

Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures	vii
Preface	ix
Using the book to learn R	xiii
1 R: The language and the program	1
1.1 Aims of this chapter	1
1.2 Computer programming	1
1.3 R	2
1.3.1 What is R?	2
1.3.2 R as a language	4
1.3.3 R as a computer program	4
1.3.3.1 R sessions and workspaces	5
1.3.3.2 Using R interactively	6
1.3.3.3 Using R in a “batch job”	7
1.3.3.4 Editors and IDEs	8
1.4 Reproducible data analysis	11
1.5 Finding additional information	12
1.5.1 R’s built-in help	13
1.5.2 Obtaining help from online forums	14
1.5.2.1 Netiquette	14
1.5.2.2 StackOverflow	15
1.5.2.3 Reporting bugs	15
1.6 What is needed to run the examples in this book?	16
1.7 Further reading	17
2 The R language: “Words” and “sentences”	19
2.1 Aims of this chapter	19
2.2 Natural and computer languages	20
2.3 Numeric values and arithmetic	20
2.4 Logical values and Boolean algebra	31
2.5 Comparison operators and operations	33
2.6 Sets and set operations	38
2.7 Character values	42
2.8 The ‘mode’ and ‘class’ of objects	43
2.9 ‘Type’ conversions	44
2.10 Vector manipulation	48
2.11 Matrices and multidimensional arrays	54
2.12 Factors	59

2.13 Lists	65
2.13.1 Member extraction and subsetting	65
2.13.2 Adding and removing list members	67
2.13.3 Nested lists	67
2.14 Data frames	70
2.14.1 Operating within data frames	77
2.14.2 Re-arranging columns and rows	81
2.14.3 Re-encoding or adding variables	82
2.15 Attributes of R objects	85
2.16 Saving and loading data	87
2.16.1 Data sets in R and packages	87
2.16.2 .rda files	87
2.16.3 .rds files	89
2.17 Looking at data	90
2.18 Plotting	92
2.19 Further reading	95
Bibliography	97
General index	99
Index of R names by category	103
Alphabetic index of R names	105

List of Figures

1.1	The R console	5
1.2	The R console	6
1.3	The R console	7
1.4	Script sourced at the R console	8
1.5	Script at the Windows cmd prompt	9
1.6	Script in Rstudio	10



Preface

“Suppose that you want to teach the ‘cat’ concept to a very young child. Do you explain that a cat is a relatively small, primarily carnivorous mammal with retractable claws, a distinctive sonic output, etc.? I’ll bet not. You probably show the kid a lot of different cats, saying ‘kitty’ each time, until it gets the idea. To put it more generally, generalizations are best made by abstraction from experience.”

R. P. Boas

Can we make mathematics intelligible?, 1981

Why have I chosen the title “*Learn R: As a Language*”? This book is based on exploration and practice that aims at teaching how to express various operations on data using the R language. It focuses on the language, rather than on specific types of data analysis, and exposes the reader to current usage and does not spare the quirks of the language. When we use our native language in everyday life, we do not think about grammar rules or sentence structure, except for the trickier or unfamiliar situations. My aim is for this book to help readers grow to use R in this same way, i.e., to become fluent in R. The book is structured around the elements of languages with chapter titles that highlight the parallels between natural languages like English and the R language.

Nowadays, many students of biological and environmental sciences learn R in courses about statistics or data analysis. However, frequently not in enough depth to effectively use it in scripts for automating data analyses or documenting the whole data analysis workflow to ensure reproducibility. Students in the humanities and also in other fields, may find it easier to learn the R language separately from data analysis and statistics. There are also many who are already familiar with statistical principles and willing to switch from other software to R. *Learn R: As a Language* is written with these readers in mind to serve both as a text book and as a reference.

A language is a system of communication. Basic concepts and operations are based on abstractions that are shared across programming languages and relevant to programs of all sizes and complexities; these abstractions are explained in the book together their implementation in the R language. Other abstractions and programming concepts, outside the scope of this book, are relevant to large and complex pieces of software meant to be widely distributed. In other words, *Learn R: As a Language* aims at teaching and supporting *programming in the small*: the

use of R to automate the drudgery of data manipulation, including the different steps spanning from data input and exploration to the production of publication-quality illustrations and their documentation.

Using a language actively is the most efficient way of learning it. By using it, I mean actually reading, writing, and running scripts or programs. *Learn R: As a Language* supports learning the R language in a way comparable to how children learn to speak: they work out what the rules are, simply by listening to people speak and trying to utter what they want to tell their parents. Of course, small children receive some guidance, but they are not taught a prescriptive set of rules like when learning a second language at school. Instead of listening, readers will read and run code, and instead of speaking, readers will write and try to execute R code statements on a computer. I do provide explanations and guidance, but the idea of this book is for readers to play with the numerous examples by creating new variations upon them, to find out by themselves the patterns behind the R language. Instead of parents being the sounding board for the first utterances of readers new to R, the computer will play this role.

This revised second edition reflects changes that took place in R and packages described. Very few code chunks from the first edition had stopped working but deprecations meant that some examples triggered messages or warnings, and will eventually fail. Recent (> 4.0.0) versions of R have significant enhancements such as the new pipe operator. Packages have also evolved acquiring new features. Feedback from readers and reviewers has highlighted some gaps in the contents and unclear explanations. Re-reading myself the book after some time allowed me to think of other improvements. I have updated the book accordingly. I have added diagrams and flowcharts to facilitate comprehension programming concepts. I edited the text from the first edition to fix all errors and outdated examples or explanations known to me and to improve the clarity of previously unclear explanations.

I encourage you to approach R like a child approaches his or her mother tongue when first learning to speak: do not struggle, just play, and fool around with R! If the going gets difficult and frustrating, take a break! If you get a new insight, take a break to enjoy the victory!

Acknowledgements

I thank Jaakko Heinonen for introducing me to the then new R. Along the way many well known and not so famous experts have answered my questions in usenet and more recently in Stackoverflow. I wish to warmly thank members of my own research group, students participating in the courses I have taught, colleagues I have collaborated with, authors of the books I have read and people I have only met online or at conferences. All of them have made it possible for me to write this book. I am indebted to Tarja Lehto, Titta Kotilainen, Tautvydas Zalnierius, Fang Wang, Yan Yan, Neha Rai, Markus Laurel, Brett Cooper, colleagues, students and anonymous reviewers for many very helpful comments on the draft manuscript and/or the published first edition. Rob Calver, editor of both editions, provided

encouragement with great patience, Lara Spieker, Vaishali Singh, and Paul Boyd for their help with different aspects of this project.

In many ways this text owes much more to people who are not authors than to myself. However, as I am the one who has written *Learn R: As a Language* and decided what to include and exclude, I take full responsibility for any errors and inaccuracies.

Helsinki, January 17, 2023



Using the book to learn R

Few people like problems. Hence the natural tendency in problem-solving is to pick the first solution that comes to mind and run with it. ... A better strategy ... is to select the most attractive path from many ideas, or concepts.

J. L. Adams
Conceptual blockbusting, 1987

Approach and structure


Depending on previous experience, reading *Learn R: As a Language* will be about exploring a new world or revisiting a familiar one. In both cases *Learn R: As a Language* aims to be a travel guide, neither a traveler's account, nor a cookbook of R recipes. It can be used as a course book, supplementary reading or for self instruction, and also as a reference.


In R, like in most “rich” languages, there are multiple ways of coding the same operations. I have included code examples that aim to strike a balance between execution speed and readability. One could write equivalent R books using substantially different code examples. Keep this in mind when reading the book and using R. Keep also in mind that it is impossible to remember everything about R and as a user you will frequently need to consult the documentation, even while doing the exercises in this book. The R language, in a broad sense, is vast because it can be expanded with independently developed packages. Learning to use R mainly consists of learning the basics plus developing the skill of finding your way in R, its documentation and on-line question and answer forums.


The contents of the book are organized so that it can be used both as a text book for learning R and as a reference. It starts with simple concepts and language elements progressing towards more complex language structures and uses. Along the way readers will find, in each chapter, descriptions and examples for the common (usual) cases and the exceptions. Some books hide the exceptions and counterintuitive features from learners to make the learning easier, I instead have included these but marked using icons and marginal bars. There are two reasons for choosing this approach. First, the boundary between boringly easy and


frustratingly challenging is different for each of us, and varies depending on the subject dealt with. So, I hope the marks will help readers predict what to expect, how much effort to put in each section and even what to read or skip. Second, if I had hidden the tricky bits of the R language, I would have made reader's later use of R more difficult. It would have also made the book less useful as a reference.


The key to the marginal bars and icons is given next. They inform about what content is advanced or included with a specific aim.

 Signals text providing general information not directly related to the R language.

 Signals in-depth explanations of specific points that may require you to spend time thinking, which in general can be skipped on first reading, but to which you should return at a later peaceful time, preferably with a cup of coffee or tea.

 Signals important bits of information that must be remembered when using R—i.e., explain some unusual feature of the language.

 Signals *playground* sections which contain open-ended exercises—ideas and pieces of R code to play with at the R console. Most code chunks show only the R code listing. Readers are expected to run this code both as is and modified, study the output returned by R for each variation.

 Signals *advanced playground* sections which will require more time to play with before grasping concepts than regular *playground* sections.

Readers new to R should read at least chapters 1 to ?? sequentially. Possibly, starting by reading parts and doing exercises not marked as advanced. However, I expect to be most useful to these readers, not to completely skip the description of unusual features and special cases, but rather to skim enough from them so as to get an idea of what special situations they may face as R users. Playground exercises should not be skipped, as they are a key component of the didactic approach used.

Readers already familiar with R will be able to read the chapters in the book in any order, as need arises. In the long run, I expect *Learn R: As a Language* to remain useful as a reference to those using it as a textbook, both for refreshing the mainstream features and to deal with the oddities and quirks of the language. To make its use as a reference easy, I have been thorough with indexing, including many carefully chosen terms, their synonyms and the names of all R objects and constructs discussed, collecting them in three alphabetical indexes: *General index*, *Index of R names by category*, and *Alphabetic index of R names* starting at pages 101, 104 and ??, respectively. I have also included cross references that link related sections.

Readers should not aim at remembering all the details presented in the book. Using this and other books, and documentation effectively as references, depends on a good grasp of the larger picture of R and how to navigate the documentation; i.e., it is more important to remember abstractions and in what situations they are

used, and function names, than the details of how to use them. Developing a sense of when one needs to be careful not to fall in a “language trap” is also important.

Typography and syntax highlighting

I use the notation `<value>`, `<statement>`, etc., as a generic placeholder in diagrams indicating *any valid value*, *any valid R statement*, etc.

R code chunks are typeset in a typewriter font, and using colour to highlight the different elements of the syntax, such as variables, functions, constant values, etc. R code elements embedded in the text are similarly typeset but always black. For example in the “code chunk” below `mean()` and `print()` are functions; 1, 5 and 3 are constant numeric values, and `z` is the name of a variable where the result of the computation done in the first line of code is stored. The line starting with `##` shows what is printed to the screen when executing the second statement: `[1] 1`.

```
z <- mean(1, 5, 3)
print(z)
## [1] 1
```



1

R: The language and the program

In a world of ... relentless pressure for more of everything, one can lose sight of the basic principles—simplicity, clarity, generality—that form the bedrock of good software.

Brian W. Kernighan and Rob Pike
The Practice of Programming, 1999

1.1 Aims of this chapter

In this chapter you will learn some facts about the history and design aims behind the R language, its implementation in the R program, and how it is used in actual practice when sitting at a computer. You will learn the difference between typing commands interactively, reading each partial response from R on the screen as you type versus using R scripts to execute a “job” which saves results for later inspection by the user.

I will describe the advantages and disadvantages of textual command languages such as R compared to menu-driven user interfaces as frequently used in other statistics software and occasionally also with R. I will discuss the role of textual languages in the very important question of reproducibility of data analyses.

Finally you will learn about the different types and sources of help available to R users, and how to best make use of them.

1.2 Computer programming

We can distinguish two phases in the development of a computer program or script. The design phase is the initial step: deciding what the computer should do and what algorithms will be used. Coding is the second phase, and consists in translating a design into a given computer language, such as R. The distinction is not absolutely clear cut, as usually when programming one re-uses available code.

In a language like C++ we use libraries of routines, classes and templates. In R we use *packages* that provide extensions to the language. So, in most cases, the design stage for a data-analysis script in R centres, once the statistical procedure to use has been decided, in selecting what package, if any, to use, and the identification of the steps needed to import the data, possibly validate them, pass them to the functions in the packages used, and reporting the results either graphically or as text. In fact, most of the ad-hoc data-analysis code users write in their scripts is to transfer data among ready made “black boxes” and display the results. In contrast, writing new packages, requires much more effort towards design, of both computations and the interface to users’ code. In this book, we focus on the design of scripts and their coding.

Abstraction plays a central role in designing solutions suitable for families of similar problems. According to Wirth (1974) “Our most important mental tool for coping with complexity is abstraction. Therefore, a complex problem should not be regarded immediately in terms of computer instructions ... but rather in terms and entities natural to the problem itself, abstracted in some suitable sense.” Zimmer (1985) adds “Abstraction is the way we carry out a divide-and-conquer approach to the solution of complex problems.” A simple example of an abstraction centred on objects is the concept of *fruit* that describes properties shared among apples, oranges, pears, and many other fruits. An example of an abstraction centred on an action is the verb *show*, which depending on the context may signify different actions that share similar aims or purposes.

New concepts and styles of programming have appeared since Wirth and Zimmer wrote the texts quoted above and new terms are in use, but the role of abstraction remains as important. An important distinction is in the focus of the abstractions: actions vs. objects. These, oversimplifying things, give rise to procedural and object-oriented approaches (or paradigms) to computer programming, respectively. Which approach yields the most useful abstraction of a problems depends on the nature of the problem (see Coplien 1999). As we will see through the book, the R language is eclectic in this respect, and supports multiple approaches and their combined use. R itself relies quite heavily on a rather simple approach to object-oriented programming. When writing scripts, it is unusual to define new classes of objects, but in almost every script we make use of classes of objects and their corresponding methods, both defined in R and in extension packages. R also supports functional programming because functions are treated similarly to other objects and can be saved and operated upon. It is even possible in R to write functions that accept other functions as arguments and/or construct and return new functions dynamically.

1.3 R

1.3.1 What is R?

Most people think of R as a computer program. R is indeed a computer program—a piece of software— but it is also a computer language, implemented in the R

program. Does this make a difference? Yes. Until recently we had only one mainstream implementation of R, the program R. Recently another implementation has gained some popularity, Microsoft R Open (MRO), which is directly based on the R program from *The R Project for Statistical Computing*. MRO is described as an enhanced distribution of R. These two very similar implementations are not the only ones available, but others are not in widespread use. In other words, the R language can be used not only in the R program, and it is feasible that other implementations will be developed in the future.

The name “base R ” is used to distinguish R itself, as in the R executable included in the R distribution, from R in a broader sense, which includes packages. A few packages are included in the R distribution, while most R packages are independently developed extensions that can be loaded from separately distributed files.

Being that R is essentially a command-line application, it can be used on what nowadays are frugal computing resources, equivalent to a personal computer of three decades ago. R can run even on the Raspberry Pi, a micro-controller board with the processing power of a modest smart phone. At the other end of the spectrum, on really powerful servers, R can be used for the analysis of big data sets with millions of observations. How powerful a computer you will need will depend on the size of the data sets you want to analyze, on how patient you are, on your ability select efficient algorithms and to write “good” code.

One could think of R as a dialect of an earlier language, called S. S evolved into S-Plus (Becker et al. 1988). S and S-Plus are commercial programs, and variations in the language appeared only between versions. R started as a poor man’s home-brewed implementation of S, for use in teaching. Initially R, the program, implemented a subset of the S language. The R program evolved until only relatively few differences between S and R remained, and these differences are intentional—thought of as significant improvements. As R overtook S-Plus in popularity, some of the new features in R made their way back into S-Plus. R is free and open-source and the name Gnu S is sometimes used to refer to R.

What makes R different from SPSS, SAS, etc., is that S was designed from the start as a computer programming language. This may look unimportant for someone not actually needing or willing to write software for data analysis. However, in reality it makes a huge difference because R is easily extensible. By this we mean that new functionality can be easily added, and shared, and this new functionality is to the user indistinguishable from that built into R. In other words, instead of having to switch between different pieces of software to do different types of analyses or plots, one can usually find an R package that will provide the tools to do the job within R. For those routinely doing similar analyses the ability to write a short program, sometimes just a handful of lines of code, allows automation of routine analyses. For those willing to spend time programming, they have the door open to building the tools they need when these do not already exist.

However, the most important advantage of using a language like R is that it makes it easy to do data analyses in a way that ensures that they can be exactly repeated. In other words, the biggest advantage of using R, as a language, is not in communicating with the computer, but in communicating to other people what has been done, in a way that is unambiguous. Of course, other people may want to

run the same commands in another computer, but still it means that a translation from a set of instructions to the computer into text readable to humans—say the materials and methods section of a paper—and back is avoided together with the ambiguities usually creeping in.

1.3.2 R as a language

R is a computer language designed for data analysis and data visualization, however, in contrast to some other scripting languages, it is, from the point of view of computer programming, a complete language—it is not missing any important feature. In other words, no fundamental operations or data types are lacking (Chambers 2016). I attribute much of its success to the fact that its design achieves a very good balance between simplicity, clarity and generality. R excels at generality thanks to its extensibility at the cost of only a moderate loss of simplicity, while clarity is ensured by enforced documentation of extensions and support for both object-oriented and functional approaches to programming. The same three principles can be also easily respected by user code written in R.

As mentioned above, R started as a free and open-source implementation of the S language (Becker and Chambers 1984; Becker et al. 1988). We will describe the features of the R language in later chapters. Here I mention, for those with programming experience, that it does have some features that make it different from other frequently used programming languages. For example, R does not have the strict type checks of Pascal or C++. It has operators that can take vectors and matrices as operands allowing more concise program statements for such operations than other languages. Writing programs, specially reliable and fast code, requires familiarity with some of these idiosyncracies of the R language. For those using R interactively, or writing short scripts, these idiosyncratic features make life a lot easier by saving typing.

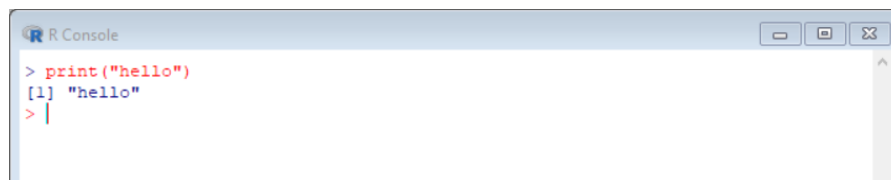


Some languages have been standardized, and their grammar has been formally defined. R, in contrast is not standardized, and there is no formal grammar definition. So, the R language is defined by the behavior of the R program.

1.3.3 R as a computer program

The R program itself is open-source, and the source code is available for anybody to inspect, modify and use. A small fraction of users will directly contribute improvements to the R program itself, but it is possible, and those contributions are important in making R reliable. The executable, the R program we actually use, can be built for different operating systems and computer hardware. The members of the R developing team make an important effort to keep the results obtained from calculations done on all the different builds and computer architectures as consistent as possible. The aim is to ensure that computations return consistent results not only across updates to R but also across different operating systems like Linux, Unix (including OS X), and MS-Windows, and computer hardware.

The R program does not have a graphical user interface (GUI), or menus from which to start different types of analyses. Instead, the user types the commands at the R console (Figure 1.1). The same textual commands can also be saved into

**FIGURE 1.1**

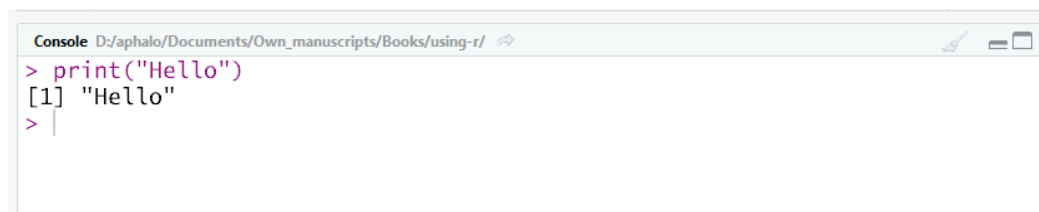
The R console where the user can type textual commands one by one. Here the user has typed `print("hello")` and *entered* it by ending the line of text by pressing the “enter” key. The result of running the command is displayed below the command. The character at the head of the input line, a “>” in this case, is called the command prompt, signaling where a command can be typed in. Commands entered by the user are displayed in red, while results returned by R are displayed in blue.

a text file, line by line, and such a file, called a “script” can substitute repeated typing of the same sequence of commands. When we work at the console typing in commands one by one, we say that we use R interactively. When we run script, we may say that we run a “batch job.”

The two approaches described above are part of the R program itself. However, it is common to use a second program as a front-end or middleman between the user and the R program. Such a program allows more flexibility and has multiple features that make entering commands or writing scripts easier. Computations are still done by exactly the same R program. The simplest option is to use a text editor like Emacs to edit the scripts and then run the scripts in R from within the editor. With some editors like Emacs, rather good integration is possible. However, nowadays there are also Integrated Development Environments (IDEs) available for R. An IDE both gives access to the R console in one window and provides a text editor for writing scripts in another window. Of the available IDEs for R, RStudio is currently the most popular by a wide margin.

1.3.3.1 R sessions and workspaces

We use *session* to describe the interactive execution from start to finish of one running instance of R. We use *workspace* to name the imaginary space where all objects currently available in a session are stored. In R the whole workspace can be stored in file on disk at the end or during a session and restored later into another session, possibly on a different computer. Usually when working with R we dedicate a folder in disk storage to store all files from a given data analysis project. We normally keep in this folder files with data to read in, scripts, a file storing the whole contents of the workspace, named by default `.Rdata` and a text file with the history of commands entered interactively, named by default `.Rhistory`. The user’s files within this folder can be located in nested folders. There are no strict rules on how the files should be organised or on their number. The recommended practice is to avoid crowded folders and folders containing unrelated files. It is a good idea to keep in a given folder and workspace the work in progress for a single data-analysis project or experiment, so that the workspace can be saved and restored easily between sessions and work continued from where one left it independently

**FIGURE 1.2**

The R console embedded in RStudio. The same commands have been typed in as in Figure 1.1. Commands entered by the user are displayed in purple, while results returned by R are displayed in black.

of work done on other workspaces. The folder where files are currently read and saved is in R documentation called the *current working directory*. When opening an `.Rdata` file the current working directory is automatically set to the folder where the `.Rdata` file is stored.

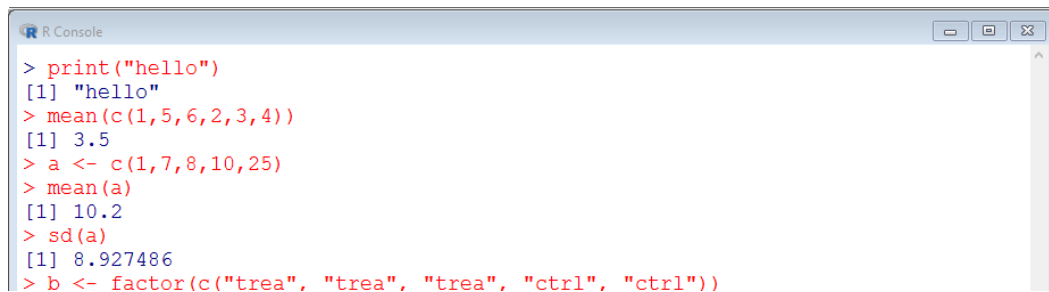
As described below, if we use a front-end program, it will save additional files, possibly a whole hierarchy of folders, to keep track of its own state and local settings between sessions. If we use a program like `git` to track our edits to R scripts and changes in other files in the folder, additional folders and files will be kept within the same folder.

1.3.3.2 Using R interactively

Decades ago users communicated with computers through a physical terminal (keyboard plus text-only screen) that was frequently called a *console*. A text-only interface to a computer program, in most cases a window or a pane within a graphical user interface, is still called a console. In our case, the R console (Figure 1.1). This is the native user interface of R.

Typing commands at the R console is useful when one is playing around, rather aimlessly exploring things, or trying to understand how an R function or operator we are not familiar with works. Once we want to keep track of what we are doing, there are better ways of using R, which allow us to keep a record of how an analysis has been carried out. The different ways of using R are not exclusive of each other, so most users will use the R console to test individual commands and plot data during the first stages of exploration. As soon as we decide how we want to plot or analyze the data, it is best to start using scripts. This is not enforced in any way by R, but scripts are what really brings to light the most important advantages of using a programming language for data analysis. In Figure 1.1 we can see how the R console looks. The text in red has been typed in by the user, except for the prompt `>`, and the text in blue is what R has displayed in response. It is essentially a dialogue between user and R. The console can *look* different when displayed within an IDE like RStudio, but the only difference is in the appearance of the text rather than in the text itself (cf. Figures 1.1 and 1.2).

The two previous figures showed the result of entering a single command. Figure 1.3 shows how the console looks after the user has entered several commands, each as a separate line of text.




```
> print("hello")
[1] "hello"
> mean(c(1,5,6,2,3,4))
[1] 3.5
> a <- c(1,7,8,10,25)
> mean(a)
[1] 10.2
> sd(a)
[1] 8.927486
> b <- factor(c("trea", "trea", "trea", "ctrl", "ctrl"))
```

FIGURE 1.3


The R console after several commands have been entered. Commands entered by the user are displayed in red, while results returned by R are displayed in blue.

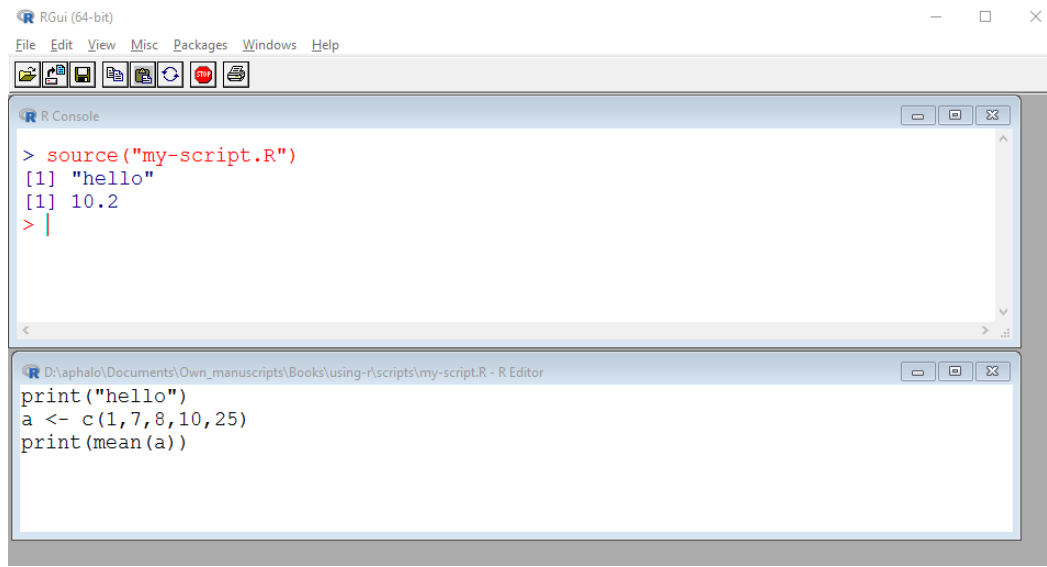
The examples in this book require only the console window for user input. Menu-driven programs are not necessarily bad, they are just unsuitable when there is a need to set very many options and choose from many different actions. They are also difficult to maintain when extensibility is desired, and when independently developed modules of very different characteristics need to be integrated. Textual languages also have the advantage, to be addressed in later chapters, that command sequences can be stored in human- and computer-readable text files. Such files constitute a record of all the steps used, and in most cases, makes it trivial to manually reproduce the same steps at a later time. Scripts are a very simple and handy way of communicating to other users how a given data analysis has been done or can be done.

 In the console one types commands at the `>` prompt. When one ends a line by pressing the return or enter key, if the line can be interpreted as an R command, the result will be printed at the console, followed by a new `>` prompt. If the command is incomplete, a `+` continuation prompt will be shown, and you will be able to type in the rest of the command. For example if the whole calculation that you would like to do is `1 + 2 + 3`, if you enter in the console `1 + 2 +` in one line, you will get a continuation prompt where you will be able to type `3`. However, if you type `1 + 2`, the result will be calculated, and printed.

1.3.3.3 Using R in a “batch job”

To run a script we need first to prepare a script in a text editor. Figure 1.4 shows the console immediately after running the script file shown in the text editor. As before, red text, the command `source("my-script.R")`, was typed by the user, and the blue text in the console is what was displayed by R as a result of this action. The title bar of the console, shows “R-console,” while the title bar of the editor shows the *path* to the script file that is open and ready to be edited followed by “R-editor.”

 When working at the command prompt, most results are printed by default. However, within scripts one needs to use function `print()` explicitly when a result is to be displayed.


**FIGURE 1.4**

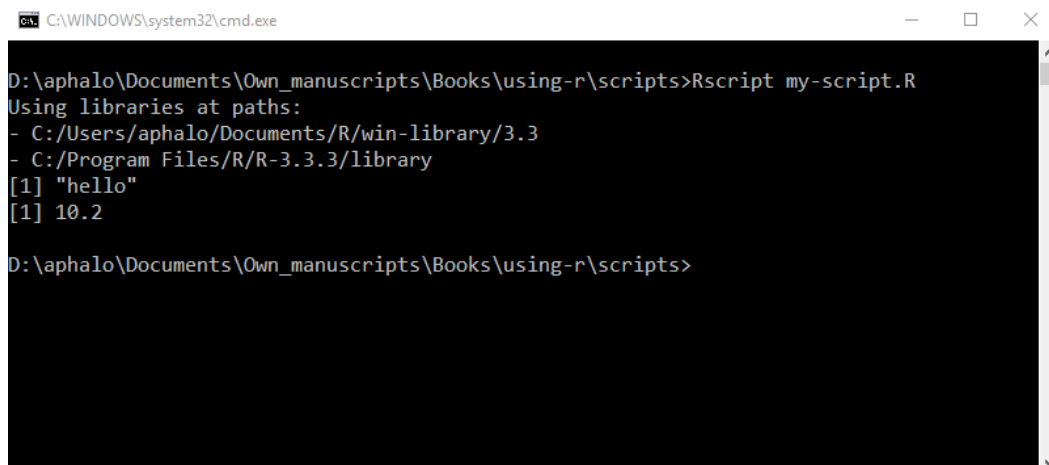
Screen capture of the R console and editor just after running a script. The upper pane shows the R console, and the lower pane, the script file in an editor.

A true “batch job” is not run at the R console but at the operating system command prompt, or shell. The shell is the console of the operating system—Linux, Unix, OS X, or MS-Windows. Figure 1.5 shows how running a script at the Windows command prompt looks. A script can be run at the operating system prompt to do time-consuming calculations with the output saved to a file. One may use this approach on a server, say, to leave a large data analysis job running overnight or even for several days.

1.3.3.4 Editors and IDEs

Integrated Development Environments (IDEs) are used when developing computer programs. IDEs provide a centralized user interface from within which the different tools used to create and test a computer program can be accessed and used in coordination. Most IDEs include a dedicated editor capable of syntax highlighting, and even report some mistakes, related to the programming language in use. One could describe such an editor as the equivalent of a word processor with spelling and grammar checking, that can alert about spelling and syntax errors for a computer language like R instead of for a natural language like English. In the case of RStudio, the languages supported are R and Python.

 RStudio and other IDEs provide a more comfortable user interface (UI) to R. What they add is an easier way of editing scripts, running the scripts with R and well as direct access to various tools. RStudio also makes it easier the access to help. Many menu entries and dialogue boxes in RStudio and other IDEs call behind the scenes R functions that are also available through the R console and scripts. It is important to keep this in mind, as R at least when working with small data sets

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The command prompt shows the following text:

```
D:\aphalo\Documents\Own_manuscripts\Books\using-r\scripts>Rscript my-script.R
Using libraries at paths:
- C:/Users/aphalo/Documents/R/win-library/3.3
- C:/Program Files/R/R-3.3.3/library
[1] "hello"
[1] 10.2
D:\aphalo\Documents\Own_manuscripts\Books\using-r\scripts>
```

FIGURE 1.5

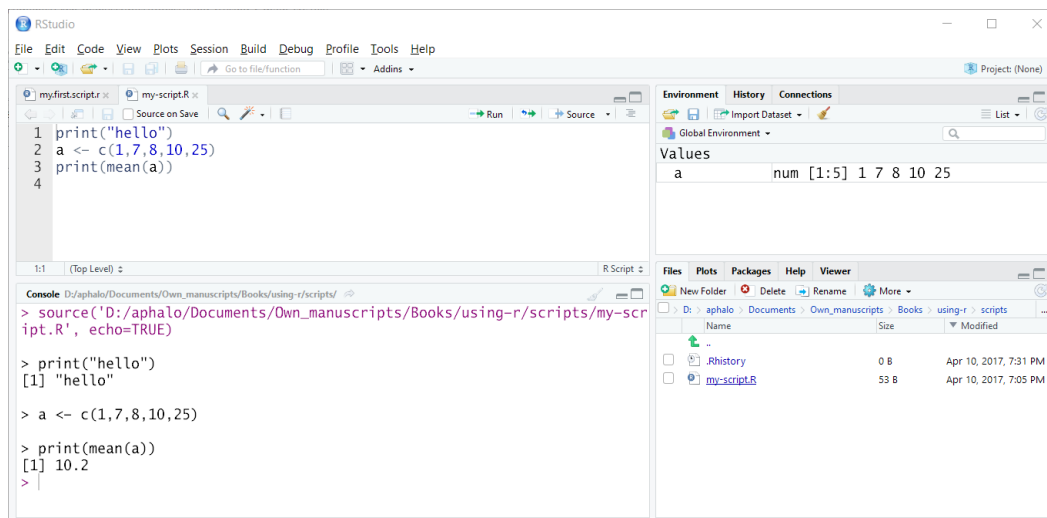
Screen capture of the MS-Windows command console just after running the same script. Here we use `Rscript` to run the script; the exact syntax will depend on the operating system in use. In this case, R prints the results at the operating system console or shell, rather than in its own R console.

needs much less computing power and memory resources than RStudio. R scripts created in RStudio run unchanged in the absence of RStudio or other IDEs.

This book provides only a minimum of guidance on the use of RStudio. Additional documentation on RStudio is available through the Resources menu entry at the book website at <https://www.learnr-book.info/>.

The main window of IDEs usually displays more than one pane simultaneously. From within the RStudio IDE, one has access to the R console, a text editor, a file-system browser, a pane for graphical output, and access to several additional tools such as for installing and updating extension packages. Although RStudio supports very well the development of large scripts and packages, it is currently, in my opinion, also the best possible way of using R at the console as it has the R help system very well integrated both in the editor and R console. Figure 1.6 shows the main window displayed by RStudio after running the same script as shown above at the R console (Figure 1.4) and at the operating system command prompt (Figure 1.5). We can see by comparing these three figures how RStudio is really a layer between the user and an unmodified R executable. The script was sourced by pressing the “Source” button at the top of the editor pane. RStudio, in response to this, generated the code needed to source the file and “entered” it at the console, the same console, where we would type any R commands.

When a script is run, if an error is triggered, RStudio automatically finds the location of the error. Some features are beyond what one needs for simple everyday data analysis and aimed at package development and report generation. Integration of debugging tools, trace-back on errors, code profiling, bench marking of code and unit tests, make it possible to analyze and improve performance as well help with quality assurance and certification. It also integrates support for file version control, which is not only useful for package development, but also for

**FIGURE 1.6**

The RStudio interface just after running the same script. Here we used the “Source” button to run the script. In this case, R prints the results to the R console in the lower left pane.

keeping track of the progress or work together with collaborators in the analysis of data.

The “desktop” version of RStudio that one uses locally, runs on most modern operating systems, such as Linux, Unix, OS X, and MS-Windows. There is also a server version that runs on Linux, as well as a cloud service (<https://posit.cloud/>), which can be both used remotely in a web browser. The user interface is the same in all cases. Desktop and server version are distributed each as free software and commercial software.

RStudio supports saving of its state and settings per working folder under the name of *project*, so that work on a project can be interrupted and restored, even on a different computer. As mentioned in section 1.3.3.1 on page 5, when working with R we keep related files in a folder. RStudio projects are implemented as a folder with a name ending in `.Rproj`, located under the same folder where scripts, data, `.Rdata` and `.Rhistory` files are stored.

RStudio is under active development. Two books (Hillebrand and Nierhoff 2015; Loo and Jonge 2012) describe and teach how to use RStudio without going in depth into data analysis or statistics, however, as RStudio is under very active development, several recently added important features are not described in these books. To learn more about RStudio, please, read the documentation available through RStudio’s help menu and keep at hand a printed copy of the RStudio cheat sheet while learning how to use it. This and other R-related cheatsheets can be downloaded at <https://posit.co/resources/cheatsheets/>.

1.4 Reproducible data analysis

Reproducible data analysis is much more than a fashionable buzzword. Under any situation where accountability is important, from scientific research to decision making in commercial enterprises, industrial quality control and safety and environmental impact assessments, being able to reproduce a data analysis reaching the same conclusions from the same data is crucial. Most approaches to reproducible data analysis are based on automating report generation and including, as part of the report, all the computer commands used to generate the results presented.

A fundamental requirement for reproducibility is a reliable record of what commands have been run on which data. Such a record is especially difficult to keep when issuing commands through menus and dialogue boxes in a graphical user interface or interactively at a console. Even working interactively at the R console using copy and paste to include commands and results in a report is error prone, and laborious.

A further requirement is to be able to match the output of the R commands to the input. If the script saves the output to separate files, then the user will need to take care that the script saved or shared as a record of the data analysis was the one actually used for obtaining the reported results and conclusions. This is another error-prone stage in the reporting of data analysis. To solve this problem an approach was developed, inspired in what is called *literate programming* (Knuth 1984). The idea is that running the script will produce a document that includes the listing of the R code used, the results of running this code and any explanatory text needed to understand and interpret the analysis.

Although a system capable of producing such reports with R, called ‘Sweave’ (Leisch 2002), has been available for a couple decades, it was rather limited and not supported by an IDE, making its use rather tedious. A more recently developed system called ‘knitr’ (Xie 2013) together with its integration into RStudio has made the use of this type of reports very easy. The most recent development is what has been called R *notebooks* produced within RStudio. This new feature, can produce the readable report of running the script as an HTML file, displaying the code used interspersed with the results within the viewable file as in earlier approaches. However, this newer approach goes even further: the actual source script used to generate the report is embedded in the HTML file of the report and can be extracted and run very easily and consequently re-used. This means that anyone who gets access to the output of the analysis in human readable form also gets access to the code used to generate the report, in computer executable format.

Because of these recent developments, R is an ideal language to use when the goal of reproducibility is important. During recent years the problem of the lack of reproducibility in scientific research has been broadly discussed and analysed (Gandrud 2015). One of the problems faced when attempting to reproduce experimental work, is reproducing the data analysis. R together with these modern tools can help in avoiding this source of lack of reproducibility.


How powerful are these tools and how flexible? They are powerful and flexible enough to write whole books, such as this very book you are now reading, produced

with R, ‘knitr’ and \LaTeX . All pages in the book are generated directly, all figures are generated by R and included automatically, except for the figures in this chapter that have been manually captured from the computer screen. Why am I using this approach? First because I want to make sure that every bit of code as you will see printed, runs without error. In addition, I want to make sure that the output that you will see below every line or chunk of R language code is exactly what R returns. Furthermore, it saves a lot of work for me as author, as I can just update R and all the packages used to their latest version, and build the book again, to keep it up to date and free of errors.

Although the use of these tools is important, they are outside the scope of this book and well described in other books (Gandrud 2015; Xie 2013). Still when writing code, using a consistent style for formatting and indentation, carefully choosing variable names, and adding textual explanations in comments when needed, helps very much with readability for humans. I have tried to be as consistent as possible throughout the whole book in this respect, with only small personal deviations from the usual style.

1.5 Finding additional information

When searching for answers, asking for advice or reading books, you will be confronted with different ways of approaching the same tasks. Do not allow this to overwhelm you; in most cases it will not matter as many computations can be done in R, as in any language, in several different ways, still obtaining the same result. The different approaches may differ mainly in two aspects: 1) how readable to humans are the instructions given to the computer as part of a script or program, and 2) how fast the code runs. Unless computation time is an important bottleneck in your work, just concentrate on writing code that is easy to understand to you and to others, and consequently easy to check and reuse. Of course, do always check any code you write for mistakes, preferably using actual numerical test cases for any complex calculation or even relatively simple scripts. Testing and validation are extremely important steps in data analysis, so get into this habit while reading this book. Testing how every function works, as I will challenge you to do in this book, is at the core of any robust data analysis or computing programming.

 Error messages tend to be terse in R, and may require some lateral thinking and/or “experimentation” to understand the real cause behind problems. When you are not sure you understand how some command works, it is useful in many cases to try simple examples for which you know the correct answer and see if you can reproduce them with R. Because of this, this book includes some code examples that trigger errors. Learning to interpret error messages is part of what is needed to become a proficient user of R. To test your understanding of how a code statement or function works, it is good to try your hand at testing its limits, testing which variations of a piece code are valid or not.

1.5.1 R's built-in help

To access help pages through the command prompt we use function `help()` or a question mark. Every object exported by an R package (functions, methods, classes, data) is documented. Sometimes a single help page documents several R objects. Usually at the end of the help pages, some examples are given, which tend to help very much in learning how to use the functions described. For example, one can search for a help page at the R console.

```
help("sum")  
?sum
```



Look at help for some other functions like `mean()`, `var()`, `plot()` and, why not, `help()` itself!

```
help(help)
```

When using RStudio there are easier ways of navigating to a help page than using function `help()`, for example, with the cursor on the name of a function in the editor or console, pressing the F1 key opens the corresponding help page in the help pane. Letting the cursor hover for a few seconds over the name of a function at the R console will open “bubble help” for it. If the function is defined in a script or another file that is open in the editor pane, one can directly navigate from the line where the function is called to where it is defined. In RStudio one can also search for help through the graphical interface.

In addition to help pages, R's distribution includes useful manuals as PDF or HTML files. These can be accessed most easily through the Help menu in RStudio or RGUI. Extension packages provide help pages for the functions and data they export. When a package is loaded into an R session, its help pages are added to the native help of R. In addition to these individual help pages, each package provides an index of its corresponding help pages for users to browse. Many packages, contain *vignettes* such as User Guides or articles describing the algorithms used.

There are some web sites that give access to R documentation through a web server. These sites can be very convenient when exploring whether a certain package could be useful for a certain problem, as they allow browsing and searching the documentation without need of installing the packages. Some package maintainers have web sites with additional documentation for their own packages. The DESCRIPTION or README of packages provide contact information for the maintainer, links to web sites, and instructions on how to report bugs. As packages are contributed by independent authors, they should be cited in addition to citing R itself. R function `citation()` when called with the name of a package as its argument provides the reference that should be cited for the package, and without an explicit argument, the reference to cite for the version of R in use.

```
citation()  
##  
## To cite R in publications use:  
##
```

```
## R Core Team (2022). R: A language and environment for statistical
## computing. R Foundation for Statistical Computing, Vienna, Austria.
## URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
## @Manual{,
##   title = {R: A Language and Environment for Statistical Computing},
##   author = {{R Core Team}},
##   organization = {R Foundation for Statistical Computing},
##   address = {Vienna, Austria},
##   year = {2022},
##   url = {https://www.R-project.org/},
## }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```



Look at the help page for function `citation()` for a discussion of why it is important for users to cite R and packages when using them.

1.5.2 Obtaining help from online forums

When consulting help pages, vignettes, and possibly books at hand fails to provide the information needed, the next step to follow is to search internet forums for existing answers to one's questions. When these steps fail to solve a problem, then it is time to ask for help, either from local experts or by posting your own question in a suitable online forum. When posting requests for help, one needs to abide by what is usually described as “netiquette.”

1.5.2.1 Netiquette

In most internet forums, a certain behavior is expected from those asking and answering questions. Some types of misbehavior, like use of offensive or inappropriate language, will usually result in the user losing writing rights in a forum. Occasional minor misbehavior, will usually result in the original question not being answered and instead the problem highlighted in the reply. In general following the steps listed below will greatly increase your chances of getting a detailed and useful answer.

- Do your homework: first search for existing answers to your question, both online and in the documentation. (Do mention that you attempted this without success when you post your question.)
- Provide a clear explanation of the problem, and all the relevant information. Say if it concerns R, the version, operating system, and any packages loaded and their versions.
- If at all possible, provide a simplified and short, but self-contained, code example that reproduces the problem (sometimes called *reprex*).

- Be polite.
- Contribute to the forum by answering other users' questions when you know the answer.


1.5.2.2 StackOverflow

Nowadays, StackOverflow (<http://stackoverflow.com/>) is the best question-and-answer (Q&A) support site for R. In most cases, searching for existing questions and their answers, will be all that you need to do. If asking a question, make sure that it is really a new question. If there is some question that looks similar, make clear how your question is different.

StackOverflow has a user-rights system based on reputation, and questions and answers can be up- and down-voted. Those with the most up-votes are listed at the top of searches. If the questions or answers you write are up-voted, after you accumulate enough reputation, you acquire badges and rights, such as editing other users' questions and answers or later on, even deleting wrong answers or off-topic questions from the system. This sounds complicated, but works extremely well at ensuring that the base of questions and answers is relevant and correct, without relying on nominated *moderators*. When using StackOverflow, do contribute by accepting correct answers, up-voting questions and answers that you find useful, down-voting those you consider poor, and flagging or correcting errors you may discover.

1.5.2.3 Reporting bugs

Being careful in the preparation of a reproducible example is especially important when you intend to report a bug to the maintainer of any piece of software. For the problem to be fixed, the person revising the code, needs to be able to reproduce the problem, and after modifying the code, needs to be able to test if the problem has been solved or not. However, even if you are facing a problem caused by your misunderstanding of how R works, the simpler the example, the more likely that someone will quickly realize what your intention was when writing the code that produces a result different from what you expected.

 How to prepare a reproducible example (“reprex”). A *reprex* is a self-contained and as simple as possible piece of computer code that triggers (and so demonstrates) a problem. If possible, when you need to use data, either use a data set included in base R or generate artificial data within the reprex code. If you can reproduce the problem only with your own data, then you need to provide a minimal subset of it that triggers the problem.

While preparing the *reprex* you will need to simplify the code, and sometimes this step allows you to diagnose the problem. Always, before posting a reprex online, it is wise to check it with the latest versions of R and any package being used.

I would say that about two out of three times I prepare a *reprex*, it allows me to much better understand the problem and find the root of the problem and a solution or a work-around on my own.

1.6 What is needed to run the examples in this book?

The book is written with the expectation that you will run most of the code examples and try as many other variations as needed until you are sure you understand the basic “rules” of the R language and how each function or command described works. As mentioned above, you are expected to use this book as a travel guide for your exploration of the world of R.

R is all that is needed to work through all the examples in this book, but it is not a convenient way of doing this. I recommend that you use an editor or an IDE, in particular RStudio. RStudio is user friendly, actively maintained, free, open-source and available both in desktop and server versions. The desktop version runs on MS-Windows, Linux, and OS X and other Unix distributions. Visit <https://posit.co/products/open-source/rstudio/> for an up-to-date description and download and installation instructions. Any edition of RStudio, including the free open edition, is suitable for working through the code examples and exercises in the book. When using the book to teach a class, RStudio Cloud simplifies the teacher’s job.

Of course when choosing which editor to use, personal preferences and previous familiarity play an important role. Currently, for the development of packages, I use RStudio exclusively. For writing this book I have used both RStudio and the text editor WinEdt which has support for R together with excellent support for \LaTeX . When working on a large project or collaborating with other data analysts or researchers, one big advantage of a system based on plain text files such as R scripts, is that the same files can be edited with different programs and under different operating systems as needed or wished by the different persons involved in a project.

When I started using R, nearly two decades ago, I was using other editors, using the operating system shell a lot more, and struggling with debugging as no IDE was available. The only reasonably good integration with an editor was for Emacs, which was widely available only under Unix-like systems. Given my past experience, I encourage you to use an IDE for R. RStudio is nowadays very popular, but if you do not like it, need a different set of features, such as integration with ImageJ, or are already familiar with the Eclipse IDE, you may want to try the Bio7 IDE, available from <http://bio7.org>.

The examples in this book make use of several freely available R extension packages, which can be installed from CRAN. One of them, ‘learnrbook’, also available through CRAN, contains data sets and files specific to this book. The ‘learnrbook’ package contains installation instructions and saved lists of the names of all other packages used in the book. The package also contains one script file per chapter with the R code for all the examples and exercises. Instructions on installing R, Git, RStudio, compilers and other tools are available online. In many cases the IT staff at your employer or school will know how to install them, or they may even be included in the default computer setup. In addition, a web site supporting the book is available at: <http://www.learnr-book.info>.

1.7 Further reading

Suggestions for further reading are dependent on how you plan to use R. If you envision yourself running batch jobs under Linux or Unix, you would profit from learning to write shell scripts. Because `bash` is widely used nowadays, *Learning the bash Shell* (Newham and Rosenblatt 2005) can be recommended. If you aim at writing R code that is going to be reused, and have some familiarity with C, C++ or Java, reading *The Practice of Programming* (Kernighan and Pike 1999) will provide a mostly language-independent view of programming as an activity and help you master the all-important tricks of the trade.



2

The R language: “Words” and “sentences”

The desire to economize time and mental effort in arithmetical computations, and to eliminate human liability to error, is probably as old as the science of arithmetic itself.

Howard Aiken

Proposed automatic calculating machine, 1937; reprinted 1964

2.1 Aims of this chapter

In my experience, for those not familiar with computer programming languages, the best first step in learning the R language is to use it interactively by typing textual commands at the *console* or command line. This will teach not only the syntax and grammar rules, but also give you a glimpse at the advantages and flexibility of this approach to data analysis.

In the first part of the chapter we will use R to do everyday calculations that should be so easy and familiar that you will not need to think about the operations themselves. This easy start will give you a chance to focus on learning how to issue textual commands at the command prompt.

Later in the chapter, you will gradually need to focus more on the R language and its grammar and less on how commands are entered. By the end of the chapter you will be familiar with most of the kinds of “words” used in the R language and you will be able to write simple “sentences” in R.

Along the chapter, I will occasionally show the equivalent of the R code in mathematical notation. If you are not familiar with the mathematical notation, you can safely ignore it, as long as you understand the R code.

2.2 Natural and computer languages

Computer languages have strict rules and interpreters and compilers are unforgiving about errors. They will issue error messages, but in contrast to human readers or listeners, will not guess your intentions and continue. However, computer languages have a much smaller set of words than natural languages, such as English. If you are new to computer programming, understanding the parallels between computer and natural languages may be useful.


One can think of constant values and variables (values stored under a name) as nouns and of operators and functions as verbs. A complete command, or statement, is the equivalent of a natural language sentence: “a comprehensible utterance.” The simple statement `a + 1` has three components: `a`, a variable, `+`, an operator and `1` a constant. The statement `sqrt(4)` has two components, a function `sqrt()` and a numerical constant `4`. We say that “to compute $\sqrt{4}$ we *call* `sqrt()` with `4` as its *argument*.”

Although all values manipulated in a digital computer are stored as *bits* in memory, multiple interpretations are possible. Numbers, letters, logical values, etc., can be encoded into bits and decoded as long as their type or *mode* is known. The concept of `class` is not directly related to how values are encoded when stored in computer memory, but instead their interpretation as part of a computer program. We can have, for example, RGB color values, stored as three numbers such as `0, 0, 255`, as hexadecimal numbers stored as characters `#0000FF`, or even use fancy names stored as character strings like `"blue"`. We could create a `class` for colors using any of these representations, based on two different modes: `numeric` and `character`.

In this chapter we will focus on individual program statements, the equivalent of sentences in natural language. In later chapters you will learn how to combine them to create compound statements, the equivalent of natural-language paragraphs, and scripts, the equivalent of essays. You will also learn how to define new verbs, user-defined functions and operators, and new nouns, user-defined classes.

2.3 Numeric values and arithmetic

When working in R with arithmetic expressions, the normal mathematical precedence rules are respected, but parentheses can be used to alter this order. Parentheses can be nested, but in contrast to the usual practice in mathematics, the same parenthesis symbol is used at all nesting levels.

 Both in mathematics and programming languages *operator precedence rules* determine which subexpressions are evaluated first and which later. Contrary to primitive electronic calculators, R evaluates numeric expressions containing operators according to the rules of mathematics. In the expression $3 + 2 \times 3$, the product 2×3 has precedence over the addition, and is evaluated first, yielding

as the result of the whole expression, 9. In programming languages, similar rules apply to all operators, even those taking as operands non-numeric values.

It is important to keep in mind that in R trigonometric functions interpret numeric values representing angles as being expressed in radians.

The equivalent of the math expression

$$\frac{3 + e^2}{\sin \pi}$$

is, in R, written as follows:

```
(3 + exp(2)) / sin(pi)
## [1] 8.483588e+16
```

It can be seen above that mathematical constants and functions are part of the R language. One thing to remember when translating complex fractions as above into R code, is that in arithmetic expressions the bar of the fraction generates a grouping that alters the normal precedence of operations. In contrast, in an R expression this grouping must be explicitly signaled with additional parentheses.

If you are in doubt about how precedence rules work, you can add parentheses to make sure the order of computations is the one you intend. Redundant parentheses have no effect.

```
1 + 2 * 3
## [1] 7

1 + (2 * 3)
## [1] 7

(1 + 2) * 3
## [1] 9
```

The number of opening (left side) and closing (right side) parentheses must be balanced, and they must be located so that each enclosed term is a valid mathematical expression, i.e., code that can be evaluated to return a value, a value that can be inserted in place of the expression enclosed in parenthesis before evaluating the remaining of the expression. For example, $(1 + 2) * 3$ after evaluating $(1 + 2)$ becomes $3 * 3$ yielding 9. In contrast, $(1 +) 2 * 3$ is a syntax error as $1 +$ is incomplete and does not yield a number.



Here results are not shown. These are examples for you to type at the command prompt. In general you should not skip them, as in many cases, as with the statements highlighted with comments in the code chunk below, they have something to teach or demonstrate. You are strongly encouraged to *play*, in other words, create new variations of the examples and execute them to explore how R works.

```

1 + 1
2 * 2
2 + 10 / 5
(2 + 10) / 5
10^2 + 1
sqrt(9)
pi # whole precision not shown when printing
print(pi, digits = 22)
sin(pi) # oops! Read on for explanation.
log(100)
log10(100)
log2(8)
exp(1)

```

Variables are used to store values. After we *assign* a value to a variable, we can use in our code the name of the variable in place of the stored value. The “usual” assignment operator is `<-`. In R, all names, including variable names, are case sensitive. Variables `a` and `A` are two different variables. Variable names can be long in R although it is not a good idea to use very long names. Here I am using very short names, something that is usually also a very bad idea. However, in the examples in this chapter where the stored values have no connection to the real world, simple names emphasize their abstract nature. In the chunk below, `a` and `b` are arbitrarily chosen variable names; I could have used names like `my.variable.a` or `outside.temperature` if they had been useful to convey information.

```

a <- 1
a + 1
## [1] 2

a
## [1] 1

b <- 10
b <- a + b
b
## [1] 11

3e-2 * 2.0
## [1] 0.06

```

Entering the name of a variable *at the R console* implicitly calls function `print()` displaying the stored value on the console. The same applies to any other statement entered *at the R console*: `print()` is implicitly called with the result of executing the statement as its argument.

```

a
## [1] 1

print(a)
## [1] 1

a + 1
## [1] 2

print(a + 1)
## [1] 2

```



There are some syntactically legal assignment statements that are not very frequently used, but you should be aware that they are valid, as they will not trigger error messages, and may surprise you. The most important thing is to write code consistently. The “backwards” assignment operator `->` and resulting code like `1 -> a` are valid but less frequently used. The use of the equals sign (`=`) for assignment in place of `<=` although valid is discouraged. Chaining assignments as in the first statement below can be used to signal to the human reader that `a`, `b` and `c` are being assigned the same value.

```
a <- b <- c <- 0.0
a
b
c
1 -> a
a
a = 3
a
```



In R, all numbers belong to mode `numeric` (we will discuss the concepts of *mode* and *class* in section 2.8 on page 43). We can query if the mode of an object is `numeric` with function `is.numeric()`.

```
mode(1)
## [1] "numeric"

a <- 1
is.numeric(a)
## [1] TRUE
```

Because numbers can be stored in different formats, requiring different amounts of computer memory per value, most computing languages implement several different types of numbers. In most cases R’s `numeric()` can be used everywhere that a number is expected. However, in some cases it has advantages to explicitly indicate that we will store or operate on whole numbers, in which case we can use class `integer`, with integer constants indicated by a trailing capital “L,” as in `32L`.

```
is.numeric(1L)
## [1] TRUE

is.integer(1L)
## [1] TRUE

is.double(1L)
## [1] FALSE
```

Real numbers are a mathematical abstraction, and do not have an exact equivalent in computers. Instead of Real numbers, computers store and operate on numbers that are restricted to a broad but finite range of values and have a finite resolution. They are called, *floats* (or *floating-point* numbers); in R they go by the name of `double` and can be created with the constructor `double()`.

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

is.integer(1)
## [1] FALSE

is.double(1)
## [1] TRUE
```

The name `double` originates from the C language, in which there are different types of floats available. With the name `double` used to mean “double-precision floating-point numbers.” Similarly, the use of `L` stems from the `long` type in C, meaning “long integer numbers.”

In R numeric objects are always `vector`s, and can contain zero, one or more values. We will later see why this is important. As you have seen above, the results of calculations were printed preceded with `[1]`. This is the index or position in the vector of the first number (or other value) displayed at the head of the current line. Vectors, in mathematical notation, are represented with positional subindexes,

$$a_{1\dots n} = a_1, a_2, \dots, a_i, \dots, a_n, \quad (2.1)$$

where $a_{1\dots n}$ represents the whole vector and a_1 its first member. The length of $a_{1\dots n}$ is n as it contains n members.

One can use `c()` “concatenate” to create a vector from other vectors, including vectors of length 1, or even vectors of length 0, such as the `numeric` constants in the statements below.

```
a <- c(3, 1, 2)
a
## [1] 3 1 2

b <- c(4, 5, 0)
b
## [1] 4 5 0

c <- c(a, b)
c
## [1] 3 1 2 4 5 0

d <- c(b, a)
d
## [1] 4 5 0 3 1 2

e <- c(d, numeric())
```

Method `c()` accepts as arguments two or more vectors and concatenates them, one after another. Quite frequently we may need to insert one vector in the middle of another. For this operation, `c()` is not useful by itself. One could use indexing combined with `c()`, but this is not needed as R provides a function capable of directly doing this operation. Although it can be used to “insert” values, it is named `append()`, and by default, it indeed appends one vector at the end of another.

```
append(a, b)
## [1] 3 1 2 4 5 0
```

The output above is the same as for `c(a, b)`, however, `append()` accepts as an argument an index position after which to “append” its second argument. This results in an *insert* operation when the index points at any position different from the end of the vector.

```
append(a, values = b, after = 2L)
## [1] 3 1 4 5 0 2
```

Both `c()` and `append()` can also be used with lists (described in section 2.13 on page 65).



One can create sequences using function `seq()` or the operator `:`, or repeat values using function `rep()`. In this case, I leave to the reader to work out the rules by running these and his/her own examples, with the help of the documentation, available through `help(seq)` and `help(rep)`.

```
a <- -1:5
a
b <- 5:-1
b
c <- seq(from = -1, to = 1, by = 0.1)
c
d <- rep(-5, 4)
d
```

Next, something that makes R different from most other programming languages: vectorized arithmetic. Operators and functions that are vectorized accept, as arguments, vectors of arbitrary length, in which case the result returned is equivalent to having applied the same function or operator individually to each element of the vector.

```
a + 1 # we add one to vector a defined above
## [1] 4 2 3

(a + 1) * 2
## [1] 8 4 6

a + b
## [1] 7 6 2

a - a
## [1] 0 0 0
```

As it can be seen in the first line above, another peculiarity of R, is what is frequently called “recycling” of arguments: as vector `a` is of length 6, but the constant 1 is a vector of length 1, this short constant vector is extended, by recycling its value, into a vector of six ones—i.e., a vector of the same length as the longest vector in the statement, `a`.

Make sure you understand what calculations are taking place in the chunk above, and also the one below.

```

a <- rep(1, 6)
a
## [1] 1 1 1 1 1 1

a + 1:2
## [1] 2 3 2 3 2 3

a + 1:3
## [1] 2 3 4 2 3 4

a + 1:4
## Warning in a + 1:4: longer object length is not a multiple of shorter object
length
## [1] 2 3 4 5 2 3

```



A useful thing to know: a vector can have length zero. Vectors of length zero may seem at first sight quite useless, but in fact they are very useful. They allow the handling of “no input” or “nothing to do” cases as normal cases, which in the absence of vectors of length zero would require to be treated as special cases. Constructors for built in classes like `numeric()` return vectors of a length given by their first argument, which defaults to zero. I describe here a useful function, `length()` which returns the length of a vector or list.

```

z <- numeric(0)
z
## numeric(0)

length(z)
## [1] 0

```

```

z1 <- numeric()
z1
## numeric(0)

z2 <- numeric(length = 0)
z2
## numeric(0)

```

Vectors and lists of length zero, behave in most cases, as expected—e.g., they can be concatenated as shown here.

```

length(c(a, numeric(0), b))
## [1] 9

length(c(a, b))
## [1] 9

```

Many functions, such as R’s maths functions and operators, will accept numeric vectors of length zero as valid input, returning also a vector of length zero, issuing neither a warning nor an error message. In other words, *these are valid operations* in R.

```
log(numeric(0))
## numeric(0)

5 + numeric(0)
## numeric(0)
```

Even when of length zero, vectors do have to belong to a class acceptable for the operation: `5 + character(0)` is an error.

It is possible to *remove* variables from the workspace with `rm()`. Function `ls()` returns a *list* of all objects visible in the current environment, or by supplying a `pattern` argument, only the objects with names matching the `pattern`. The pattern is given as a regular expression, with `[]` enclosing alternative matching characters, `^` and `$`, indicating the extremes of the name (start and end, respectively). For example, `"^z$"` matches only the single character 'z' while `"^z"` matches any name starting with 'z'. In contrast `"^[zy]$"` matches both 'z' and 'y' but neither 'zy' nor 'yz', and `"^[a-z]"` matches any name starting with a lowercase ASCII letter. If you are using RStudio, all objects are listed in the Environment pane, and the search box of the panel can be used to find a given object.

```
ls(pattern="^z$")
## [1] "z"

rm(z)
ls(pattern="^z$")
## character(0)
```

There are some special values available for numbers. `NA` meaning “not available” is used for missing values. Calculations can also yield the following values `NaN` “not a number”, `Inf` and `-Inf` for ∞ and $-\infty$. As you will see below, calculations yielding these values do **not** trigger errors or warnings, as they are arithmetically valid. `Inf` and `-Inf` are also valid numerical values for input and constants.

```
a <- NA
a
## [1] NA

-1 / 0
## [1] -Inf

1 / 0
## [1] Inf

Inf / Inf
## [1] NaN

Inf + 4
## [1] Inf

b <- -Inf
b * -1
## [1] Inf
```

Not available (`NA`) values are very important in the analysis of experimental data,

as frequently some observations are missing from an otherwise complete data set due to “accidents” during the course of an experiment. It is important to understand how to interpret `NA`’s. They are simple placeholders for something that is unavailable, in other words, *unknown*.

```
A <- NA
A
## [1] NA

A + 1
## [1] NA

A + Inf
## [1] NA
```



When to use vectors of length zero, and when `NA`s? Make sure you understand the logic behind the different behavior of functions and operators with respect to `NA` and `numeric()` or its equivalent `numeric(0)`. What do they represent? Why `NA`’s are not ignored, while vectors of length zero are?

```
123 + numeric()
123 + NA
```

Model answer: `NA` is used to signal a value that “was lost” or “was expected” but is unavailable because of some accident. A vector of length zero, represents no values, but within the normal expectations. In particular, if vectors are expected to have a certain length, or if index positions along a vector are meaningful, then using `NA` is a must.

Any operation, even tests of equality, involving one or more `NA`’s return an `NA`. In other words, when one input to a calculation is unknown, the result of the calculation is unknown. This means that a special function is needed for testing for the presence of `NA` values.

```
is.na(c(NA, 1))
## [1] TRUE FALSE
```

In the example above, we can also see that `is.na()` is vectorized, and that it applies the test to each of the two elements of the vector individually, returning the result as a logical vector of length two.

One thing to be aware of are the consequences of the fact that numbers in computers are almost always stored with finite precision and/or range: the expectations derived from the mathematical definition of Real numbers are not always fulfilled. See the box on page 35 for an in-depth explanation.

```
1 - 1e-20
## [1] 1
```

When comparing `integer` values these problems do not exist, as integer arithmetic is not affected by loss of precision in calculations restricted to integers. Because of the way integers are stored in the memory of computers, within the

representable range, they are stored exactly. One can think of computer integers as a subset of whole numbers restricted to a certain range of values.

```
1L + 3L
## [1] 4

1L * 3L
## [1] 3

1L %% 3L
## [1] 0

1L %/% 3L
## [1] 1

1L / 3L
## [1] 0.3333333
```

The last statement in the example immediately above, using the “usual” division operator yields a floating-point `double` result, while the integer division operator `%/%` yields an `integer` result, and `%%` returns the remainder from the integer division. If as a result of an operation the result falls outside the range of representable values, the returned value is `NA`.

```
1000000L * 1000000L

## Warning in 1000000L * 1000000L: NAs produced by integer overflow
## [1] NA
```

Both doubles and integers are considered numeric. In most situations, conversion is automatic and we do not need to worry about the differences between these two types of numeric values. The next chunk shows returned values that are either `TRUE` or `FALSE`. These are `logical` values that will be discussed in the next section.

```
is.numeric(1L)
## [1] TRUE

is.integer(1L)
## [1] TRUE

is.double(1L)
## [1] FALSE

is.double(1L / 3L)
## [1] TRUE

is.numeric(1L / 3L)
## [1] TRUE
```



Study the variations of the previous example shown below, and explain why the two statements return different values. Hint: `1` is a `double` constant. You can use `is.integer()` and `is.double()` in your explorations.

```
1 * 1000000L * 1000000L
1000000L * 1000000L * 1
```

Both when displaying numbers or as part of computations, we may want to decrease the number of significant digits or the number of digits after the decimal marker. Be aware that in the examples below, even if printing is being done by default, these functions return `numeric` values that are different from their input and can be stored and used in computations. Function `round()` is used to round numbers to a certain number of decimal places after or before the decimal marker, while `signif()` rounds to the requested number of significant digits.

```
round(0.0124567, digits = 3)
## [1] 0.012

signif(0.0124567, digits = 3)
## [1] 0.0125

round(1789.1234, digits = 3)
## [1] 1789.123

signif(1789.1234, digits = 3)
## [1] 1790

round(1789.1234, digits = -1)
## [1] 1790

a <- 0.12345
b <- round(a, digits = 2)
a == b
## [1] FALSE

a - b
## [1] 0.00345

b
## [1] 0.12
```



Being `digits`, the second parameter of these functions, the argument can also be passed by position. However, code is usually easier to understand for humans when parameter names are made explicit.

```
round(0.0124567, digits = 3)
## [1] 0.012

round(0.0124567, 3)
## [1] 0.012
```

Functions `trunc()` and `ceiling()` return the non-fractional part of a numeric value as a new numeric value. They differ in how they handle negative values, and neither of them rounds the returned value to the nearest whole number.



What does value truncation mean? Function `trunc()` truncates a numeric value, but it does not return an `integer`.

- Explore how `trunc()` and `ceiling()` differ. Test them both with positive and negative values.

- **Advanced** Use function `abs()` and operators `+` and `-` to reproduce the output of `trunc()` and `ceiling()` for the different inputs.
- Can `trunc()` and `ceiling()` be considered type conversion functions in R?

2.4 Logical values and Boolean algebra

What in Mathematics are usually called Boolean values, are called `logical` values in R. They can have only two values `TRUE` and `FALSE`, in addition to `NA` (not available). They are vectors as all other atomic types in R (by *atomic* we mean that each value is not composed of “parts”). There are also logical operators that allow Boolean algebra. In the chunk below we operate on `logical` vectors of length one.

```
a <- TRUE
b <- FALSE
mode(a)
## [1] "logical"

a
## [1] TRUE

!a # negation
## [1] FALSE

a && b # logical AND
## [1] FALSE

a || b # logical OR
## [1] TRUE

xor(a, b) # exclusive OR
## [1] TRUE
```

As with arithmetic operators, vectorization is available with *some* logical operators. The availability of two kinds of logical operators is one of the most troublesome aspects of the R language for beginners. Pairs of “equivalent” logical operators behave differently, use similar syntax and use similar symbols! The vectorized operators have single-character names `&` and `|`, while the non-vectorized ones have double-character names `&&` and `||`. There is only one version of the negation operator `!` that is vectorized. In some, but not all cases, a warning will indicate that there is a possible problem.

```
a <- c(TRUE, FALSE)
b <- c(TRUE, TRUE)
a
## [1] TRUE FALSE

b
## [1] TRUE TRUE
```

```

a & b # vectorized AND
## [1] TRUE FALSE

a | b # vectorized OR
## [1] TRUE TRUE

a && b # not vectorized

## warning in a && b: 'length(x) = 2 > 1' in coercion to 'logical(1)'
## warning in a && b: 'length(x) = 2 > 1' in coercion to 'logical(1)'
## [1] TRUE

a || b # not vectorized

## warning in a || b: 'length(x) = 2 > 1' in coercion to 'logical(1)'
## [1] TRUE

```

Functions `any()` and `all()` take zero or more logical vectors as their arguments, and return a single logical value “summarizing” the logical values in the vectors. Function `all()` returns `TRUE` only if all values in the vectors passed as arguments are `TRUE`, and `any()` returns `TRUE` unless all values in the vectors are `FALSE`.

```

any(a)
## [1] TRUE

all(a)
## [1] FALSE

any(a & b)
## [1] TRUE

all(a & b)
## [1] FALSE

```

Another important thing to know about logical operators is that they “short-cut” evaluation. If the result is known from the first part of the statement, the rest of the statement is not evaluated. Try to understand what happens when you enter the following commands. Short-cut evaluation is useful, as the first condition can be used as a guard protecting a later condition from being evaluated when it would trigger an error.

```

TRUE || NA
## [1] TRUE

FALSE || NA
## [1] NA

TRUE && NA
## [1] NA

FALSE && NA
## [1] FALSE

TRUE && FALSE && NA
## [1] FALSE

TRUE && TRUE && NA
## [1] NA

```

When using the vectorized operators on vectors of length greater than one, ‘short-cut’ evaluation still applies for the result obtained at each index position.

```
a & b & NA
## [1]      NA FALSE

a & b & c(NA, NA)
## [1]      NA FALSE

a | b | c(NA, NA)
## [1] TRUE TRUE
```



Based on the description of “recycling” presented on page 25 for numeric operators, explore how “recycling” works with vectorized logical operators. Create logical vectors of different lengths (including length one) and *play* by writing several code statements with operations on them. To get you started, one example is given below. Execute this example, and then create and run your own, making sure that you understand why the values returned are what they are. Sometimes, you will need to devise several examples or test cases to tease out of R an understanding of how a certain feature of the language works, so do not give up early, and make use of your imagination!

```
x <- c(TRUE, FALSE, TRUE, NA)
x & FALSE
x | c(TRUE, FALSE)
```

2.5 Comparison operators and operations

Comparison operators return vectors of `logical` values as results.

```
1.2 > 1.0
## [1] TRUE

1.2 >= 1.0
## [1] TRUE

1.2 == 1.0 # be aware that here we use two = symbols
## [1] FALSE

1.2 != 1.0
## [1] TRUE

1.2 <= 1.0
## [1] FALSE

1.2 < 1.0
## [1] FALSE

a <- 20
a < 100 && a > 10
## [1] TRUE
```

These operators can be used on vectors of any length, returning as a result a logical vector as long as the longest operand. In other words, they behave in the same way as the arithmetic operators described on page 25: their arguments are recycled when needed. Hint: if you do not know what to expect as a value for the vector returned by `1:10`, execute the statement `print(a)` after the first code statement below, or, alternatively, `1:10` without saving the result to a variable.

```
a <- 1:10
a > 5
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE

a < 5
## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

a == 5
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE

all(a > 5)
## [1] FALSE

any(a > 5)
## [1] TRUE

b <- a > 5
b
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE

any(b)
## [1] TRUE

all(b)
## [1] FALSE
```

Precedence rules also apply to comparison operators and they can be overridden by means of parentheses.

```
a > 2 + 3
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE

(a > 2) + 3
## [1] 3 3 4 4 4 4 4 4 4 4
```



Use the statement below as a starting point in exploring how precedence works when logical and arithmetic operators are part of the same statement. *Play* with the example by adding parentheses at different positions and based on the returned values, work out the default order of operator precedence used for the evaluation of the example given below.

```
a <- 1:10
a > 3 | a + 2 < 3
```

Again, be aware of “short-cut evaluation”. If the result does not depend on the missing value, then the result, `TRUE` or `FALSE` is returned. If the presence of the `NA` makes the end result unknown, then `NA` is returned.

```

c <- c(a, NA)
c > 5
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE NA

all(c > 5)
## [1] FALSE

any(c > 5)
## [1] TRUE

all(c < 20)
## [1] NA

any(c > 20)
## [1] NA

is.na(a)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

is.na(c)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

any(is.na(c))
## [1] TRUE

all(is.na(c))
## [1] FALSE

```

The behavior of many of base-R's functions when `NA`s are present in their input arguments can be modified. `TRUE` passed as an argument to parameter `na.rm`, results in `NA` values being *removed* from the input **before** the function is applied.

```

all(c < 20)
## [1] NA

any(c > 20)
## [1] NA

all(c < 20, na.rm=TRUE)
## [1] TRUE

any(c > 20, na.rm=TRUE)
## [1] FALSE

```



Here I give some examples for which the finite resolution of computer machine floats, as compared to Real numbers as defined in mathematics, can cause serious problems. In R, numbers that are not integers are stored as *double-precision floats*. In addition to having limits to the largest and smallest numbers that can be represented, the precision of floats is limited by the number of significant digits that can be stored. Precision is usually described by “epsilon” (ϵ), abbreviated *eps*, defined as the largest value of ϵ for which $1 + \epsilon = 1$. The finite resolution of floats can lead to unexpected results when testing for equality. In the second example below, the result of the subtraction is still exactly 1 due to insufficient resolution.


```
0 - 1e-20
## [1] -1e-20

1 - 1e-20
## [1] 1
```

The finiteness of floats also affects tests of equality, which is more likely to result in errors with important consequences.

```
1e20 == 1 + 1e20
## [1] TRUE

1 == 1 + 1e-20
## [1] TRUE

0 == 1e-20
## [1] FALSE
```

As R can run on different types of computer hardware, the actual machine limits for storing numbers in memory may vary depending on the type of processor and even compiler used to build the R program executable. However, it is possible to obtain these values at run time from the variable `.Machine`, which is part of the R language. Please see the help page for `.Machine` for a detailed and up-to-date description of the available constants.

```
.Machine$double.eps
## [1] 2.220446e-16

.Machine$double.neg.eps
## [1] 1.110223e-16

.Machine$double.max
## [1] 1024

.Machine$double.min
## [1] -1022
```

The last two values refer to the exponents of 10, rather than the maximum and minimum size of numbers that can be handled as objects of class `double`. Values outside these limits are stored as `-Inf` or `Inf` and enter arithmetic as infinite values according the mathematical rules.

```
1e1026
## [1] Inf

1e-1026
## [1] 0

Inf + 1
## [1] Inf

-Inf + 1
## [1] -Inf
```

As `integer` values are stored in machine memory without loss of precision, `epsilon` is not defined for `integer` values.

```
.Machine$integer.max
## [1] 2147483647

2147483699L
## [1] 2147483699
```

In those statements in the chunk below where at least one operand is `double` the `integer` operands are *promoted* to `double` before computation. A similar promotion does not take place when operations are among `integer` values, resulting in *overflow*, meaning numbers that are too big to be represented as `integer` values.

```
2147483600L + 99L

## Warning in 2147483600L + 99L: NAs produced by integer overflow
## [1] NA

2147483600L + 99
## [1] 2147483699

2147483600L * 2147483600L

## Warning in 2147483600L * 2147483600L: NAs produced by integer overflow
## [1] NA


2147483600L * 2147483600
## [1] 4.611686e+18
```

We see next that the exponentiation operator `^` forces the promotion of its arguments to `double`, resulting in no overflow. In contrast, as seen above, the multiplication operator `*` operates on integers resulting in overflow.

```
2147483600L * 2147483600L

## Warning in 2147483600L * 2147483600L: NAs produced by integer overflow
## [1] NA

2147483600L^2L
## [1] 4.611686e+18
```

 In many situations, when writing programs one should avoid testing for equality of floating point numbers ('floats'). Here we show how to gracefully handle rounding errors. As the example shows, rounding errors may accumulate, and in practice `.Machine$double.eps` is not always a good value to safely use in tests for "zero," and a larger value may be needed. Whenever possible according to the logic of the calculations, it is best to test for inequalities, for example using `x <= 1.0` instead of `x == 1.0`. If this is not possible, then the tests should be done replacing tests like `x == 1.0` with `abs(x - 1.0) < eps`. Function `abs()` returns the absolute value, in simpler words, makes all values positive or zero, by changing the sign of negative values, or in mathematical notation $|x| = |-x|$.

```

a == 0.0 # may not always work
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

abs(a) < 1e-15 # is safer
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

sin(pi) == 0.0 # angle in radians, not degrees!
## [1] FALSE

sin(2 * pi) == 0.0
## [1] FALSE

abs(sin(pi)) < 1e-15
## [1] TRUE

abs(sin(2 * pi)) < 1e-15
## [1] TRUE

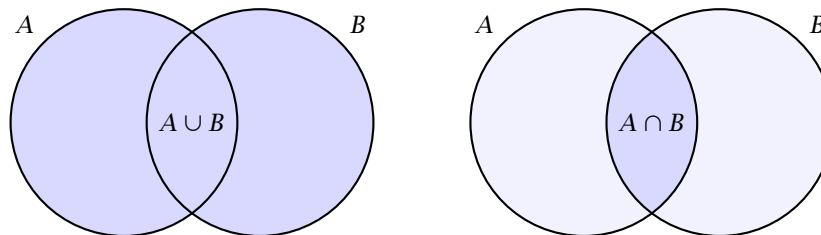
sin(pi)
## [1] 1.224606e-16

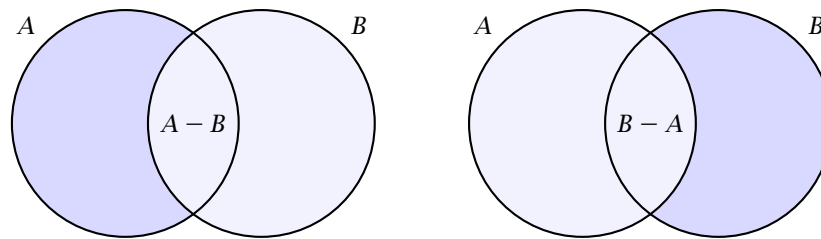
sin(2 * pi)
## [1] -2.449213e-16

```

2.6 Sets and set operations

The R language supports set operations on vectors. They can be useful in many different contexts when manipulating and comparing vectors of values. Set algebra operations and their equivalents in mathematical notation and R functions are: *union*, \cup , `union()`; *intersection*, \cap , `intersect()`; *difference (asymmetrical)*, $-$, `setdiff()`; *equality test* `setequal()`; *membership*, `is.element()` and `%in%`. The first three functions return vector of the same mode as their inputs, and the last three a `logical` vector. The action of the first three operations is most easily illustrated with Venn diagrams.





In Bioinformatics it is usual, for example, to have character vectors of gene tags. We may have a vector for each of a set of different samples, and need to compare them. However, we start by using a more mundane example, everyday shopping, as illustration, followed by explanations.

```
fruits <- c("apple", "pear", "orange", "lemon", "tangerine")
bakery <- c("bread", "buns", "cake", "cookies")
dairy <- c("milk", "butter", "cheese")
shopping <- c("bread", "butter", "apple", "cheese", "orange")
intersect(fruits, shopping)
## [1] "apple" "orange"

intersect(bakery, shopping)
## [1] "bread"

intersect(dairy, shopping)
## [1] "butter" "cheese"

"lemon" %in% dairy
## [1] FALSE

"lemon" %in% fruits
## [1] TRUE

dairy %in% shopping
## [1] FALSE TRUE TRUE

setdiff(union(bakery, dairy), shopping)
## [1] "buns" "cake" "cookies" "milk"
```

For explanations we use abstract (symbolic) examples.


```
my.set <- c("a", "b", "c", "b")
```

To test if a given value belongs to a set, we use operator `%in%` or its function equivalent `is.element()`. In the algebra of sets notation, this is written $a \in A$, where A is a set and a a member. The second statement shows that the `%in%` operator is vectorized on its left-hand-side (lhs) operand, returning a logical vector.

```
is.element("a", my.set)
## [1] TRUE

"a" %in% my.set
## [1] TRUE

c("a", "a", "z") %in% my.set
## [1] TRUE TRUE FALSE
```

 Keep in mind that inclusion is an asymmetrical (not reflective) operation among sets. The rhs argument is interpreted as a set, while the lhs argument is interpreted as a vector of values to test for inclusion. In other words, any duplicate member in the lhs will be retained while the rhs is interpreted as a set of unique values. The returned logical vector has the same length as the lhs.

```
my.set %in% "a"
## [1] TRUE FALSE FALSE FALSE
```


The negation of inclusion is $a \notin A$, and coded in R by applying the negation operator `!` to the result of the test done with `%in%` or function `is.element()`.

```
!is.element("a", my.set)
## [1] FALSE

!"a" %in% my.set
## [1] FALSE

!c("a", "a", "z") %in% my.set
## [1] FALSE FALSE TRUE
```

Although inclusion is a set operation, it is also very useful for the simplification of `if()...else` statements by replacing multiple tests for alternative constant values of the same mode chained by multiple `|` operators. A useful property of `%in%` and `is.element()` is that they never return `NA`.

 Use operator `%in%` to write more concisely the following comparisons. Hint: see section 2.4 on page 31 for the difference between `|` and `||` operators.

```
x <- c("a", "a", "z")
x == "a" | x == "b" | x == "c" | x == "d"
```

Convert the logical vectors of length 3 into a vector of length one. Hint: see help for functions `all()` and `any()`.

With `unique()` we convert a vector of possibly repeated values into a set of unique values. In the algebra of sets, a certain object belongs or not to a set. Consequently, in a set, multiple copies of the same object or value are meaningless.

```
unique(my.set)
## [1] "a" "b" "c"


c("a", "a", "z") %in% unique(my.set)
## [1] TRUE TRUE FALSE
```

In the notation used in algebra of sets, the set union operator is \cup while the intersection operator is \cap . If we have sets A and B , their union is given by $A \cup B$ —in the next three examples, `c("a", "a", "z")` is a constant, while `my.set` is a variable.

```
union(c("a", "a", "z"), my.set)
## [1] "a" "z" "b" "c"
```


If we have sets A and B , their intersection is given by $A \cap B$.

```
intersect(c("a", "a", "z"), my.set)
## [1] "a"
```

 What do you expect to be the difference between the values returned by the three statements in the code chunk below? Before running them, write down your expectations about the value each one will return. Only then run the code. Independently of whether your predictions were correct or not, write down an explanation of what each statement's operation is.

```
union(c("a", "a", "z"), my.set)
c(c("a", "a", "z"), my.set)
c("a", "a", "z", my.set)
```

In the algebra of sets notation $A \subseteq B$, where A and B are sets, indicates that A is a subset or equal to B . For a true subset, the notation is $A \subset B$. The operators with the reverse direction are \supseteq and \supset . Implement these four operations in four R statements, and test them on sets (represented by R vectors) with different “overlap” among set members.

 All set algebra examples above use character vectors and character constants. This is just the most frequent use case. Sets operations are valid on vectors of any atomic class, including `integer`, and computed values can be part of statements. In the second and third statements in the next chunk, we need to use additional parentheses to alter the default order of precedence between arithmetic and set operators.

```
9L %in% 2L:4L
## [1] FALSE

9L %in% ((2L:4L) * (2L:4L))
## [1] TRUE

c(1L, 16L) %in% ((2L:4L) * (2L:4L))
## [1] FALSE TRUE
```

Empty sets are an important component of the algebra of sets, in R they are represented as vectors of zero length. Vectors and lists of zero length, which the R language fully supports, can be used to “encode” emptiness also in other contexts. These vectors do belong to a class such as `numeric` or `character` and must be compatible with other operands in an expression. By default, constructors for vectors, construct empty vectors.

```
length(integer())
## [1] 0

1L %in% integer()
## [1] FALSE

setdiff(1L:4L, union(1L:4L, integer()))
## integer(0)
```

Although set operators are defined for `numeric` vectors, rounding errors in ‘floats’ can result in unexpected results (see section 2.5 on page 35). The next two examples do, however, return the correct answers.

```
9 %in% (2:4)^2
## [1] TRUE

c(1, 5) %in% (1:10)^2
## [1] TRUE FALSE
```


2.7 Character values

Character variables can be used to store any character. Character constants are written by enclosing characters in quotes. There are three types of quotes in the ASCII character set, double quotes `"`, single quotes `'`, and back ticks ```. The first two types of quotes can be used as delimiters of `character` constants.

```
a <- "A"
a
## [1] "A"

b <- 'A'
b
## [1] "A"

a == b
## [1] TRUE
```

 In many computer languages, vectors of characters are distinct from vectors of character strings. In these languages, character vectors store at each index position a single character, while vectors of character strings store at each index position strings of characters of various lengths, such as words or sentences. If you are familiar with C or C++, you need to keep in mind that C’s `char` and R’s `character` are not equivalent and that in R, `character` vectors are vectors of character strings. In contrast to these other languages, in R there is no predefined class for vectors of individual characters and character constants enclosed in double or single quotes are not different.

Concatenating character vectors of length one does not yield a longer character string, it yields instead a longer vector.

```
a <- 'A'
b <- "bcdefg"
c <- "123"
d <- c(a, b, c)
d
## [1] "A"      "bcdefg" "123"
```

Having two different delimiters available makes it possible to choose the type of quotes used as delimiters so that other quotes can be included in a string.

```
a <- "He said 'hello' when he came in"
a
## [1] "He said 'hello' when he came in"

b <- 'He said "hello" when he came in'
b
## [1] "He said \"hello\" when he came in"
```

The outer quotes are not part of the string, they are “delimiters” used to mark the boundaries. As you can see when `b` is printed special characters can be represented using “escape sequences”. There are several of them, and here we will show just four, new line (`\n`) and tab (`\t`), `\` the escape code for a quotation mark within a string and `\\` the escape code for a single backslash `\`. We also show here the different behavior of `print()` and `cat()`, with `cat()` *interpreting* the escape sequences and `print()` displaying them as entered.

```
c <- "abc\\ndef\\tx\\"yz\\"\\\\"tm"
print(c)
## [1] "abc\\ndef\\tx\\"yz\\"\\\\"tm"

cat(c)
## abc
## def x"yz"\\ m
```

The *escape codes* work only in some contexts, as when using `cat()` to generate the output. For example, the new-line escape (`\n`) can be embedded in strings used for axis-label, title or label in a plot to split them over two or more lines.

2.8 The ‘mode’ and ‘class’ of objects

Variables have a *mode* that depends on what is stored in them. But different from other languages, assignment to a variable of a different mode is allowed and in most cases its mode changes together with its contents. However, there is a restriction that all elements in a vector, array or matrix, must be of the same mode. While this is not required for lists, which can be heterogenous. In practice this means that we can assign an object, such as a vector, with a different mode to a name already in use, but we cannot use indexing to assign an object of a different mode to individual members of a vector, matrix or array. Functions with names starting with `is.` are tests returning a logical value, `TRUE`, `FALSE` or `NA`. Function `mode()` returns the mode of an object, as a character string and `typeof()` returns R’s internal type or storage mode.

```
my_var <- 1:5
mode(my_var) # no distinction of integer or double
## [1] "numeric"
```



```
typeof(my_var)
## [1] "integer"

is.numeric(my_var) # no distinction of integer or double
## [1] TRUE

is.double(my_var)
## [1] FALSE

is.integer(my_var)
## [1] TRUE

is.logical(my_var)
## [1] FALSE

is.character(my_var)
## [1] FALSE

my_var <- "abc"
mode(my_var)
## [1] "character"
```

While *mode* is a fundamental property, and limited to those modes defined as part of the R language, the concept of *class*, is different in that new classes can be defined in user code. In particular, different R objects of a given mode, such as `numeric`, can belong to different `classes`. The use of classes for dispatching functions is discussed in section ?? on page ??, in relation to object-oriented programming in R. Method `class()` is used to query the class of an object, and method `inherits()` is used to test if an object belongs to a specific class or not (including “parent” classes, to be later described).

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

inherits(my_var, "character")
## [1] TRUE

inherits(my_var, "numeric")
## [1] FALSE
```

2.9 ‘Type’ conversions

The least-intuitive type conversions are those related to logical values. All others are as one would expect. By convention, functions used to convert objects from one mode to a different one have names starting with `as.`¹.

```
as.character(1)
## [1] "1"
```

¹Except for some packages in the ‘tidyverse’ that use names starting with `as_` instead of `as.`

```
as.numeric("1")
## [1] 1

as.logical("TRUE")
## [1] TRUE

as.logical("NA")
## [1] NA
```

Conversion takes place automatically in arithmetic and logical expressions.

```
TRUE + 10
## [1] 11

1 || 0
## [1] TRUE

FALSE | -2:2
## [1] TRUE TRUE FALSE TRUE TRUE
```



There is some flexibility in the conversion from character strings into `numeric` and `logical` values. Use the examples below plus your own variations to get an idea of what strings are acceptable and correctly converted and which are not. Do also pay attention at the conversion between `numeric` and `logical` values.

```
as.character(3.0e10)
as.numeric("5E+5")
as.numeric("A")
as.numeric(TRUE)
as.numeric(FALSE)
as.logical("T")
as.logical("t")
as.logical("true")
as.logical(100)
as.logical(0)
as.logical(-1)
```



Compare the values returned by `trunc()` and `as.integer()` when applied to a floating point number, such as `12.34`. Check for the equality of values, and for the *class* of the returned objects.



Using conversions, the difference between the length of a `character` vector and the number of characters composing each member “string” within a vector is obvious.

```
f <- c("1", "2", "3")
length(f)
## [1] 3

g <- "123"
length(g)
## [1] 1

as.numeric(f)
## [1] 1 2 3

as.numeric(g)
## [1] 123
```

Other functions relevant to the “conversion” of numbers and other values are `format()`, and `sprintf()`. These two functions return character strings, instead of numeric or other values, and are useful for printing output. One could think of these functions as advanced conversion functions returning formatted, and possibly combined and annotated, character strings. However, they are usually not considered normal conversion functions, as they are very rarely used in a way that preserves the original precision of the input values. We show here the use of `format()` and `sprintf()` with numeric values, but they can also be used with values of other modes.

When using `format()`, the format used to display numbers is set by passing arguments to several different parameters. As `print()` calls `format()` to make numbers *pretty* it accepts the same options.

```
x = c(123.4567890, 1.0)
format(x) # using defaults
## [1] "123.4568" " 1.0000"

format(x[1]) # using defaults
## [1] "123.4568"

format(x[2]) # using defaults
## [1] "1"

format(x, digits = 3, nsmall = 1)
## [1] "123.5" " 1.0"

format(x[1], digits = 3, nsmall = 1)
## [1] "123.5"

format(x[2], digits = 3, nsmall = 1)
## [1] "1.0"

format(x, digits = 3, scientific = TRUE)
## [1] "1.23e+02" "1.00e+00"
```

Function `sprintf()` is similar to C’s function of the same name. The user interface is rather unusual, but very powerful, once one learns the syntax. All the formatting is specified using a character string as template. In this template, placeholders for data and the formatting instructions are embedded using special codes.

These codes start with a percent character. We show in the example below the use of some of these: `f` is used for `numeric` values to be formatted according to a “fixed point,” while `g` is used when we set the number of significant digits and `e` for exponential or *scientific* notation.

```
x = c(123.4567890, 1.0)
sprintf("The numbers are: %4.2f and %.0f", x[1], x[2])
## [1] "The numbers are: 123.46 and 1"

sprintf("The numbers are: %4.2g and %.2g", x[1], x[2])
## [1] "The numbers are: 123.5 and 1"

sprintf("The numbers are: %4.2e and %.0e", x[1], x[2])
## [1] "The numbers are: 1.23e+02 and 1e+00"
```

In the template "The numbers are: %4.2f and %.0f", there are two placeholders for `numeric` values, %4.2f and %.0f, so in addition to the template, we pass two values extracted from the first two positions of vector `x`. These could have been two different vectors of length one, or even numeric constants. The template itself does not need to be a `character` constant as in these examples, as a variable can be also passed as argument.



Function `format()` may be easier to use, in some cases, but `sprintf()` is more flexible and powerful. Those with experience in the use of the C language will already know about `sprintf()` and its use of templates for formatting output. Even if you are familiar with C, look up the help pages for both functions, and practice, by trying to create the same formatted output by means of the two functions. Do also play with these functions with other types of data like `integer` and `character`.



We have above described `NA` as a single value ignoring modes, but in reality `NA` s come in various flavors. `NA_real_`, `NA_character_`, etc. and `NA` defaults to an `NA` of class `logical`. `NA` is normally converted on the fly to other modes when needed, so in general `NA` is all we need to use.

```
a <- c(1, NA)
is.numeric(a[2])
## [1] TRUE

is.numeric(NA)
## [1] FALSE

b <- c("abc", NA)
is.character(b[2])
## [1] TRUE

is.character(NA)
## [1] FALSE

class(NA)
## [1] "logical"

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

```
c <- NA
c(c, 2:3)
## [1] NA 2 3
```

However, even the statement below works transparently.

```
a[3] <- b[2]
```

2.10 Vector manipulation

If you have read earlier sections of this chapter, you already know how to create a vector. R’s vectors are equivalent to what would be written in mathematical notation as $a_{1\dots n} = a_1, a_2, \dots, a_i, \dots, a_n$, they are not the equivalent to the vectors, common in Physics, which are symbolized with an arrow as an “accent,” such as \vec{F} .

In this section we are going to see how to extract or retrieve, replace, and move elements such as a_2 from a vector. Elements are extracted using an index enclosed in single square brackets. The index indicates the position in the vector, starting from one, following the usual mathematical tradition. We here use a vector of characters to make it clear the distinction between contents of the vector from the numerical indexes. What in maths notation would be a_i for a vector $a_{1\dots n}$, in R is represented as `a[i]` and the whole vector, by excluding the brackets and indexing vector, as `a`.

```
a <- letters[1:10]
a
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[2]
## [1] "b"
```



Four constant vectors are available in R: `letters`, `LETTERS`, `month.name` and `month.abb`, of which we used `letters` in the example above. These vectors are always for English, irrespective of the locale.




In R, indexes always start from one, while in some other programming languages such as C and C++, indexes start from zero. It is important to be aware of this difference, as many computation algorithms are valid only under a given indexing convention.

It is possible to extract a subset of the elements of a vector in a single operation, using a vector of indexes. The positions of the extracted elements in the result (“returned value”) are determined by the ordering of the members of the vector of indexes—easier to demonstrate than to explain.

```
a[c(3, 2)]
## [1] "c" "b"

a[10:1]
## [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

 The length of the indexing vector is not restricted by the length of the indexed vector. However, only numerical indexes that match positions present in the indexed vector can extract values. Those values in the indexing vector pointing to positions that are not present in the indexed vector, result in `NA`s. This is easier to learn by *playing* with R, than from explanations. Play with R, using the following examples as a starting point.

```
length(a)
a[c(3, 3, 3, 3)]
a[c(10:1, 1:10)]
a[c(1, 11)]
a[11]
```


Have you tried some of your own examples? If not yet, do *play* with additional variations of your own before continuing.

Negative indexes have a special meaning; they indicate the positions at which values should be excluded. Be aware that it is *illegal* to mix positive and negative values in the same indexing operation.

```
a[-2]
## [1] "a" "c" "d" "e" "f" "g" "h" "i" "j"

a[-c(3,2)]
## [1] "a" "d" "e" "f" "g" "h" "i" "j"

a[-3:-2]
## [1] "a" "d" "e" "f" "g" "h" "i" "j"
```

 Results from indexing with special values and zero may be surprising. Try to build a rule from the examples below, a rule that will help you remember what to expect next time you are confronted with similar statements using “subscripts” which are special values instead of integers larger or equal to one—this is likely to happen sooner or later as these special values can be returned by different R expressions depending on the value of operands or function arguments, some of them described earlier in this chapter.

```
a[ ]
a[0]
a[numeric(0)]
a[NA]
a[c(1, NA)]
a[NULL]
a[c(1, NULL)]
```

Another way of indexing, which is very handy, but not available in most other programming languages, is indexing with a vector of `logical` values. The `logical` vector used for indexing is usually of the same length as the vector from which elements are going to be selected. However, this is not a requirement, because if the `logical` vector of indexes is shorter than the indexed vector, it is “recycled” as discussed above in relation to other operators.

```
a[TRUE]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[FALSE]
## character(0)

a[c(TRUE, FALSE)]
## [1] "a" "c" "e" "g" "i"

a[c(FALSE, TRUE)]
## [1] "b" "d" "f" "h" "j"

a > "c"
## [1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

a[a > "c"]
## [1] "d" "e" "f" "g" "h" "i" "j"
```

Indexing with logical vectors is very frequently used in R because comparison operators are vectorized. Comparison operators, when applied to a vector, return a `logical` vector, a vector that can be used to extract the elements for which the result of the comparison test was `TRUE`.



The examples in this text box demonstrate additional uses of logical vectors: 1) the logical vector returned by a vectorized comparison can be stored in a variable, and the variable used as a “selector” for extracting a subset of values from the same vector, or from a different vector.

```
a <- letters[1:10]
b <- 1:10
selector <- a > "c"
selector
a[selector]
b[selector]
```

Numerical indexes can be obtained from a logical vector by means of function `which()`.

```
indexes <- which(a > "c")
indexes
a[indexes]
b[indexes]
```

Make sure to understand the examples above. These constructs are very widely used in R because they allow for concise code that is easy to understand once you are familiar with the indexing rules. However, if you do not command these rules, many of these terse statements will be unintelligible to you.

Indexing can be used on either side of an assignment expression. In the chunk below, we use the extraction operator on the left-hand side of the assignments to replace values only at selected positions in the vector. This may look rather esoteric at first sight, but it is just a simple extension of the logic of indexing described above. It works, because the low precedence of the `<-` operator results in both the left-hand side and the right-hand side being fully evaluated before the assignment takes place. To make the changes to the vectors easier to follow, we use identical vectors with different names for each of these examples.

```
a <- 1:10
a
## [1] 1 2 3 4 5 6 7 8 9 10

a[1] <- 99
a
## [1] 99 2 3 4 5 6 7 8 9 10

b <- 1:10
b[c(2,4)] <- -99 # recycling
b
## [1] 1 -99 3 -99 5 6 7 8 9 10

c <- 1:10
c[c(2,4)] <- c(-99, 99)
c
## [1] 1 -99 3 99 5 6 7 8 9 10

d <- 1:10
d[TRUE] <- 1 # recycling
d
## [1] 1 1 1 1 1 1 1 1 1 1

e <- 1:10
e <- 1 # no recycling
e
## [1] 1
```


We can also use subscripting on both sides of the assignment operator, for example, to swap two elements.

```
a <- letters[1:10]
a[1:2] <- a[2:1]
a
## [1] "b" "a" "c" "d" "e" "f" "g" "h" "i" "j"
```



Do play with subscripts to your heart's content, really grasping how they work and how they can be used, will be very useful in anything you do in the future with R. Even the contrived example below follows the same simple rules, just study it bit by bit. Hint: the second statement in the chunk below, modifies `a`, so, when studying variations of this example you will need to recreate `a` by executing the first statement, each time you run a variation of the second statement.


```
a <- letters[1:10]
a[5:1] <- a[c(TRUE,FALSE)]
a
## [1] "i" "g" "e" "c" "a" "f" "g" "h" "i" "j"
```

 In R, indexing with positional indexes can be done with `integer` or `numeric` values. Numeric values can be floats, but for indexing, only integer values are meaningful. Consequently, `double` values are converted into `integer` values when used as indexes. The conversion is done invisibly, but it does slow down computations slightly. When working on big data sets, explicitly using `integer` values can improve performance.

```
b <- LETTERS[1:10]
b
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[1]
## [1] "A"

b[1.1]
## [1] "A"

b[1.9999] # surprise!!
## [1] "A"

b[2]
## [1] "B"
```

From this experiment, we can learn that if positive indexes are not whole numbers, they are truncated to the next smaller integer.

```
b <- LETTERS[1:10]
b
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-1]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-1.1]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-1.9999]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-2]
## [1] "A" "C" "D" "E" "F" "G" "H" "I" "J"
```

From this experiment, we can learn that if negative indexes are not whole numbers, they are truncated to the next larger (less negative) integer. In conclusion, `double` index values behave as if they were sanitized using function `trunc()`.

This example also shows how one can tease out of R its rules through experimentation.

A frequent operation on vectors is sorting them into an increasing or decreasing order. The most direct approach is to use `sort()`.

```
my.vector <- c(10, 4, 22, 1, 4)
sort(my.vector)
## [1] 1 4 4 10 22


sort(my.vector, decreasing = TRUE)
## [1] 22 10 4 4 1
```

An indirect way of sorting a vector, possibly based on a different vector, is to generate with `order()` a vector of numerical indexes that can be used to achieve the ordering.

```
order(my.vector)
## [1] 4 2 5 1 3

my.vector[order(my.vector)]
## [1] 1 4 4 10 22

another.vector <- c("ab", "aa", "c", "zy", "e")
another.vector[order(my.vector)]
## [1] "zy" "aa" "e" "ab" "c"
```

 A problem linked to sorting that we may face is counting how many copies of each value are present in a vector. We need to use two functions `sort()` and `rle()`. The second of these functions computes *run length* as used in *run length encoding* for which *rle* is an abbreviation. A *run* is a series of consecutive identical values. As the objective is to count the number of copies of each value present, we need first to sort the vector.

```
my.letters <- letters[c(1,5,10,3,1,4,21,1,10)]
my.letters
## [1] "a" "e" "j" "c" "a" "d" "u" "a" "j"

sort(my.letters)
## [1] "a" "a" "a" "c" "d" "e" "j" "j" "u"

rle(sort(my.letters))
## Run Length Encoding
## lengths: int [1:6] 3 1 1 1 2 1
## values : chr [1:6] "a" "c" "d" "e" "j" "u"
```

The second and third statements are only to demonstrate the effect of each step. The last statement uses nested function calls to compute the number of copies of each value in the vector.

2.11 Matrices and multidimensional arrays

Vectors have a single dimension, and, as we saw above, we can query their length with method `length()`. Matrices have two dimensions, which can be queried with `dim()`, `ncol()` and `nrow()`. R arrays can have any number of dimensions, even a single dimension, which can be queried with method `dim()`. As expected `is.vector()`, `is.matrix()` and `is.array()` can be used to query the class. In mathematical notation a matrix is denoted as follows

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix}$$

where A represents the whole matrix and the a_{ij} its elements, with i indexing rows and j indexing columns. The two dimensions of the matrix are given by m and n , for rows and columns.

In R we can create a matrix using the `matrix()` or `as.matrix()` constructors. The first argument of `matrix()` is a vector. In the same way as vectors, matrices are homogeneous, all elements are of the same type.

```
matrix(1:15, ncol = 3)
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15

matrix(1:15, nrow = 3)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
```

When a matrix is printed in R the row and column indexes are indicated on the edges left and top margins, in the same way as they would be used to extract whole rows and columns.

When a vector is converted to a matrix, R’s default is to allocate the values in the vector to the matrix starting from the leftmost column, and within the column, down from the top. Once the first column is filled, the process continues from the top of the next column, as can be seen above. This order can be changed as you will discover in the playground below.



Check in the help page for the `matrix` constructor how to use the `byrow` parameter to alter the default order in which the elements of the vector are allocated to columns and rows of the new matrix.

```
help(matrix)
```

While you are looking at the help page, also consider the default number of columns and rows.

```
matrix(1:15)
```

And to start getting a sense of how to interpret error and warning messages, run the code below and make sure you understand which problem is being reported. Before executing the statement, analyze it and predict what the returned value will be. Afterwards, compare your prediction, to the value actually returned.

```
matrix(1:15, ncol = 2)
```

Subscripting of matrices and arrays is consistent with that used for vectors; we only need to supply an indexing vector, or leave a blank space, for each dimension. A matrix has two dimensions, so to access any element or group of elements, we use two indices. The only complication is that there are two possible orders in which, in principle, indexes could be supplied. In R, indexes for matrices are written “row first.” In simpler words, the first index value selects rows, and the second one, columns.

```
A <- matrix(1:20, ncol = 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

A[1, 1]
## [1] 1
```

Remind yourself of how indexing of vectors works in R (see section 2.10 on page 48). We will now apply the same rules in two dimensions. The first or leftmost indexing vector corresponds to rows and the second one to columns, so R uses a rows-first convention for indexing. Missing indexing vectors are interpreted as meaning *extract all rows* and *extract all columns*, respectively.

```
A[1, ]
## [1] 1 6 11 16


A[ , 1]
## [1] 1 2 3 4 5

A[2:3, c(1,3)]
##      [,1] [,2]
## [1,]    2   12
## [2,]    3   13

A[3, 4] <- 99
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   99
## [4,]    4    9   14   19
## [5,]    5   10   15   20

A[4:3, 2:1] <- A[3:4, 1:2]
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    9    4   13   99
## [4,]    8    3   14   19
## [5,]    5   10   15   20
```

 Matrices can be indexed as vectors, without triggering an error or warning.

```
A <- matrix(1:20, ncol = 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

dim(A)
## [1] 5 4

A[10]
## [1] 10

A[5, 2]
## [1] 10
```


The next code example demonstrates that indexing as a vector with a single index, always works column-wise even if matrix `B` was created by assigning vector elements by row.

```
B <- matrix(1:20, ncol = 4, byrow = TRUE)
B
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20

dim(B)
## [1] 5 4

B[10]
## [1] 18

B[5, 2]
## [1] 18
```

 In R, a `matrix` can have a single row, a single column, a single element or no elements. However, in all cases, a `matrix` will have a *dimensions* attribute of length two defined.

```
my.vector <- 1:6
dim(my.vector)
## NULL

one.col.matrix <- matrix(1:6, ncol = 1)
dim(one.col.matrix)
## [1] 6 1

two.col.matrix <- matrix(1:6, ncol = 2)
dim(two.col.matrix)
## [1] 3 2

one.elem.matrix <- matrix(1, ncol = 1)
dim(one.elem.matrix)
## [1] 1 1

no.elem.matrix <- matrix(numeric(), ncol = 0)
dim(no.elem.matrix)
## [1] 0 0
```

Arrays are similar to matrices, but can have more than two dimensions, which are specified with the `dim` argument to the `array()` constructor.

```
B <- array(1:27, dim = c(3, 3, 3))
B
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27
##
B[2, 2, 2]
## [1] 14
```

In the chunk above, the length of the supplied vector is the product of the dimensions, $27 = 3 \times 3 \times 3 = 3^3$. Arrays are printed in slices, with slices across

3rd and higher dimensions printed separately, with their corresponding indexes above each slice and the first two dimensions on the margins of the individual slices, similarly to how matrices are displayed.



How do you use indexes to extract the second element of the original vector, in each of the following matrices and arrays?

```
v <- 1:10
m2c <- matrix(v, ncol = 2)
m2cr <- matrix(v, ncol = 2, byrow = TRUE)
m2r <- matrix(v, nrow = 2)
m2rc <- matrix(v, nrow = 2, byrow = TRUE)
```

```
v <- 1:10
a2c <- array(v, dim = c(5, 2))
a2c <- array(v, dim = c(5, 2), dimnames = list(NULL, c("c1", "c2")))
a2r <- array(v, dim = c(2, 5))
```

Be aware that vectors and one-dimensional arrays are not the same thing, while two-dimensional arrays are matrices.

1. Use the different constructors and query methods to explore this, and its consequences.
2. Convert a matrix into a vector using `unlist()` and `as.vector()` and compare the returned values.

Operators for matrices are available in R, as matrices are used in many statistical algorithms. We will not describe them all here, only `t()` and some specializations of arithmetic operators. Function `t()` transposes a matrix, by swapping columns and rows.

```
A <- matrix(1:20, ncol = 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

t(A)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
```

As with vectors, recycling applies to arithmetic operators when applied to matrices.

```
A + 2
##      [,1] [,2] [,3] [,4]
```

```
## [1,] 3 8 13 18
## [2,] 4 9 14 19
## [3,] 5 10 15 20
## [4,] 6 11 16 21
## [5,] 7 12 17 22

A * 0:1
##      [,1] [,2] [,3] [,4]
## [1,] 0 6 0 16
## [2,] 2 0 12 0
## [3,] 0 8 0 18
## [4,] 4 0 14 0
## [5,] 0 10 0 20

A * 1:0
##      [,1] [,2] [,3] [,4]
## [1,] 1 0 11 0
## [2,] 0 7 0 17
## [3,] 3 0 13 0
## [4,] 0 9 0 19
## [5,] 5 0 15 0
```

In the examples above with the usual multiplication operator `*`, the operation described is not a matrix product, but instead, the products between individual elements of the matrix and vectors. Matrix multiplication is indicated by operator `%*%`.

```
B <- matrix(1:16, ncol = 4)
B * B
##      [,1] [,2] [,3] [,4]
## [1,] 1 25 81 169
## [2,] 4 36 100 196
## [3,] 9 49 121 225
## [4,] 16 64 144 256

B %*% B
##      [,1] [,2] [,3] [,4]
## [1,] 90 202 314 426
## [2,] 100 228 356 484
## [3,] 110 254 398 542
## [4,] 120 280 440 600
```

Other operators and functions for matrix algebra like cross-product (`crossprod()`), extracting or replacing the diagonal (`diag()`) are available in base R. Packages, including ‘matrixStats’, provide additional functions and operators for matrices.

2.12 Factors

Factors are very important in R. In contrast to other statistical software in which the role of a variable is set when defining a model to be fitted or when setting up a test, in R, models are specified exactly in the same way for ANOVA and regression anal-

ysis, both as *linear models*. The type of model that is fitted is decided by whether the explanatory variable is a factor (giving ANOVA) or a numerical variable (giving regression). This makes a lot of sense, because in most cases, considering an explanatory variable as categorical or not, depends on the quantity stored and/or the design of the experiment or survey. In other words, being categorical is a property of the data. The order of the levels in an unordered **factor** does not affect simple calculations or the values plotted, but as we will see in chapters ?? and ??, it does affect how the output is printed, the order of the levels in the scales and keys of plots, and in some cases how contrasts are applied in significance tests.


In a factor, values indicate discrete unordered categories, most frequently the treatments in an experiment, or categories in a survey. They can be created either from numerical or character vectors. The different possible values are called *levels*. Factors created with **factor()** are always unordered or categorical. R also supports ordered factors, created with function **ordered()** with identical user interface. The distinction, however, only affects how they are interpreted in statistical tests as discussed in chapter ??.

When using **factor()** or **ordered()** we create a factor from a vector, but this vector can be created on-the-fly and anonymous as shown in this example. When the vector is **numeric** and no labels are supplied, level labels are character strings matching the numbers. The default ordering of the levels is alphanumerical.

```
factor(x = c(1, 2, 2, 1, 2, 1, 1))
## [1] 1 2 2 1 2 1 1
## Levels: 1 2

ordered(x = c(1, 2, 2, 1, 2, 1, 1))
## [1] 1 2 2 1 2 1 1
## Levels: 1 < 2

factor(x = c(1, 2, 2, 1, 2, 1, 1), ordered = TRUE)
## [1] 1 2 2 1 2 1 1
## Levels: 1 < 2
```

 When the pattern of levels is regular, it is possible to use function **gl()**, *generate levels*, to construct a factor. Nowadays, it is usual to read data into R from files in which the treatment codes are already available as character strings or numeric values, however, when we need to create a factor within R, **gl()** can save some typing. In this case instead of passing a vector as argument, we pass a *recipe* to create it: **n** is the number of levels, and **k** the number of contiguous repeats (called “replicates” in R documentation) and **length** the length of the factor to be created.

```
gl(n = 2, k = 5, labels = c("A", "B"))
## [1] A A A A A B B B B B
## Levels: A B

gl(n = 2, k = 1, length = 10, labels = c("A", "B"))
## [1] A B A B A B A B A B
## Levels: A B
```

It is always preferable to use meaningful labels for levels, even if R does not

require it. Here the vector is stored in a variable named `my.vector`. In a real data analysis situation in most cases the vector would have been read from a file on disk and would be longer.

```
my.vector <- c("treated", "treated", "control", "control", "control", "treated")
factor(my.vector)
## [1] treated treated control control control treated
## Levels: control treated
```

The ordering of levels is established at the time a factor is created, and by default is alphabetical. This default ordering of levels is frequently not the one needed. We can pass an argument to parameter `levels` of function `factor()` to set a different ordering of the levels.

```
factor(x = my.vector, levels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control
```

The labels (“names”) of the levels can be set when calling `factor()`. Two vectors are passed as arguments to parameters `levels` and `labels` with levels and matching labels in the same position. The argument passed to `levels` determines the order of the levels based on their old names or values, and the argument passed to `labels` gives new names to the levels.

```
factor(x = c(1, 1, 0, 0, 0, 1), levels = c(1, 0), labels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control
```


In the examples above we passed a numeric vector or a character vector as an argument for parameter `x` of function `factor()`. It is also possible to pass a `factor` as an argument to parameter `x`. This makes it possible to modify the ordering of levels or replace the labels in a factor.

```
my.factor <- factor(x = my.vector)
my.factor
## [1] treated treated control control control treated
## Levels: control treated

factor(x = my.factor, levels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control

factor(x = my.factor, labels = c(control = "cooled", treated = "heated"))
## [1] heated heated cooled cooled cooled heated
## Levels: cooled heated

factor(x = my.factor,
      levels = c("treated", "control"),
      labels = c("heated", "cooled"))
## [1] heated heated cooled cooled cooled heated
## Levels: heated cooled
```

 **Merging factor levels.** We use `factor()` as shown below, setting the same label for the levels we want to merge.

```
my.factor1 <- gl(4, 3, labels = c("A", "F", "B", "Z"))
my.factor1
## [1] A A A F F F B B B Z Z Z
## Levels: A F B Z

factor(my.factor1,
       levels = c("A", "B", "F", "Z"),
       labels = c("A", "B", "C", "C"))
## [1] A A A C C C B B B C C C
## Levels: A B C
```

We can use indexing on factors in the same way as with vectors. In the next example, we use a test returning a logical vector to extract all “controls.” We use function `levels()` to look at the levels of the factors, as with vectors, `length()` to query the number of values stored.

```
my.factor
## [1] treated treated control control control treated
## Levels: control treated

levels(my.factor)
## [1] "control" "treated"

length(my.factor)
## [1] 6


control.factor <- my.factor[my.factor == "control"]
control.factor
## [1] control control control
## Levels: control treated

levels(control.factor) # same as in my.factor
## [1] "control" "treated"

length(control.factor) # shorter than my.factor
## [1] 3

control.factor <- factor(control.factor)
levels(control.factor) # the unused level was dropped
## [1] "control"
```

It can be seen above that subsetting does not drop unused factor levels, and that `factor()` can be used to explicitly drop the unused factor levels.


 How to convert factors into numeric vectors is not obvious, even when the factor was created from a `numeric` vector.

```
my.vector2 <- rep(3:5, 4)
my.vector2
## [1] 3 4 5 3 4 5 3 4 5 3 4 5

my.factor2 <- factor(my.vector2)
my.factor2
## [1] 3 4 5 3 4 5 3 4 5 3 4 5
## Levels: 3 4 5

as.numeric(my.factor2)
## [1] 1 2 3 1 2 3 1 2 3 1 2 3


as.numeric(as.character(my.factor2))
## [1] 3 4 5 3 4 5 3 4 5 3 4 5
```


 **Why is a double conversion needed?** Internally, factor values are stored as running integers starting from one, each distinct integer value corresponding to a level. These underlying integer values are returned by `as.numeric()` when applied to a factor instead of the level labels. The labels of the factor levels are always stored as character strings, even when these characters are digits. In contrast to `as.numeric()`, `as.character()` returns the character labels of the levels for each of the values stored in the factor. If these character strings represent numbers, they can be converted, in a second step, using `as.numeric()` into the original numeric values. Use of `class` and `mode` is described on section 2.8 on page 43, and `str()` on page 66.

```
class(my.factor2)
## [1] "factor"

mode(my.factor2)
## [1] "numeric"

str(my.factor2)
## Factor w/ 3 levels "3","4","5": 1 2 3 1 2 3 1 2 3 1 ...
```

 Create a factor with levels labeled with words. Create another factor with the levels labeled with the same words, but ordered differently. After this convert both factors to numeric vectors using `as.numeric()`. Explain why the two numeric vectors differ or not from each other.

 **Safely reordering and renaming factor levels.** The simplest approach is to use `factor()` and its `levels` parameter as shown above. In these more advanced examples we use `levels()` to retrieve the names of the levels from the factor itself to protect from possible bugs due to typing mistakes, or for changes in the naming conventions used.

Reverse previous order using `rev()`.

```
my.factor2 <- factor(c("treated", "treated", "control", "control", "control", "treated"))
levels(my.factor2)
## [1] "control" "treated"

my.factor2 <- factor(my.factor2, levels = rev(levels(my.factor2)))
levels(my.factor2)
## [1] "treated" "control"
```

Sort in decreasing order, i.e., opposite to default.

```
my.factor2 <- factor(my.factor2,
                     levels = sort(levels(my.factor2), decreasing = TRUE))
levels(my.factor2)
## [1] "treated" "control"
```

Alter ordering using subscripting; especially useful with three or more levels.

```
my.factor2 <- factor(my.factor2, levels = levels(my.factor2)[c(2, 1)])
levels(my.factor2)
## [1] "control" "treated"
```

Reordering the levels of a factor based on summary quantities from data stored in a numeric vector is very useful, especially when plotting. Function `reorder()` can be used in this case. It defaults to using `mean()` for summaries, but other suitable summary functions, such as `median()` can be supplied in its place.

```
my.factor3 <- gl(2, 5, labels = c("A", "B"))
my.vector3 <- c(5.6, 7.3, 3.1, 8.7, 6.9, 2.4, 4.5, 2.1, 1.4, 2.0)
my.factor3
## [1] A A A A A B B B B B
## Levels: A B

my.factor3ord <- reorder(my.factor3, my.vector3)
levels(my.factor3ord)
## [1] "B" "A"

my.factor3rev <- reorder(my.factor3, -my.vector3) # a simple trick
levels(my.factor3rev)
## [1] "A" "B"
```

In the last statement, using the unary negation operator, which is vectorized, allows us to easily reverse the ordering of the levels, while still using the default function, `mean()`, to summarize the data.



Reordering factor values. It is possible to arrange the values stored in a factor either alphabetically according to the labels of the levels or according to the order of the levels. (The use of `rep()` is explained on page 25.)

```
# gl() keeps order of levels
my.factor4 <- gl(4, 3, labels = c("A", "F", "B", "Z"))
my.factor4
as.integer(my.factor4)
# factor() orders levels alphabetically
my.factor5 <- factor(rep(c("A", "F", "B", "Z"), rep(3,4)))
my.factor5
as.integer(my.factor5)
levels(my.factor5)[as.integer(my.factor5)]
```

We see above that the integer values by which levels in a factor are stored, are equivalent to indices or “subscripts” referencing the vector of labels. Function `sort()` operates on the values’ underlying integers and sorts according to the order of the levels while `order()` operates on the values’ labels and returns a vector of indices that arrange the values alphabetically.

```
sort(my.factor4)
my.factor4[order(my.factor4)]
my.factor4[order(as.integer(my.factor4))]
```

Run the examples in the chunk above and work out why the results differ.

2.13 Lists

Lists’ main difference from vectors is, in R, that they can be heterogeneous. In R, the members of a list can be considered as following a sequence, and accessible through numerical indexes, the same as members of vectors. Members of a list as well as members of a vector can be named, and retrieved (indexed) through their names. In practice, named lists are more frequently used than named vectors. Lists are created using function `list()` similarly as `c()` is used for vectors. Members of a list can be vectors differing both in their class and in their length.

```
a.list <- list(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.list
## $x
## [1] 1 2 3 4 5 6
##
## $y
## [1] "a"
##
## $z
## [1] TRUE FALSE
```

2.13.1 Member extraction and subsetting

Using double square brackets for indexing a list extracts the element stored in the list, in its original mode. In the example above, `a.list[["x"]]` returns a numeric vector, while `a.list[1]` returns a list containing the numeric vector `x`. `a.list$x`

returns the same value as `a.list[["x"]]`, a numeric vector. `a.list[c(1,3)]` returns a list of length two, while `a.list[[c(1,3)]]` is an error.

```
a.list$x
## [1] 1 2 3 4 5 6

a.list[["x"]]
## [1] 1 2 3 4 5 6


a.list[[1]]
## [1] 1 2 3 4 5 6

a.list["x"]
## $x
## [1] 1 2 3 4 5 6

a.list[1]
## $x
## [1] 1 2 3 4 5 6

a.list[c(1,3)]
## $x
## [1] 1 2 3 4 5 6
##
## $z
## [1] TRUE FALSE

try(a.list[[c(1,3)]])
## [1] 3
```

 *Lists*, as usually defined in languages like C, are based on pointers to memory locations, with pointers stored at each node. These pointers chain or link the different member nodes (this allows, for example, sorting of lists in place by modifying the pointers). In such implementations, indexing by position is not possible, or at least requires “walking” down the list, node by node. In R, `list` members can be accessed through positional indexes, similarly to vectors. Of course, insertions and deletions in the middle of a list, whatever their implementation, alter (or *invalidate*) any position-based indexes.

To investigate the members contained in a list, function `str()` (*structure*) is of help, especially when lists have many members. The `print()` method for the structure formats lists more compactly than function `print()` applied directly to a list.

```
str(a.list)
## List of 3
## $ x: int [1:6] 1 2 3 4 5 6
## $ y: chr "a"
## $ z: logi [1:2] TRUE FALSE
```

2.13.2 Adding and removing list members

The two most common simple operations on lists are insertions and deletions. In R, function `append()` can be used both to append elements at the end of a list and insert elements into the head or any position in the middle of a list.

```
another.list <- append(a.list, list(yy = 1:10, zz = letters[5:1]), 2L)
another.list
## $x
## [1] 1 2 3 4 5 6
##
## $y
## [1] "a"
##
## $yy
## [1] 1 2 3 4 5 6 7 8 9 10
##
## $zz
## [1] "e" "d" "c" "b" "a"
##
## $z
## [1] TRUE FALSE
```

To delete a member from a list we assign `NULL` to it.

```
a.list$y <- NULL
a.list
## $x
## [1] 1 2 3 4 5 6
##
## $z
## [1] TRUE FALSE
```

2.13.3 Nested lists

Lists can be nested, i.e., lists of lists can be constructed to an arbitrary depth.


```
a.list <- list("a", "aa", "aaa")
b.list <- list("b", "bb")
nested.list <- list(A = a.list, B = b.list)
str(nested.list)
## List of 2
## $ A:List of 3
## ..$ : chr "a"
## ..$ : chr "aa"
## ..$ : chr "aaa"
## $ B:List of 2
## ..$ : chr "b"
## ..$ : chr "bb"
```

A nested list can alternatively be constructed within a single statement in which several member lists are created. Here we combine the first three statements in the earlier chunk into a single one.



```
nested.list <- list(A = list("a", "aa", "aaa"), B = list("b", "bb"))
str(nested.list)
## List of 2
## $ A:List of 3
## ..$ : chr "a"
## ..$ : chr "aa"
## ..$ : chr "aaa"
## $ B:List of 2
## ..$ : chr "b"
## ..$ : chr "bb"
```

A list can contain a combination of list and vector members.


```
nested.list <- list(A = list("a", "aa", c(2:5)),
                  B = list("b", "bb"),
                  C = c(1, 3, 9))
str(nested.list)
## List of 3
## $ A:List of 3
## ..$ : chr "a"
## ..$ : chr "aa"
## ..$ : int [1:4] 2 3 4 5
## $ B:List of 2
## ..$ : chr "b"
## ..$ : chr "bb"
## $ C: num [1:3] 1 3 9
```

 The logic behind extraction of members of nested lists using indexing is the same as for simple lists, but applied recursively—e.g., `nested.list[[2]]` extracts the second member of the outermost list, which is another list. As, this is a list, its members can be extracted using again the extraction operator: `nested.list[[2]][[1]]`. It is important to remember that these concatenated extraction operations are written so that the leftmost operator is applied to the outermost list.

The example above uses the `[[]]` operator, but the left to right precedence also applies to concatenated calls to `[]`.

 What do you expect each of the statements below to return? *Before running the code*, predict what value and of which mode each statement will return. You may use implicit or explicit calls to `print()`, or calls to `str()` to visualize the structure of the different objects.

```
nested.list <- list(A = list("a", "aa", "aaa"), B = list("b", "bb"))
str(nested.list)
nested.list[2:1]
nested.list[1]
nested.list[[1]][2]
nested.list[[1]][[2]]
nested.list[2]
nested.list[2][[1]]
```

 When dealing with deep lists, it is sometimes useful to limit the number of levels of nesting returned by `str()` by means of a `numeric` argument passed to parameter `max.level`s.

```
str(nested.list, max.level = 1)
## List of 3
## $ A:List of 3
## $ B:List of 2
## $ C: num [1:3] 1 3 9
```

Sometimes we need to flatten a list, or a nested structure of lists within lists. Function `unlist()` is what should be normally used in such cases.

The list `nested.list` is a nested system of lists, but all the “terminal” members are character strings. In other words, terminal nodes are all of the same mode, allowing the list to be “flattened” into a character vector.

```
nested.list <- list(A = list("a", "aa", "aaa"), B = list("b", "bb"))
c.vec <- unlist(nested.list)
c.vec
##      A1      A2      A3      B1      B2
##      "a"     "aa"    "aaa"    "b"     "bb"

is.list(nested.list)
## [1] TRUE

is.list(c.vec)
## [1] FALSE

mode(nested.list)
## [1] "list"

mode(c.vec)
## [1] "character"

names(nested.list)
## [1] "A" "B"


names(c.vec)
## [1] "A1" "A2" "A3" "B1" "B2"
```

The returned value is a vector with named member elements. We use function `str()` to figure out how this vector relates to the original list. The names, always of mode character, are based on the names of list elements when available, while characters depicting positions as numbers are used for anonymous nodes. We can access the members of the vector either through numeric indexes or names.

```
str(c.vec)
## Named chr [1:5] "a" "aa" "aaa" "b" "bb"
## - attr(*, "names")= chr [1:5] "A1" "A2" "A3" "B1" ...

c.vec[2]
##      A2
##      "aa"

c.vec["A2"]
##      A2
##      "aa"
```

 Function `unlist()` has two additional parameters, with default argument values, which we did not modify in the example above. These parameters are `recursive` and `use.names`, both of them expecting a `logical` value as an argument. Modify the statement `c.vec <- unlist(c.list)`, by passing `FALSE` as an argument to these two parameters, in turn, and in each case, study the value returned and how it differs with respect to the one obtained above.

Function `unname()` can be used to remove names safely—i.e., without risk of altering the mode or class of the object.

```
unname(c.vec)
## [1] "a" "aa" "aaa" "b" "bb"

unname(nested.list)
## [[1]]
## [[1]][[1]]
## [1] "a"
##
## [[1]][[2]]
## [1] "aa"
##
## [[1]][[3]]
## [1] "aaa"
##
##
## [[2]]
## [[2]][[1]]
## [1] "b"
##
## [[2]][[2]]
## [1] "bb"
```

2.14 Data frames

Data frames are a special type of list, in which each element is a vector or a factor of the same length (or rarely a matrix with the same number of rows as the enclosing data frame). They are created with function `data.frame()` with a syntax similar to that used for lists—in object-oriented programming we say that data frames are derived from class `list`. As the expectation is equal length, if vectors of different lengths are supplied as arguments, the shorter vector(s) is/are recycled, possibly several times, until the required full length is reached.

```
a.df <- data.frame(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.df
##   x y    z
## 1 1 a TRUE
## 2 2 a FALSE
## 3 3 a TRUE
## 4 4 a FALSE
## 5 5 a TRUE
## 6 6 a FALSE
```

```

str(a.df)
## 'data.frame': 6 obs. of  3 variables:
##  $ x: int  1 2 3 4 5 6
##  $ y: chr  "a" "a" "a" "a" ...
##  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE

class(a.df)
## [1] "data.frame"

mode(a.df)
## [1] "list"

is.data.frame(a.df)
## [1] TRUE

is.list(a.df)
## [1] TRUE

```

Extraction of individual member variables or “columns” can be done like in a list with operator `[[]]`.

```

a.df$x
## [1] 1 2 3 4 5 6

a.df[["x"]]
## [1] 1 2 3 4 5 6

a.df[[1]]
## [1] 1 2 3 4 5 6

class(a.df[["x"]])
## [1] "integer"

```

With function `class()` we can query the class of an R object (see section 2.8 on page 43). As we saw in the two previous chunks, `list` and `data.frame` objects belong to two different classes. However, their relationship is based on a hierarchy of classes. We say that class `data.frame` is derived from class `list`. Consequently, data frames inherit the methods and characteristics of lists, as long as they have not been hidden by new ones defined for data frames.

In the same way as with lists, we can add members to data frames.

```

a.df$x2 <- 6:1
a.df$x3 <- "b"
str(a.df)
## 'data.frame': 6 obs. of  5 variables:
##  $ x : int  1 2 3 4 5 6
##  $ y : chr  "a" "a" "a" "a" ...
##  $ z : logi  TRUE FALSE TRUE FALSE TRUE FALSE
##  $ x2: int   6 5 4 3 2 1
##  $ x3: chr   "b" "b" "b" "b" ...

```

We have added two columns to the data frame, and in the case of column `x3` recycling took place. This is where lists and data frames differ substantially in their behavior. In a data frame, although class and mode can be different for different variables (columns), they are required to be vectors or factors of the same length

(or a matrix with the same number of rows). In the case of lists, there is no such requirement, and recycling never takes place when adding a node. Compare the values returned below for `a.ls`, to those in the example above for `a.df`.

```
a.ls <- list(x = 1:6, y = "a", z = c(TRUE, FALSE))
str(a.ls)
## List of 3
## $ x: int [1:6] 1 2 3 4 5 6
## $ y: chr "a"
## $ z: logi [1:2] TRUE FALSE

a.ls$x2 <- 6:1
a.ls$x3 <- "b"
str(a.ls)
## List of 5
## $ x : int [1:6] 1 2 3 4 5 6
## $ y : chr "a"
## $ z : logi [1:2] TRUE FALSE
## $ x2: int [1:6] 6 5 4 3 2 1
## $ x3: chr "b"
```



Usually data frames are created from lists or by passing individual vectors and factors to the constructors. It is also possible to construct data frames starting from matrices, other data frames and named vectors and combinations of them. In these cases additional nuances become important. We give only some examples here, as the details are well described in `help(data.frame)`.

We use a named numeric vector, and a factor. The names are moved from the vector to the rows of the data frame! Consult `help(data.frame)` for an explanation.

```
my.vector <- c(one = 1, two = 2, three = 3, four = 4)
my.factor <- as.factor(c(1, 2, 3, 2))
df1 <- data.frame(my.factor, my.vector)
df1
##      my.factor my.vector
## one          1          1
## two          2          2
## three        3          3
## four          2          4

df1$my.vector
## [1] 1 2 3 4
```

If we protect the vector with R’s identity function `I()` the names are not removed from the vector as can be seen by extracting the column from the data frame.

```
df2 <- data.frame(my.factor, I(my.vector))
df2
##      my.factor my.vector
## one          1         1
## two          2         2
## three        3         3
## four         2         4

df2$my.vector
##   one  two three  four
##    1   2    3    4
```

If we start with a matrix instead of a vector, the matrix is split into separate columns in the data frame. If the matrix has no column names, new ones are created.

```
my.matrix <- matrix(1:12, ncol = 3)
df4 <- data.frame(my.factor, my.matrix)
df4
##   my.factor x1 x2 x3
## 1          1  1  5  9
## 2          2  2  6 10
## 3          3  3  7 11
## 4          2  4  8 12
```

If we protect the matrix with function `I()`, it is not split, and the whole matrix becomes a column in the data frame.

```
df5 <- data.frame(my.factor, I(my.matrix))
df5
##   my.factor my.matrix.1 my.matrix.2 my.matrix.3
## 1          1          1          5          9
## 2          2          2          6         10
## 3          3          3          7         11
## 4          2          4          8         12

df5$my.matrix
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

If we start with a list, each member with a suitable number of elements, each member becomes a column in the data frame. In the case of a too short one, recycling is applied.

```
my.list <- list(a = 4:1, b = letters[4:1], c = "n", d = "z")
df6 <- data.frame(my.factor, my.list)
df6
##   my.factor a b c d
## 1          1 4 d n z
## 2          2 3 c n z
## 3          3 2 b n z
## 4          2 1 a n z
```


If we protect the list, then the list is added in whole, similarly as in a tibble (see chapter ?? starting on page ?? for details about the ‘tidyverse’).

```
df7<- data.frame(my.factor, I(my.list))
df7
##   my.factor   my.list
## a         1 4, 3, 2, 1
## b         2 d, c, b, a
## c         3          n
## d         2          z

df7$my.list
## $a
## [1] 4 3 2 1
##
## $b
## [1] "d" "c" "b" "a"
##
## $c
## [1] "n"
##
## $d
## [1] "z"
```

What is this exercise about? Do check the documentation carefully and think of uses where the flexibility gained by use of function `I()` to protect arguments passed to the `data.frame()` constructor can be useful. In addition, write code to extract individual members of embedded matrices and lists using indexing in a single R statement in each case.

Data frames are extremely important to anyone analyzing or plotting data using R. One can think of data frames as tightly structured work-sheets, or as lists. As you may have guessed from the examples earlier in this section, there are several different ways of accessing columns, rows, and individual observations stored in a data frame. The columns can be treated as members in a list, and can be accessed both by name or index (position). When accessed by name, using `$` or double square brackets, a single column is returned as a vector or factor. In contrast to lists, data frames are always “rectangular” and for this reason the values stored can also be accessed in a way similar to how elements in a matrix are accessed, using two indexes. As we saw for vectors, indexes can be vectors of integer numbers or vectors of logical values. For columns they can, in addition, be vectors of character strings matching the names of the columns. When using indexes it is extremely important to remember that the indexes are always given **row first** in R.

 Indexing of data frames can in all cases be done as if they were lists, which is preferable, as it ensures compatibility with regular R lists and with newer implementations of data-frame-like structures like those defined in package ‘tibble’. Using this approach, extracting two values from the second and third positions in the first column of `a.df` is done as follows, using numerical indexes.

```
a.df[[1]][2:3]
## [1] 2 3
```

Or using the column name.

```
a.df[["x"]][2:3]
## [1] 2 3
```

Matrix-like indexing is done as follows, with the first index indicating rows and the second one indicating columns. This notation allows simultaneous extraction from multiple columns, which is not possible with lists. The value returned is a “smaller” data frame.

```
a.df[2:3, 1:2]
##   x y
## 2 2 a
## 3 3 a
```

If the length of the column indexing vector is one, by default the returned single column value is simplified into a vector or factor, which is not consistent with the previous example which returned a data frame. This is not only surprising in everyday use, but can be the source of bugs when coding algorithms in which the length of the second index vector cannot be guaranteed to be always more than one.

```
a.df[2:3, 1]
## [1] 2 3
```

Simplification can be prevented by overriding the default argument of parameter `drop`.

```
a.df[2:3, 1, drop = FALSE]
##   x
## 2 2
## 3 3
```

In contrast, the extraction operator `[]` of tibbles—defined in package ‘tibble’—never simplifies returned one-column tibbles into vectors (see section ?? on page ?? for details on the differences between data frames and tibbles).

```
# first column, a.df[[1]] preferred
a.df[, 1]
## [1] 1 2 3 4 5 6

# first column, a.df[["x"]] or a.df$x preferred
a.df[, "x"]
## [1] 1 2 3 4 5 6

# first row
a.df[1, ]
##   x y    z x2 x3
## 1 1 a TRUE 6  b

# first two rows of the third and fourth columns
a.df[1:2, c(FALSE, FALSE, TRUE, TRUE, FALSE)]
```



```
##           z x2
## 1  TRUE  6
## 2 FALSE  5

# the rows for which z is true
a.df[a.df$z , ]
##    x y    z x2 x3
## 1 1 a TRUE  6  b
## 3 3 a TRUE  4  b
## 5 5 a TRUE  2  b

# the rows for which x > 3 keeping all columns except the third one
a.df[a.df$x > 3, -3]
##    x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b
```

As explained earlier for vectors (see section 2.10 on page 48), indexing can be present both on the right-hand side and left-hand side of an assignment. The next few examples do assignments to “cells” of `a.df`, either to one whole column, or individual values. The last statement in the chunk below copies a number from one location to another by using indexing of the same data frame both on the right side and left side of the assignment.

```
a.df[1, 1] <- 99
a.df
##    x y    z x2 x3
## 1 99 a TRUE  6  b
## 2  2 a FALSE 5  b
## 3  3 a TRUE  4  b
## 4  4 a FALSE 3  b
## 5  5 a TRUE  2  b
## 6  6 a FALSE 1  b

a.df[, 1] <- -99
a.df
##    x y    z x2 x3
## 1 -99 a TRUE  6  b
## 2 -99 a FALSE 5  b
## 3 -99 a TRUE  4  b
## 4 -99 a FALSE 3  b
## 5 -99 a TRUE  2  b
## 6 -99 a FALSE 1  b

a.df[["x"]] <- 123
a.df
##    x y    z x2 x3
## 1 123 a TRUE  6  b
## 2 123 a FALSE 5  b
## 3 123 a TRUE  4  b
## 4 123 a FALSE 3  b
## 5 123 a TRUE  2  b
## 6 123 a FALSE 1  b

a.df[1, 1] <- a.df[6, 4]
a.df
```

```
##      x y      z x2 x3
## 1    1 a  TRUE  6  b
## 2  123 a FALSE  5  b
## 3  123 a  TRUE  4  b
## 4  123 a FALSE  3  b
## 5  123 a  TRUE  2  b
## 6  123 a FALSE  1  b
```



We mentioned above that indexing by name can be done either with double square brackets, `[[]]`, or with `$`. In the first case the name of the variable or column is given as a character string, enclosed in quotation marks, or as a variable with mode `character`. When using `$`, the name is entered as a constant, without quotation marks, and cannot be a variable.

```
x.list <- list(abcd = 123, xyzw = 789)
x.list[["abcd"]]
## [1] 123

a.var <- "abcd"
x.list[[a.var]]
## [1] 123
```

```
x.list$abcd
## [1] 123

x.list$ab
## [1] 123

x.list$a
## [1] 123
```

Both in the case of lists and data frames, when using double square brackets, by default an exact match is required between the name in the object and the name used for indexing. In contrast, with `$`, an unambiguous partial match is silently accepted. For interactive use, partial matching is helpful in reducing typing. However, in scripts, and especially R code in packages, it is best to avoid the use of `$` as partial matching to a wrong variable present at a later time, e.g., when someone else revises the script, can lead to very difficult-to-diagnose errors. In addition, as `$` is implemented by first attempting a match to the name and then calling `[[]]`, using `$` for indexing can result in slightly slower performance compared to using `[[]]`. It is possible to set an R option so that partial matching triggers a warning, which can be very useful when debugging.

2.14.1 Operating within data frames

When the names of data frames are long, complex conditions become awkward to write using indexing—i.e., subscripts. In such cases `subset()` is handy because evaluation is done in the “environment” of the data frame, i.e., the names of the columns are recognized if entered directly when writing the condition. Function `subset()` “filters” rows, usually corresponding to observations or experimental

units. The condition is computed for each row, and if it returns **TRUE**, the row is included in the returned data frame, and excluded if **FALSE**.

```
a.df <- data.frame(x = 1:6, y = "a", z = c(TRUE, FALSE))
subset(a.df, x > 3)
##   x y    z
## 4 4 a FALSE
## 5 5 a  TRUE
## 6 6 a FALSE
```



What is the behavior of `subset()` when the condition is **NA**? Find the answer by writing code to test this, for a case where tests for different rows return **NA**, **TRUE** and **FALSE**.

When calling functions that return a vector, data frame, or other structure, the extraction operators `[]`, `[[]]`, or `$` can be appended to the rightmost parenthesis of the function call, in the same way as to the name of a variable holding the same data.

```
subset(a.df, x > 3)[ , -3]
##   x y
## 4 4 a
## 5 5 a
## 6 6 a

subset(a.df, x > 3)[ , "x", drop = FALSE]
##   x
## 4 4
## 5 5
## 6 6

subset(a.df, x > 3)[ , "x"]
## [1] 4 5 6
```



When do extraction operators applied to data frames return a vector or factor, and when do they return a data frame?




In the case of `subset()` we can select columns directly as shown below, while for most other functions, extraction using operators `[]`, `[[]]` or `$` is needed.

```
subset(a.df, x > 3, select = 2)
##   y
## 4 a
## 5 a
## 6 a

subset(a.df, x > 3, select = x)
##   x
## 4 4
## 5 5
## 6 6

subset(a.df, x > 3, select = "x")
##   x
## 4 4
## 5 5
## 6 6
```

None of the examples in the last four code chunks alters the original data frame `a.df`. We can store the returned value using a new name if we want to preserve `a.df` unchanged, or we can assign the result to `a.df`, deleting in the process, the previously stored value.

 In the examples above, the names in the expression passed as the second argument to `subset()` were searched within `a.df` and found. However, if not found in the data frame objects with matching names are searched for in the environment. There being no variable `A` in the data frame `a.df`, vector `A` from the environment is silently used in the chunk below resulting in a returned data frame with no rows as `A > 3` returns `FALSE`.

```
A <- 1
subset(a.df, A > 3)
## [1] x y z
## <0 rows> (or 0-length row.names)
```

This also applies to the expression passed as argument to parameter `select`, here shown as a way of selecting columns based on names stored in a character vector.

```
columns <- c("x", "z")
subset(a.df, select = columns)
##   x      z
## 1 1  TRUE
## 2 2 FALSE
## 3 3  TRUE
## 4 4 FALSE
## 5 5  TRUE
## 6 6 FALSE
```


The use of `subset()` is convenient, but more prone to bugs compared to directly using the extraction operator `[]`. This same “cost” to achieving convenience applies to functions like `attach()` and `with()` described below. The longer time that a script is expected to be used, adapted and reused, the more careful we should

be when using any of these functions. An alternative way of avoiding excessive verbosity is to keep the names of data frames short.

A frequently used way of deleting a column by name from a data frame is to assign `NULL` to it—i.e., in the same way as members are deleted from `lists`. This approach modifies `a.df` in place.


```
aa.df <- a.df
colnames(aa.df)
## [1] "x" "y" "z"

aa.df[["y"]] <- NULL
colnames(aa.df)
## [1] "x" "z"
```


 Alternatively, we can use negative indexing to remove columns from a copy of a data frame. In this example we remove a single column. As base R does not support negative indexing by name with the extraction operator, we need to find the numerical index of the column to delete. (See the examples above using `subset()` with bare names to delete columns.)

```
a.df[, -which(colnames(a.df) == "y")]
##   x     z
## 1 1 TRUE
## 2 2 FALSE
## 3 3 TRUE
## 4 4 FALSE
## 5 5 TRUE
## 6 6 FALSE
```

Instead of using the equality test, we can use the operator `%in%` or function `grep()` to create a `logical` vector useful to delete or select multiple columns in a single statement.


 In the previous code chunk we deleted the last column of the data frame `a.df`. Here is an esoteric trick for you to first untangle how it changes the positions of columns and rows, and then for you to think how and why it can be useful to use indexing with the extraction operator `[]` on both sides of the assignment operator `<=`.

```
a.df[1:6, c(1,3)] <- a.df[6:1, c(3,1)]
a.df
```

 Although in this last example we used numeric indexes to make it more interesting, in practice, especially in scripts or other code that will be reused, do use column or member names instead of positional indexes whenever possible. This makes code much more reliable, as changes elsewhere in the script could alter the order of columns and *invalidate* numerical indexes. In addition, using meaningful names makes programmers’ intentions easier to understand.

2.14.2 Re-arranging columns and rows

The most direct way of changing the order of columns and/or rows in data frames (and matrices and arrays) is to use subscripting as described above. Once we know the original position and target position we can use numerical indexes on both right-hand side and left-hand side of an assignment.

 When using the extraction operator `[]` on both the left-hand-side and right-hand-side to swap columns, the vectors or factors are swapped, while the names of the columns are not! The same applies to row names, which makes storing important information in them inconvenient and error prone.


To retain the correspondence between column naming and column contents after swapping or rearranging the columns, we need to separately move the names of the columns. This seems counter intuitive, unless we think in terms of positions being named rather than the contents of the columns being linked to the names.

```
my_data_frame.df <- data.frame(A = 1:10, B = 3)
head(my_data_frame.df, 2)
##      A B
## 1 1 3
## 2 2 3

my_data_frame.df[, 1:2] <- my_data_frame.df[, 2:1]
head(my_data_frame.df, 2)
##      A B
## 1 3 1
## 2 3 2

colnames(my_data_frame.df)[1:2] <- colnames(my_data_frame.df)[2:1]
head(my_data_frame.df, 2)
##      B A
## 1 3 1
## 2 3 2
```

Taking into account that `order()` returns the indexes needed to sort a vector (see page 52), we can use `order()` to generate the indexes needed to sort rows of a data frame. In this case, the argument to `order()` is usually a column of the data frame being arranged. However, any vector of suitable length, including the result of applying a function to one or more columns, can be passed as an argument to `order()`. Function `order()` is very rarely useful for sorting columns of data frames as it requires a vector across columns as input. In the case of `matrix` and `array` this approach can be applied to any of their dimensions as all their elements homogeneously belong to one class.

 The first task to be completed is to sort a data frame based on the values in one column, using indexing and `order()`. Create a new data frame and with three numeric columns with three different haphazard sequences of values. Call these columns `A`, `B` and `C`. 1) Sort the rows of the data frame so that the values in `A` are in decreasing order. 2) Sort the rows of the data frame according to increasing values of the sum of `A` and `B` without adding a new column to the data frame or storing the vector of sums in a variable. In other words, do the sorting based on sums calculated on the fly.



Repeat the tasks in the playground immediately above but using factors instead of numeric vectors as columns in the data frame. Hint: revisit the exercise on page 64 where the use of `order()` on factors is described.

2.14.3 Re-encoding or adding variables

It is common that some variables need to be added to an existing data frame based on existing variables, either as a computed value or based on mapping for example treatments to sample codes already in a data frame. In the second case, named vectors can be used to replace values in a variable or to add a variable to a data frame.

Mapping is possible because the length of the value returned by the extraction operator `[]` is given by the length of the indexing vector (see section 2.10 on page 48). Although we show toy-like examples, this approach is most useful with data frames containing many rows.

If the existing variable is a character vector or factor, we need to create a named vector with the new values as data and the existing values as names.

```
my.df <-
  data.frame(genotype = rep(c("WT", "mutant1", "mutant2"), 2),
             value = c(1.5, 3.2, 4.5, 8.2, 7.4, 6.2))
mutant <- c(WT = FALSE, mutant1 = TRUE, mutant2 = TRUE)
my.df$mutant <- mutant[my.df$genotype]
my.df
##   genotype value mutant
## 1      WT    1.5  FALSE
## 2  mutant1    3.2   TRUE
## 3  mutant2    4.5   TRUE
## 4      WT    8.2  FALSE
## 5  mutant1    7.4   TRUE
## 6  mutant2    6.2   TRUE
```

If the existing variable is an `integer` vector, we can use a vector without names, being careful that the positions in the *mapping* vector match the values of the existing variable

```
my.df <- data.frame(individual = rep(1:3, 2),
                   value = c(1.5, 3.2, 4.5, 8.2, 7.4, 6.2))
genotype <- c("WT", "mutant1", "mutant2")
my.df$genotype <- genotype[my.df$individual]
my.df
##   individual value genotype
## 1          1    1.5       WT
## 2          2    3.2  mutant1
## 3          3    4.5  mutant2
## 4          1    8.2       WT
## 5          2    7.4  mutant1
## 6          3    6.2  mutant2
```




Add a variable named `genotype` to the data frame below so that for individual 4 its value is "WT", for individual 1 its value is "mutant1", and for individual 2 its value is "mutant2".

```
my.df <- data.frame(individual = rep(c(2, 4, 1), 2),  
                    value = c(1.5, 3.2, 4.5, 8.2, 7.4, 6.2))
```



```
##      B A      E D      C
## 1 3 1 1.333333 3 4.0
## 2 3 2 1.666667 6 2.5
```

 Repeatedly pre-pending the name of a *container* such as a list or data frame to the name of each member variable being accessed can make R code verbose and difficult to understand. Functions `attach()` and its matching `detach()` allow us to change where R looks for the names of objects we include in a code statement. When using a long name for a data frame, entering a simple calculation can easily result in a difficult to read statement. (Method `head()` is used here to limit the displayed value to the first two rows—`head()` is described in section 2.17 on page 90.)

```
my_data_frame.df <- data.frame(A = 1:10, B = 3)
my_data_frame.df$C <-
  (my_data_frame.df$A + my_data_frame.df$B) / my_data_frame.df$A
head(my_data_frame.df, 2)
##      A B      C
## 1 1 3 4.0
## 2 2 3 2.5
```

Using `attach()` we can alter how R looks up names and consequently simplify the statement. With `detach()` we can restore the original state. It is important to remember that here we can only simplify the right-hand side of the assignment, while the “destination” of the result of the computation still needs to be fully specified on the left-hand side of the assignment operator. We include below only one statement between `attach()` and `detach()` but multiple statements are allowed. Furthermore, if variables with the same name as the columns exist in the search path, these will take precedence, something that can result in bugs or crashes, or as seen below, a message warns that variable `A` from the global environment will be used instead of column `A` of the attached `my_data_frame.df`. The returned value is, of course, not the desired one.

```
my_data_frame.df$C <- NULL
attach(my_data_frame.df)

## The following object is masked _by_ .GlobalEnv:
##
##      A

my_data_frame.df$C <- (A + B) / A
detach(my_data_frame.df)
head(my_data_frame.df, 2)
##      A B C
## 1 1 3 4
## 2 2 3 4
```

Use of `attach()` and `detach()`, which function as a pair of ON and OFF switches, can result in an undesired after-effect on name lookup if the script terminates after `attach()` is executed but before `detach()` is called, as cleanup is not automatic. In contrast, `with()` and `within()`, being self-contained, guarantee that cleanup takes place. Consequently, the usual recommendation is to give preference to the use of

`with()` and `within()` over `attach()` and `detach()`. Use of these functions not only saves typing but also makes code more readable.

2.15 Attributes of R objects

R objects can have attributes. Attributes are named slots normally used to store ancillary data such as object properties. There are no restrictions on the class of what is assigned to an attribute. They are used by R itself to store things like column names in data frames and labels of factor levels. All these attributes are visible to user code, and user code can read and write objects' attributes. However, they are rarely displayed explicitly when an object is printed. They can be also used to store metadata accompanying the data stored in an object, which is important for reproducible research and data sharing.

Attribute `"comment"` is meant to be set by users to store a character string—e.g., to store metadata as text together with data. As comments are frequently used, R has functions for accessing and setting comments.

```
a.df <- data.frame(x = 1:6, y = "a", z = c(TRUE, FALSE))
comment(a.df)
## NULL

comment(a.df) <- "this is stored as a comment"
comment(a.df)
## [1] "this is stored as a comment"
```

Methods like `names()`, `dim()` or `levels()` return values retrieved from attributes stored in R objects, and methods like `names()<-`, `dim()<-` or `levels()<-` set (or un-set with `NULL`) the value of the respective attributes. Specific query and set methods do not exist for all attributes. Methods `attr()`, `attr()<-` and `attributes()` can be used with any attribute. With `attr()` we access, and with `attr()<-` we set individual attributes by name. With `attributes()` we retrieve all attributes of an object as a named list. In addition, method `str()` displays all components and structure of R objects including their attributes.

Continuing with the previous example, we can retrieve and set the comment using these functions. In the second statement we delete the value stored in the `"comment"` attribute by assigning `NULL` to it.

```
attr(a.df, "comment")
## [1] "this is stored as a comment"

attr(a.df, "comment") <- NULL
attr(a.df, "comment")
## NULL

comment(a.df) # same as previous line
## NULL
```

The `"names"` attribute of `a.df` was set by the `data.frame()` constructor when

it was created above. In the next example, in the first statement we retrieve the names, and implicitly print them. In the second statement, read from right to left, we retrieve the names, convert them to upper case and save them back to the same attribute.

```
names(a.df)
## [1] "x" "y" "z"

names(a.df) <- toupper(names(a.df))
names(a.df)
## [1] "X" "Y" "Z"

attr(a.df, "names") # same as previous line
## [1] "X" "Y" "Z"
```

We can add a new attribute, under our own control, as long as its name does not clash with that of existing attributes.

```
attr(a.df, "my.attribute") <- "this is stored in my attribute"
attributes(a.df)
## $names
## [1] "x" "y" "z"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5 6
##
## $my.attribute
## [1] "this is stored in my attribute"
```



There is no restriction to the creation, setting, resetting and reading of attributes, but not all methods and operators that can be used to modify objects will preserve non-standard attributes. This can be a problem when using some R packages, such as some popular packages from the ‘tidyverse’. So, using private attributes is a double-edged sword that usually is worthwhile considering only when designing a new class together with the corresponding methods for it. A good example of extensive use of class-specific attributes are the values returned by model fitting functions like `lm()` (see section ?? on page ??).

Even the class of S3 objects is stored as an attribute that is accessible as any other attribute—this is in contrast to the mode and atomic class of an object. Object-oriented programming in R is explained in section ?? on page ??.

```
numbers <- 1:10
mode(numbers)
## [1] "numeric"

class(numbers)
## [1] "integer"

attributes(numbers)
## NULL
```

```
a.factor <- factor(numbers)
mode(a.factor)
## [1] "numeric"

class(a.factor)
## [1] "factor"

attributes(a.factor)
## $levels
## [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
##
## $class
## [1] "factor"
```

2.16 Saving and loading data

2.16.1 Data sets in R and packages

To be able to present more meaningful examples, we need some real data. Here we use `cars`, one of the many data sets included in base R. Function `data()` is used to load data objects that are included in R or contained in packages. It is also possible to import data saved in files with *foreign* formats, defined by other software or commonly used for data exchange. Package ‘foreign’, included in the R distribution, as well as contributed packages make available functions capable of reading and decoding various foreign formats. How to read or import “foreign” data is discussed in R documentation in *R Data Import/Export*, and in this book, in chapter ?? starting on page ?. It is also good to keep in mind that in R, URLs (Uniform Resource Locators) are accepted as arguments to the `file` or `path` parameter of many functions (see section ?? starting on page ?).

In the next example we load data included in R as R objects by calling function `data()`. The loaded R object `cars` is a data frame.

```
data(cars)
```

Once we have a data set available, the first step is usually to explore it, and we will do this with `cars` in section 2.17 on page 90.

2.16.2 .rda files

By default, at the end of a session, the current workspace containing the results of your work is saved into a file called `.RData`. In addition to saving the whole workspace, it is possible to save one or more R objects present in the workspace to disk using the same file format (with file name tag `.rda` or `.Rda`). One or more objects, belonging to any mode or class can be saved into a single file using function `save()`. Reading the file restores all the saved objects into the current workspace with their original names. These files are portable across most R versions—i.e., old formats can be read and written by newer versions of R, although the newer,

default format may be not readable with earlier R versions. Whether compression is used, and whether the “binary” data is encoded into ASCII characters, allowing maximum portability at the expense of increased size can be controlled by passing suitable arguments to `save()`.

We create a data frame object and then save it to a file.

```
my.df <- data.frame(x = 1:5, y = 5:1)
my.df
##      x y
## 1 1 5
## 2 2 4
## 3 3 3
## 4 4 2
## 5 5 1

save(my.df, file = "my-df.rda")
```

We delete the data frame object and confirm that it is no longer present in the workspace.

```
rm(my.df)
ls(pattern = "my.df")
## character(0)
```

We read the file we earlier saved to restore the object.

```
load(file = "my-df.rda")
ls(pattern = "my.df")
## [1] "my.df"

my.df
##      x y
## 1 1 5
## 2 2 4
## 3 3 3
## 4 4 2
## 5 5 1
```

The default format used is binary and compressed, which results in smaller files.



In the example above, only one object was saved, but one can simply give the names of additional objects as arguments. Just try saving more than one data frame to the same file. Then the data frames plus a few vectors. After creating each file, clear the workspace and then restore from the file the objects you saved.

Sometimes it is easier to supply the names of the objects to be saved as a vector of character strings passed as an argument to parameter `list`. One case is when wanting to save a group of objects based on their names. We can use `ls()` to list the names of objects matching a simple `pattern` or a complex regular expression. The example below does this in two steps, first saving a character vector with the names of the objects matching a pattern, and then using this saved vector as an argument to `save`'s `list` parameter.

```
objects <- ls(pattern = "*.df")
save(list = objects, file = "my-df1.rda")
```

The two statements above can be combined into a single statement by nesting the function calls.

```
save(list = ls(pattern = "*.df"), file = "my-df1.rda")
```



Practice using different patterns with `ls()`. You do not need to save the objects to a file. Just have a look at the list of object names returned.

As a coda, we show how to clean up by deleting the two files we created. Function `unlink()` can be used to delete any files for which the user has enough rights.

```
unlink(c("my-df.rda", "my-df1.rda"))
```

2.16.3 .rds files

The RDS format can be used to save individual objects instead of multiple objects (usually using file name tag `.rds`). They are read and saved with functions `readRDS()` and `saveRDS()`, respectively. When RDS files are read, different from when RDA files are loaded, we need to assign the object read to a possibly different name for it to be added to the search path. Of course, it is also possible to use the returned object as an argument to a function or in an expression without saving it to a variable.

```
saveRDS(my.df, "my-df.rds")
```

If we read the file, by default the read R object will be printed at the console.

```
readRDS("my-df.rds")
##      x y
## 1 1 5
## 2 2 4
## 3 3 3
## 4 4 2
## 5 5 1
```

In the next example we assign the read object to a different name, and check that the object read is identical to the one saved.

```
my_read.df <- readRDS("my-df.rds")
identical(my.df, my_read.df)
## [1] TRUE
```

As above, we clean up by deleting the file.

```
unlink("my-df.rds")
```

2.17 Looking at data

There are several functions in R that let us obtain different views into objects. Function `print()` is useful for small data sets, or objects. Especially in the case of large data frames, we need to explore them step by step. In the case of named components, we can obtain their names with `colnames()`, `rownames()`, and `names()`. If a data frame contains many rows of observations, `head()` and `tail()` allow us to easily restrict the number of rows printed. Functions `nrow()` and `ncol()` return the number of rows and columns in the data frame (also applicable to matrices but not to lists or vectors where we use `length()`). As mentioned earlier, function `str()` concisely displays the structure of R objects.

```
class(cars)
## [1] "data.frame"

head(cars)
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10

tail(cars)
##   speed dist
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85

nrow(cars)
## [1] 50


ncol(cars)
## [1] 2

names(cars)
## [1] "speed" "dist"

colnames(cars)
## [1] "speed" "dist"

head(rownames(cars))
## [1] "1" "2" "3" "4" "5" "6"


str(cars)
## 'data.frame': 50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```


 Look up the help pages for `head()` and `tail()`, and edit the code above to print only the first two lines, or only the last three lines of `cars`, respectively.

The different columns of a data frame can be factors, or vectors of various modes (e.g., numeric, logical, character, etc.) (see section 2.14 on page 70). To explore the mode of the columns of `cars`, we can use an *apply* function. In the present case, we want to apply function `class()` to each column of the data frame `cars`. (Apply functions are described in section ?? on page ??.)

```
sapply(X = cars, FUN = class)
##      speed      dist
## "numeric" "numeric"
```

The statement above returns a vector of character strings, with the mode of each column. Each element of the vector is named according to the name of the corresponding “column” in the data frame. For this same statement to be used with any other data frame or list, we need only to substitute the name of the object, the argument to the first parameter called `x`, to the one of current interest.


 Data set `airquality` contains data from air quality measurements in New York, and, being included in the R distribution, can be loaded with `data(airquality)`. Load it, and repeat the steps above, to learn what variables (columns) it contains, their classes, the number of rows, etc.

 Although the R language allows data frame columns of class `matrix`, their use is infrequent. On the other hand, columns belonging to class `list` are disallowed in data frames. The reverse is true for tibbles (described in section ?? on page ??).

Function `summary()` can be used to obtain a summary from objects of most R classes, including data frames. We can also use `sapply()`, `lapply()` or `vapply()` to apply any suitable function to individual columns.

```
summary(cars)
##      speed      dist
## Min.   : 4.0    Min.   : 2.00
## 1st Qu.:12.0    1st Qu.: 26.00
## Median :15.0    Median : 36.00
## Mean   :15.4    Mean   : 42.98
## 3rd Qu.:19.0    3rd Qu.: 56.00
## Max.   :25.0    Max.   :120.00

sapply(cars, range)
##      speed dist
## [1,]     4    2
## [2,]    25  120
```

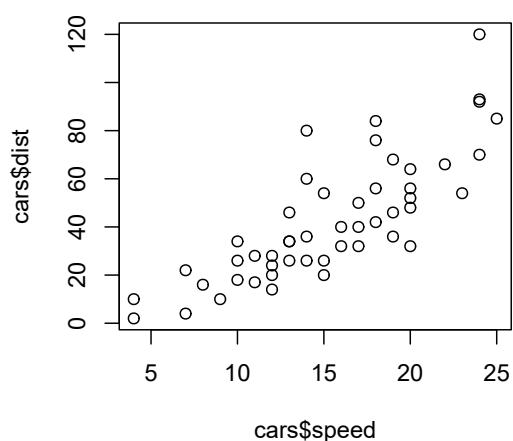
 Obtain the summary of `airquality` with function `summary()`, but in addition, write code with an *apply* function to count the number of non-missing values in each column. Hint: using `sum()` on a `logical` vector returns the count of `TRUE` values as `TRUE`, and `FALSE` are transparently converted into numeric 1 and 0, respectively, when `logical` values are used in arithmetic expressions.

2.18 Plotting

The base-R generic method `plot()` can be used to plot different data. It is a generic method that has specializations suitable for different kinds of objects (see section ?? on page ?? for a brief introduction to objects, classes and methods). In this section we only very briefly demonstrate the use of the most common base-R graphics functions. They are well described in the book *R Graphics* (Murrell 2019). We will not describe the Lattice (based on S's Trellis) approach to plotting (Sarkar 2008). Instead we describe in detail the use of the *grammar of graphics* and plotting with package ‘ggplot2’ in chapter ?? starting on page ??.

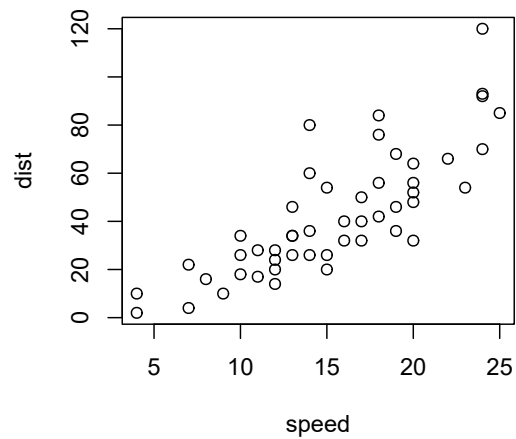
It is possible to pass two variables (here columns from a data frame) directly as arguments to the `x` and `y` parameters of `plot()`.

```
plot(x = cars$speed, y = cars$dist)
```



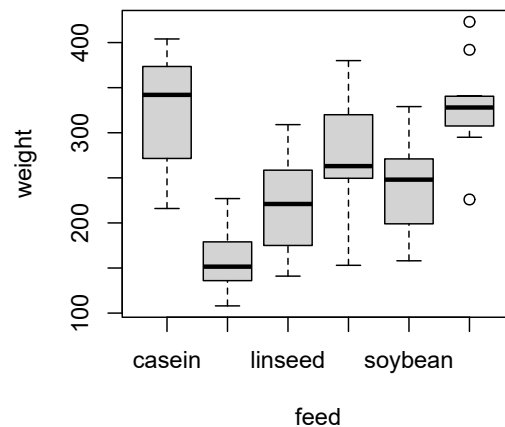
It is also possible, and usually more convenient, to use a *formula* to specify the variables to be plotted on the x and y axes, passing additionally as an argument to parameter `data` the name of the data frame containing these variables. The formula `dist ~ speed`, is read as `dist` explained by `speed`—i.e., `dist` is mapped to the y -axis as the dependent variable and `speed` to the x -axis as the independent variable.

```
plot(dist ~ speed, data = cars)
```



Within R there exist different specializations, or “flavors,” of method `plot()` that become active depending on the class of the variables passed as arguments: passing two numerical variables results in a scatter plot as seen above. In contrast passing one factor and one numeric variable to `plot()` results in a box-and-whiskers plot being produced. To exemplify this we need to use a different data set, here `chickwts` as `cars` does not contain any factors. Use `help("chickwts")` to learn more about this data set, also included in R.

```
plot(weight ~ feed, data = chickwts)
```



Method `plot()` and variants defined in R, when used for plotting return their graphical output to a *graphical output device*.

When R is used interactively, a software device is opened automatically to output the graphical output to a physical device, usually the computer screen. The

name of the R software device used may depend on the operating system (e.g., MS-Windows or Linux), or on the IDE (e.g., RStudio).

In R, software graphical devices not necessarily generate output on a physical device like a printer, as several of these devices translate the plotting commands into a file format and save it to disk. Several different graphical devices are available in R and they differ in the kind of output they produce: raster files (e.g., TIFF, PNG and JPEG formats), vector graphics files (e.g., SVG, EPS and PDF), or output to a physical device like the screen of a computer. Additional devices are available through contributed R packages.

Devices follow the paradigm of ON and OFF switches, opening and closing a destination for `print()`, `plot()` and related functions. Some devices producing a file as output, can save their output one plot at a time to single-page graphic files or only when the device is closed, possibly as a multi-page file.

When opening a device the user supplies additional information. For the PDF device that produces output in a vector-graphics format, width and height of the output are specified in *inches*. A default file name is used unless we pass a `character` string as an argument to parameter `file`.

```
pdf(file = "output/my-file.pdf", width = 6, height = 5, onefile = TRUE)
plot(dist ~ speed, data = cars)
plot(weight ~ feed, data = chickwts)
dev.off()
## cairo_pdf
##      2
```

Raster devices return bitmaps and `width` and `height` are specified in *pixels*.

```
png(file = "output/my-file.png", width = 600, height = 500)
plot(weight ~ feed, data = chickwts)
dev.off()
## cairo_pdf
##      2
```

The approach of direct output to a software device is used in base R, and the addition of plot components, as shown below, is done directly to the output device.

```
png(file = "output/my-file.png", width = 600, height = 500)
plot(dist ~ speed, data = cars)
text(x = 10, y = 110, labels = "some texts to be added")
dev.off()
## cairo_pdf
##      2
```

i This is not the only approach available. As we will see in chapter ?? starting on page ??, an alternative approach is to build a *plot object* as a list of member components that is later rendered as a whole on a graphical device by calling `print()` once.

2.19 Further reading

For further reading on the aspects of R discussed in the current chapter, I suggest the books *R Programming for Data Science* (Peng) and *The Art of R Programming: A Tour of Statistical Software Design* (Matloff).



Bibliography

- Adams, J. L. (1987). *Conceptual Blockbusting: A Guide to Better Ideas*. Penguin Books Ltd. ISBN: 9780140098426.
- Aiken, H., A. G. Oettinger, and T. C. Bartee (Aug. 1964). "Proposed automatic calculating machine". In: *IEEE Spectrum* 1.8, pp. 62-69. DOI: 10.1109/mspec.1964.6500770.
- Becker, R. A. and J. M. Chambers (1984). *S: An Interactive Environment for Data Analysis and Graphics*. Chapman and Hall/CRC. ISBN: 0-534-03313-X (cit. on p. 4).
- Becker, R. A., J. M. Chambers, and A. R. Wilks (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics*. Chapman & Hall. ISBN: 0-534-09192-X (cit. on pp. 3, 4).
- Boas, R. P. (1981). "Can we make mathematics intelligible?" In: *The American Mathematical Monthly* 88.10, pp. 727-731.
- Chambers, J. M. (2016). *Extending R*. The R Series. Chapman and Hall/CRC. ISBN: 1498775713 (cit. on p. 4).
- Coplien, J. O. (1999). *Multi-paradigm design for C++*. Addison-Wesley, p. 280. ISBN: 0201824671 (cit. on p. 2).
- Gandrud, C. (2015). *Reproducible Research with R and R Studio*. 2nd ed. Chapman & Hall/CRC The R Series). Chapman and Hall/CRC. 323 pp. ISBN: 1498715370 (cit. on pp. 11, 12).
- Hillebrand, J. and M. H. Nierhoff (2015). *Mastering RStudio: Develop, Communicate, and Collaborate with R*. Packt Publishing. 348 pp. ISBN: 9781783982554 (cit. on p. 10).
- Kernighan, B. W. and R. Pike (1999). *The Practice of Programming*. Addison Wesley. 288 pp. ISBN: 020161586X (cit. on p. 17).
- Knuth, D. E. (1984). "Literate programming". In: *The Computer Journal* 27.2, pp. 97-111 (cit. on p. 11).
- Leisch, F. (2002). "Dynamic generation of statistical reports using literate data analysis". In: *Proceedings in Computational Statistics*. Compstat 2002. Ed. by W. Härdle and B. Rönz. Heidelberg, Germany: Physika Verlag, pp. 575-580. ISBN: 3-7908-1517-9 (cit. on p. 11).
- Loo, M. P. van der and E. de Jonge (2012). *Learning RStudio for R Statistical Computing*. 1st ed. Birmingham: Packt Publishing, p. 126. ISBN: 9781782160601 (cit. on p. 10).
- Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843 (cit. on p. 95).
- Murrell, P. (2019). *R Graphics*. 3rd ed. Portland: Chapman and Hall/CRC. 423 pp. ISBN: 1498789056 (cit. on p. 92).

- Newham, C. and B. Rosenblatt (June 1, 2005). *Learning the bash Shell*. O'Reilly UK Ltd. 352 pp. ISBN: 0596009658 (cit. on p. 17).
- Peng, R. D. (2016). *R Programming for Data Science*. Leanpub. 182 pp. URL: <https://leanpub.com/rprogramming> (visited on 07/31/2019) (cit. on p. 95).
- Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. 1st ed. Springer, p. 268. ISBN: 0387759689 (cit. on p. 92).
- Wirth, N. (1974). "On the Composition of Well-Structured Programs". In: *Computing Surveys* 6.4, pp. 247-259. DOI: 10.1145/356635.356639 (cit. on p. 2).
- Xie, Y. (2013). *Dynamic Documents with R and knitr*. The R Series. Chapman and Hall/CRC, p. 216. ISBN: 1482203537 (cit. on pp. 11, 12).
- Zimmer, J. A. (1985). *Abstraction for Programmers*. New York: McGraw-Hill, p. 251. ISBN: 0070728321 (cit. on p. 2).

General index

- algebra of sets, 38
- arithmetic overflow, 37
 - type promotion, 37
- arrays, 54–59
 - dimensions, 57
- assignment, 22
 - chaining, 23
 - leftwise, 23
- attributes, 85–87
- base R, 3
- bash, 17
- batch job, 7
- Bio7, 16
- Boolean arithmetic, 31
- C, 17, 24, 42, 46–48, 66
- C++, 2, 4, 17, 42, 48
- categorical variables, *see* factors
- character escape codes, 43
- character string delimiters, 43
- character strings, 42
- classes and modes
 - character, 42–43
 - logical, 31–38
 - numeric, integer, double, 20–31
- comparison of floating point numbers, 37–38
- comparison operators, 33–38
- console, 6
- data
 - exploration at the R console, 90–95
 - loading data sets, 87
- data frames, 70–85
 - “filtering rows”, 77
 - attaching, 84
 - operating within, 77
 - ordering columns, 81
 - ordering rows, 81, 82
 - subsetting, 77
- deleting objects, *see* removing objects
- Eclipse, 16
- editor for R scripts, 16
- Emacs, 5, 16
- EPS (ε), *see* machine arithmetic precision
- factors, 59–65
 - arrange values, 64
 - convert to numeric, 63
 - drop unused levels, 62
 - labels, 61
 - levels, 61
 - merge levels, 62
 - ordered, 60
 - reorder levels, 63
 - reorder values, 64
- floating point numbers
 - arithmetic, 35–37
- floats, *see* floating point numbers
- ‘foreign’, 87
- formatted character strings from numbers, 46
- further reading
 - shell scripts in Unix and Linux, 17
 - using the R language, 95
- ‘ggplot2’, 92
- Git, 16
- git, 6
- Gnu S, 3
- IDE, *see* integrated development environment

- IDE for R, 16
- ImageJ, 16
- inequality and equality tests, 37–38
- integrated development environment, 8
- Java, 17
- ‘knitr’, 11, 12
- languages
 - C, 17, 24, 42, 46–48, 66
 - C++, 2, 4, 17, 42, 48
 - Java, 17
 - LaTeX, 12
 - natural and computer, 20
 - Pascal, 4
 - Python, 8
 - S, 3, 4
 - S-Plus, 3
- LaTeX, 16
- LaTeX, 12
- ‘learnrbook’, 16
- Linux, 4, 8, 10, 16, 17, 94
- lists, 65–70
 - append to, 67
 - convert into vector, 70
 - flattening, 69
 - insert into, 67
 - member extraction, 65–66
 - nested, 67–69
 - structure, 69
- logical operators, 31
- logical values and their algebra, 31–33
- loss of numeric precision, 37
- machine arithmetic
 - precision, 35–37
 - rounding errors, 35
- math functions, 20
- math operators, 20
- matrices, 54–59
- matrix
 - dimensions, 57
- ‘matrixStats’, 59
- Microsoft R Open, 3
- MS-Windows, 4, 8–10, 16, 94
- named vectors
 - mapping with, 82
- netiquette, 14
- network etiquette, 14
- numbers
 - double, 29
 - floating point, 28
 - integer, 28, 29
 - whole, 28
- numbers and their arithmetic, 20–31
- numeric values, 20
- numeric, integer and double values, 23
- objects
 - mode, 43
- operating systems
 - Linux, 4, 8, 10, 16, 94
 - MS-Windows, 4, 8–10, 16, 94
 - OS X, 4, 8, 10, 16
 - Unix, 4, 8, 10, 16
- operators
 - comparison, 33–38
 - set, 38–42
- OS X, 4, 8, 10, 16
- overflow, *see* arithmetic overflow
- packages
 - ‘foreign’, 87
 - ‘ggplot2’, 92
 - ‘knitr’, 11, 12
 - ‘learnrbook’, 16
 - ‘matrixStats’, 59
 - ‘Sweave’, 11
 - ‘tibble’, 74, 75
 - ‘tidyverse’, 74
 - ‘tidyverse’, 86
- Pascal, 4
- plots
 - base R graphics, 92
- precision
 - math operations, 28
- programmes
 - bash, 17
 - Bio7, 16
 - Eclipse, 16
 - Emacs, 5, 16
 - Git, 16

- git, 6
- Gnu S, 3
- ImageJ, 16
- Linux, 17
- Microsoft R Open, 3
- RGUI, 13
- RStudio, 5, 6, 8–11, 13, 16, 27, 94
- SAS, 3
- SPSS, 3
- Unix, 17
- WinEdt, 16
- Python, 8
- R
 - help, 13
- R as a language, 2, 4
- R as a program, 2
- R as a program, 4
- Raspberry Pi, 3
- Real numbers and computers, 35
- recycling of arguments, 25
- removing objects, 27
- reprex, *see* reproducible example
- reproducible data analysis, 11–12
- reproducible example, 15
- RGUI, 13
- RStudio, 5, 6, 8–11, 13, 16, 27, 94
- S, 3, 4
- S-Plus, 3
- SAS, 3
- script, 7
- sequence, 25
- sets, 38–42
- special values
 - NA, 27
 - NaN, 27
- SPSS, 3
- StackOverflow, 15
- ‘Sweave’, 11
- ‘tibble’, 74, 75
- ‘tidyverse’, 74
- ‘tidyverse’, 86
- type conversion, 44–48
- type promotion, 37
- Unix, 4, 8, 10, 16, 17
- variables, 22
- vector
 - run length encoding, 53
- vectorized arithmetic, 25
- vectors
 - indexing, 48–53
 - member extraction, 48
 - sorting, 53
 - zero length, 28
- WinEdt, 16
- ‘worksheet’, *see* data frame
- zero length objects, 28



Index of R names by category

classes and modes
 array, 54, 81
 character, 41, 42, 46
 data.frame, 70, 71
 double, 23, 36, 37, 52
 factor, 59
 integer, 23, 28, 36, 37, 52
 list, 65, 70, 91
 logical, 31, 47
 matrix, 54, 57, 81, 91
 numeric, 20, 23, 41, 42, 52
 vector, 24
constant and special values
 `-Inf`, 27, 36
 `.Machine$double.eps`, 36
 `.Machine$double.max`, 36
 `.Machine$double.min`, 36
 `.Machine$double.neg.eps`, 36
 `.Machine$integer.max`, 36
 `FALSE`, 29
 `Inf`, 27, 36
 `LETTERS`, 48
 `letters`, 48
 `month.abb`, 48
 `month.name`, 48
 `NA`, 27, 28, 47
 `NA_character_`, 47
 `NA_real_`, 47
 `NaN`, 27
 `pi`, 21
 `TRUE`, 29
control of execution
 `lapply()`, 91
 `sapply`, 91
 `sapply()`, 91
 `vapply()`, 91
functions and methods
 `abs()`, 31, 37
 `all()`, 32
 `any()`, 32
 `append()`, 25, 67
 `array()`, 57
 `as.character()`, 44, 45, 63
 `as.integer()`, 45
 `as.logical()`, 44, 45
 `as.matrix()`, 54
 `as.numeric()`, 44, 45, 63
 `as.vector()`, 58
 `attach()`, 79
 `attr()`, 85
 `attr()<-`, 85
 `attributes()`, 85
 `c()`, 24, 65
 `cat()`, 43
 `ceiling()`, 30, 31
 `citation()`, 13
 `class()`, 44, 71
 `colnames()`, 90
 `comment()`, 85
 `comment()<-`, 85
 `crossprod()`, 59
 `data()`, 87
 `data.frame()`, 70, 74
 `diag()`, 59
 `dim()`, 54, 85
 `dim()<-`, 85
 `double()`, 23
 `exp()`, 21
 `factor()`, 60–63
 `format()`, 46, 47
 `gl()`, 60
 `head()`, 90, 91
 `help()`, 13
 `I()`, 72–74
 `inherits()`, 44
 `is.array()`, 54

- is.character(), 43
- is.element(), 39, 40
- is.logical(), 43
- is.matrix(), 54
- is.na(), 28
- is.numeric(), 23, 43
- is.vector(), 54
- length(), 26, 45, 54, 90
- levels(), 62, 63, 85
- levels()<-, 85
- list(), 65
- lm(), 86
- load(), 88
- log(), log10(), log2(), 21
- ls(), 27, 88, 89
- matrix(), 54
- mode(), 43
- names(), 85, 90
- names()<-, 85
- ncol(), 54, 90
- nrow(), 54, 90
- numeric(), 23, 26
- order(), 53, 65, 81, 82
- ordered(), 60
- plot(), 92, 93
- print(), 7, 43, 46, 68, 90
- readRDS(), 89
- reorder(), 64
- rep(), 25
- rev(), 63
- rle(), 53
- rm(), 27
- round(), 30
- rownames(), 90
- save(), 87, 88
- saveRDS(), 89
- seq(), 25
- signif(), 30
- sin(), 21
- sort(), 53, 65
- sprintf(), 46, 47
- sqrt(), 21
- str(), 66, 68, 69, 85, 90
- subset(), 77-79
- summary(), 91

- t(), 58
- tail(), 90, 91
- trunc(), 30, 31, 45
- typeof(), 43
- unique(), 40
- unlink(), 89
- unlist(), 58, 69, 70
- unname(), 70
- with(), 79

names and their scope

- attach(), 84, 85
- detach(), 84, 85
- transform(), 83
- with(), 83-85
- within(), 83-85

operators

- *, 20, 37
- +, 20, 31
- , 20, 31
- >, 23
- /, 20
- ., 25
- <, 33
- <-, 22, 23, 51, 80
- <=, 33
- =, 23
- ==, 33
- >, 33
- >=, 33
- [], 68, 75, 78, 80, 82
- [[]], 68, 71, 77, 78
- [[]], 65, 76, 77
- [], 76, 81
- \$, 74, 77, 78
- %*%, 59
- %/%, 29
- %%, 29
- %in%, 39, 40, 42
- &, 31
- &&, 31
- ^, 37
- |, 31
- ||, 31

Alphabetic index of R names

`*`, 20, 37
`+`, 20, 31
`-`, 20, 31
`->`, 23
`-Inf`, 27, 36
`.Machine$double.eps`, 36
`.Machine$double.max`, 36
`.Machine$double.min`, 36
`.Machine$double.neg.eps`, 36
`.Machine$integer.max`, 36
`/`, 20
`:`, 25
`<`, 33
`<-`, 22, 23, 51, 80
`<=`, 33
`=`, 23
`==`, 33
`>`, 33
`>=`, 33
`[]`, 68, 75, 78, 80, 82
`[[]]`, 68, 71, 77, 78
`[[]]`, 65, 76, 77
`[]`, 76, 81
`$`, 74, 77, 78
`%*%`, 59
`%/%`, 29
`%%`, 29
`%in%`, 39, 40, 42
`&`, 31
`&&`, 31
`^`, 37
`|`, 31
`||`, 31

`abs()`, 31, 37
`all()`, 32
`any()`, 32
`append()`, 25, 67
`array`, 54, 81
`array()`, 57
`as.character()`, 44, 45, 63
`as.integer()`, 45
`as.logical()`, 44, 45
`as.matrix()`, 54
`as.numeric()`, 44, 45, 63
`as.vector()`, 58
`attach()`, 79, 84, 85
`attr()`, 85
`attr()<-`, 85
`attributes()`, 85

`c()`, 24, 65
`cat()`, 43
`ceiling()`, 30, 31
`character`, 41, 42, 46
`citation()`, 13
`class()`, 44, 71
`colnames()`, 90
`comment()`, 85
`comment()<-`, 85
`crossprod()`, 59

`data()`, 87
`data.frame`, 70, 71
`data.frame()`, 70, 74
`detach()`, 84, 85
`diag()`, 59
`dim()`, 54, 85
`dim()<-`, 85
`double`, 23, 36, 37, 52
`double()`, 23

`exp()`, 21

`factor`, 59
`factor()`, 60–63
`FALSE`, 29
`format()`, 46, 47

`gl()`, 60
`head()`, 90, 91
`help()`, 13
`I()`, 72–74
`Inf`, 27, 36
`inherits()`, 44
`integer`, 23, 28, 36, 37, 52
`is.array()`, 54
`is.character()`, 43
`is.element()`, 39, 40
`is.logical()`, 43
`is.matrix()`, 54
`is.na()`, 28
`is.numeric()`, 23, 43
`is.vector()`, 54

`lapply()`, 91
`length()`, 26, 45, 54, 90
`LETTERS`, 48
`letters`, 48
`levels()`, 62, 63, 85
`levels()<=`, 85
`list`, 65, 70, 91
`list()`, 65
`lm()`, 86
`load()`, 88
`log()`, `log10()`, `log2()`, 21
`logical`, 31, 47
`ls()`, 27, 88, 89

`matrix`, 54, 57, 81, 91
`matrix()`, 54
`mode()`, 43
`month.abb`, 48
`month.name`, 48

`NA`, 27, 28, 47
`NA_character_`, 47
`NA_real_`, 47
`names()`, 85, 90
`names()<=`, 85
`NaN`, 27
`ncol()`, 54, 90
`nrow()`, 54, 90
`numeric`, 20, 23, 41, 42, 52
`numeric()`, 23, 26

`order()`, 53, 65, 81, 82
`ordered()`, 60

`pi`, 21
`plot()`, 92, 93
`print()`, 7, 43, 46, 68, 90

`readRDS()`, 89
`reorder()`, 64
`rep()`, 25
`rev()`, 63
`rle()`, 53
`rm()`, 27
`round()`, 30
`rownames()`, 90

`sapply`, 91
`sapply()`, 91
`save()`, 87, 88
`saveRDS()`, 89
`seq()`, 25
`signif()`, 30
`sin()`, 21
`sort()`, 53, 65
`sprintf()`, 46, 47
`sqrt()`, 21
`str()`, 66, 68, 69, 85, 90
`subset()`, 77–79
`summary()`, 91

`t()`, 58
`tail()`, 90, 91
`transform()`, 83
`TRUE`, 29
`trunc()`, 30, 31, 45
`typeof()`, 43

`unique()`, 40
`unlink()`, 89
`unlist()`, 58, 69, 70
`unname()`, 70

`vapply()`, 91
`vector`, 24

`with()`, 79, 83–85
`within()`, 83–85