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).

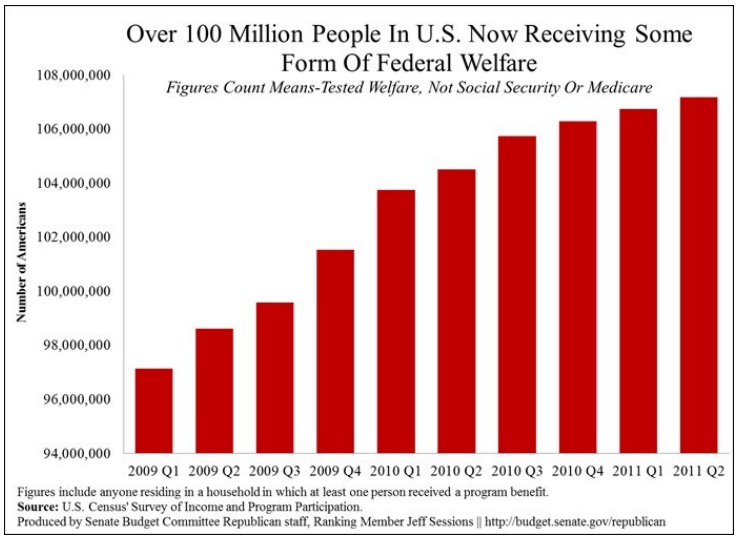

FIGURE 1.1: This diagram (Halper, 2012) is misleading because the y-axis (i.e. the vertical axis) does not start from zero.

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

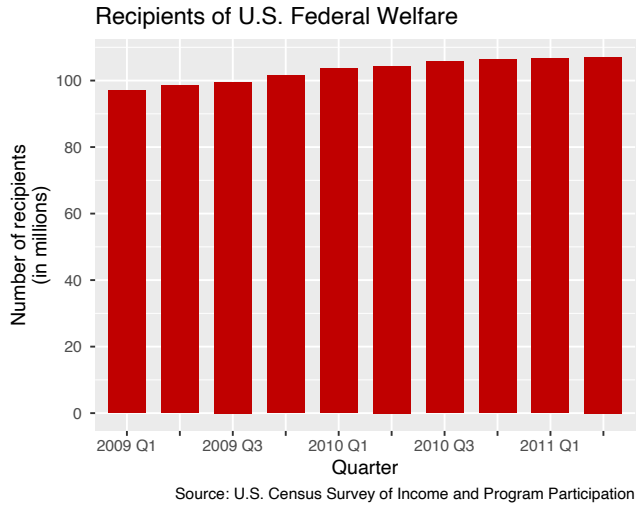

FIGURE 1.2: Unlike figure 1.1, this diagram satisfies the area principle because the y-axis starts from zero.

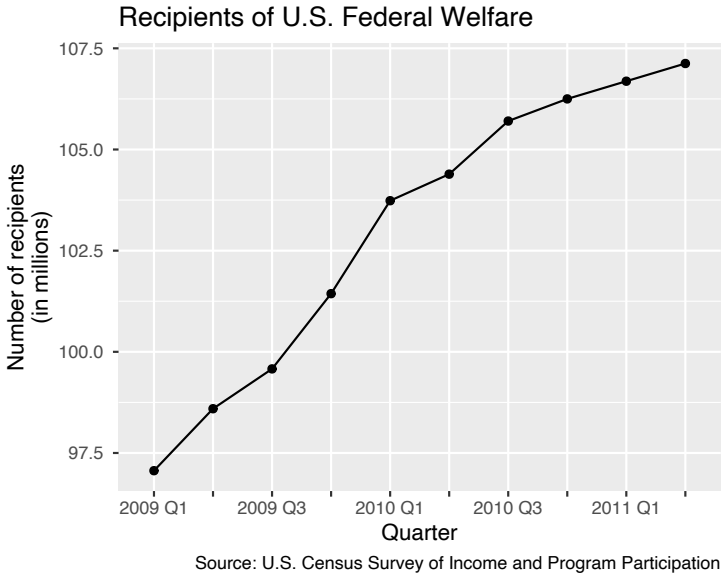

FIGURE 1.3: If we want to use a y-axis range that is close to the range of the data, we should not use a bar chart but a point-to-point chart.

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

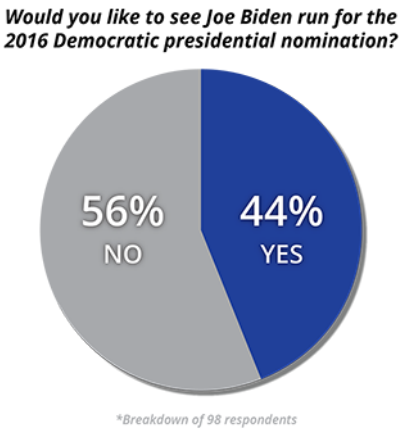

FIGURE 1.4: This pie chart (Barnes, 2015) contains little quantitative information. It would be more efficient to present the data as text instead of a diagram.

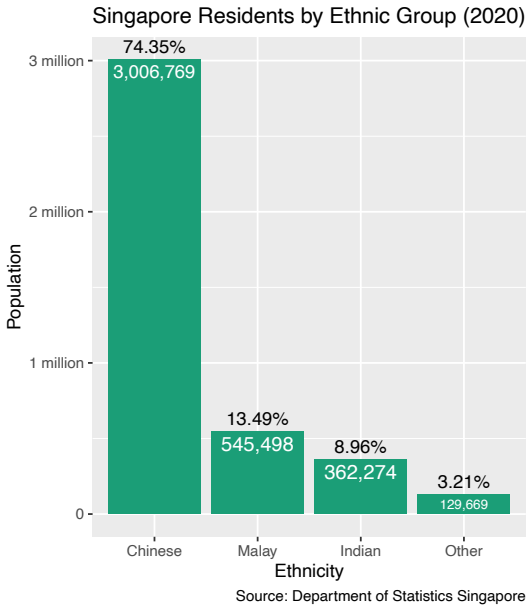

FIGURE 1.5: This bar chart contains four independent pieces of quantitative data (one number for each ethnic group). In this case, a bar chart is better than long-winded text. The percentage values shown above each bar are not strictly necessary, but they are a reader-friendly addition.

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 application exercises of chapter 7.

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

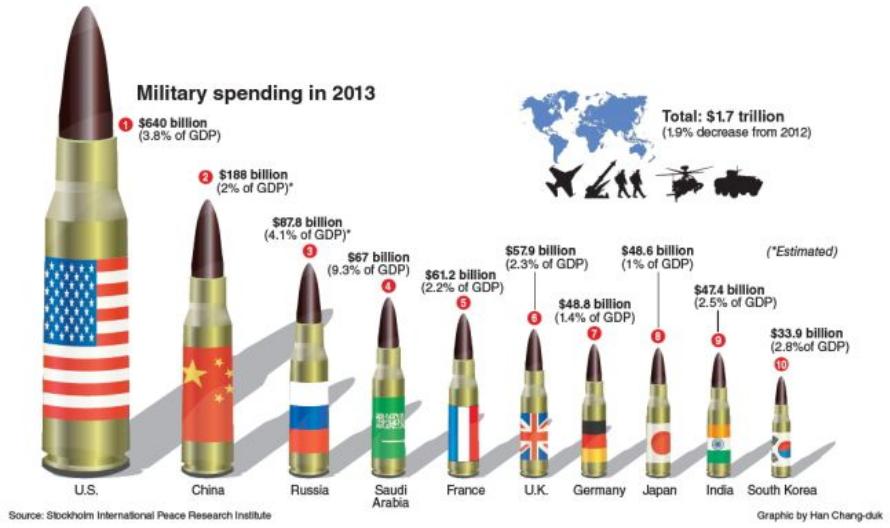

FIGURE 1.7: This plot (Chang-Duk, 2014) has artistic value, but it does not communicate data effectively.

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 XII. 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.

Application 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.

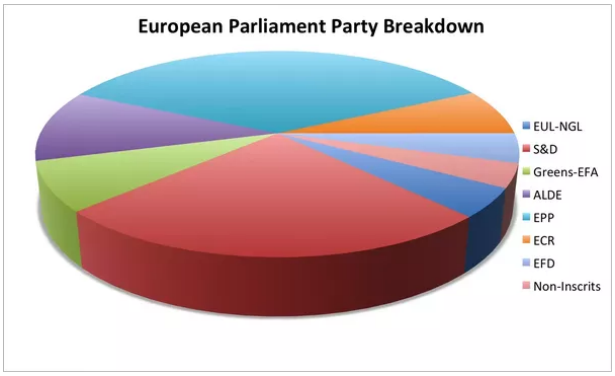

FIGURE 1.9: Diagram by Hickey (2013).

(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?

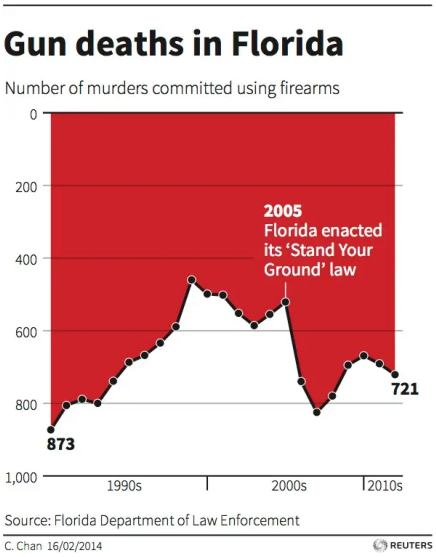

FIGURE 1.10: Diagram by Chan (2014).

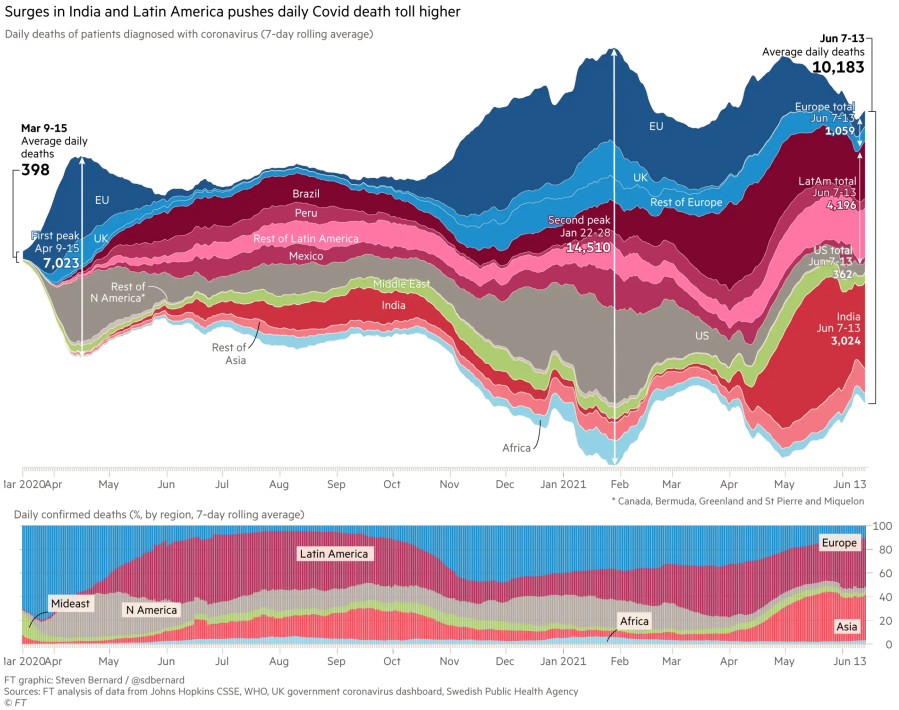

FIGURE 1.11: Diagram by Bernard (2020).

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 beer' (i.e. there is no cost to the users).
- R is free 'as in 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 beer and as in 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.

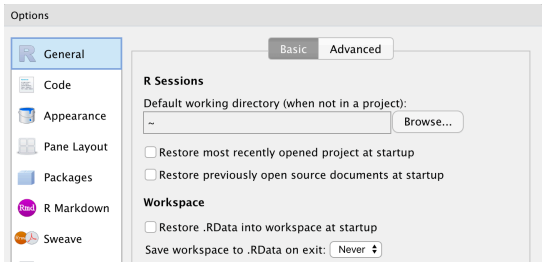

FIGURE 2.1: Recommended global options. The top three boxes are unticked, and we do not save `.Rdata` on exit.

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.

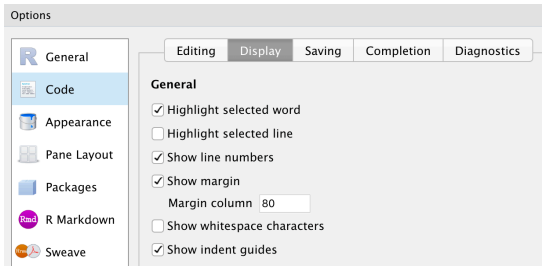

FIGURE 2.2: Recommended editor settings.

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.

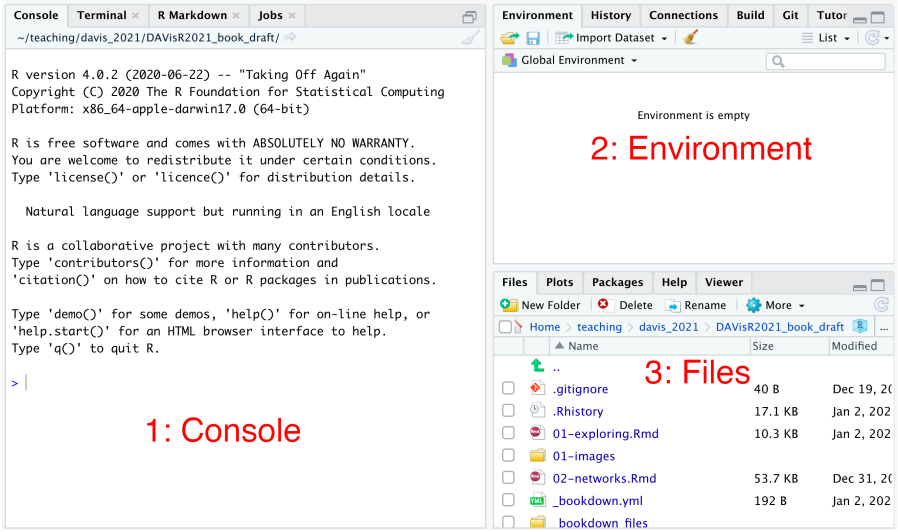

FIGURE 2.3: The RStudio window is by default divided into three panes: console, environment and files.

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.

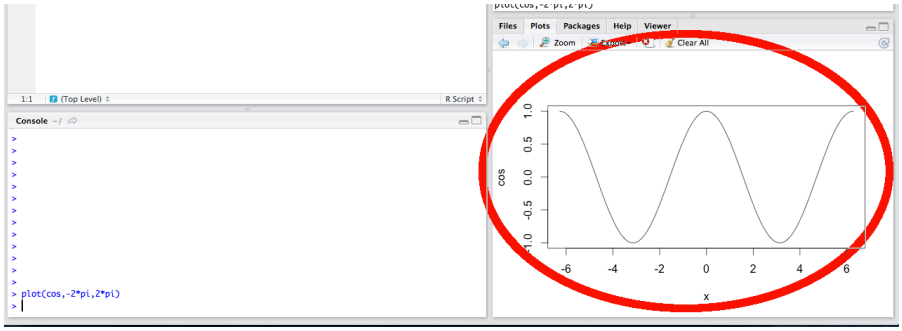

FIGURE 2.4: In RStudio, plots appear in the bottom right pane. By clicking on the tab names at the top of the pane (e.g. 'Files' or 'Plots'), you can navigate between different tabs.

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'.

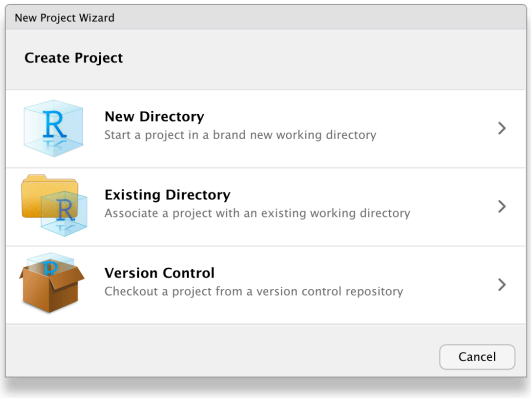

FIGURE 2.5: Pop-up window that appears when you start a new project.

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.

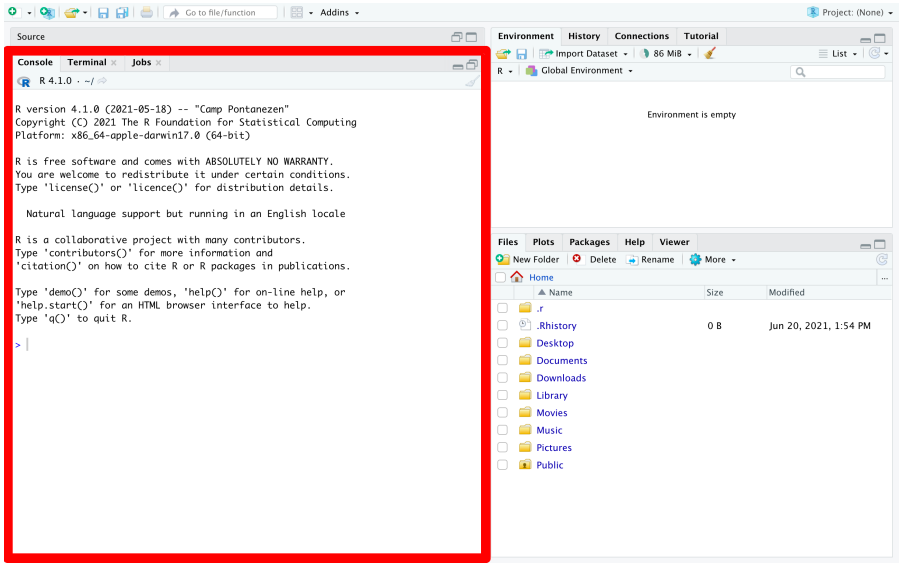

FIGURE 2.6: RStudio interface with left pane highlighted in red.

(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).

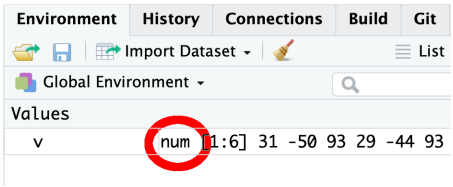

FIGURE 3.1: RStudio displays an abbreviation of a vector's class in the environment tab.

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).

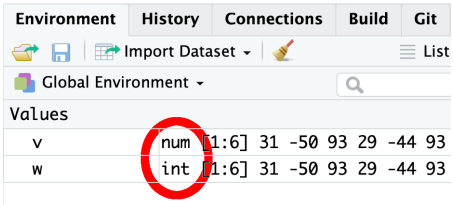

FIGURE 3.2: The abbreviations `num` and `int` in the environment tab show that `v` and `w` are of different classes: `numeric` versus `integer`.

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.

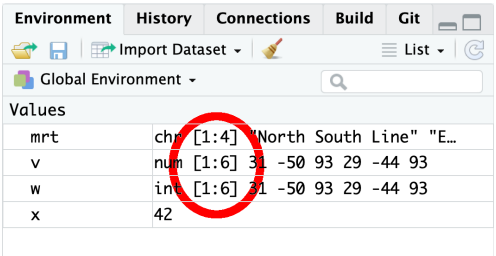

FIGURE 3.3: In the environment tab, `[1:4]` indicates that the elements have indices from 1 to 4 (i.e. the length of the vector is 4; note that indices in R start from `1`, not `0`). For vectors of length 1 (e.g. `x` in the depicted example), RStudio does not explicitly show the length, but it is obvious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