCE A." "HERE IS SENTENCE B."
str_to_lower(x)
## [1] "here is sentence a." "here is sentence b."
```

27.2.6 Replacing characters and patterns

Changing the case of all lower-case characters to upper case is straightforward thanks to `str_to_upper()`, and `str_to_lower()` performs the inverse

action. For more complicated substitutions, **stringr** offers the functions `str_replace()` and `str_replace_all()`. Both functions need three arguments:

- `string`: a character vector.
- `pattern`: a pattern of characters to be replaced.
- `replacement`: a character vector of replacements.

The function `str_replace()` replaces only the first match of the `pattern`, whereas `str_replace_all()` replaces all matches.

```
x <- c("understated funder", "underpaid and underappreciated")
str_replace(x, "under", "un")
## [1] "unstated funder"           "unpaid and underappreciated"
str_replace_all(x, "under", "un")
## [1] "unstated fun"             "unpaid and unappreciated"
```

A common replacement is the removal of certain characters. In this case, we can use `str_remove()` or `str_remove_all()`.

```
y <- c("raging bull with a string")
str_remove(y, "r")
## [1] "aging bull with a string"
str_remove_all(y, "r")
## [1] "aging bull with a sting"
```

The functions `str_replace()`, `str_replace_all()`, `str_remove()` and `str_remove_all()` accept ‘regular expressions’ as `pattern` and `replacement`. So, what is the big deal about ‘regular expressions’? Some people seem to have a lot of fun with them (figure 27.1). Let us join the fun in the next section.

27.3 Regular expressions

A regular expression, also known as a regex or regexp, is a search pattern for strings, interpreted according to a set of standardised rules. Many programming languages, including R, natively understand regular expressions, and even more languages have add-on libraries to offer regex capabilities.

Regular expressions were invented in 1951 when the mathematician Stephen Cole Kleene described regular languages with a mathematical notation called regular sets (see https://en.wikipedia.org/wiki/Regular_expression). Although the rules for regular expressions are complex, their syntax is very powerful and well worth learning.



FIGURE 27.1: An xkcd cartoon (licensed under CC-BY-NC 2.5).

27.3.1 Literal and special characters

We have already unwittingly used regular expressions earlier in this chapter, for example in this code chunk from section 27.2.4:

```
str_subset(state.name, "al")
```

The second argument "al" is a regular expression consisting only of *literal characters*. A character in a regular expression is literal if it is meant to be taken at face value. In this example, R interprets both a and l as the characters we are familiar with.

There are 13 *special characters* that are reserved for building more complex regular expressions than what would be possible with literal characters alone.

- Backslash: \
- Caret: ^
- Dollar: \$
- Plus sign: +
- Opening parenthesis: (
- Closing parenthesis:)
- Opening square bracket: [, but the closing square bracket] is, by default, a literal character
- Opening curly brace: {
- Closing curly brace: }
- Dot: .
- Vertical bar: |
- Question mark: ?
- Asterisk: *

If we intend to match a special character literally, we must, in most cases, put an ‘escape sequence’ before them. We talk in more detail about escape sequences in section 27.3.7.

27.3.2 Search for patterns at the beginning of a string with ^

Often we want to find substrings at a particular position in a text. For example, we may want to determine which US state names start with a "v". When we search with a regular expression "v", we include "West Virginia" in the results, because regular expressions seek matches everywhere in a string, not only at the start.

```
str_subset(state.name, "v")
## [1] "Vermont"      "Virginia"       "West Virginia"
```

We restrict the search to a v only at the *beginning* of the string by preceding v with a caret ^.

```
str_subset(state.name, "^v")
## [1] "Vermont"  "Virginia"
```

We can search for patterns extending beyond the first character. For example, here is how we find all states whose names contain "Ca" anywhere in the string:

```
str_subset(state.name, "Ca")
## [1] "California"   "North Carolina" "South Carolina"
```

And here are only the states whose *start* with "Ca":

```
str_subset(state.name, "^Ca")
## [1] "California"
```

27.3.3 Search for patterns at the end of a string with \$

While a preceding `^` restricts the search to patterns at the *beginning* of a string, a trailing `$` limits the search to patterns at the *end*. For example, here is how we find the states whose names contain "ta" anywhere in the string:

```
str_subset(state.name, "ta")
## [1] "Minnesota"    "Montana"      "North Dakota" "South Dakota" "Utah"
```

And here are the states whose names *end* with "ta":

```
str_subset(state.name, "ta$")
## [1] "Minnesota"    "North Dakota" "South Dakota"
```

27.3.4 Search Patterns at the start and end of a word with \\b

A *word* is a substring separated from the rest of the string by a whitespace. Whitespace is a technical term defined as space, tab, line feed, form feed and carriage return. To find a certain pattern at the beginning of a word, but not necessarily at the beginning of a string, we can use `\b`. Here are, for example, all states whose name contains a word starting with "D":

```
str_subset(state.name, "\bD")
## [1] "Delaware"     "North Dakota" "South Dakota"
```

We can also put `\b` at the *end* of a pattern. In that case, we are looking for strings containing a word, not necessarily the last one, ending with this pattern. Here are the states whose names contain a word ending with "t":

```
str_subset(state.name, "t\b")
## [1] "Connecticut"   "Vermont"      "West Virginia"
```

While `\b` matches word boundaries, `\B` matches all characters that are *not* word boundaries. For example, here are all states whose names contain a "t" that is not at the end of a word. Note that "Connecticut" is in the set because it contains a "t" in the middle, not only at the end.

```
str_subset(state.name, "t\\B")
## [1] "Connecticut"    "Kentucky"       "Massachusetts"  "Minnesota"
## [5] "Montana"        "North Carolina" "North Dakota"   "South Carolina"
## [9] "South Dakota"   "Utah"          "Washington"
```

27.3.5 Character classes

When working with text, we often encounter alternative (e.g. American and British) spellings of substrings. We can allow multiple different characters to qualify as a match by inserting them between square brackets. Here is how we search for all words ending in either "ise" or "ize":

```
example_1 <- c("maximise", "maximize", "granite", "bowtie")
str_subset(example_1, "i[sz]e$")
## [1] "maximise" "maximize"
```

The output of the next code chunk highlights with a grey background behind the corresponding substrings where the regular expressions match in each element of `example_1`. The result from `str_view_all()` appears in RStudio's viewer pane in the bottom right.

```
str_view_all(example_1, "i[sz]e$")
```

| |
|----------|
| maximise |
| maximize |
| granite |
| bowtie |

We can also specify more than two alternatives inside the brackets.

```
example_2 <- c("at", "bat", "cat", "hat", "rat")
str_view_all(example_2, "[bhr]at")
```

| |
|-----|
| at |
| bat |
| cat |
| hat |
| rat |

The characters inside the square brackets are called a *character class*. We can list, as we have done so far, all characters individually. However, it is also possible to specify ranges of characters using hyphens. Here are some examples:

- all capital letters: [A-Z],
- all letters: [a-zA-Z],
- "a", "b", "c", "d", "e", "w", "x", "y", "z": [a-ew-z]
- all numbers: [0-9]

Here is an example:

```
str_view_all(c("copper", "hum", "explain", "unite"), "[a-ew-z]")
```

```
copper
hum
explain
unite
```

In all previous examples, we matched a string by specifying characters that it should contain. However, sometimes we would like to do the opposite: match a string because certain characters are absent. For this purpose, we use a caret ^ as the first symbol inside square brackets to indicate that the remaining characters in the square brackets should *not* belong to the character set. For example, here we match those strings that have an "i" as third to last letter, "e" as last letter and neither an "s" nor "z" as second-to-last letter:

```
example_1
## [1] "maximise" "maximize" "granite"  "bowtie"
str_view_all(example_1, "i[^sz]e$")
```

```
maximise
maximize
granite
bowtie
```

27.3.6 The dot . as a wildcard

Suppose we want to find all strings that contain a "t" followed by any arbitrary character that is then followed by an "o". The regular expression for this application is "t.o" where the dot is a 'wildcard' (i.e. a special character that represents any arbitrary character).

```
example_3 <- c("two", "to", "too", "toe")
str_view_all(example_3, "t.o")
```

```
two
to
too
toe
```

If we want to literally match a dot, we must ‘escape’ it with two backslashes.

```
str_view_all(c("End", "of", "sentence."), "\\.")
```

```
End
of
sentence.
```

27.3.7 How to make special characters literal

The rule we have just seen at the end of section 27.3.6 can be generalised: two preceding backslashes turn any of the special characters listed in section 27.3.1 into a literal character. Here is an example that shows how to escape a dollar sign.

```
example_4 <- c("enj", "eoj", "e$j", "e\\$j")
str_view_all(example_4, "e\\$j")
```

```
enj
eoj
e$]
e\j
```

Backslashes start piling up when we want to match a literal backslash with a regular expression because each literal backslash must be escaped by another backslash. Both of these backslashes also need to be escaped themselves. Ultimately, the regular expression must contain four(!) backslashes to match one literal backslash.³

```
str_view_all(example_4, "\\\\\")
```

```
enj
eoj
e$]
e\\j
```

We cannot match those backslashes that are part of an escape sequence (e.g. the newline character `\n`), but we can match the escape sequence as an entity (i.e. using only one backslash in the regular expression).

³In other languages, we escape special regular expression characters with only one backslash. We can for all practical purposes think of R’s double-backslash rule as a dialect of the conventional rule. There is a deeper reason behind R’s rule: R first interprets our input as a conventional character string, in which every literal backslash must be escaped.

```
example_5 <- c("enq", "e\\nq", "e\\nq")
str_view_all(example_5, "\\n")
```

enq
e\nq
e\nq

27.3.8 Shorthand character classes

There are shorthand notations for many frequently used character classes. Table 27.1 lists the most important shorthand notations. Here is an example that uses the whitespace character class "\s":

```
example_6 <- c(
  "linebreak\\nnext line",
  "no.break",
  "whitespace: the final frontier"
)
str_view_all(example_6, "\\s")
```

linebreak\nnext line
no.break
whitespace: the final frontier

Here is another example that uses the punctuation character class `[[:punct:]]`:

```
str_view_all(example_6, "[[:punct:]]")
```

linebreak next line
no.\break
whitespace; the final frontier

27.3.9 Alternation with a vertical bar |

Character classes only allow specifying alternatives for *individual* characters. If the alternatives consist of sequences of *multiple* characters, we must instead use a vertical bar `|` to symbolise a logical ‘or’. In the parlance of regular expressions, the ‘or’ is called an *alternation*. The next code chunk shows how to use an alternation that matches strings containing “cat” or “mice”:

```
hidden_animals <- c(
  "undogmatic",
```

TABLE 27.1: Shorthand character classes.

| Shorthand | Replacement for | Description |
|-----------|--------------------------------|----------------------------|
| \w | [A-Za-z0-9_] | word character |
| \W | [^A-Za-z0-9_] | not a word character |
| \d | [0-9] | decimal digit |
| \D | [^0-9] | not a decimal digit |
| \s | [\t\r\n\f] | whitespace character |
| \S | [^ \t\r\n\f] | not a whitespace character |
| [:alpha:] | [A-Za-z] | alphabetical character |
| [:lower:] | [a-z] | lowercase letter |
| [:upper:] | [A-Z] | uppercase letter |
| [:punct:] | character class with .,;, etc. | punctuation character |

```
"education",
"semicentennial",
"pharmaceutical"
)
str_view_all(hidden_animals, "cat|mice")
```

undogmatic
 education
 semicentennial
 pharmaceutical

It is also possible to have more than two alternatives, as the next example shows:

```
str_view_all(hidden_animals, "cat|mice|dog")
```

undogmatic
 education
 semicentennial
 pharmaceutical

The strings on either side of the vertical bar can contain character classes and wildcards. In the next example, the square-bracket and dot operators are carried out before the vertical bar because the latter has a lower precedence. For the rules of operator precedence in regular expressions, see https://docs.tore.mik.ua/oreilly/perl/lperl/ch08_05.htm.

```
str_view_all(hidden_animals, "[cm]at|m.ce")
undogmatic
education
semicentennial
pharmaceutical
```

If the options are part of a longer expression, we put them inside parentheses.

```
str_view_all(hidden_animals, "d(ogm|uc)a")
undogmatic
education
semicentennial
pharmaceutical
```

27.3.10 Allowing options with ?

A question mark makes the preceding character optional.

```
example_7 <- c("lie", "life", "lime")
str_view_all(example_7, "if?e")
lie
life
lime
```

If the options concern more than one character, we must put the corresponding items in parentheses.

```
dates <- c(
  "November 23rd",
  "Nov 23",
  "Nov 23rd",
  "November, the 23rd",
  "Nov 23, 1992"
)
str_view_all(dates, "Nov(ember)? 23(rd)?")
```

```
November 23rd
Nov 23
Nov 23rd
November, the 23rd
Nov 23, 1992
```

27.3.11 Allowing repetitions

The question mark checks whether the preceding symbols appear 0 or 1 time in the string. When we replace ? by *, we allow the string to appear 0, 1, 2, ... times. For example, the next regular expression looks for a space followed by nothing or by a purely numerical substring at the end of a word.

```
str_view_all(dates, "\d*\b", dates)
```

```
November 23rd
Nov 23
Nov 23rd
November, the 23rd
Nov 23, 1992
```

Note that "\d*\b" matches the pattern " 23" instead of only the space in front of "23" because regular expressions are 'greedy' by default. That is, they match the longest available substring. We learn more about greedy matching in section 27.3.13.

The + operator has a similar effect as the * operator. Both operators look for repetition. However, the + operator demands that the preceding pattern is matched *at least* once, whereas the * does not require that the pattern must be present. For example, words that match the regular expression in the next code chunk must end in a number and cannot terminate with a space:

```
str_view_all(dates, "\d+\b")
```

```
November 23rd
Nov 23
Nov 23rd
November, the 23rd
Nov 23, 1992
```

We can indicate that a repetition must occur exactly m times by putting {m} behind the subpattern.

```
str_view_all(dates, "[[:lower:]]{7}\b")
```

```
November 23rd
Nov 23
Nov 23rd
November, the 23rd
Nov 23, 1992
```

If we put {m,} behind the subpattern, we search for $\geq m$ repetitions.

```
str_view_all(dates, "[[:lower:]]{3,}\b")
```

```
November 23rd
Nov 23
Nov 23rd
November, the 23rd
Nov 23, 1992
```

If we specify two numbers in the curly braces, as in $\{m,n\}$, the number of repetitions must be between m and n .

```
str_view_all(dates, "[[:lower:]]{2,3}\b")
```

```
November 23rd
Nov 23
Nov 23rd
November, the 23rd
Nov 23, 1992
```

27.3.12 Capturing groups

When working with text, we often want to extract substrings and temporarily keep them in the computer's memory so that we can refer back to them. In such applications, we can take advantage of *captured groups* in regular expressions. Groups to be captured must be coded inside parentheses. For example, table 27.2 shows the groups captured by the regular expression `"^(.)(.*)(..)$"`. Within a regular expression, we can refer to the groups as `\1`, `\2`, ... in the order of opening parentheses from left to right. In the parlance of regular expressions, the patterns `\1`, `\2`, ... are called *backreferences*. We can view the captured groups with the function `str_match()` in the `stringr` package. The first column in the returned matrix shows the entire matched substring. The remaining columns show each captured group. Note that we need to add 1 to the captured group's number to find its column index in the matrix: `\1` is in the 2nd column, `\2` is in the 3rd column etc.

```
dates
## [1] "November 23rd"      "Nov 23"          "Nov 23rd"
## [4] "November, the 23rd"  "Nov 23, 1992"
str_match(dates, "^(.)(.*)(..)$")
##      [,1]      [,2]      [,3]      [,4]
## [1,] "November" "N"     "ovember" "23rd"
## [2,] "23rd"     "N"     "ov"      "23"
## [3,] "rd"       "N"     "ov 23"   "rd"
```

TABLE 27.2: Groups captured by the regular expression `(.)(.*)(..)`

| Captured group | What is it? | Where is it stored? |
|--------------------|---|---------------------|
| <code>(.)</code> | first character | <code>\1</code> |
| <code>(..*)</code> | everything except the first character and the last two characters | <code>\2</code> |
| <code>(..)</code> | last two characters | <code>\3</code> |

```
## [4,] "November, the 23rd" "N"  "ovember, the 23" "rd"
## [5,] "Nov 23, 1992"      "N"  "ov 23, 19"      "92"
```

Captured groups can, for example, identify duplicated words and eliminate the duplicate.

```
str_replace(
  "The chromosomes duplicated duplicated themselves.",
  "(\\b\\w+\\b) \\1",
  "\\1"
)
## [1] "The chromosomes duplicated themselves."
```

We can also identify extensions of file names.

```
files <- c("IMG8935.png", "page389.html")
str_replace(
  files,
  "^([[:alpha:]]*\\d+)\\.([\\w{3,}])$",
  "File \\1 has extension \\2."
)
## [1] "File IMG8935 has extension png." "File page389 has extension html."
```

We can even nest captured groups.

```
str_replace(
  files,
  "^([[:alpha:]]*(\\d+))\\.([\\w{3,}])$",
  "File \\1 has index \\2 and extension \\3"
)
## [1] "File IMG8935 has index 8935 and extension png"
## [2] "File page389 has index 389 and extension html"
```

Captured groups are not our first encounter with parentheses in regular ex-

pressions. We also used parentheses for alternations in section 27.3.9 and for optional patterns in section 27.3.10. These parentheses also automatically capture the group in parentheses.

```
example_8 <-
  c("dog tail", "duct tape", "duck tales", "dining hall table")
  str_match(example_8, "(d(o|u|i).*) ta(bl)?")
##      [,1]          [,2]      [,3] [,4]
## [1,] "dog ta"       "dog"     "o"   NA
## [2,] "duct ta"      "duct"    "u"   NA
## [3,] "duck ta"      "duck"    "u"   NA
## [4,] "dining hall tabl" "dining hall" "i"   "bl"
```

What should we do if we have to insert parentheses to satisfy regular expression syntax, but we do not want to capture the subpattern inside these parentheses? In that case, we start the parenthesised expression with "(?::" instead of only "(:". In regular expression jargon, any group between "(?:" and ")" is called a *non-capturing group*. The next matrix neither contains the alternation "o|u|i" nor the optional "bl" because we made those groups non-capturing.

```
str_match(example_8, "(d(?:o|u|i).*) ta(?:bl)?")
##      [,1]          [,2]
## [1,] "dog ta"       "dog"
## [2,] "duct ta"      "duct"
## [3,] "duck ta"      "duck"
## [4,] "dining hall tabl" "dining hall"
```

27.3.13 Greedy and lazy matching

The output of the following code snippet is at first sight surprising.

```
angle_brackets <- "<text1> <text2> <text3>"
str_replace_all(angle_brackets, "<(.*>", "(\\1)")
## [1] "(text1) <text2> <text3>"
```

We might instead have expected that all angle brackets would be changed to parentheses.

```
## [1] "(text1) (text2) (text3)"

<text1>, <text2> and <text3> all match the regular expression "<(.*>". Should the parenthesised expressions \\1 then not match text1, text2 and text3? If yes, why does str_replace_all() leave the angle brackets in the middle unchanged?
```

The loophole in this argument is that not only <text1>, <text2> and <text3> match the regular expression "<(.*)>". The entire string angle_brackets is another match because it starts with < and ends with >. Why does str_replace_all() decide to match the larger substring?

By default, repetition operators (e.g. * and +) are ‘greedy’. They try to match as much text as possible. We can make them ‘lazy’ by putting a ? behind them. We can think of *? or +? as new repetition operators that match as few characters as possible. In the next code chunk we see the effect of lazy matching.

```
str_replace_all(angle_brackets, "<(.*)>", "(\\1)")
## [1] "(text1) (text2) (text3)"
```

Let us take a look again at homicides.txt from the start of this chapter. Here is the first line.

```
39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class="address">3400 Clifton Ave.<br />Baltimore, MD
21216</dd><dd>black male, 17 years old</dd><dd>Found on January 1,
2007</dd><dd>Victim died at Shock Trauma</dd><dd>Cause: shooting
</dd></dl>'
```

The date is between the substrings "<dd>Found on" and "</dd>". We must use *lazy* matching so that we do not include too much text.

```
homicide_dates <- str_replace(
  homicides,
  ".*<dd>Found on (.*)</dd>.*",
  "\\1"
)
head(homicide_dates)
## [1] "January 1, 2007" "January 2, 2007" "January 2, 2007" "January 3, 2007"
## [5] "January 5, 2007" "January 5, 2007"
```

27.3.14 General regular expression wisdom

Regular expressions are not for the faint-hearted. However, the fact that we obtained all dates from homicides.txt with a single(!) command shows the power of regular expressions. When we need assistance with debugging regular expressions, we can use str_view_all() or, if we are only interested in the first matched pattern in a string, str_view().

```
str_view_all(x, "un")
```

understated funder
 underpaid and underappreciated

```
str_view(x, "un")
```

understated funder
 underpaid and underappreciated

There are often many alternative solutions when using regular expressions. When we face a concrete problem involving regular expressions, it is best to search the World Wide Web for hints and elegant solutions.

There are many features of regular expressions that we have not covered in this chapter. Here is a recommended website if you want to dig deeper: <https://www.regular-expressions.info/index.html>. Please bear in mind that there are slight differences in the regular expression dialects of different programming languages.

27.4 Just checking

Find the correct answer options for the following questions. First, try to answer the questions without running the code on your computer. Afterwards, you can find the solution either by running the code in RStudio or by looking up solutions in appendix A.17.

- (I) Suppose we have defined the following vector called `x`.

```
x <- c("treacherous", "actress", "theatre", "centre")
```

We want to change these words from British to US spelling. That is, we want to generate a vector with the following elements.

```
## [1] "treacherous" "actress"      "theater"       "center"
```

Which of the following commands can we use in this example?

- (a) `str_replace(x, "re$", "er")`
- (b) `str_replace(x, "re?", "er")`
- (c) `str_replace(x, "re+", "er")`
- (d) `str_replace(x, "re^", "er")`

(II) Suppose we have defined a vector called `v` as follows.

```
v <- c("at", "bat", "cat", "hat", "mat", "rat")
```

Which command below produces the following vector?

- ## [1] "at" "bat" "cat" "hat"
- (a) `str_subset(v, "[^bch]+at")`
- (b) `str_subset(v, "^bch+at")`
- (c) `str_subset(v, "bch?at")`
- (d) `str_subset(v, "bch?at")`

(III) Here is part of the R documentation for `str_count()`.

str_count {stringr}	R Documentation
<p>Count the number of matches in a string.</p> <p>Description</p> <p>Vectorised over <code>string</code> and <code>pattern</code>.</p> <p>Usage</p> <pre>str_count(string, pattern = "")</pre> <p>Arguments</p> <p><code>string</code> Input vector. Either a character vector, or something coercible to one.</p> <p><code>pattern</code> Pattern to look for. The default interpretation is a regular expression, as described in stringi::stringi-search-regex. Control options with regex(.).</p>	

Judging by this description, what is the return value of the following command?

```
str_count("Here is a sentence.", pattern = ".")
```

- (a) ## [1] 1
- (b) ## [1] 4
- (c) ## [1] 16
- (d) ## [1] 19

- (IV) Suppose we have created a character vector called `movies` that contains information about movie titles and years:

```
movies
## [1] "Godzilla vs. Kong (2021)"      "The Unholy (2021)"
## [3] "French Exit (2020)"            "Voyagers (2021)"
## [5] "Mortal Kombat (2021)"          "Monday (2020)"
## [7] "Vanquish (2021)"              "The Asset (2021)"
## [9] "Wrath of Man (2021)"           "Here Are the Young Men (2020)"
```

Note that all titles have the year added in parentheses at the end. How can we remove the year so that we obtain the following vector?

```
## [1] "Godzilla vs. Kong"      "The Unholy"      "French Exit"
## [4] "Voyagers"                "Mortal Kombat"   "Monday"
## [7] "Vanquish"                "The Asset"       "Wrath of Man"
## [10] "Here Are the Young Men"

(a) str_subset(movies, "\\d{4}")
(b) str_split(movies, " \\(")
(c) str_replace(movies, "(....)", "")
(d) str_sub(movies, end = -8)
```

- (V) Suppose we saved the names of various files in a character vector called `files`.

```
files
## [1] "access.lock"        "documentation.html"  "favicon.gif"
## [4] "img0912.jpg"         "img0912.jpg.tmp"     "updated_img0912.png"
## [7] "workspace.docx"
```

We want to extract the names of all image files. Image files are defined as those ending with the extensions ".jpg", ".png" or ".gif". We exclude temporary files ending in ".tmp". We also want to remove the extensions of the image files so that the output is as follows.

```
img_file_names
## [1] "favicon"           "img0912"           "updated_img0912"
```

Which regular expression `regex` must replace the ellipsis (...) in the following code chunk to produce the correct vector `img_file_names`?

```
regex <- ...
img_file_names <-
  files |>
  str_subset(regex) |>
  str_replace(regex, "\\\1")  
  
(a) "\\w+\\.(:gif|jpg|png)$"
(b) "(\\w+)\\.(:gif|jpg|png)"
(c) "(\\w+)\\.(:gif|jpg|png)$"
(d) "\\w+\\.(:gif|jpg|png)"
```