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).

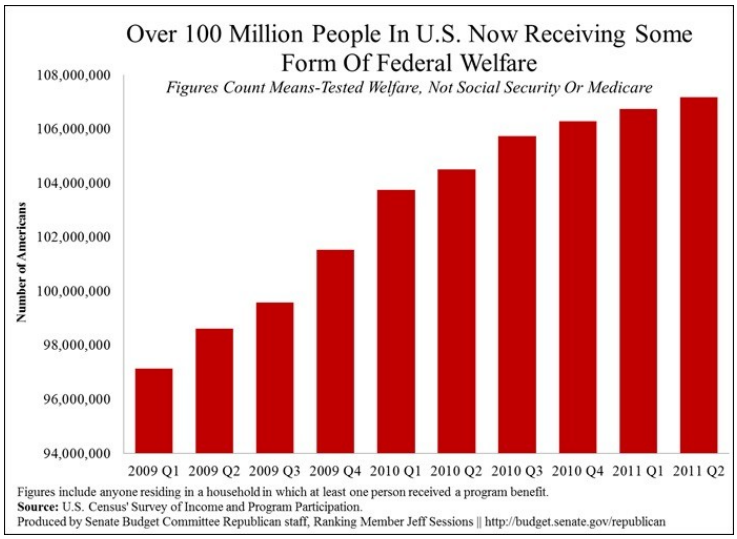


FIGURE 1.1: This diagram (Halper, 2012) is misleading because the y-axis (i.e. the vertical axis) does not start from zero.

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

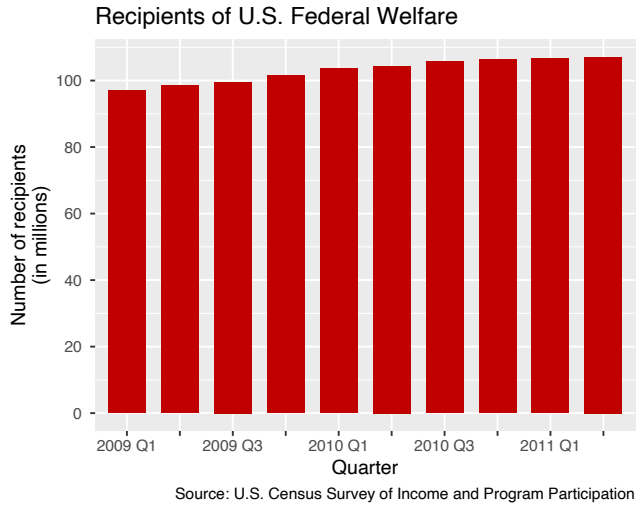


FIGURE 1.2: Unlike figure 1.1, this diagram satisfies the area principle because the y-axis starts from zero.

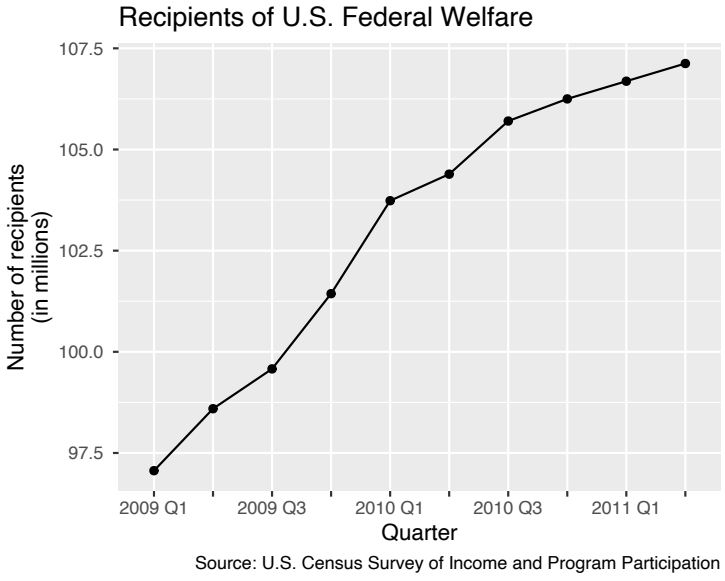


FIGURE 1.3: If we want to use a y-axis range that is close to the range of the data, we should not use a bar chart but a point-to-point chart.

recipients, and each plot has its strengths and weaknesses. While figure 1.2 emphasises the total number or 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

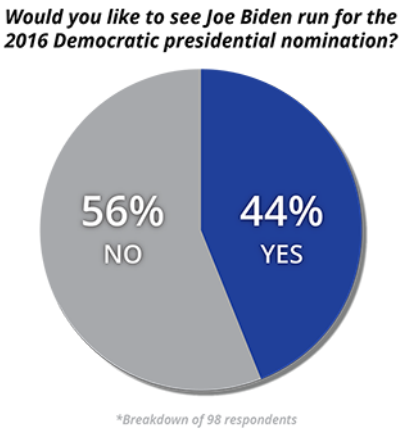


FIGURE 1.4: This pie chart (Barnes, 2015) contains little quantitative information. It would be more efficient to present the data as text instead of a diagram.

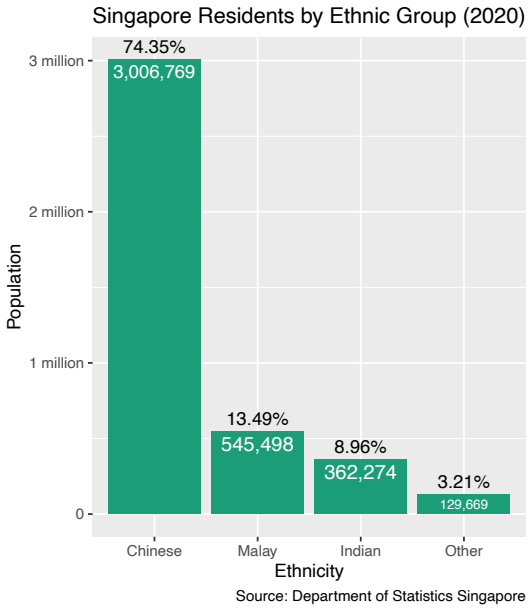


FIGURE 1.5: This bar chart contains four independent pieces of quantitative data (one number for each ethnic group). In this case, a bar chart is better than long-winded text. The percentage values shown above each bar are not strictly necessary, but they are a reader-friendly addition.

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

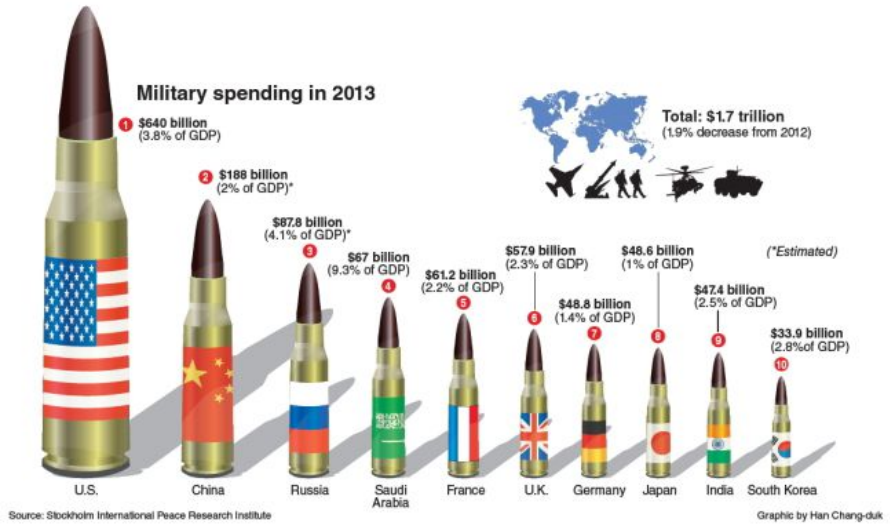


FIGURE 1.7: This plot (Chang-Duk, 2014) has artistic value, but it does not communicate data effectively.

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 **gganimate** 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.

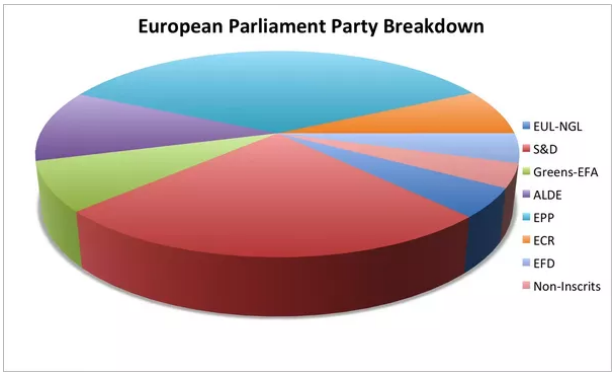


FIGURE 1.9: Diagram by Hickey (2013).

(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?

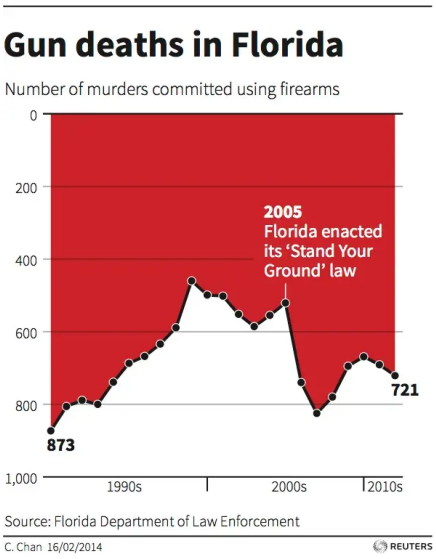


FIGURE 1.10: Diagram by Chan (2014).

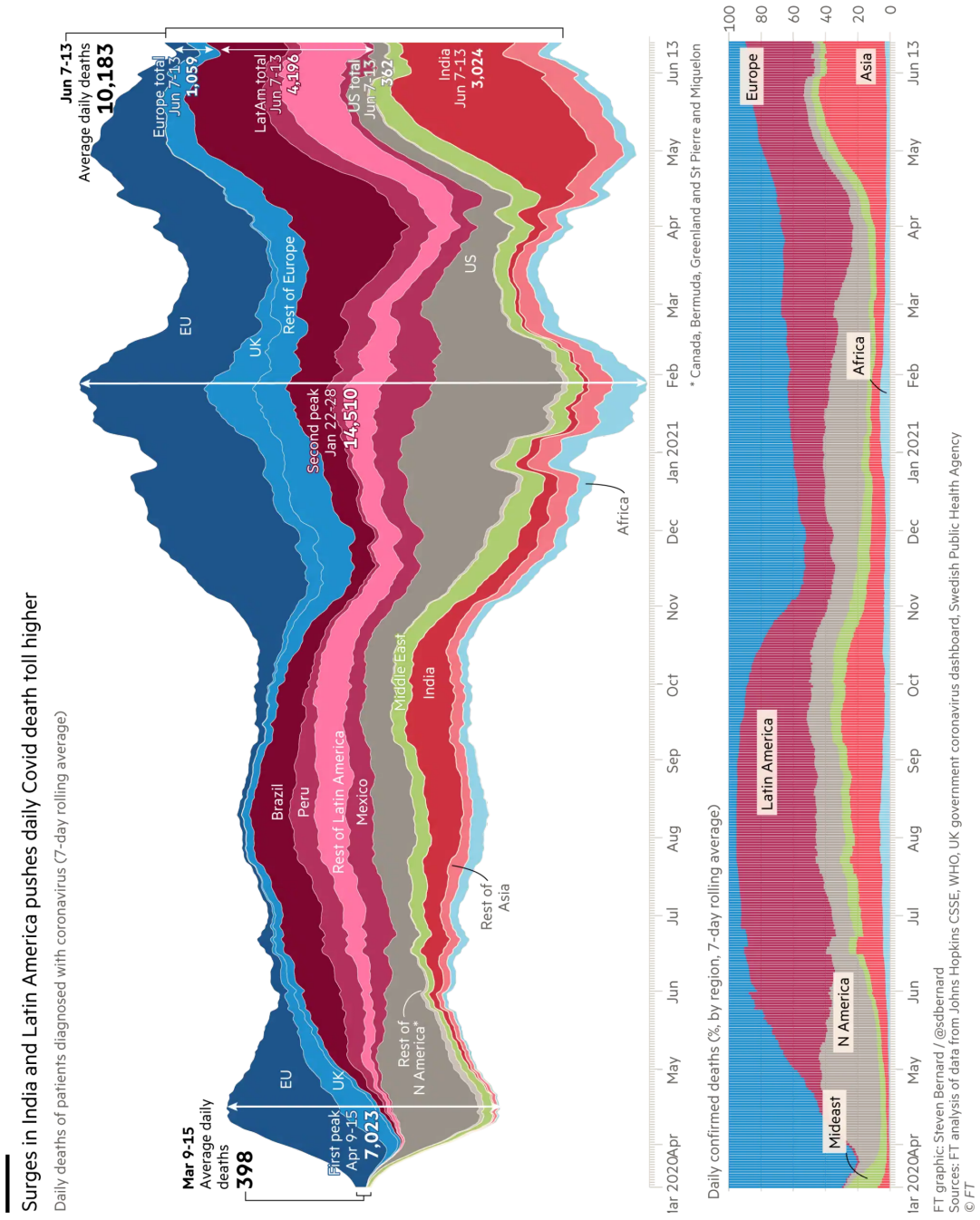


FIGURE 1.11: Diagram by Bernard (2020).

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,

[1] There 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-R[2] for Windows or RKWard[3] for Linux. At present, by far the most popular IDE for R is RStudio[4] 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.

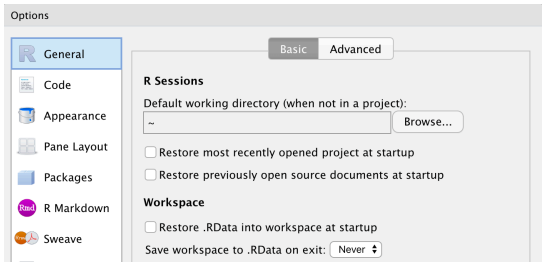


FIGURE 2.1: Recommended global options. The top three boxes are unticked, and we do not save `.Rdata` on exit.

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'. 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.

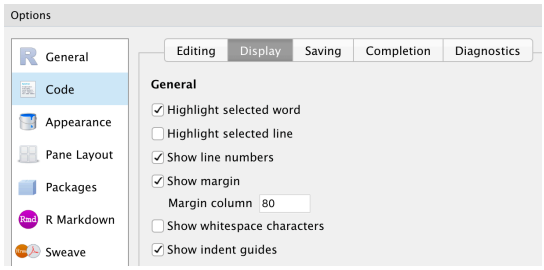


FIGURE 2.2: Recommended editor settings.

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.

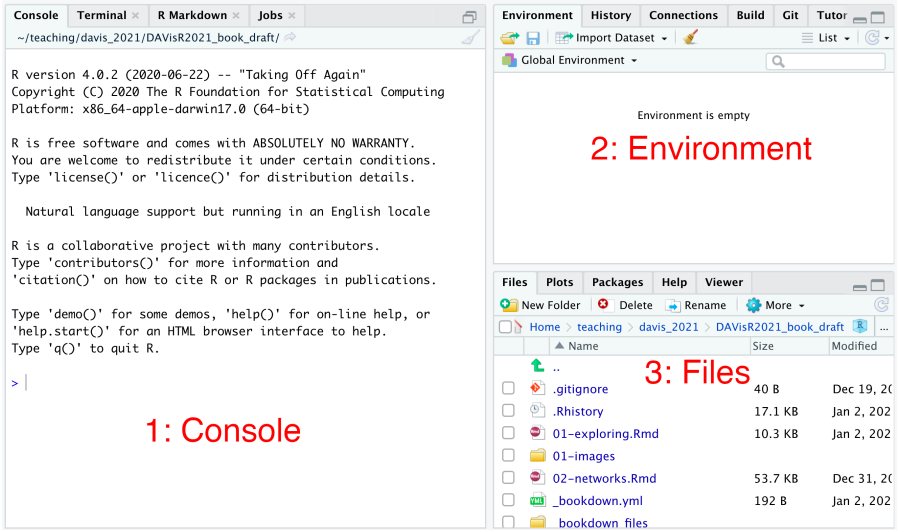


FIGURE 2.3: The RStudio window is by default divided into three panes: console, environment and files.

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.

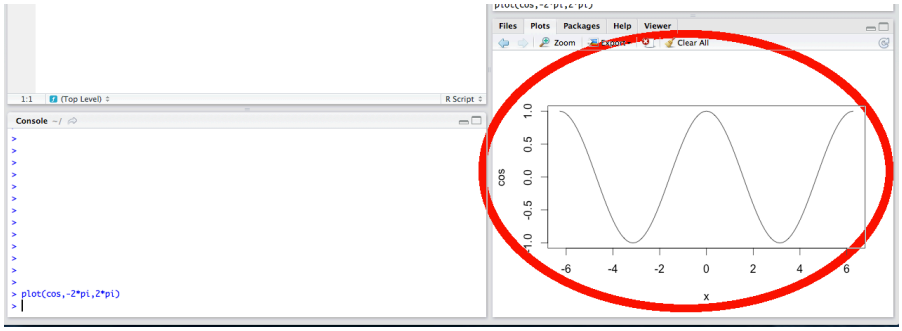


FIGURE 2.4: In RStudio, plots appear in the bottom right pane. By clicking on the tab names at the top of the pane (e.g. 'Files' or 'Plots'), you can navigate between different tabs.

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'.

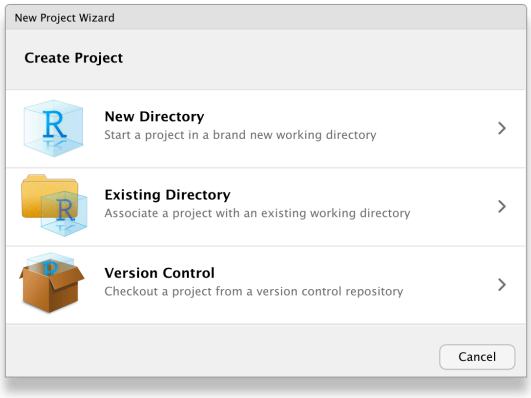


FIGURE 2.5: Pop-up window that appears when you start a new project.

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.

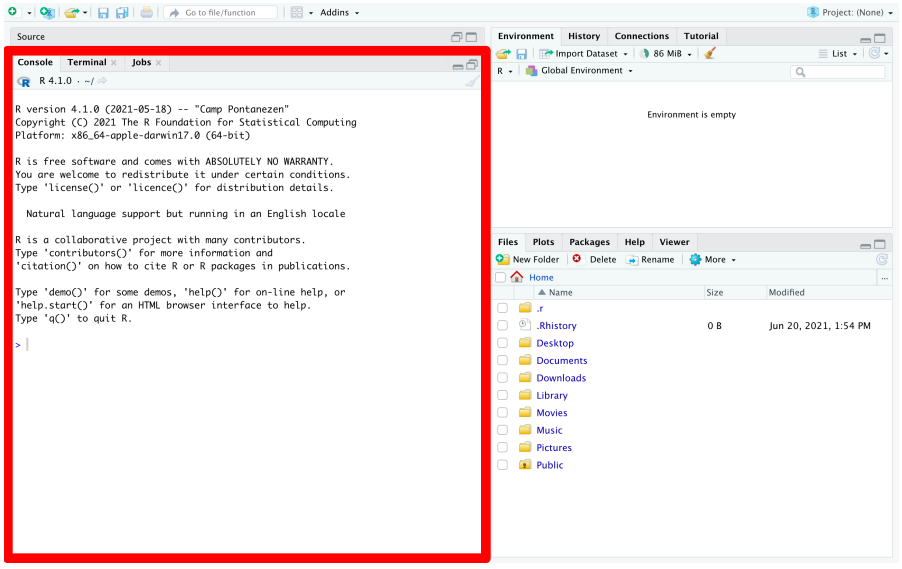


FIGURE 2.6: RStudio interface with left pane highlighted in red.

(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).

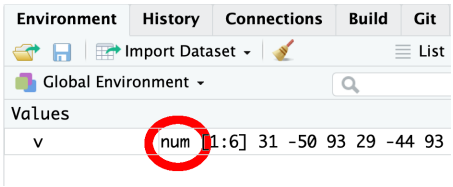


FIGURE 3.1: RStudio displays an abbreviation of a vector's class in the environment tab.

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).

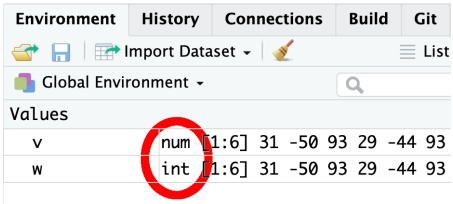


FIGURE 3.2: The abbreviations `num` and `int` in the environment tab show that `v` and `w` are of different classes: `numeric` versus `integer`.

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

[1] The 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.

[2] As we learn in section 11.1, logical vectors can also contain missing values in the form of `NA` (*N*ot *A*ssigned).

[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.

[4] In 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).

[5] Technically, 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.

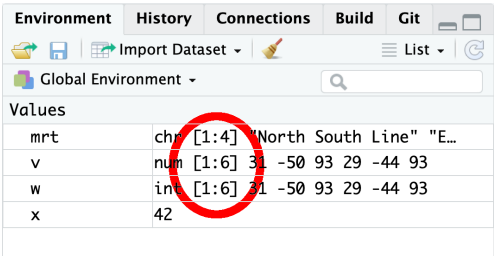


FIGURE 3.3: In the environment tab, `[1:4]` indicates that the elements have indices from 1 to 4 (i.e. the length of the vector is 4; note that indices in R start from `1`, not `0`). For vectors of length 1 (e.g. `x` in the depicted example), RStudio does not explicitly show the length, but it is obvious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

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

Like the basic arithmetic operators of section 3.4.1, all of these mathematical functions are vectorised: if we insert arguments with multiple elements, 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 $(v_1, \ldots, v_n)$, defined as $\bar{v} = \left(\sum_{i=1}^{n} v_i\right)/n$, 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 $\text{var}(v_1, \ldots, v_n) = \left(\sum_{i=1}^{n}(v_i - \bar{v})^2\right)/(n-1)$ 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.
## -15.000  -6.000   4.500   1.833   8.250  17.000
```

[6] If `q` is not a multiple of `1 / (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.

[7] There 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 `table()`'s argument.

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 values of the argument are `TRUE`.

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

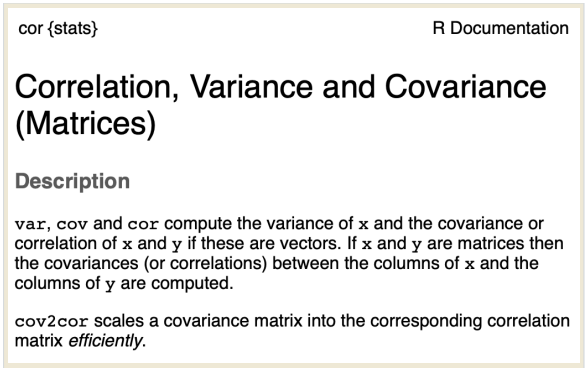
The help page opens in RStudio's bottom right pane. Let us take a look at the information shown in this pane.

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 called `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.



cor {stats} R Documentation

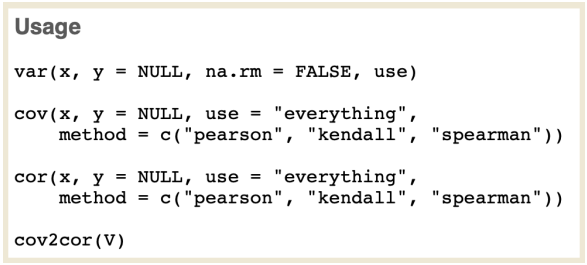
Correlation, Variance and Covariance (Matrices)

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

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



Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

FIGURE 4.2: In the 'Usage' section of R's built-in help pages, there is a summary of the arguments with which we call the functions.

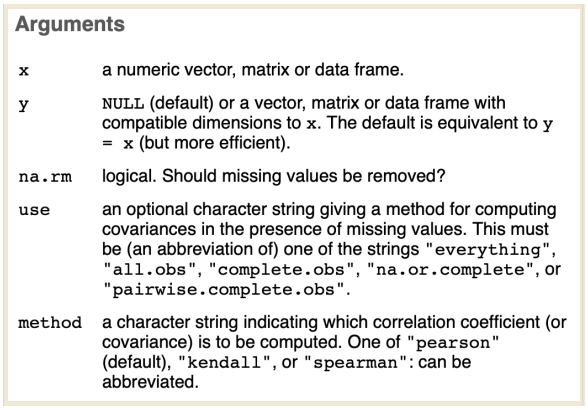
The third section, *Arguments*, shows more information about the arguments (e.g. their data types and data structures). The text also indicates what the arguments mean (figure 4.3).

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).

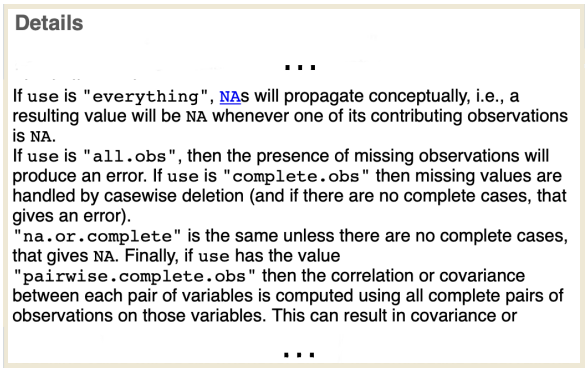
An example often explains more than many words. For this reason, almost every built-in help document in R ends with examples. We can copy and



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 `y = x` (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.



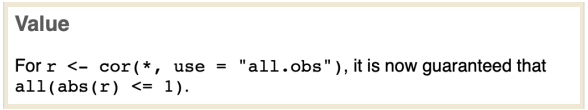
Details

...

If `use` is `"everything"`, NAs will propagate conceptually, i.e., a resulting value will be `NA` whenever one of its contributing observations is `NA`.
If `use` is `"all.obs"`, then the presence of missing observations will produce an error. If `use` 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 `NA`. Finally, if `use` 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 `r <- cor(*, use = "all.obs")`, it is now guaranteed that `all(abs(r) <= 1)`.

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.

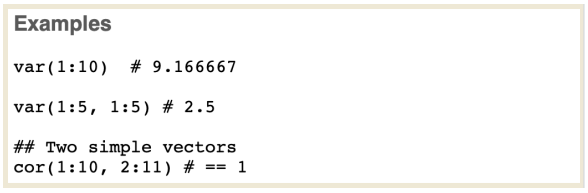


FIGURE 4.6: Examples at the end of a help page illuminate how a function can be used.

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-help[2] 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 set the arguments of 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 to the function `append()`: `x`, `values` and `after`. The last argument is followed by `=` in the Usage information, whereas there is no `=` after the first and second arguments. 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)
```

[1] We 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()`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.

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).

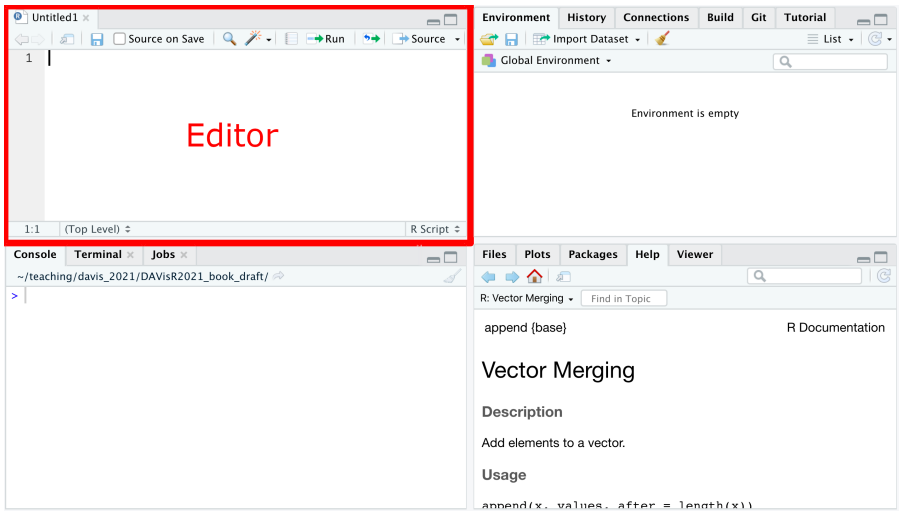


FIGURE 6.1: By default, the editor opens in the top left pane of the RStudio window.

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

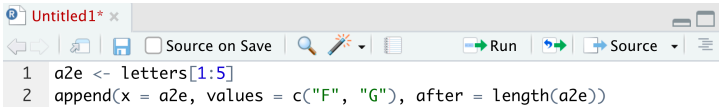


FIGURE 6.2: A simple script with only two commands.

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.

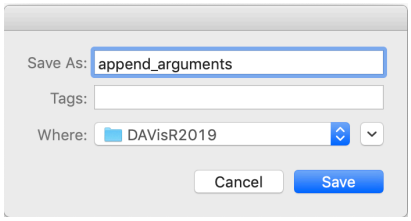


FIGURE 6.3: The 'Save File' dialogue box.

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).

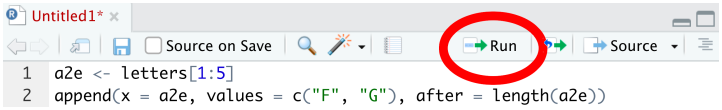


FIGURE 6.4: The 'Run' button is highlighted by the red ellipse.

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.

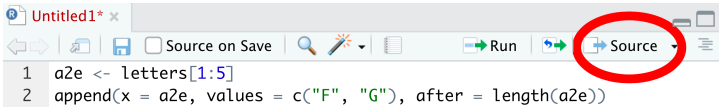


FIGURE 6.5: The 'Source' button is highlighted by the red ellipse.

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. We can confirm this observation with the operator `%in%` which checks whether the value to its left is in the vector to its right.

```
> "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 run `print()` directly with the argument `append(x = a2e, values = c("something", "in", "between"), after = 2)`.

```
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 inserted into 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 with 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.

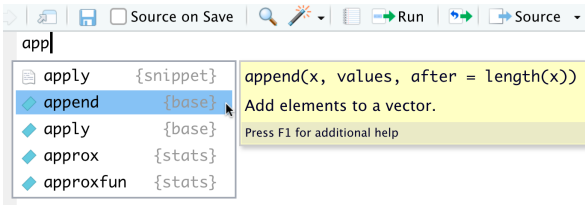


FIGURE 6.6: RStudio shows code suggestions when we type the first few letters of a function or variable name.

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

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

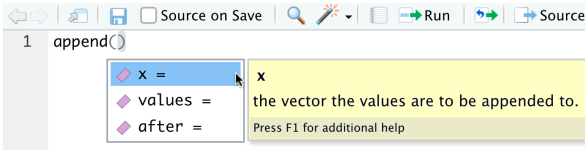


FIGURE 6.7: RStudio shows the arguments of a function in a popup menu.

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).

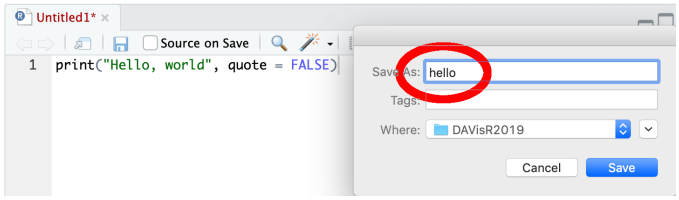


FIGURE 6.8: A 'Hello, world' program is a rite of passage when learning any programming language. Here is the R version.

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.

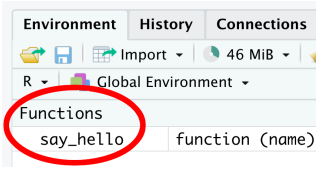


FIGURE 6.9: Besides vectors, the global environment can also contain functions.

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

```
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`. 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 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 'con*cat*enate 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.