Deep R Programming

Marek Gagolewski

v0.1.13.9001 (draft)

Dr habil. **Marek Gagolewski**
Deakin University, Australia
Systems Research Institute, Polish Academy of Sciences
Warsaw University of Technology, Poland
https://www.gagolewski.com

This project received no funding, administrative, technical, or editorial support from Deakin University, Warsaw University of Technology, Polish Academy of Sciences, or any other source.

Anyway, any bug reports/corrections/feature requests are welcome. To make this textbook even better, please file them at https://github.com/gagolews/deepr.

Typeset with XeLaTeX. Please be understanding: it was an algorithmic process, hence the results are $\in$ [good enough, perfect).

Homepage: https://deepr.gagolewski.com/

Datasets: https://github.com/gagolews/teaching-data

Release: v0.1.13.9001 (draft) (2023-01-15T14:50:57+1100)

ISBN: 978-0-6455719-2-9 (reserved) (vX.Y.Z; 2023; Melbourne: Marek Gagolewski)

DOI: 10.5281/zenodo.7490464 (Zenodo)

Contents

Deep R Programming is a **comprehensive course on one of the most popular languages in data science** (statistical computing, graphics, machine learning, data wrangling and analytics). It **introduces the base language** in-depth and is aimed at ambitious students, practitioners, and researchers who would like to become **independent users** of this powerful environment.

This early draft is distributed in the hope that it will be useful.

For many students around the world, educational resources are hardly affordable. Therefore, I have decided that this book should **remain an independent, non-profit, open-access project** (available both in PDF[1] and HTML[2] forms). Whilst, for some people, the presence of a "designer tag" from a major publisher might still be a proxy for quality, it is my hope that this publication will prove useful to those who seek knowledge for knowledge's sake.

Please spread the news about it by sharing the above URLs with your mates, peers, or students. Thank you.

Also, check out my other book, *Minimalist Data Wrangling with Python*[3] [20].

Any bug/typos reports/fixes[4] are appreciated. Although available online, this is a whole course, and should be read from the beginning to the end. Please refer to the *Preface* for general introductory remarks and design philosophy.

Consider citing this book as: Gagolewski M. (2023), *Deep R Programming*, Zenodo, Melbourne, DOI: 10.5281/zenodo.7490464[5], ISBN: 978-0-6455719-2-9, URL: https://deepr.gagolewski.com/.

[1] https://deepr.gagolewski.com/deepr.pdf
[2] https://deepr.gagolewski.com/
[3] https://datawranglingpy.gagolewski.com/
[4] https://github.com/gagolews/deepr/issues
[5] https://dx.doi.org/10.5281/zenodo.7490464

0

Preface

0.1 To R, or not to R

R [52] has been named the eleventh most dreaded programming language in the 2022 StackOverflow Developer Survey[6].

Also, it is a free app, so there must be something wrong with it, right?

But whatever, R is deprecated anyway; the "modern" way is to use **`tidyverse`**. Or we should all just switch to Python[7].

Well, not really[8].

0.2 R as a Language and an Environment

Let us get one thing straight: R is *not* just a *statistical package*. It is a general-purpose, high-level programming language, that just happens to be very powerful for any kind of numerical, data-intense computing. It offers extensive support for statistical, machine learning, data analysis, data wrangling, and data visualisation applications, but there is a lot more.

Initially, R was written "for statisticians by statisticians". Therefore, it may be thought of as a free yet more capable alternative to Stata, SAS, SPSS, Statistica, Minitab, Weka, etc. Unlike some of them, however, a spreadsheet-like GUI is not the main gateway for performing computations on data. In R, a user must *write code* to get things actually done. Despite the learning curve's being a little steeper for non-programmers, in the long run, it empowers their users because they are not limited only to the most common scenarios. If some functionality is missing or does not suit their needs, they can easily implement everything themselves.

It is thus very convenient for rapid prototyping. It helps turn our ideas into operational code that can be tested, extended, polished, run in production, and otherwise enjoyed overall. As an interpreted language, it can be run not only in an interactive read-eval-

[6] https://survey.stackoverflow.co/2022/
[7] https://datawranglingpy.gagolewski.com/
[8] Or, as Aussies would say, *yeah, nah*.

print loop (command–result, question–answer, ...), but also in batch mode (running whole, standalone scripts).

Thus, we would rather position R amongst such tools/languages for numerical or scientific computing as Python with the NumPy ecosystem, Julia, GNU Octave, Scilab, MATLAB, etc. However, it is more *specialised* in data science applications than all of them. Hence, it provides a much smoother experience. This is why, over the years, R has become the de facto standard in statistics and many other related fields.

Important R is a whole ecosystem (environment). It not only consists of the R language interpreter, but also features advanced:

- graphical capabilities (see Chapter 13),
- a help system (Section 1.4),
- ways for convenient interfacing with compiled code (`Chapter %s`),
- a package system and centralised package repositories (such as CRAN and Bioconductor; Section 7.3.1),
- a lively community of users and developers – curious and passionate people, just like you and me.

Note R's predecessor is the popular S system designed in the 1980s by John M. Chambers and his colleagues at Bell Labs S: [3, 4, 8, 42]. R is called GNU S, a free, open-source version of its commercial counterpart developed in the mid-1990s[9] by Robert Gentleman and Ross Ihaka of the Statistics Department, University of Auckland, and a large number of contributors; see [7, 31] for some historical notes.

R has a C language-like syntax that involves the use of {curly braces}. Still, in principle, it is a beautiful, functional programming language: its design was heavily inspired by Scheme (see [1] and `Chapter %s` for more details). It is also somewhat object-oriented (Chapter 10).

0.3 Aims, Scope, and Design Philosophy

Many users have been introduced to R by means of some very advanced operations involving data frames, formulas, and functions that rely on nonstandard evaluation (metaprogramming), like:

[9] R version 0.49 released in April 1997 (the first for which source code[2] is available on CRAN), was already quite feature-rich (e.g., implemented S3 methods, formulas, and data frames introduced in the 1991 version of S [8]).

[9] https://cloud.r-project.org/src/base/R-0/

```
lm(
    Ozone~Solar.R+Temp,
    data=subset(airquality, Temp>60, select=-(Month:Day))
) |> summary()
```

This is horrible.

Another group has been isolated from the base R through a thick layer of third-party packages that feature an overwhelming number of functions (every operation, regardless of its complexity, has a different name), often duplicating the core functionality, and sometimes being quite incompatible with our traditional system.

Both families should be fine — as long as they limit themselves to solving only the simplest and most common data processing problems.

But *we* yearn for more. We do not want hundreds of prefabricated *recipes* for popular dishes that we can mindlessly apply without much understanding.

Our aim is to learn *base R*, which is *supposed to be* the common language (lingua franca) to all R users. We want to be able to write code that everybody *should* be able to understand, and which will be likely to work without modifications ten years from now (no slang!).

We want to be able to tackle *any* data-intense problem. Furthermore, we want to develop skills that are *transferable*, so that learning new tools such as Julia or Python with NumPy and Pandas will be much easier later (because R is not the only notable environment out there).

Anyway, enough preaching. This graduate[10]-level textbook is for independent readers who do not mind a slightly steeper learning curve at the beginning, but who are able to appreciate a more cohesively and comprehensively[11] organised material.

Some will benefit from it as a first introduction to R (but without all the pampering). For others[12], this will be a good course from intermediate to advanced (do not skip the first chapters, though).

Either way, do not forget to solve *all* the prescribed exercises.

Good luck.

[10] The author taught similar courses for his wonderfully ambitious undergraduate data/computer science and maths students at Warsaw University of Technology, where such an approach has proven not difficult at all. It requires a more independent, curious, and motivated mindset, though. And that's the way to go, in the long run.

[11] Yours truly is neither a historian, a stenographer, nor a grammarian. We allow ourselves to make a few noninvasive idealisations for didactic purposes. Languages evolve over time, R now is different from what it used to be, and we can shape it (slowly, because we value its stable API) to become something even better in the future.

[12] It might also happen that for some, this will not be a good course at all, either at this stage of their career (come back later) or in general (no dramas). This is a non-profit, open-access project, but it does not mean it is ideal for everyone – in such a case, give other sources a try, e.g., [5, 10, 36, 43, 44, 45, 51], etc. Some of them are also freely available.

0.4 ⨯ Classification of R Data Types and Book Structure

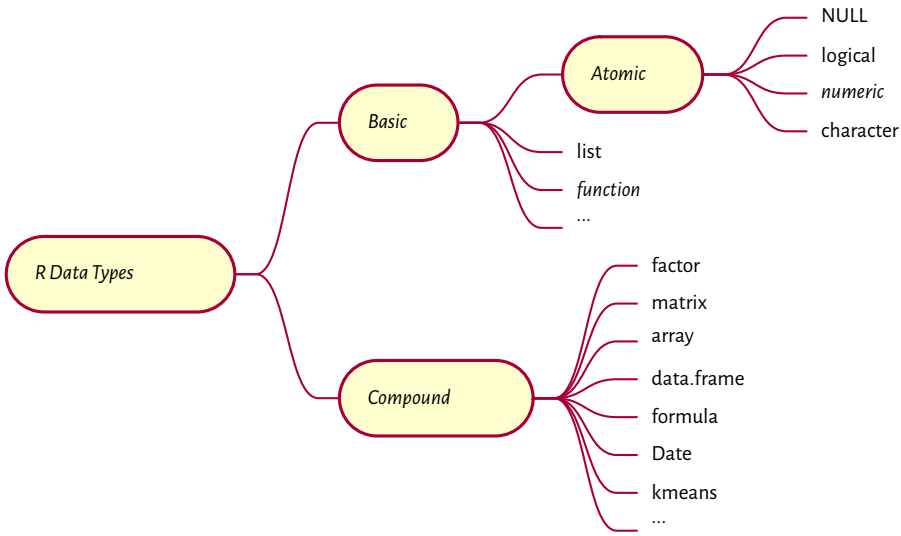

Figure 1: An overview of the most prevalent R data types (see Figure 16.1 for a more comprehensive list)

The most commonly used R data types can be classified as follows; see also Figure 1.

1. *Basic types* – which we discuss in the first part of this book – internal or built-in types, upon which more complex ones are hinged:
    - *atomic vectors* – represent whole sequences of values, where every element is of the same type:
        - `logical` (Chapter 3) – includes items that are `TRUE` ("yes", "present"), `FALSE` ("no", "absent"), or `NA` ("not available", "missing");
        - `numeric` (Chapter 2) – features real numbers, such as `1`, `3.14`, `-0.0000001`, etc.;
        - `character` (Chapter 6) – contains strings of characters, e.g., `"groß"`, `"123"`, or "Добрий день";
    - `function` (Chapter 7) – used to group a series of expressions (code lines) so that they can be applied on different kinds of input data to generate the (hopefully) desired outcomes, for instance, **`cat`**, **`print`**, **`plot`**, **`sample`**, and **`sum`**;
    - `list` (Chapter 4) a.k.a. a generic vector – can store elements of mixed types;

    The above will be complemented with a discussion on vector indexing (Chapter 5) and ways to control the program flow (Chapter 8).

2. *Compound types* – discussed in the second part – wrappers around objects of basic types that might behave differently from the underlying primitives thanks to the dedicated operations *overloaded* for them. They are

    - `factor` (Section 10.3.3) – a vector-like object that represents qualitative data (on a nominal or an ordered scale);
    - `matrix` (Chapter 11) – stores tabular data, i.e., arranged into rows and columns, where each cell is usually of the same type;
    - `data.frame` (Chapter 12) – also used for depositing tabular data, but this time such that each column can be of different type;
    - and many more, which we or third-parties can define arbitrarily using, amongst others, the principles of S3-style object orientated-programming (Chapter 10).

    In this part of the book, we also discuss the principles of sustainable coding (Chapter 9) as well as introduce the basic ways to prepare publication-quality graphics (Chapter 13).

3. ⨯Some more advanced material that, in most cases, we can easily do without, but which is still essential to gain a full understanding of and control over the environment, is discussed in the first part. This includes, amongst others, the following data types:

    - `externalptr` (`sec:to-do`);
    - `environment` (`sec:to-do`);
    - `symbol` (`name`), `call`, `expression` (`sec:to-do`);
    - `formula` (Section 16.5) – used by some functions to specify supervised learning models or define operations to be performed within data subgroups, amongst others;

    ⨯Also, we will discuss other things, but this is an early draft of this book, so right now, we only provide a placeholder therefor (`sec:to-do`). Please come back later.

Note The above classification is just a first approximation to the complete type classification that we give in Figure 16.1.

Also, we should not be surprised that above we do not see any of the data types defined by a few very popular[13] third-party packages. We will later see that we can most often do without them. If that is not the case, we will become skilled enough to learn them easily ourselves.

[13] Which does not automatically mean *good*. For instance, sugar, salt, and some drugs are very popular, but it does not make them healthy.

0.5 About the Author

I, Marek Gagolewski[14] (pronounced like Ma'rek Gong-olive-ski), am currently a Senior Lecturer in Applied AI at Deakin University in Melbourne, VIC, Australia and an Associate Professor in Data Science at the Systems Research Institute of the Polish Academy of Sciences.

My research interests are related to data science, in particular: modelling complex phenomena, developing usable, general-purpose algorithms, studying their analytical properties, and finding out how people use, misuse, understand, and misunderstand methods of data analysis in research, commercial, and decision-making settings. I'm an author of 90+ publications, including journal papers in outlets such as *Proceedings of the National Academy of Sciences (PNAS)*, *Information Fusion*, *International Journal of Forecasting*, *Statistical Modelling*, *Journal of Statistical Software*, *Information Sciences*, *Knowledge-Based Systems*, *IEEE Transactions on Fuzzy Systems*, and *Journal of Informetrics*.

In my "spare" time, I write books for my students (also check out my *Minimalist Data Wrangling with Python*[15] [20]) and develop open-source (libre) data analysis software, such as **`stringi`**[16] (one of the most often downloaded R packages), **`genieclust`**[17] (a fast and robust clustering algorithm in both Python and R), and many others[18].

0.6 Acknowledgements

Deep R Programming is based on my experience as an R user (since ~2003), developer of open-source packages (see above), tutor/lecturer (since ~2008), and an author of a quite successful Polish textbook *Programowanie w języku R* (see [19]) which was published by PWN (1st ed. 2014, 2nd ed. 2016). Even though the current book is an entirely different work, its predecessor served as an excellent testbed for many ideas conveyed here.

In particular, the teaching style exercised in this book has proven successful in many similar courses that yours truly has been responsible for, including at Warsaw University of Technology, Data Science Retreat (Berlin), and Deakin University (Melbourne). I thank all my students and colleagues for the feedback given over the last 15-odd years.

We describe R version 4.2.2 Patched (2022-11-10 r83330). However, we expect 99.9% of material covered here to be valid in future releases (consider filing a bug report if you discover that this is not the case).

[14] https://www.gagolewski.com
[15] https://datawranglingpy.gagolewski.com/
[16] https://stringi.gagolewski.com
[17] https://genieclust.gagolewski.com
[18] https://github.com/gagolews

This book was prepared in a Markdown superset called MyST[19], **Sphinx**[20], and TeX (XeLaTeX). Code chunks were processed with the R package **knitr** [46]. All figures were plotted with the low-level **graphics** package using the author's own style template. A little help from Makefiles, custom shell scripts, and **Sphinx** plugins (**sphinxcontrib-bibtex**[21], **sphinxcontrib-proof**[22]) dotted the *j*'s and crossed the *f*'s. The **Ubuntu Mono**[23] font is used for the display of `code`. Typesetting of the main text relies upon the *Alegreya*[24] and *Lato*[25] typefaces.

This book received no funding, administrative, technical, or editorial support from Deakin University, Warsaw University of Technology, Polish Academy of Sciences, or any other source.

[19] https://myst-parser.readthedocs.io/en/latest/index.html
[20] https://www.sphinx-doc.org/
[21] https://pypi.org/project/sphinxcontrib-bibtex/
[22] https://pypi.org/project/sphinxcontrib-proof/
[23] https://design.ubuntu.com/font/
[24] https://www.huertatipografica.com/en
[25] https://www.latofonts.com/

Part I

Deep

1

Introduction

1.1 Hello, World!

Traditionally, every programming journey starts with the printing of a "Hello, World"-like greeting. Let's then get it over with asap:

```
cat("My hovercraft is full of eels.")
## My hovercraft is full of eels.
```

By calling the **cat** function, we printed out a given character string that we enclosed in double quote characters.

Documenting code is a good development practice. It is thus worth knowing that any text followed by a hash sign (that is not part of a string) is a *comment*, ignored by the interpreter.

```
# This is a comment.
# This is another comment.
cat("I cannot wait", "till lunchtime.")  # two arguments (another comment)
## I cannot wait till lunchtime.
cat("# I will not buy this record.\n# It is scratched.")  # `\n` == newline
## # I will not buy this record.
## # It is scratched.
```

By convention, in this book, the textual outputs generated by R itself are always preceded by two hashes. This makes copy-pasting all code chunks easier in the case where the kind reader would like to experiment with them by themself (which is always highly encouraged).

Whenever a call to some function is to be made, the round brackets are obligatory. All objects within the parentheses (they are separated by commas) constitute the input data to be consumed by the operation. Thus, the syntax is: **`some_function_to_be_called`**`(argument1, argument2, etc.)`.

1.2 Setting up the Development Environment

1.2.1 Installing R

It is quite natural to pine for the ability to execute the above code ourselves – we cannot learn programming without getting our hands dirty.

The official precompiled binary distributions of R can be downloaded from https://cran.r-project.org/.

For serious programming work[1], we recommend, sooner rather than later, switching to[2] one of the Unix-like operating systems. This includes the free, open-source (== good) variants of GNU/Linux, amongst others, or the proprietary (== very far from good) m**OS. The users thereof might employ their favourite package manager (e.g., **`apt`**, **`dnf`**, **`pacman`**, or **`Homebrew`**) to install R.

Users of other operating systems (such as Wi***ws) might consider installing Anaconda or Miniconda if they require some level of interoperability with the Python environment, e.g., they would like to work with the Jupyter environment (Section 1.2.5).

Below we review several ways in which we can write and execute R code. It is up to the benign reader to research, setup, and learn the development environment that suits their needs. As usual in real life, there is no single universal approach that always works best in all the scenarios.

1.2.2 Interactive Mode

R's *read-eval-print loop* (REPL) can give us instant gratification whenever we would like to compute something quickly, e.g., determine basic aggregates of a few numbers entered by hand or evaluate a mathematical expression like "`2+2`".

How to start the R console varies from system to system, e.g., users of Unix-like boxes can simply execute **`R`** from the terminal (shell). Wi***ws folks can fire up the **`RGui`** from the *Start* menu.

Important When working interactively, the default[3] command prompt, "`>`", means: *I am awaiting an order*. Moreover, "`+`" denotes: *Please continue*. In such a case, we should either complete the unfinished expression, or cancel the operation by pressing `ESC` or `CTRL+C` (depends on the operating system).

```
> cat("And now
```

(continues on next page)

[1] For instance, when an easy interoperability with other programming languages/environments is required or when we think about scheduling jobs on Linux-based computing/container clusters.

[2] Or at least trying out – by installing a copy of GNU/Linux on a virtual machine (VM).

[3] It can be changed; see **`help`**`("options")`.

(continued from previous page)

```
+ for something
+ completely different
+
+
+ it is an unfinished expression...
+ awaiting another double quote character and then the closing bracket...
+
+ press ESC or CTRL+C to abort input
>
```

For readability, we never print out the command prompt characters in this book.

1.2.3 Batch Mode: Working with R Scripts (**)

The interactive mode of operation is unsuitable for more complicated tasks, though.

The users of Unix-like operating systems will be interested in another extreme, which involves writing standalone R scripts that can be executed one by line, without any user intervention.

To do so, in the terminal (command line, shell), we can invoke:

```
Rscript file.R
```

where `file.R` is the path to some source file.

Exercise 1.1 *(**) In your favourite text editor (e.g., **Notepad++**, **Kate**, **vi**, **Emacs**, **RStudio**, or **VSCodium**), create a file named `test.R`. Write a few calls to the **`cat`** function. Then, execute this script from the terminal by invoking the **`Rscript`** program.*

1.2.4 Weaving: Automatic Report Generation (**)

Reproducible data analysis[4] requires us to keep the results (text, tables, plots, auxiliary files) synchronised with their generating code and data.

`utils::Sweave` (the **`Sweave`** function from the **`utils`** package) and **`knitr`** [46] are two example template processors that evaluate R code chunks within documents written in LaTeX, HTML, or other markup languages. The chunks are replaced by the outputs they yield.

This book is a showcase of such an approach – all the results, including Figure 2.3 and the above "Hello, World", were generated programmatically. Thanks to its being written in the highly universal Markdown[5] language, it could be easily converted to a single

[4] The idea dates back to Knuth's literate programming concept; see [32].
[5] https://daringfireball.net/projects/markdown/

PDF document[6] as well as the whole website[7]. Tools like **pandoc** and **docutils** facilitate such operations.

Exercise 1.2 *(**) Install the* ***knitr*** *package by calling* ***install.packages****("knitr") from within an R session. Then, create a text file named* `test.Rmd` *with the following content:*

```
# Hello, Markdown!

This is my first automatically generated report,
where I print stuff.

```{r}
print("G'day!")
print(2+2)
```

Thank you for your attention.
```

Assuming that the file is located in the current working directory (compare Section 7.3.3), call ***knitr::knit****("test.Rmd") from within the R console or run the following in the terminal:*

```
Rscript -e 'knitr::knit("test.Rmd")'
```

Then, inspect the generated Markdown file, `test.md`*.*

Furthermore, if you have the ***pandoc*** *tool installed, to generate a standalone HTML file, execute in the terminal:*

```
pandoc test.md --standalone -o test.html
```

Alternatively, for ways to call external programs from R, see Section 7.3.3.

1.2.5 Semi-Interactive Modes (Jupyter Notebooks, Sending Code to an Associated R Console, etc.)

The nature of the most frequent use cases of R encourages a semi-interactive workflow, where we progress with prototyping fast by trial-and-error.

In this mode, we write a series of short code fragments inside a standalone R script.

Each fragment implements a simple, well-defined task, such as the loading of data files, data cleansing, feature visualisation, computations of some information aggregates, etc.

Importantly, any code chunk can be sent to the associated R console and executed therein. This way, we can inspect the results it generates at any time. If we are not happy with the outcome, we can apply any corrections that are necessary.

[6] https://deepr.gagolewski.com/deepr.pdf
[7] https://deepr.gagolewski.com

There are quite a few integrated development environments (IDEs; sometimes requiring additional plugins) that enable such a workflow, including **`JupyterLab`**, **`Emacs`**, **`RStudio`**, and **`VSCodium`**.

Executing an individual code line or a whole text selection is usually done by pressing a (configurable) keyboard shortcut such as `Ctrl+Enter` or `Shift+Enter`.

Exercise 1.3 *(*) **`JupyterLab`**[8] is a development environment that runs in a web browser. It was programmed in Python, but supports many programming languages. Thanks to **`IRkernel`**[9], we can use it with R.*

1. *Install **`JupyterLab`** and **`IRkernel`** (for instance, if you use Anaconda, run `conda install -c r r-essentials`).*
2. *From the* File *menu, select* Create a new R source file *and save it as, e.g., `test.R`.*
3. *From the* File *menu, select* Create a new console for editor running the R kernel.
4. *Type some print "Hello, World"-like calls.*
5. *Press `Shift+Enter` (whilst working in the editor) to send different code fragments onto the console and execute them. Inspect the results.*

See Figure 1.1 for an illustration.

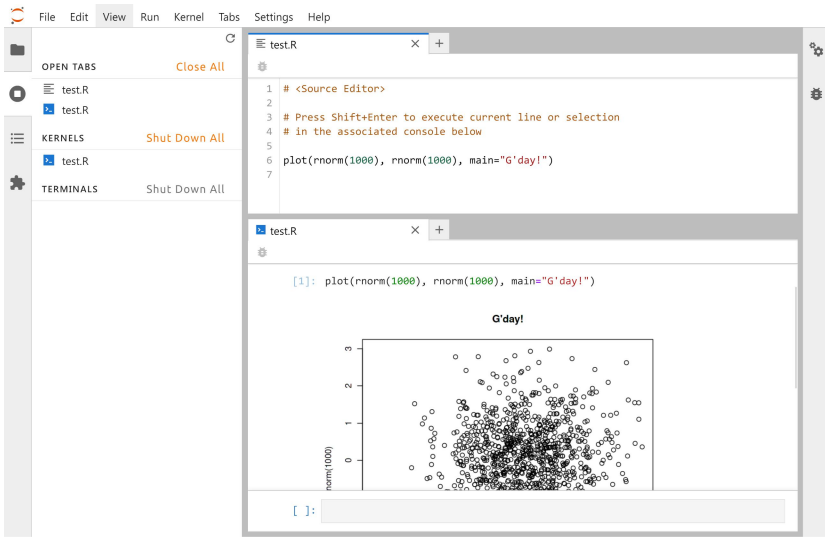

Figure 1.1: JupyterLab: A source file editor and the associated R console, where we can run arbitrary code fragments

Example 1.4 *(*) The Jupyter project, whose **`JupyterLab`** is part of, also supports the handling of dedicated Notebooks. There, editable and executable code chunks and results they generate can*

[8] https://jupyterlab.readthedocs.io/en/stable/
[9] https://irkernel.github.io/

be kept together in a single `.ipynb` *(JSON) file; see Figure 1.2 for an illustration and Chapter 1 of [20] for a quick introduction (from the Python language kernel perspective).*

This environment is quite convenient for live coding (e.g., for teachers) or performing exploratory data analyses. However, for more serious programming work, the code can get quite messy (luckily, there is always an option to export a notebook to an executable, plain text R script).

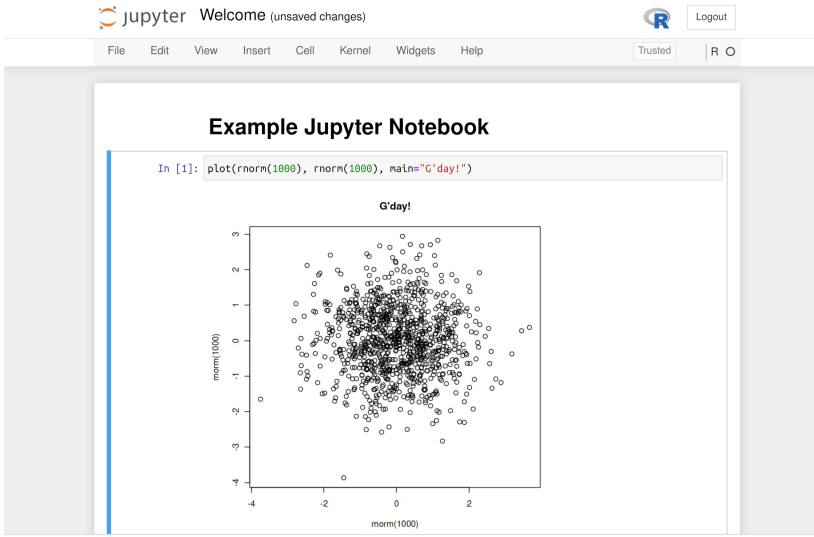

Figure 1.2: An example Jupyter Notebook, where we can keep the code and the results together

1.3 Atomic Vectors at a Glance

After the printing of the "Hello, World" message, a typical programming course would normally proceed with the discussion on basic data types for storing individual numeric or logical values. Next, we would be introduced to arithmetic and comparison operations on such *scalars*, followed by the definition of whole arrays or other collections of such values, complemented by the methods to iterate over them, one element after another.

In R, no separate types representing individual values have been defined. Instead, what seems to be a single datum, is already a *vector* (sequence, array) of length 1.

```
2.71828          # input a number (here: the same as print(2.71828))
## [1] 2.7183
length(2.71828)  # it is a vector featuring one element
## [1] 1
```

To create a vector of any length, we can call the **c** function, which combines given arguments into a single sequence:

```
c(1, 2, 3)  # three vectors of length 1  ->  one vector of length 3
## [1] 1 2 3
length(c(1, 2, 3))
## [1] 3
```

In Chapter 2, Chapter 3, and Chapter 6, we will discuss the most prevalent types of atomic vectors: numeric, logical, and character ones, respectively.

```
c(0, 1, -3.14159, 12345.6)           # four numbers
## [1]     0.0000     1.0000    -3.1416 12345.6000
c(TRUE, FALSE)                       # two logical values
## [1]  TRUE FALSE
c("spam", "spam", "bacon and spam")  # three character strings
## [1] "spam"           "spam"           "bacon and spam"
```

We call them *atomic*, because they can only group together values of the same type. Lists, which we will discuss in Chapter 4, are, on the other hand, referred to as *generic* vectors – they can be used for storing items of mixed types – other lists as well.

Note Not having separate scalar types greatly simplifies the programming of numerical computing tasks. Vectors are prevalent in our main areas of interest – statistics, simulations, data science, machine learning, and all other data-oriented computing. For example, columns and rows in tables (values of different features describing clients, ratings of items given by users) or time series (stock market prices, readings from temperature sensors) are all best represented by means of such sequences.

Moreover, the fact that vectors are the core part of the R language makes their use very natural – as opposed to the languages that require special add-ons for vector processing, e.g., **numpy** for Python [29]. By learning different ways to process them *as a whole*, instead of one element at a time, we will assure that our ideas can quickly be turned into working code (rapid prototyping). For instance, computing summary statistics such as, say, the mean absolute deviation of some sequence `x`, will be as effortless as writing `mean(abs(x-mean(x)))`. Such a code is not only easy to read and maintain, but it is also fast to run.

1.4 Getting Help

Our aim is to become independent, advanced R programmers.

Independent, however, does not mean omniscient. The *R help system* is the authorit-

ative source of knowledge about specific functions or more general topics. To open a help page, we call:

```
help("topic")  # equivalently: ?"topic"
```

Exercise 1.5 *Sight (without going into detail) the manual on the* `length` *function by calling* `help("length")`*. Note that most help pages are structured as follows:*

1. Header: *"*`package:base`*" means that the function is a base one (see Section 7.3.1 for more details on the R package system);*
2. Title*;*
3. Description*: a short description of what the function does;*
4. Usage*: the list of formal arguments (parameters) to the function;*
5. Arguments*: the meaning of each formal argument explained;*
6. Details*: technical information;*
7. Value*: return value explained;*
8. References*: further reading;*
9. See Also*: links to other help pages;*
10. Examples*: R code that is worth to run and study by yourself.*

We can also search within all the installed help pages by calling:

```
help.search("vague topic")  # equivalently: ??"vague topic"
```

Oftentimes, this way we will be able to find answers to our questions more reliably than when asking DuckDuckGo or G**gle (which commonly feature many low quality/irrelevant/distracting results that can make us lose the sacred code writer's flow).

Important All code chunks, including code comments and textual outputs, form an integral part of this book's text. They should not be skipped by the reader. On the contrary, they should become objects of our intense reflection and thorough investigation.

For instance, whenever we introduce a few function, it may be a good idea to look it up in the help system. Moreover, playing with the presented code (running, modifying, experimenting, etc.) is also very beneficial. We should develop the habit of asking ourselves questions like "what would happen if…", and then finding the answers on our own.

We are now ready to discuss the most significant operations on numeric vectors, which constitute the main theme of the next chapter. See you there.

1.5 Exercises

Exercise 1.6 *What are the three most important types of atomic vectors?*

Exercise 1.7 *According to the classification of the R data types we introduced in the previous chapter, are atomic vectors basic or compound types?*

2

Numeric Vectors

In this chapter, we discuss the uttermost common operations on numeric vectors. They are so fundamental that we will also find them in other scientific computing environments, including Python with **`NumPy`** or **`TensorFlow`**, Julia, MATLAB, GNU Octave, or Scilab.

At first blush, the number of functions we are going to explore may seem quite large. Still, the reader is kindly asked to place some trust (a rare thing these days) in yours truly. It is because our selection is comprised only of the most representative and educational amongst the plethora of possible choices. More complex building blocks can either be reduced to a creative combination of the former or be easily found – should the need arise – in a number additional packages or libraries (e.g., the GNU GSL [23]).

A solid understanding of base R programming is necessary for the effective dealing with the popular packages (such as **`data.table`**, **`dplyr`**, or **`caret`**). Most importantly, base R's API is *stable*, hence the code we write today will most likely work the same way in 10 years. This is often not the case when we rely on third-party add-ons.

In the sequel, we will be advocating a minimalist, keep-it-simple approach to the art of programming of data processing pipelines, one that is a good balance between "doing it all by oneself", "minimising the information overload", "being lazy", and "standing on the shoulders of giants".

Note The exercises that we suggest below are all self-contained, unless explicitly stated otherwise. The use of language constructs that are yet to be formally introduced (in particular, **`if`**, **`for`**, and **`while`** which we will explain in Chapter 8) is not only unnecessary, but discouraged. Moreover, we recommend against taking shortcuts by looking up partial solutions on the internet. Rather, to get the most out of this course, the reader should be seeking relevant information within the current and preceding chapters as well as the R help system.

2.1 Creating Numeric Vectors

2.1.1 Numeric Constants

The simplest numeric vectors are those of length one:

```
-3.14
## [1] -3.14
1.23e-4
## [1] 0.000123
```

The latter is in what we call the *scientific notation* which is convenient means of entering numbers of very large or small order of magnitude. Here, "e" stands for "... times 10 to the power of...". Therefore, `1.23e-4` is equal to $1.23 \times 10^{-4} = 0.000123$. In other words, given 1.23, we move the decimal separator by 4 digits towards the left.

In real life, some information items may be inherently or temporarily missing, unknown, or Not Available. R is data processing-oriented, hence it is equipped with a special indicator:

```
NA_real_  # numeric NA (missing value)
## [1] NA
```

This is similar to the *Null* marker in database query languages such as SQL. Note that `NA_real_` is displayed simply as "NA", chiefly for readability.

Moreover, `Inf` denotes the infinity (∞; a value that is larger than the largest representable double precision – 64 bit – floating point number) and `NaN` stands for *not-a-number* (it is returned as the result of some illegal operations, e.g., $0/0$ or $\infty - \infty$).

2.1.2 Concatenating Vectors with `c`

Let us provide some ways to create numeric vectors with possibly more than 1 element.

First, the **`c`** function we introduced in the previous chapter, can be used to *c*ombine (concatenate) many numeric vectors, each of any length, so as to form a single object:

```
c(1, 2, 3)  # 3 vectors of length 1  ->  1 vector of length 3
## [1] 1 2 3
c(1, c(2, NA_real_, 4), 5, c(6, c(7, Inf)))
## [1]   1   2  NA   4   5   6   7 Inf
```

Note Running **`help`**`("c")`, we will see that its usage is like "**`c`**`(...)`". In the current context, this means that the **`c`** function takes an arbitrary number of arguments. In Section 9.5.6 we will study the dot-dot-dot (ellipsis) parameter in more detail.

2.1.3 Repeating Entries with `rep`

Second, **`rep`** *rep*licates the elements in a given vector a given number of times.

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

In the second case, the whole vector (1, 2, 3) has been *recycled* (tiled) four times. Interestingly, if the second argument was a vector of the same length as the first one, the behaviour would be quite different:

```
rep(c(1, 2, 3), c(2, 1, 4))
## [1] 1 1 2 3 3 3 3
rep(c(1, 2, 3), c(4, 4, 4))
##  [1] 1 1 1 1 2 2 2 2 3 3 3 3
```

Here, *each* element is repeated the *corresponding* number of times.

If we call **help**("rep"), we will come across the notion like "**rep**(x, ...)" in the *Usage* section. Unfortunately, it is rather peculiar, but reading further we discover the dot-dot-dot stands for one of the following further parameters (see the *Arguments* section):

- times,
- length.out,
- each.

So far, we have been playing with times, which is listed second in the parameter list (after x – the vector whose elements are to be repeated).

Important It turns out that the following function calls are all equivalent:

```
rep(c(1, 2, 3), 4)   # positional matching of arguments: `x`, then `times`
rep(c(1, 2, 3), times=4)    # `times` is the second argument
rep(x=c(1, 2, 3), times=4)  # keyword arguments of the form name=value
rep(times=4, x=c(1, 2, 3))  # keyword arguments can be given in any order
rep(times=4, c(1, 2, 3))    # mixed positional and keyword arguments
```

We can also pass each or length.out (a dot has no special meaning in R; see Section 2.2), but their names should be mentioned explicitly:

```
rep(c(1, 2, 3), length.out=7)
## [1] 1 2 3 1 2 3 1
rep(c(1, 2, 3), each=3)
## [1] 1 1 1 2 2 2 3 3 3
rep(c(1, 2, 3), length.out=7, each=3)
## [1] 1 1 1 2 2 2 3
```

Note Whether it was a good programming practice to actually implement a range of varied behaviours inside a single function is a matter of taste. On the one hand, in all of the examples above, we do repeat the input elements somehow, so remembering just one function name is really convenient. Nevertheless, a drastic change in the repetition pattern depending, e.g., on the length of the `times` argument can be bug-prone. Anyway, we have been warned[1].

Zero-length vectors are also possible:

```
rep(c(1, 2, 3), 0)
## numeric(0)
```

Even though their handling might be a little tricky (compare Chapter 9), we will see later that they are useful in contexts like "create an empty data frame with a specific column structure".

2.1.4 Generating Arithmetic Progressions with `seq` and `:`

Third, we can call the **`seq`** function to create a sequence of equally-spaced numbers (on a linear scale, i.e., an arithmetic progression).

```
seq(1, 15, 2)
## [1]  1  3  5  7  9 11 13 15
```

Reading the function's help page, we note that it has the following parameters: `from`, `to`, `by`, `length.out`, amongst others.

Thus, the above call is equivalent to:

```
seq(from=1, to=15, by=2)
## [1]  1  3  5  7  9 11 13 15
```

Note that `to` actually means "up to":

```
seq(from=1, to=16, by=2)
## [1]  1  3  5  7  9 11 13 15
```

We can also pass `length.out` instead of `by`. In such a case, the increments or decrements will be computed via the formula `((to - from)/(length.out - 1))`; this *default value* is reported in the *Usage* section in **`help`**`("seq")`.

[1] Some "caring" R users might be tempted to introduce two new functions now,one for generating (1, 2, 3, 1, 2, 3, ...) only and the other outputting patterns like (1, 1, 1, 2, 2, 2, ...). They would most likely wrap them in a new package and announce that on Twitter. But this is nothing else than a multiplication of entities without actual necessity; we would end up with three functions: the original one, **`rep`**, which everyone should know anyway because it is so basic and has been and will be used everywhere by almost everybody so far, and the two redundant ones, whose user-friendliness is only illusory. See also Chapter 9 for discussion on the design of functions.

```
seq(1, 0, length.out=5)
## [1] 1.00 0.75 0.50 0.25 0.00
```

Also, this:

```
seq(length.out=5)  # default `from` is 1
## [1] 1 2 3 4 5
```

Arithmetic progressions with step equal to 1 or -1 can also be generated via the `:` operator.

```
1:10    # seq(1, 10) or seq(1, 10, 1)
##  [1]  1  2  3  4  5  6  7  8  9 10
-1:10   # seq(-1, 10) or seq(-1, 10, 1)
##  [1] -1  0  1  2  3  4  5  6  7  8  9 10
-1:-10  # seq(-1, -10) or seq(-1, -10, -1)
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

Note the order of precedence of this operator: "`-1:10`" means "`(-1):10`" and not "`-(1:10)`"; compare Section 2.4.3.

Exercise 2.1 *Take a look at the manual page of* ***`seq_along`*** *and* ***`seq_len`*** *and determine whether they can easily be done without, having* ***`seq`***[2] *at hand.*

2.1.5 Generating Pseudorandom Numbers

We can also generate sequences drawn independently from a range of univariate probability distributions.

```
runif(7)  # uniform U(0, 1)
## [1] 0.287578 0.788305 0.408977 0.883017 0.940467 0.045556 0.528105
rnorm(7)  # normal N(0, 1)
## [1]  1.23950 -0.10897 -0.11724  0.18308  1.28055 -1.72727  1.69018
```

These correspond to seven pseudorandom deviates following the uniform distribution on the unit interval (i.e., (0, 1)) and the standard normal distribution (i.e., with expectation 0 and standard deviation 1), respectively; compare Figure 2.3.

For more *named* distribution classes (frequently occurring in various real-world statistical modelling exercises), see Section 2.3.4.

Another useful function samples a number of values from a given vector, either with or without replacement:

[2] Also note that certain configurations of **`seq`** and its variants might return vectors of type `integer` instead of `double`, some of them in a compact (ALTREP) form; see Section 6.4.1.

```
sample(1:10, 20, replace=TRUE)  # 20 with replacement (allow repetitions)
##  [1]  3  3 10  2  6  5  4  6  9 10  5  3  9  9  9  3  8 10  7 10
sample(1:10, 5, replace=FALSE)  # 5 without replacement (do not repeat)
## [1] 9 3 4 6 1
```

The distribution of the sampled values does not need to be uniform; the `prob` argument may be fed with a vector of the corresponding probabilities. For example, here are 20 independent realisations of the random variable X such that $\Pr(X = 0) = 0.9$ (the probability that we obtain 0 is equal to 90%) and $\Pr(X = 1) = 0.1$:

```
sample(0:1, 20, replace=TRUE, prob=c(0.9, 0.1))
##  [1] 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
```

Note If `n` is a single number (a numeric vector of length 1), then **`sample`**`(n, ...)` is equivalent to **`sample`**`(1:n, ...)`. Similarly, **`seq`**`(n)` is a synonym for **`seq`**`(1, n)` or **`seq`**`(1, `**`length`**`(n))`, depending on the length of `n`. This is a dangerous behaviour which can occasionally backfire and lead to bugs (check what happens when `n` is, e.g., 0). Nonetheless, we have been warned and from now on are going to be extra careful (but are we really?). Read more at **`help`**`("sample")` and **`help`**`("seq")`.

Let us stress that the numbers we obtain are merely *pseudo*random, because they are generated algorithmically. R uses the Mersenne-Twister MT19937 method [37] by default; see **`help`**`("RNG")` and [16, 24, 34]. By setting the *seed* of the random number generator, i.e., re-setting its state to a given one, we can obtain results that are *reproducible*.

```
set.seed(12345)  # seeds are specified with integers
sample(1:10, 5, replace=TRUE) # a,b,c,d,e
## [1]  3 10  8 10  8
sample(1:10, 5, replace=TRUE) # f,g,h,i,j
## [1]  2  6  6  7 10
```

Setting the seed to the one used previously gives:

```
set.seed(12345)
sample(1:10, 5, replace=TRUE) # a,b,c,d,e
## [1]  3 10  8 10  8
```

We did not(?) expect that! And now for something completely different:

```
set.seed(12345)
sample(1:10, 10, replace=TRUE) # a,b,c,d,e,f,g,h,i,j
##  [1]  3 10  8 10  8  2  6  6  7 10
```

Reproducibility is a crucial feature of each truly scientific experiment. The same initial condition (here: the same seed), leads to exactly the same outcomes.

Note Some claim that the only unsuspicious seed is 42, but each programmer can have their own picks. Yours truly, for example, uses 123, 1234, and 12345 as well. When performing many runs of Monte Carlo experiments, it may be a good idea to call **`set.seed(i)`** in the `i`-th iteration of a simulation we are trying to program.

Anyhow, we should make sure that our seed settings are applied consistently across all our scripts. Otherwise, we might be accused of tampering with evidence. For instance, here is the ultimate proof that we are very lucky today:

```
set.seed(1679619)  # totally unsuspicious, right?
sample(0:1, 20, replace=TRUE)  # so random
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

This is exactly why reproducible scripts and auxiliary data should be published alongside all research reports or papers. Only open, transparent science can be fully trustworthy.

If **`set.seed`** is not called explicitly, and the random state is not restored from the previously saved R session (see Chapter 16), then the random generator is initialised based on the current wall time and the identifier of the running R instance (PID). This may give the impression that the numbers we generate are surprising.

In order to understand the "pseudo" part of the said randomness better, in Section 8.3, we will build a very simple random generator ourselves.

2.1.6 Reading Data with `scan`

The example text file named `euraud-20200101-20200630.csv`[3] gives the EUR to AUD exchange rates (how many Australian Dollars can one buy for 1 Euro) from 1 January to 30 June 2020 (remember COVID-19?). Let us preview the first couple of lines:

```
# EUR/AUD Exchange Rates
# Source: Statistical Data Warehouse of the European Central Bank System
# https://www.ecb.europa.eu/stats/policy_and_exchange_rates/
# (provided free of charge)
NA
1.6006
1.6031
NA
```

The four first lines that begin with "`#`" merely serve as comments for us, humans; they should be ignored by the interpreter. The first "real" value, `NA` corresponds to 1 January (Wednesday; New Years Day; Forex markets were closed, hence a missing observation).

[3] https://github.com/gagolews/teaching-data/raw/master/marek/euraud-20200101-20200630.csv

The **scan** function can be used to read all the inputs and convert them to a single numeric vector:

```
scan(paste0("https://github.com/gagolews/teaching-data/raw/",
    "master/marek/euraud-20200101-20200630.csv"), comment.char="#")
##  [1]     NA 1.6006 1.6031     NA     NA 1.6119 1.6251 1.6195 1.6193 1.6132
## [11]     NA     NA 1.6117 1.6110 1.6188 1.6115 1.6122     NA     NA 1.6154
## [21] 1.6177 1.6184 1.6149 1.6127     NA     NA 1.6291 1.6290 1.6299 1.6412
## [31] 1.6494     NA     NA 1.6521 1.6439 1.6299 1.6282 1.6417     NA     NA
## [41] 1.6373 1.6260 1.6175 1.6138 1.6151     NA     NA 1.6129 1.6195 1.6142
## [51] 1.6294 1.6363     NA     NA 1.6384 1.6442 1.6565 1.6672 1.6875     NA
## [61]     NA 1.6998 1.6911 1.6794 1.6917 1.7103     NA     NA 1.7330 1.7377
## [71] 1.7389 1.7674 1.7684     NA     NA 1.8198 1.8287 1.8568 1.8635 1.8226
## [81]     NA     NA 1.8586 1.8315 1.7993 1.8162 1.8209     NA     NA 1.8021
## [91] 1.7967 1.8053 1.7970 1.8004     NA     NA 1.7790 1.7578 1.7596
##  [ reached getOption("max.print") -- omitted 83 entries ]
```

We used the **paste0** function to concatenate two long (too long to fit a single line of code) strings to form a single URL; see Section 6.1.3.

We can also read the files located on our computer, for example:

```
scan("~/teaching-data/marek/euraud-20200101-20200630.csv",
    comment.char="#")
```

uses an absolute file path that starts at the user's home directory, denoted "~": yours truly's case is `/home/gagolews/`.

Note For portability reasons, we should use slashes, "/", as path separators (but see **help**("file.path") and **help**(".Platform")). These are not only recognised by all Unix-like boxes but also other popular operating systems. Note that URLs (such as https://www.r-project.org/) feature slashes too.

Paths can also be relative to the current working directory, denoted ".". It can be read via a call to **getwd**. Usually, it is the directory from where the R session has been started.

For instance, if the current working directory was `/home/gagolews/teaching-data/marek`, we could have written the file path equivalently as `./euraud-20200101-20200630.csv` or even `euraud-20200101-20200630.csv`.

On as side note, `../` would denote the parent directory of the current working directory. For instance, `../r/iris.csv` would be equivalent to `/home/gagolews/teaching-data/r/iris.csv`.

Exercise 2.2 *Read the help page about* ***scan****. Take note of the following formal arguments and their meaning:* `dec`*,* `sep`*,* `what`*,* `comment.char`*, and* `na.strings`*.*

Later we will discuss the **read.table** and **read.csv**, which are wrappers around **scan**

that can be used to read tabular data. Note that **`write`** can be used to export an atomic vector's contents to a text file.

Example 2.3 *Figure 2.1 shows the graph of the aforementioned exchange rates, which was generated by calling:*

```
plot(scan(paste0("https://github.com/gagolews/teaching-data/raw/",
    "master/marek/euraud-20200101-20200630.csv"), comment.char="#"),
    xlab="Day", ylab="EUR/AUD")
```

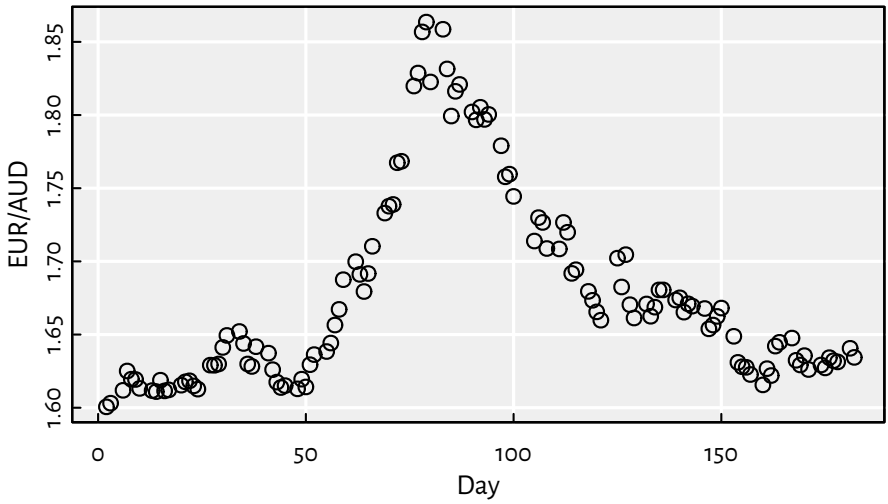

Figure 2.1: EUR/AUD exchange rates from 2020-01-01 (day 1) to 2020-06-30 (day 182)

Somewhat misleadingly (and for the reasons that will become apparent later), the documentation of **`plot`** *can be accessed by calling* **`help`**`("plot.default")`*. Read about, and experiment with, different values of the* `main`*,* `xlab`*,* `ylab`*,* `type`*,* `col`*,* `pch`*,* `cex`*,* `lty`*, and* `lwd` *arguments. More plotting routines will be discussed in Chapter 13.*

2.2 Creating Named Objects

Often, the objects we bring forth will need to be memorised so that they can be referred to in further computations. The assignment operator, **`<-`**, can be used for this very purpose:

```
x <- 1:3  # creates a numeric vector and binds the name `x` to it
```

The now-named object can be recalled and dealt with as we please:

```
print(x)  # or just `x` in the R console
## [1] 1 2 3
sum(x)    # example operation: compute the sum of all elements in `x`
## [1] 6
```

Important In R, all names are *case-sensitive*. Hence, `x` and `X` can coexist peacefully: when set, they refer to two different objects. Also, if we tried to call `Print(x)` above, we would get an error.

Typically, we will be using what we refer to as *syntactic names* (see Section 9.4.1 for an exception though). In the R help system (see **`help`**`("make.names")` and also **`help`**`("Quotes")`), we read: *A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as .2way are not valid, and neither are the reserved words.* For the list of the latter, see **`help`**`("Reserved")`.

A good name is self-explanatory and thus reader-friendly: `patients`, `mean`, and `average_scores` are way better (if they really are what they claim they are) than `xyz123`, `crap`, or `spam`. Also, it might not be such a bad idea to get used to denoting:

- vectors with `x`, `y`, `z`,
- matrices (and matrix-like objects) with `A`, `B`, ..., `X`, `Y`, `Z`,
- integer indexes with letters `i`, `j`, `k`, `l`,
- object sizes with `n`, `m`, `d`, `p` or `nx`, `ny`, etc.,

especially when they are only of temporary nature (for storing some auxiliary results, iterating over collections of objects, etc.).

There are numerous naming conventions that we can adopt, but most often they are a matter of taste; `snake_case`, `lowerCamelCase`, `UpperCamelCase`, `flatcase`, or `dot.case` are equally good as long as they are used coherently (for instance, some use `snake_case` for vectors and `UpperCamelCase` for functions). It may even be the case that we have little choice but to adhere to the naming conventions agreed upon in the project we are about to contribute to.

Note Let us stress that a dot, ".", has no special meaning (however, see Chapter 10 and Chapter 16 for some asterisks); `na.omit` is as good a name as `na_omit`, `naOmit`, `NAOMIT`, `naomit`, and `NaOmit`. Users coming from some other (C, C++, Java, Python, etc.) programming languages will need to habituate themselves to this convention.

R, as a dynamic language, allows for introducing new variables at any time. Moreover, existing names can be re-bound to new values. For instance:

```
(y <- c(1, 10, 100))  # bracketed expression - printing not suppressed
## [1]   1  10 100
x <- y
print(x)
## [1]   1  10 100
```

Now x refers to a verbatim copy of `y`.

Note Objects are automatically destroyed when there are no more names bound with them. In particular, by now the *garbage collector* should have got rid of the `1:3` vector begotten above (to which the name x was bound previously). See Section 14.4 for more details on memory management.

2.3 Vectorised Mathematical Functions

Mathematically, we will be denoting a given vector $\boldsymbol{x}$ of length n as $(x_1, x_2, \ldots, x_n)$. In other words, its i-th element is equal to x_i.

Let us review some ubiquitous operations in numerical computing.

2.3.1 `abs` and `sqrt`

R implements *vectorised* versions of the most popular mathematical functions, e.g., **`abs`** (absolute value, $|x|$) and **`sqrt`** (square root, $\sqrt{x}$).

```
abs(c(2, -1, 0, -3, NA_real_))
## [1]  2  1  0  3 NA
```

Here, *vectorised* means that instead of being defined to act on a single numeric value, the function of interest is applied on each element in a vector. The i-th resulting item is a transformed version of the i-th input. If an input is a missing value, the corresponding output will be marked as "don't know" as well.

Another example:

```
x <- c(4, 2, -1)
(y <- sqrt(x))
## Warning in sqrt(x): NaNs produced
## [1] 2.0000 1.4142    NaN
```

To attract our attention to the fact that computing the square root of a negative value yields a not-a-number, R generated an informative warning. A warning is not an error though: the result is being reckoned as usual.

Also the fact that the irrational $\sqrt{2}$ is *displayed*[4] as `1.4142` does not mean that it is such a crude approximation to 1.41421356237309504880168872420969807856967187537694 ...; it is only rounded when printing, for aesthetic reasons. In fact, in Section 3.2.3 we will point out that the computer's floating-point arithmetic allows for roughly 16 decimal digits precision (but we shall see that the devil is in the detail).

```
print(y, digits=16)  # display more significant figures
## [1] 2.000000000000000 1.414213562373095                NaN
```

2.3.2 Rounding

The following functions get rid of all or portions of fractional parts of numbers:

- **`floor`**`(x)` (rounds down to the nearest integer, denoted $\lfloor x \rfloor$),
- **`ceiling`**`(x)` (rounds up, denoted $\lceil x \rceil$),
- **`trunc`**`(x)` (rounds towards zero), and
- **`round`**`(x, digits=0)` (rounds to the nearest number with `digits` decimal digits).

For instance:

```
x <- c(7.0001, 6.9999, -4.3149, -5.19999, 123.4567, -765.4321, 0.5, 1.5, 2.5)
floor(x)
## [1]    7    6   -5   -6  123 -766    0    1    2
ceiling(x)
## [1]    8    7   -4   -5  124 -765    1    2    3
trunc(x)
## [1]    7    6   -4   -5  123 -765    0    1    2
```

Note If we call **`help`**`("round")`, we will read that its usage is like `round(x, digits=0)`, which means that the `digits` parameter is equipped with the *default value* of `0`. In other words, if rounding to `0` decimal digits is what we need, the second argument can be omitted.

```
round(x)  # the same as round(x, 0)
## [1]    7    7   -4   -5  123 -765    0    2    2
round(x, 1)
## [1]    7.0    7.0   -4.3   -5.2  123.5 -765.4    0.5    1.5    2.5
round(x, -2)
## [1]    0    0    0    0  100 -800    0    0    0
```

[4] There are a couple of settings in place that control the default behaviour of the **`print`** function; see `width`, `digits`, `max.print`, `OutDec`, `scipen`, etc. in **`help`**`("options")`.

2.3.3 Natural Exponential Function and Logarithm

Moreover:

- `exp(x)` outputs the natural exponential function, e^x, where the Euler's number $e \simeq 2.718$,
- `log(x, base=exp(1))` computes, by default, the natural logarithm of x, $\log_e x$ (which is most often denoted simply as $\log x$).

Recall that if $x = e^y$, then $\log_e x = y$, i.e., one is the inverse of the other.

```
log(c(0, 1, 2.7183, 7.3891, 20.0855))  # grows slowly
## [1] -Inf    0    1    2    3
exp(c(0, 1, 2, 3))                     # grows fast
## [1]  1.0000  2.7183  7.3891 20.0855
```

Note These functions enjoy a number of very useful identities and inequalities, including:

- $\log(x \cdot y) = \log x + \log y$,
- $\log(x^y) = y \log x$,
- $e^{x+y} = e^x \cdot e^y$.

For more properties like these, take a glance at Chapter 4 of the freely available handbook [39].

For the logarithm to a different base, say $\log_{10} x$, we can call:

```
log(c(0, 1, 10, 100, 1000, 1e10), 10)  # or log(..., base=10)
## [1] -Inf    0    1    2    3   10
```

Note that if $\log_b x = y$, then $x = b^y$, for any $1 \neq b > 0$.

Note Commonly, a logarithmic scale is used for variables that grow rapidly when expressed as functions of each other; see Figure 2.2.

```
x <- seq(0, 10, length.out=1001)
par(mfrow=c(1, 2))  # two plots in one figure (1 row, 2 columns)
plot(x, exp(x), type="l")
plot(x, exp(x), type="l", log="y")  # log-scale on the y-axis
```

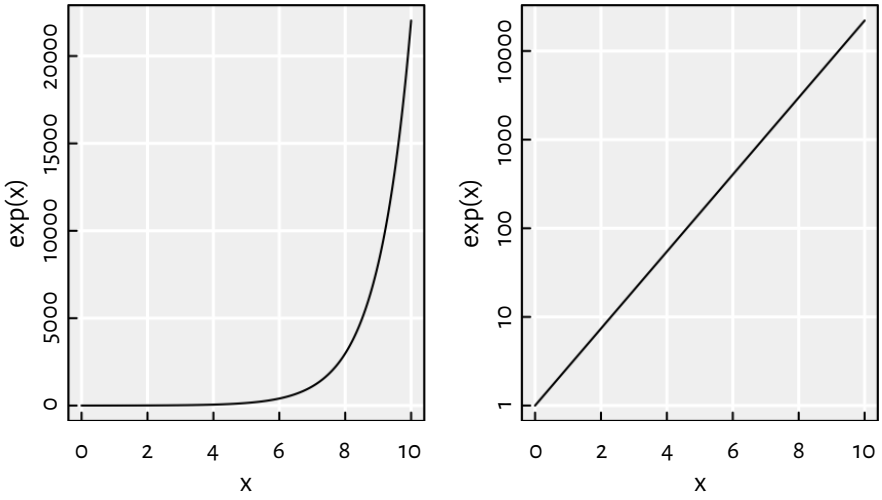

Figure 2.2: Linear- vs log-scale on the y-axis

Note that e^x on the log-scale is just a straight line. Also, keep in mind that such a transformation of the axes can only be applied in the case of values strictly greater than 0.

2.3.4 Probability Distributions (*)

It should come as no surprise that R offers an extensive support for many univariate probability distribution families, including:

- continuous distributions, which take values being arbitrary real numbers (over the whole possible range or in some interval):
  - `*unif` (uniform),
  - `*norm` (normal),
  - `*exp` (exponential),
  - `*gamma` (gamma, Γ),
  - `*beta` (beta, B),
  - `*lnorm` (log-normal),
  - `*t` (Student),
  - `*cauchy` (Cauchy–Lorentz),
  - `*chisq` (chi-squared, χ^2),
  - `*f` (Snedecor–Fisher),

  - `*weibull` (Weibull);

  with the prefix "`*`" being one of:

  - "`d`" (probability density function, PDF),
  - "`p`" (cumulative distribution function, CDF; or survival function, SF),
  - "`q`" (quantile function, being the inverse of the CDF),
  - "`r`" (generation of random deviates; already mentioned);

- discrete distributions, i.e., those whose possible outcomes can be easily enumerated (e.g., some integers).
  - `*binom` (binomial),
  - `*geom` (geometric),
  - `*pois` (Poisson),
  - `*hyper` (hypergeometric),
  - `*nbinom` (negative binomial);

  here, prefixes "`p`" and "`r`" have the same meaning as above, however:

  - "`d`" now gives the probability *mass* function (PMF),
  - "`q`" yields the quantile function, but one that is defined as a *generalised* inverse of the CDF.

Each distribution is characterised by a set of underlying parameters. For instance, a normal distribution $\mathrm{N}(\mu, \sigma)$ can be pinpointed by setting its expected value $\mu \in \mathbb{R}$ and standard deviation $\sigma > 0$. In R, these two have been named `mean` and `sd`, respectively; see **`help`**`("dnorm")`.

Note The parametrisations assumed in R can be subtly different from what we know from statistical textbooks or probability courses. For example, the normal distribution can be parameterised based on either standard deviation or variance, and the exponential distribution can be defined via its expected value or the reciprocal thereof. We thus advise the reader to study carefully the documentation of **`help`**`("dnorm")`, **`help`**`("dunif")`, **`help`**`("dexp")`, **`help`**`("dbinom")`, and the like.

It is also worth to know the typical use cases of each of the distribution listed, e.g., a Poisson distribution can describe the probability of observing the number of independent events in a fixed time interval (e.g., the number of users downloading a copy of R from CRAN per hour), and an exponential distribution can model the time between such events; compare [17].

Exercise 2.4 *A call to* ***`hist`****`(x)` draws a histogram, which can serve as an estimator of the underlying continuous probability density function of a given sample; see Figure 2.3 for an illustration.*

```
par(mfrow=c(1, 2))  # 2 plots in 1 figure
# Uniform U(0, 1)
hist(runif(10000, 0, 1), col="white", probability=TRUE, main="")
x <- seq(0, 1, length.out=101)
lines(x, dunif(x, 0, 1), lwd=2)  # draw the true density function (PDF)
# Normal N(0, 1)
hist(rnorm(10000, 0, 1), col="white", probability=TRUE, main="")
x <- seq(-4, 4, length.out=101)
lines(x, dnorm(x, 0, 1), lwd=2)  # draw the PDF
```

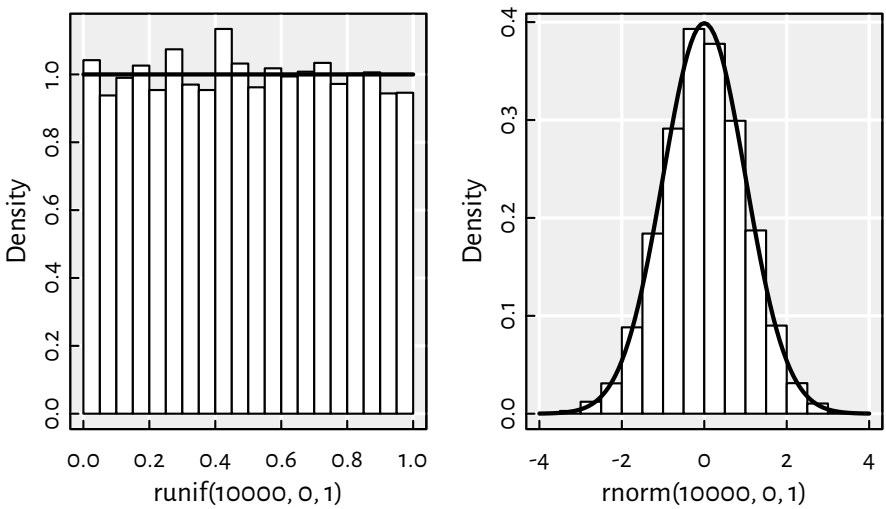

Figure 2.3: Example histograms of some pseudorandom samples and the true underlying probability density functions: the uniform distribution on the unit interval (left) and the standard normal distribution (right)

Draw a histogram of some random samples of different sizes `n` from the following distributions:

- `rnorm(n, μ, σ)` — *normal* $\mathrm{N}(\mu, \sigma)$ *with expected values* $\mu \in \{-1, 0, 5\}$ *(i.e.,* μ *being equal to either* -1, 0, *or* 5*; read "*$\in$*" as "belongs to the given set" or "in") and standard deviations* $\sigma \in \{0.5, 1, 5\}$*;*
- `runif(n, a, b)` — *uniform* $\mathrm{U}(a, b)$ *on the interval* (a, b) *with* $a = 0$ *and* $b = 1$ *as well as* $a = -1$ *and* $b = 1$*;*
- `rbeta(n, α, β)` — *beta* $\mathrm{B}(\alpha, \beta)$ *with* $\alpha, \beta \in \{0.5, 1, 2\}$*;*
- `rexp(n, λ)` — *exponential* $\mathrm{E}(\lambda)$ *with rates* $\lambda \in \{0.5, 1, 10\}$*;*

Moreover, read about and play with the `breaks`, `main`, `xlab`, `ylab`, `xlim`, `ylim`, and `col` parameters; see **`help("hist")`**.

Example 2.5 *We roll a six-sided dice 12 times. Let* C *be a random variable denoting the number*

of cases where the "1" face is thrown. C follows a binomial distribution $\mathrm{Bin}(n, p)$ *with parameters* $n = 12$ *(the number of Bernoulli trials) and* $p = 1/6$ *(the probability of* success *in a single roll).*

The probabilities that the number of "1"s rolled will be equal to 0, 1, ..., 4, i.e., $P(C = 0), P(C = 1), ..., P(C = 4)$*, respectively, can be computed based on the probability mass function (**dbinom**):*

```
dbinom(0:4, 12, 1/6)  # probability mass function at 5 different points
## [1] 0.112157 0.269176 0.296094 0.197396 0.088828
```

On the other hand, the probability that we throw more than three "1"s, $P(C > 3) = 1 - P(C \leq 3)$*, can be determined by means of the cumulative distribution function (**pbinom**) or survival function (`pbinom(..., lower.tail=FALSE)`):*

```
1-pbinom(3, 12, 1/6)  # pbinom(3, 12, 1/6, lower.tail=FALSE)
## [1] 0.12518
```

The smallest c *such that* $P(C \leq c) \geq 0.95$ *can be computed based on the quantile function:*

```
qbinom(0.5, 12, 1/6)
## [1] 2
pbinom(3:4, 12, 1/6)   # for comparison - 0.95 is in-between
## [1] 0.87482 0.96365
```

In other words, at least 95% of the time we will be observing no more than 4 successes.

Also here are some pseudorandom realisations of C *– the number of "1"s in 30 simulations of 12 independent dice rolls each:*

```
rbinom(30, 12, 1/6)
##  [1] 1 3 2 4 4 0 2 4 2 2 4 2 3 2 0 4 1 0 1 4 4 3 2 6 2 3 2 2 1 1
```

2.3.5 Special Functions (*)

Within mathematical formulae and across assorted application areas, certain functions appear more frequently than others. Hence, for the sake of notational brevity and computational precision, many of them have been assigned special names. For instance, the following may be mentioned in the definitions related to some of the probability distributions listed above:

- **`gamma(x)`** for $x > 0$ computes $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt$,
- **`beta(a, b)`** for $a, b > 0$ yields $B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} = \int_0^1 t^{a-1}(1-t)^{b-1}\, dt$.

Why do we have **`beta`** if it is merely a mix of **`gamma`**s? A specific, tailored function should be faster and more precise than its DIY version; its underlying implementation does not have to involve any calls to **`gamma`** at all.

```
beta(0.25, 250)  # okay
## [1] 0.91213
gamma(0.25)*gamma(250)/gamma(250.25)  # not okay
## [1] NaN
```

The Γ function grows so rapidly that already `gamma(172)` yields `Inf`. It is due to the fact that a computer's arithmetic is not infinitely precise; compare Section 3.2.3.

Special functions are plentiful; see the open-access [39] for one of the most definitive references (and also [2] for its predecessor). R package **`gsl`** [28] provides a vectorised interface to the famous GNU GSL [23] library, which implements many of them.

Exercise 2.6 *The Pochhammer symbol, $(a)_x = \Gamma(a+x)/\Gamma(a)$, can be computed via a call to* ***`gsl::poch(a, x)`*** *(i.e., the* ***`poch`*** *function from the* ***`gsl`*** *package; see Section 7.3.1):*

```
# call install.packages("gsl") first
library("gsl")  # load the package
poch(10, 3:6)   # calls gsl_sf_poch() from GNU GSL
## [1]    1320   17160  240240 3603600
```

Read the documentation of the corresponding ***`gsl_sf_poch`*** *function in the GNU GSL manual available here*[5]*.*

And since you are there, do not hesitate to go through the list of all the other functions, including those related to statistics, permutations, combinations, and so forth.

Many functions also have their logarithm-of versions; see, e.g., **`lgamma`** and **`lbeta`**. Also, for instance, **`dnorm`** and **`dbeta`** has the `log` parameter. Its classical use case is the (numerical) maximum likelihood estimation, which involves the sums of the *logarithms* of densities.

2.4 Arithmetic Operations

2.4.1 Vectorised Arithmetic Operators

R features the following arithmetic operators:

- `` `+` `` (addition) and `` `-` `` (subtraction),
- `` `*` `` (multiplication) and `` `/` `` (division),
- `` `%/%` `` (integer division) and `` `%%` `` (modulo, division remainder),
- `` `^` `` (exponentiation; synonym: `` `**` ``).

They are all *vectorised*: they take two vectors on input and yield another vector in result.

[5] https://www.gnu.org/software/gsl/doc/html/

```
c(1, 2, 3) * c(10, 100, 1000)
## [1]   10  200 3000
```

We note that the multiplication was performed in an *elementwise* fashion: the 1st element in the left vector was multiplied by the *corresponding* element in the right vector and the result has been stored in the 1st element of the output, then the 2nd element in the left… all right, we get the point.

Other operators are vectorised in the same manner:

```
0:10 + seq(0, 1, 0.1)
##  [1]  0.0  1.1  2.2  3.3  4.4  5.5  6.6  7.7  8.8  9.9 11.0
0:7 / rep(3, length.out=8)   # division by 3
## [1] 0.00000 0.33333 0.66667 1.00000 1.33333 1.66667 2.00000 2.33333
0:7 %/% rep(3, length.out=8) # integer division
## [1] 0 0 0 1 1 1 2 2
0:7 %% rep(3, length.out=8)  # division remainder
## [1] 0 1 2 0 1 2 0 1
```

Note that operations involving missing values also yield `NA`s:

```
c(1, NA_real_, 3, NA_real_) + c(NA_real_, 2, 3, NA_real_)
## [1] NA NA  6 NA
```

2.4.2 Recycling Rule

Some of the above statements can be written more concisely. When the operands are of different lengths, the shorter one is *recycled* (think: `rep(y, length.out=length(x))`) as many times as necessary.

```
0:7 / 3
## [1] 0.00000 0.33333 0.66667 1.00000 1.33333 1.66667 2.00000 2.33333
1:10 * c(-1, 1)
##  [1] -1  2 -3  4 -5  6 -7  8 -9 10
2 ^ (0:10)
##  [1]    1    2    4    8   16   32   64  128  256  512 1024
```

We call this the *recycling rule*.

If an operand cannot be recycled in its entirety, a warning[6] is generated, but the output is still available.

[6] A few built-in functions do not warn at all when incomplete recycling is performed (e.g., **`paste`**) or can even give an error (e.g., **`as.data.frame.list`**). Consider this inconsistency an annoying bug and hope it will be fixed in the next decade or so.

```
c(1, 10, 100) * 1:8
## Warning in c(1, 10, 100) * 1:8: longer object length is not a multiple of
## shorter object length
## [1]   1  20 300   4  50 600   7  80
```

Note Some functions are also deeply vectorised, i.e., with respect to multiple arguments. For example,

```
runif(3, c(10, 20, 30), c(11, 22, 33))
## [1] 10.288 21.577 31.227
```

generates three random numbers uniformly distributed over the intervals $(10, 11)$, $(20, 22)$, and $(30, 33)$, respectively.

Also, **`pmin`** and **`pmax`** return the *parallel* minimum and maximum of the corresponding elements of the input vectors:

```
pmin(c(1, 2, 3, 4), c(4, 2, 3, 1))
## [1] 1 2 3 1
pmin(3, 1:5)
## [1] 1 2 3 3 3
pmax(0, pmin(1, c(0.25, -2, 5, -0.5, 0, 1.3, 0.99)))  # clipping to [0, 1]
## [1] 0.25 0.00 1.00 0.00 0.00 1.00 0.99
```

Note Vectorisation and the recycling rule are perhaps most useful when applying binary operators on sequences of identical lengths or when performing vector-scalar (i.e., a sequence vs a single value) operations. However, there is much more: schemes like "every k-th element" appear in Taylor series expansions (multiply by **`c`**`(-1, 1)`), k-fold cross validation, etc.; see also Section 11.3.4 for use cases in matrix/tensor processing.

2.4.3 Operator Precedence

Apart from the seven binary arithmetic operators, other noteworthy, already mentioned ones include: the unary `` `-` `` (change of sign), `` `:` `` (sequence generation), and `` `<-` `` (assignment).

Expressions involving multiple operations need a set of rules governing the order of computations (unless we enforce it using round brackets). We have said that "`-1:10`" means "`(-1):10`" rather than "`-(1:10)`". But what about, say, "`1+1+1+1+1*0`" or "`3*2^0:5+10`"?

Let us list the aforementioned operators in their order of precedence, from the least to the most binding (see also **`help`**`("Syntax")`):

1. `<-` (right-to-left),
2. `+` and `-`,
3. `*` and `/`,
4. `%%` and `%/%`,
5. `:`,
6. `+` and `-` (unary),
7. `^` (right-to-left).

Hence, "`-2^2/3+3*4`" means "`((-(2^2))/3)+(3*4)`" and not, for example, `-((2^(2/(3+3)))*4)`.

Note that `+` and `-`, `*` and `/`, as well as `%%` and `%/%` have the same priority. Expressions involving a series of operations in the same group, are evaluated left-to-right, with the exception of `^` and `<-`, which are performed from right to left.

Therefore:

- "`2*3/4*5`" is equivalent to "`((2*3)/4)*5`",
- "`2^3^4`" is the same as "`2^(3^4)`" (which, mathematically, we would write as $2^{3^4} = 2^{81}$),
- "`x <- y <- 4*3%%8/2`" binds both `y` and `x` with `6` and not `x` with the previous value of `y`.

And let us remember: when in doubt, we can always bracket a subexpression to make sure it is executed in the intended order (which can also increase readability of the code).

2.4.4 Accumulating

The `+` and `*` operators as well as the **`pmin`** and **`pmax`** functions implement element-wise operations that are applied on the corresponding elements taken from two given vectors:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \\ \vdots \\ x_n + y_n \end{pmatrix}.$$

However, we can also scan through all the values in a *single* vector and combine the successive elements that we inspect using the corresponding operation:

- **`cumsum`**`(x)` gives the cumulative sum of the elements in a vector,
- **`cumprod`**`(x)` computes the cumulative product,
- **`cummin`**`(x)` yields the cumulative minimum,

- **cummax**(x) generates the cumulative maximum.

The i-th element in the output vector will consist of the sum/product/min/max of the first i inputs:

$$\operatorname{cumsum}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_1 + x_2 \\ x_1 + x_2 + x_3 \\ \vdots \\ x_1 + x_2 + x_3 + \cdots + x_n \end{pmatrix}.$$

For example:

```
cumsum(1:8)
## [1]  1  3  6 10 15 21 28 36
cumprod(1:8)
## [1]     1     2     6    24   120   720  5040 40320
cummin(c(3, 2, 4, 5, 1, 6, 0))
## [1] 3 2 2 2 1 1 0
cummax(c(3, 2, 4, 5, 1, 6, 0))
## [1] 3 3 4 5 5 6 6
```

If we are interested only in the last cumulant, summarising all the inputs, we have the following functions at our disposal:

- **sum**(x) computes the sum of elements in a vector, $\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$,
- **prod**(x) outputs the product of all elements, $\prod_{i=1}^{n} x_i = x_1 x_2 \cdots x_n$,
- **min**(x) computes the minimum,
- **max**(x) reckons the greatest value.

For example:

```
sum(1:8)
## [1] 36
prod(1:8)
## [1] 40320
min(c(3, 2, 4, 5, 1, 6, 0))
## [1] 0
max(c(3, 2, 4, 5, 1, 6, 0))
## [1] 6
```

Note In Chapter 7, we will discuss the **Reduce** function, which generalises the above by allowing any binary operation to be propagated over a given vector.

Example 2.7 ***diff*** *can be considered an inverse to* ***cumsum****: it computes the iterative difference.*

Namely, it subtracts the first two elements, then the 2nd from the 3rd one, the 3rd from the 4th, and so on. In other words, ***diff****(x) gives* ***y*** *such that* $y_i = x_{i+1} - x_i$.

```
x <- c(-2, 3, 6, 2, 15)
diff(x)
## [1]  5  3 -4 13
cumsum(diff(x))
## [1]  5  8  4 17
cumsum(c(-2, diff(x)))  # recreates x
## [1] -2  3  6  2 15
```

Thanks to ***diff****, we can compute the daily changes to the EUR/AUD forex rates; see Figure 2.4.*

```
aud <- scan(paste0("https://github.com/gagolews/teaching-data/raw/",
    "master/marek/euraud-20200101-20200630.csv"), comment.char="#")
aud_all <- na.omit(aud)  # remove all missing values
plot(diff(aud_all), type="s", ylab="Daily change [EUR/AUD]")
abline(h=0, lty="dotted")  # draw a horizontal line at y=0
```

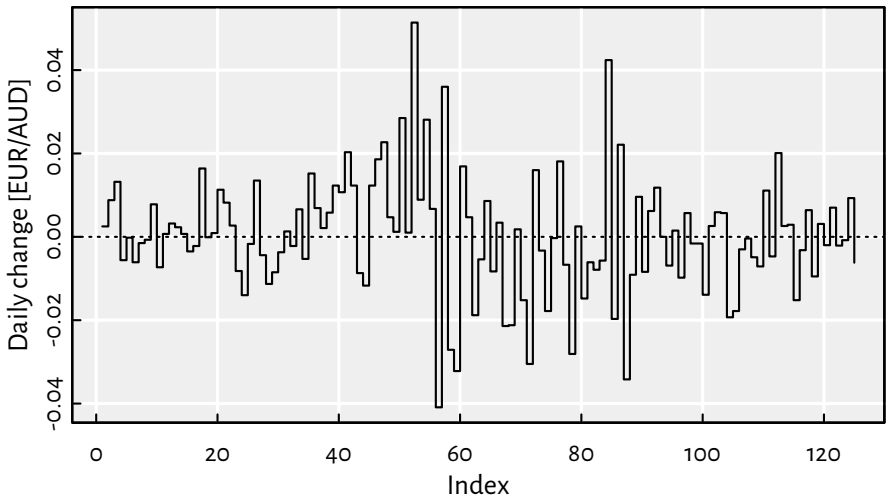

Figure 2.4: Iterative differences of the exchange rates (non-missing values only)

2.4.5 Aggregating

The above functions form the basis for some popular summary statistics[7] (sample aggregates), such as:

- **mean**(x) gives the arithmetic mean, **sum**(x)/**length**(x),

[7] Actually, **var** and **median**, amongst others, are defined by the **stats** package, but this one is automatically loaded by default, so let us not make a fuss about it now.

- **var**(x) yields the (unbiased) sample variance, **sum**((x-**mean**(x))^2)/(**length**(x)-1),
- **sd**(x) is the standard deviation, **sqrt**(**var**(x)),
- **median**(x) computes the sample median, i.e., the middle value in the sorted version of x.

For instance[8]:

```
x <- runif(1000)
c(min(x), mean(x), median(x), max(x), sd(x))
## [1] 0.00046535 0.49727780 0.48995025 0.99940453 0.28748391
```

Exercise 2.8 *Let* $\boldsymbol{x}$ *be any vector of length* n *with positive elements. Compute its geometric and harmonic mean, which are given by, respectively,*

$$\sqrt[n]{\prod_{i=1}^{n} x_i} = e^{\frac{1}{n}\sum_{i=1}^{n}\log x_i} \quad \text{and} \quad \frac{n}{\sum_{i=1}^{n}\frac{1}{x_i}}.$$

When solving exercises like this one, it does not really matter what data you apply these functions on (see, however, Section 9.3.4 for discussion). We are being abstract in the sense that the $\boldsymbol{x}$ *vector can be anything: from the one that features very accurate financial predictions that will help minimise inequity and make this world less miserable, through the data you have been collecting for the last the years in relation to your definitely-super-important PhD research, whatever your company asked you to crunch today, to something related to your hobby project that you enjoy doing after hours. Therefore, just test the above on something like "*`x <- runif(10)`*", and move on.*

All the aforementioned functions return a missing value if any of the input elements is unavailable. Luckily, they are equipped with the `na.rm` parameter on behalf of which we can request the removal of `NA`s.

```
aud <- scan(paste0("https://github.com/gagolews/teaching-data/raw/",
    "master/marek/euraud-20200101-20200630.csv"), comment.char="#")
c(min(aud), mean(aud), max(aud))
## [1] NA NA NA
c(min(aud, na.rm=TRUE), mean(aud, na.rm=TRUE), max(aud, na.rm=TRUE))
## [1] 1.6006 1.6775 1.8635
```

Note In the documentation, we read that the usage of some of the aforementioned functions is like **sum**(..., na.rm=FALSE). **prod**, **min**, and **max** are defined similarly. They accept any number of input vectors, each of them can be of arbitrary length. Therefore, **min**(1, 2, 3), **min**(**c**(1,2,3)) as well as **min**(**c**(1,2),3) all return the same result.

However, we can also read that we have **mean**(x, trim=0, na.rm=FALSE, ...). This

[8] Note that **min**, **median**, and **max** is a special case of **quantile**, which we will discuss much further, namely, in Section 4.4.3. This is because it returns a named vector.

time, only one vector can be aggregated and any further arguments (except `trim` and `na.rm`) are ignored.

The extra flexibility (which we do not have to rely upon, ever) of the former group is due their being associative operations: it holds, e.g., $(2+3)+4 = 2+(3+4)$. Hence, the operations can be performed in any order, in any groups.

Also note that they are more primitive operations: it is **`mean`** that is based on **`sum`**, not vice versa.

2.5 Exercises

Exercise 2.9 *Answer the following questions:*

- *What is the meaning of the dot-dot-dot parameter in the definition of the* **`c`** *function?*
- *We say that the* **`round`** *function is* vectorised*: what does that mean?*
- *What do we mean by saying that multiplication operates element-by-element?*
- *How does the recycling rule work when applying* `` `+` ``*?*
- *How to (and why) set the seed of the pseudorandom number generator?*
- *What is the difference between* `NA_real_` *and* `NaN`*?*
- *How are default arguments specified in the manual of, e.g., the* **`round`** *function?*
- *Is a call to* **`rep`**`(times=4, x=1:5)` *equivalent to* **`rep`**`(4, 1:5)`*?*
- *List a few ways to generate a sequence like (-1, -0.75, -0.5, ..., 0.75, 1).*
- *Is "*`-3:5`*" the same as "*`-(3:5)`*"? What about the precedence of operators in expressions such as "*`2^3/4*5^6`*", "*`5*6+4/17%%8`*", and "*`1+-2^3:4`*"?*
- *If* `x` *is a numeric vector of length* n *(for some* $n \geq 0$*), how many values will* **`sample`**`(x)` *output?*
- *Does* **`scan`** *support reading directly from compressed archives, e.g.,* `.csv.gz` *files?*

When in doubt, refer back to the material discussed in this chapter and/or the R manual.

Exercise 2.10 *The following code generates an example graph of arcsine and arccosine, whose preparation – thanks to vectorisation – is quite straightforward.*

```
x <- seq(-1, 1, length.out=11)  # increase length.out for a smoother curve
plot(x, asin(x),                # asin() computed for 11 points
    type="l",                   # lines
    ylim=c(-pi/2, pi),          # y axis limits like c(y_min, y_max)
    ylab="asin(x), acos(x)")    # y axis label
```

(continues on next page)

(continued from previous page)

```
lines(x, acos(x), col="red", lty="dashed")  # adds to the current plot
legend("topright", c("asin(x)", "acos(x)"),
    lty=c("solid", "dashed"), col=c("black", "red"), bg="white")
```

Inspired by the above, plot the following functions: $|\sin x^2|$, $|\sin |x||$, $\sqrt{\lfloor x \rfloor}$, *and* $1/(1+e^{-x})$. *Recall that the documentation of* ***plot*** *can be viewed by calling* ***help("plot.default")***.

Exercise 2.11 *It can be shown that:*

$$4\sum_{i=1}^{n}\frac{(-1)^{i+1}}{2i-1} = 4\left(\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

slowly converges to π *as* n *approaches* ∞. *Compute the above for* $n = 1{,}000{,}000$ *and* $n = 1{,}000{,}000{,}000$ *using the vectorised functions and operators discussed in this chapter, making use of the recycling rule as much as possible.*

Exercise 2.12 *Let* x *and* y *be two vectors of identical lengths* n, *say:*

```
x <- rnorm(100)
y <- 2*x+10+rnorm(100, 0, 0.5)
```

Compute the Pearson linear correlation coefficient given by:

$$r = \frac{\sum_{i=1}^{n}\left(x_i-\frac{1}{n}\sum_{j=1}^{n}x_j\right)\left(y_i-\frac{1}{n}\sum_{j=1}^{n}y_j\right)}{\sqrt{\sum_{i=1}^{n}\left(x_i-\frac{1}{n}\sum_{j=1}^{n}x_j\right)^2}\sqrt{\sum_{i=1}^{n}\left(y_i-\frac{1}{n}\sum_{j=1}^{n}y_j\right)^2}}.$$

To make sure you have come up with a correct implementation, compare your result to a call to the built-in ***cor(x, y)***.

Exercise 2.13 *(*) Look up on the internet an R package that features functions to compute the 5-day moving (rolling) average and median of a given vector. Apply them on the EUR/AUD currency exchange data and plot thus obtained smoothened versions of the time series.*

Exercise 2.14 *(**) Compute the* k*-moving average using a call to* ***convolve(..., type="filter")***.

In the next chapter we will study operations that involve logical values.

3

Logical Vectors

There are three logical constants in R. Wait... how many?

3.1 Creating Logical Vectors

R defines three logical constants: `TRUE`, `FALSE`, and `NA` – meant to represent "yes", "no", and "???", respectively. Each of them, when instantiated, is an atomic vector of length one.

Some of the functions we introduced in the previous chapter can be used to generate logical vectors as well:

```
c(TRUE, FALSE, FALSE, NA, TRUE, FALSE)
## [1]  TRUE FALSE FALSE    NA  TRUE FALSE
rep(c(TRUE, FALSE, NA), each=2)
## [1]  TRUE  TRUE FALSE FALSE    NA    NA
sample(c(TRUE, FALSE), 10, replace=TRUE, prob=c(0.8, 0.2))
##  [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
```

Note "`T`" is a synonym for `TRUE` and "`F`" stands for `FALSE`. However, these are not reserved keywords and can be re-assigned any other values. Therefore, we advise against relying on them and hence we will never use them throughout the course of this course.

Also note that the logical missing value is spelled simply as "`NA`" and not "`NA_logical_`". The fact that both the logical "`NA`" and the numeric `"NA_real_"` are, for the sake of our mental well-being, both *printed* as `"NA"` on the R console, does not mean they are identical; see Section 4.1 for discussion.

3.2 Comparing Elements

3.2.1 Vectorised Comparison Operators

Logical vectors frequently come into being as results of various *testing* activities.

In particular, the binary operators:

- `<` (less than),
- `<=` (less than or equal),
- `>` (greater than),
- `>=` (greater than or equal)
- `==` (equal),
- `!=` (not equal),

compare the *corresponding* elements of two numeric vectors and output a logical vector.

```
1 < 3
## [1] TRUE
c(1, 2, 3, 4) == c(2, 2, 3, 8)
## [1] FALSE  TRUE  TRUE FALSE
1:10 <= 10:1
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Thus, they operate in an elementwise manner. Moreover, the recycling rule is applied if necessary:

```
3 < 1:5  # c(3, 3, 3, 3, 3) < c(1, 2, 3, 4, 5)
## [1] FALSE FALSE FALSE  TRUE  TRUE
c(1, 4) == 1:4  # c(1, 4, 1, 4) == c(1, 2, 3, 4)
## [1]  TRUE FALSE FALSE  TRUE
```

Therefore, we can say that they are vectorised in the same manner as the arithmetic operators `+`, `*`, etc.; compare Section 2.4.1.

3.2.2 Testing for `NA`, `NaN`, and `Inf`

Comparisons against missing values and not-numbers yield `NA`s. Therefore, instead of the incorrect `x == NA_reals_` or `x == NaN`, testing for missingness should rather be performed via a call to the vectorised `is.na` function.

```
is.na(c(NA_real_, Inf, -Inf, NaN, -1, 0, 1))
## [1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
is.nan(c(NA_real_, Inf, -Inf, NaN, -1, 0, 1))
```

(continues on next page)

(continued from previous page)

```
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
is.na(c(TRUE, FALSE, NA, TRUE))  # works for logical vectors too
## [1] FALSE FALSE  TRUE FALSE
```

Moreover, **is.finite** is noteworthy, because it returns FALSE on Infs, NA_real_s and NaNs.

```
is.finite(c(NA_real_, Inf, -Inf, NaN, -1, 0, 1))
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

See also the more specific **is.nan** and **is.infinite**.

3.2.3 Dealing with Floating Point Round-Off Errors (*)

In mathematics, real numbers are merely an idealisation. In practice, however, it is impossible to store them with infinite precision (think $\pi = 3.14159265358979323846264338...$) computer memory is limited and our time is precious.

Therefore, a widely agreed upon consensus had to be reached. In R, we rely on the so-called *double-precision floating point format*. *Floating point* means that the numbers can be both small (close to zero) and large: $\pm 2.23 \times 10^{-308}$ and $\pm 1.79 \times 10^{308}$ are both acceptable.

Note

```
2.23e-308 == 0.00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000000000223

1.79e308 == 17900000000000000000000000000000000000000000000000000000000000
                 00000000000000000000000000000000000000000000000000000000
                 00000000000000000000000000000000000000000000000000000000
                 00000000000000000000000000000000000000000000000000000000
                 00000000000000000000000000000000000000000000000000000000
                 00000000000000000000000000000000000000000000000000000000
```

These two are quite distant from each other.

Every numeric value takes 8 bytes (or equivalently 64 bits) of memory. We are, however, able to store only *about* 15-17 decimal digits:

```
print(0.123456789012345678901234567890123456789012345678901234, digits=22)  # 22 is max
## [1] 0.12345678901234566773699
```

which limits the precision of our computations. The *about* part is – unfortunately – due to the numbers' being written in the computer-friendly *binary*, not human-aligned *decimal*, base. This can lead to some unexpected outcomes.

In particular:

- `0.1` cannot be represented exactly, because it cannot be written as a finite series of reciprocals of powers of 2 (it holds $0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + ...$). This leads to surprising results such as:

  ```
  0.1 + 0.1 + 0.1 == 0.3
  ## [1] FALSE
  ```

  Despite the fact that what follows does not *show* anything suspicious:

  ```
  c(0.1, 0.1 + 0.1 + 0.1, 0.3)
  ## [1] 0.1 0.3 0.3
  ```

  Printing involves rounding, hence, in the above context, is misleading. Above, we have something more like:

  ```
  print(c(0.1, 0.1 + 0.1 + 0.1, 0.3), digits=22)
  ## [1] 0.1000000000000000055511 0.3000000000000000444089
  ## [3] 0.2999999999999999888978
  ```

- All integers between -2^{53} and 2^{53} all stored exactly – this is good news. However, the next integer is beyond the representable range:

  ```
  2^53 + 1 == 2^53
  ## [1] TRUE
  ```

- The above suggests that, more generally, the order of operations may matter, in particular, the associativity property may be violated when dealing with numbers of different orders of magnitude:

  ```
  2^53 + 2^-53 - 2^53 - 2^-53  # should be == 0.0
  ## [1] -1.1102e-16
  ```

- Some numbers may just be just too large, too small, or too close to zero to be represented exactly:

  ```
  c(sum(2^((1023-52):1023)), sum(2^((1023-53):1023)))
  ## [1] 1.7977e+308         Inf
  c(2^(-1022-52), 2^(-1022-53))
  ## [1] 4.9407e-324  0.0000e+00
  ```

Important The double-precision floating point format (IEEE 754) is not specific to R: it is used by most other computing environments, including Python and C++.

For discussion, see [27, 30, 34] ([26] can be of particular interest to the general statistical/data analysis audience).

Can we do anything about these issues?

First, when dealing with integers of *reasonable* order of magnitude (a frequent case where we are dealing various resource or case IDs in our datasets), rest assured that we are safe: their comparison, addition, subtraction, and multiplication is always precise.

In all other cases (including applying other operations on integers, e.g., division or **`sqrt`**), we need to be very careful with comparisons, especially involving testing for equality, **`==`**.

The sole fact that $\sin \pi = 0$, mathematically speaking, does not mean that we should expect that:

```
sin(pi) == 0
## [1] FALSE
```

Instead, they are so close to each other that we can *treat the difference between them as negligible*. Thus, in practice, instead of testing if $x = y$, we will be considering:

- $|x - y|$ (absolute error) or
- $\frac{|x-y|}{|y|}$ (relative error; which takes the order of magnitude of the numbers into account but obviously cannot be applied if y is very close of 0),

and determining if these are less than some assumed error margin, $\varepsilon > 0$, say, 10^{-8} or 2^{-26}.

For example:

```
abs(sin(pi) - 0) < 2^-26
## [1] TRUE
```

Note Note that rounding can sometimes have a similar effect as testing for almost-equality in terms of the absolute error.

```
round(sin(pi), 8) == 0
## [1] TRUE
```

Important Our recommendations are valid for the most popular applications of R,

i.e., statistical and, more generally, scientific computing[1]. The datasets we handle on a daily basis do not represent accurate measurements themselves, bah, the World itself is far from ideal, therefore we do not have to lose sleep over our not being able to precisely pinpoint the *exact* solution.

3.3 Logical Operations

3.3.1 Vectorised Logical Operators

The comparison operators such as `==` and `>` accept only *two* arguments. Their chaining is forbidden; a test which we would mathematically write as $0 \leq x \leq 1$ (or $x \in [0, 1]$) cannot be expressed as "`0<=x<=1`" in R.

Therefore, we need a way to combine two logical conditions so as to be able to state that "$x \geq 0$ *and, at the same time*, $x \leq 1$".

In such situations, the following logical operators and functions come in handy:

- `!` (not, negation; unary),
- `&` (and, conjunction; are both predicates true?),
- `|` (or, alternation; is at least one true?),
- **xor** (exclusive-or, exclusive disjunction, either-or; is one and only one of the predicates true?).

They again act elementwisely and implement the recycling rule if necessary (and applicable).

```
x <- c(-10, -1, -0.25, 0, 0.5, 1, 5, 100)
(x >= 0) & (x <= 1)
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
(x < 0) | (x > 1)
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
!((x < 0) | (x > 1))
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
xor(x >= -1, x <= 1)
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

[1] However, in financial applications, we should rather rely on base-10 numbers (compare the 0.1 problem above). Also note that there exist some libraries implementing higher precision floating-point numbers or even interval arithmetic that keeps track of error propagation operation chains.

Important The vectorised `&` and `|` operators should not be confused with their scalar, short-circuit counterparts, `&&` and `||`, which we discuss in Section 8.1.4.

3.3.2 Operator Precedence Revisited

The operators introduced in this chapter have lower precedence than the arithmetic ones. In particular, the binary `+` and `-`. Calling `help("Syntax")` reveals that we can extend our listing from Section 2.4.3 as follows:

1. `<-` *(right-to-left; least binding)*,
2. `|`,
3. `&`,
4. `!` (unary),
5. `<`, `>`, `<=`, `>=`, `==`, and `!=`,
6. `+` *and* `-`,
7. `*` *and* `/`,
8. ...

3.3.3 Dealing with Missingness

Operations involving missing values follow the principles of the Łukasiewicz's three-valued logic, which is based on common sense. For instance, "`NA | TRUE`" is `TRUE`, because *or* needs at least one argument to be `TRUE` to generate such a result. On the other hand, "`NA | FALSE`" is `NA`, because the result would be different depending on what we substituted `NA` for.

Let us take a moment to contemplate the operations' *truth tables* for all the possible combinations of inputs:

```
u <- c(TRUE, FALSE, NA,  TRUE,  FALSE, NA,    TRUE, FALSE, NA)
v <- c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, NA,   NA,    NA)
!u
## [1] FALSE  TRUE    NA FALSE  TRUE    NA FALSE  TRUE    NA
u & v
## [1]  TRUE FALSE    NA FALSE FALSE FALSE    NA FALSE    NA
u | v
## [1]  TRUE  TRUE  TRUE  TRUE FALSE    NA  TRUE    NA    NA
xor(u, v)
## [1] FALSE  TRUE    NA  TRUE FALSE    NA    NA    NA    NA
```

3.3.4 Aggregating with `all`, `any`, and `sum`

Just like in the case of numeric vectors, we can summarise the contents of logical sequences.

`all` tests whether every element in a logical vector is equal to `TRUE` and **`any`** determines if there exists an element that is `TRUE`.

```
x <- runif(10000)
all(x <= 0.2)  # are all values in x <= 0.2?
## [1] FALSE
any(x <= 0.2)  # is there at least one element in x that is <= 0.2?
## [1] TRUE
```

Note The **`all`** function will frequently be used in conjunction with "`==`". This is because the latter, as we have said above, is itself vectorised: it does *not* test whether a vector *as a whole* is equal to another one.

```
z <- c(1, 2, 3)
z == 1:3  # elementwise equal
## [1] TRUE TRUE TRUE
all(z == 1:3)  # elementwise equal summarised
## [1] TRUE
```

However, let us keep in mind the warning about the testing for *exact* equality of floating-point numbers stated in Section 3.2.3. Sometimes, considering absolute or relative errors might be more appropriate.

```
z <- sin((0:10)*pi)  # sin(0), sin(pi), sin(2*pi), ..., sin(10*pi)
all(z == 0.0)  # danger zone! please don't...
## [1] FALSE
all(abs(z - 0.0) < 1e-9)  # are the absolute errors negligible?
## [1] TRUE
```

We can also call **`sum`** on a logical vector. Taken into account that it interprets `TRUE` as numeric `1` and `FALSE` as `0` (more on this in Section 4.1), it will give us the number of elements equal to `TRUE`.

```
sum(x <= 0.2)  # how many elements in x are <= 0.2?
## [1] 1998
```

Also, by computing **`sum`**`(x)/`**`length`**`(x)`, we can obtain the proportion (fraction) of values equal to `TRUE` in `x`. Equivalently:

```
mean(x <= 0.2)  # proportion of elements <= 0.2
## [1] 0.1998
```

Naturally, we *expect* `mean(runif(n) <= 0.2)` to be equal to 0.2 (20%), but with randomness we can never be sure.

3.3.5 Simplifying Predicates

Each aspiring programmer needs to become fluent with the rules governing the transformations of logical conditions, for example, that the negation of "`(x >= 0) & (x < 1)`" is equivalent to "`(x < 0) | (x >= 1)`".

Each such rule is called a *tautology*. Here are some of them:

- `!(!p)` is equivalent to `p` (double negation),
- `!(p & q)` holds if and only if `!p | !q` (De Morgan's law),
- `!(p | q)` is `!p & !q` (another De Morgan's law),
- `all(p)` is equivalent to `!any(!p)`.

Various combinations thereof are of course possible. Some further simplifications are enabled by other properties of the binary operations:

- commutativity (symmetry), e.g., $a + b = b + a$, $a * b = b * a$,
- associativity, e.g., $(a + b) + c = a + (b + c)$, $\max(\max(a, b), c) = \max(a, \max(b, c))$,
- distributivity, e.g., $a * b + a * c = a * (b + c)$, $\min(\max(a, b), \max(a, c)) = \max(a, \min(b, c))$,

and relations, including:

- transitivity, e.g., if $a \leq b$ and $b \leq c$ then surely $a \leq c$.

Exercise 3.1 *Assuming that `a`, `b`, and `c` are numeric vectors, simplify the following expressions:*

- *`!(b>a & b<c)`,*
- *`!(a>=b & b>=c & a>=c)`,*
- *`a>b & a<c | a<c & a>d`,*
- *`a>b | a<=b`,*
- *`a<=b & a>c | a>b & a<=c`,*
- *`a<=b & (a>c | a>b) & a<=c`,*
- *`!all(a > b & b < c)`.*

3.4 Choosing Elements with `ifelse`

The **`ifelse`** function is a vectorised version of the scalar **`if`**...**`else`** conditional statement which we will do without for as long as until Chapter 8.

It allows us to select an element from either one or another vector based on some logical condition.

A call to **`ifelse`**`(l, t, f)`, where `l` is a logical vector, returns a vector `y` such that:

$$y_i = \begin{cases} t_i & \text{if } l_i \text{ is TRUE}, \\ f_i & \text{if } l_i \text{ is FALSE}. \end{cases}$$

In other words, the i-th element of the result vector is equal to t_i if l_i is `TRUE` and to f_i otherwise.

For example:

```
(z <- rnorm(6))  # example vector
## [1] -0.560476 -0.230177  1.558708  0.070508  0.129288  1.715065
ifelse(z >= 0, z, -z)  # like abs(z)
## [1] 0.560476 0.230177 1.558708 0.070508 0.129288 1.715065
```

or:

```
(x <- rnorm(6))  # example vector
## [1]  0.46092 -1.26506 -0.68685 -0.44566  1.22408  0.35981
(y <- rnorm(6))  # example vector
## [1]  0.40077  0.11068 -0.55584  1.78691  0.49785 -1.96662
ifelse(x >= y, x, y)   # like pmax(x, y)
## [1]  0.46092  0.11068 -0.55584  1.78691  1.22408  0.35981
```

By now, we should not be surprised that the recycling rule is fired up if necessary:

```
ifelse(x > 0, x^2, 0)  # squares of positive xs and 0 otherwise
## [1] 0.21244 0.00000 0.00000 0.00000 1.49838 0.12947
```

Note Keep in mind that all arguments are evaluated in their entirety before deciding on which element should be selected. Therefore, the following call will generate a warning:

```
ifelse(z >= 0, log(z), NA_real_)
## Warning in log(z): NaNs produced
## [1]       NA       NA  0.44386 -2.65202 -2.04571  0.53945
```

This is because with **`log`**`(z)`, we are computing the logarithms of negative values anyway. To fix this, we can write:

```
log(ifelse(z >= 0, z, NA_real_))
## [1]       NA       NA  0.44386 -2.65202 -2.04571  0.53945
```

The calls to **ifelse** can naturally be nested in the case where we yearn for an **if**…**else** **if**…**else**-type expression.

Example 3.2 *A version of* ***pmax(pmax(x, y), z)*** *can be written as:*

```
ifelse(x >= y,
    ifelse(z >= x, z, x),
    ifelse(z >= y, z, y)
)
## [1] 0.46092 0.11068 1.55871 1.78691 1.22408 1.71506
```

However, determining the three intermediate logical vectors is not necessary; we can save one call to ***`>=`*** *by introducing an auxiliary variable:*

```
xy <- ifelse(x >= y, x, y)
ifelse(z >= xy, z, xy)
## [1] 0.46092 0.11068 1.55871 1.78691 1.22408 1.71506
```

Exercise 3.3 *Figure 3.1 depicts a realisation of the mixture* $Z = 0.2X + 0.8Y$ *of two normal distributions* $X \sim \mathrm{N}(-2, 0.5)$ *and* $Y \sim \mathrm{N}(3, 1)$.

```
n <- 100000
z <- ifelse(runif(n) <= 0.2, rnorm(n, -2, 0.5), rnorm(n, 3, 1))
hist(z, breaks=101, probability=TRUE, main="", col="white")
```

In other words, we generated a variate from the normal distribution that has expected value of -2 with probability 20% and from the one with expectation of 3 otherwise.

Inspired by the above, generate the following Gaussian mixtures:

- $\frac{2}{3}X + \frac{1}{3}Y$, *where* $X \sim \mathrm{N}(100, 16)$ *and* $Y \sim \mathrm{N}(116, 8)$,
- $0.3X + 0.4Y + 0.3Z$, *where* $X \sim \mathrm{N}(-10, 2)$, $Y \sim \mathrm{N}(0, 2)$, *and* $Z \sim \mathrm{N}(10, 2)$.

(*) On a side note, knowing that if X *follows* $\mathrm{N}(0, 1)$, *then the scaled-shifted* $\sigma X + \mu$ *is distributed* $\mathrm{N}(\mu, \sigma)$, *the above can be equivalently written as:*

```
w <- (runif(n) <= 0.2)
z <- rnorm(n, 0, 1)*ifelse(w, 0.5, 1) + ifelse(w, -2, 3)
```

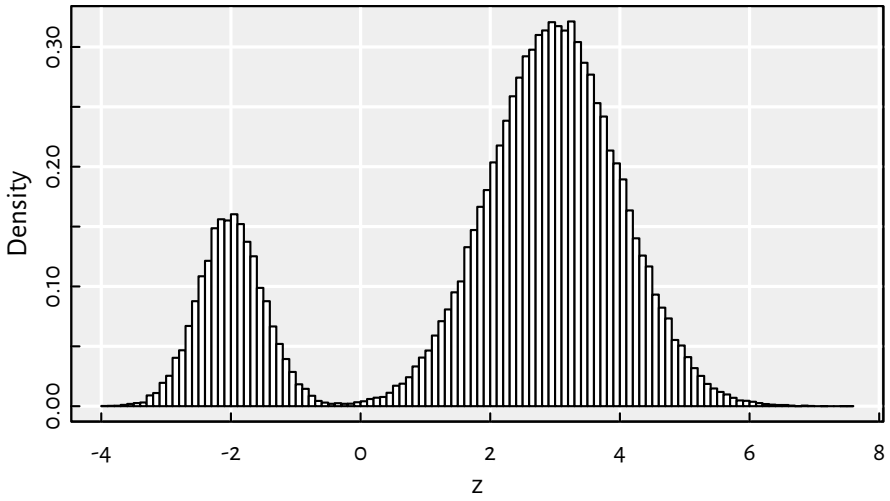

Figure 3.1: A mixture of two Gaussians generated with **`ifelse`**

3.5 Exercises

Exercise 3.4 *Answer the following questions:*

- *Why the statement "Earth is flat or the smallpox vaccine is proven effective" is obviously true?*
- *What is the difference between* `NA` *and* `NA_real_`*?*
- *Why is "*`FALSE & NA`*" equal to* `FALSE`*, but "*`TRUE & NA`*" is* `NA`*?*
- *Why has "*`ifelse(x>=0, sqrt(x), NA_real_)`*" a tendency to generate warnings and how to rewrite it so as to prevent that from happening?*
- *What is the interpretation of "*`mean(x >= 0 & x <= 1)`*"?*
- *For some integer x and y, how to verify whether $0 < x < 100$, $0 < y < 100$, and $x < y$, all at the same time?*
- *Mathematically, for all real $x, y > 0$, it holds $\log xy = \log x + \log y$. Why then "*`all(log(x*y) == log(x)+log(y))`*" can sometimes return* `FALSE`*? How to fix this?*
- *Is "*`x/y/z`*" always equal to "*`x/(y/z)`*"? How to fix this?*
- *What is the purpose of very specific functions such as* **`log1p`** *and* **`expm1`** *(see their help page) and many other ones listed in, e.g., the GNU GSL library [23]? Is our referring to them a violation of the beloved "let us be minimalist" approach?*
- *If we know that x may be subject to error, how to test whether $x > 0$ in a robust manner?*
- *Is "*`y<-5`*" the same as "*`y <- 5`*" or rather "*`y < -5`*"?*

Exercise 3.5 *Compute the cross-entropy loss between a numeric vector* $\boldsymbol{p}$ *with values in the interval* $(0, 1)$ *and a logical vector* $\boldsymbol{y}$*, both of length* n *(you can generate them randomly or manually, it does not matter, it is just an exercise):*

$$\mathcal{L}(\boldsymbol{p}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1}^{n} \ell_i,$$

where

$$\ell_i = \begin{cases} -\log p_i & \text{if } y_i \text{ is TRUE}, \\ -\log(1 - p_i) & \text{if } y_i \text{ is FALSE}. \end{cases}$$

Interpretation: in classification problems, $y_i \in \{FALSE, TRUE\}$ *denotes the true class of the* i*-th object (say, whether the* i*-th hospital patient is symptomatic) and* $p_i \in (0, 1)$ *a machine learning algorithm's* confidence *that* i *belongs to class* `TRUE` *(e.g., how sure a decision tree model is that the corresponding person is unwell). Ideally, if* y_i *is* `TRUE`*,* p_i *should be close to* 1 *and to* 0 *otherwise. The cross-entropy loss quantifies by how much a classifier differs from the omniscient one. The use of the logarithm penalises strong beliefs in the wrong answer.*

By the way! If you have solved any of the exercises encountered so far by referring to **`if`** statements, **`for`** loops, vector indexing like `x[...]`, or any external R package, please go back and re-write your code. Let us keep it simple (effective, readable) by using the *base* R's vectorised operations that we have introduced.

4

Lists and Attributes

After two brain-teasing chapters, it is time to cool it down a little. In this more technical part, we will introduce lists, which serve as universal containers for R objects of any size and type. Moreover, we will also show that each R object can be equipped with a number of optional attributes, thanks to which we will not only be able to label elements in any vector, but also – later – introduce new complex data types such as matrices and data frames.

4.1 Type Hierarchy and Conversion

So far, we were dealing with three types of atomic vectors:

1. `logical` (Chapter 3),
2. `numeric` (Chapter 2),
3. `character` (which we have barely touched upon yet, but rest assured that they will be covered in detail very soon; see Chapter 6).

To determine the type of an object programmatically, we can call the **`typeof`** function.

```
typeof(c(1, 2, 3))
## [1] "double"
typeof(c(TRUE, FALSE, TRUE, NA))
## [1] "logical"
typeof(c("spam", "spam", "bacon", "gluten-free spam"))
## [1] "character"
```

It turns out that we can easily convert between these types, either on our explicit demand (*type casting*), or on-the-fly (*coercion*, when we perform an operation that expects something different from the kind of input it was fed with).

Note (*) Numeric vectors are reported as being either of type `double` (double-precision floating-point numbers) or `integer` (32-bit; it is a subset of `double`); see Section 6.4.1. In most practical cases, this is a technical detail which we can safely ignore; compare also the **`mode`** function.

4.1.1 Explicit Type Casting

We can use functions such as **`as.logical`**, **`as.numeric`**, and **`as.character`** to *coerce* (convert) given objects to the corresponding types.

```
as.numeric(c(TRUE, FALSE, NA, TRUE, NA, FALSE))
## [1]  1  0 NA  1 NA  0
as.logical(c(-2, -1, 0, 1, 2, 3, NA_real_, -Inf, NaN))
## [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE    NA  TRUE    NA
```

Important It is easily seen that the rules are:

- `TRUE` → `1`,
- `FALSE` → `0`,
- `NA` → `NA_real_`,

and:

- `0` → `FALSE`,
- `NA_real_` and `NaN` → `NA`,
- anything else → `TRUE`.

The distinction between zero and non-zero is commonly applied in other programming languages as well.

Moreover, in the case of the conversion involving character strings, we have:

```
as.character(c(TRUE, FALSE, NA, TRUE, NA, FALSE))
## [1] "TRUE"  "FALSE" NA      "TRUE"  NA      "FALSE"
as.character(c(-2, -1, 0, 1, 2, 3, NA_real_, -Inf, NaN))
## [1] "-2"   "-1"   "0"    "1"    "2"    "3"    NA     "-Inf" "NaN"
as.logical(c("TRUE", "True", "true", "T",
             "FALSE", "False", "false", "F",
             "anything other than these", NA_character_))
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE    NA    NA
as.numeric(c("0", "-1.23e4", "pi", "2+2", "NaN", "-Inf", NA_character_))
## Warning: NAs introduced by coercion
## [1]      0 -12300     NA     NA    NaN   -Inf     NA
```

4.1.2 Implicit Conversion (Coercion)

Recall that we referred to the three vector types as *atomic* ones: they can only be used to store elements of the *same type*.

If we make an attempt at composing an object of mixed types with **`c`**, the common type

will be determined in such a way that storing the data is done without information loss:

```
c(-1, FALSE, TRUE, 2, "three", NA)
## [1] "-1"    "FALSE" "TRUE"  "2"     "three" NA
c("zero", TRUE, NA)
## [1] "zero" "TRUE" NA
c(-1, FALSE, TRUE, 2, NA)
## [1] -1  0  1  2 NA
```

Hence, we see that `logical` is the least, whereas `character` – the most general of the three.

Note The logical `NA` is converted to `NA_real_` and `NA_character_` in the above examples. R users tend to rely on implicit type conversion when they write `c(1, 2, NA, 4)` instead of the more explicit `c(1, 2, NA_real_, 4)`. In most cases, this is fine.

However, occasionally, it will be wiser to be more unequivocal. For instance, `rep(NA_real_, 1e9)` pre-allocates a long numeric vector, instead of a logical one.

Some functions that expect vectors of specific types can apply coercion by themselves (or act as if they do so):

```
c(NA, FALSE, TRUE) + 10 # implicit conversion logical -> numeric
## [1] NA 10 11
c(-1, 0, 1) & TRUE  # implicit conversion numeric -> logical
## [1]  TRUE FALSE  TRUE
sum(c(TRUE, TRUE, FALSE, TRUE, FALSE))  # same as sum(as.numeric(...))
## [1] 3
cumsum(c(TRUE, TRUE, FALSE, TRUE, FALSE))
## [1] 1 2 2 3 3
cummin(c(TRUE, TRUE, FALSE, TRUE, FALSE))
## [1] 1 1 0 0 0
```

Exercise 4.1 *In one of the previous exercises, we have computed the cross-entropy loss between a logical vector $\boldsymbol{y} \in \{0,1\}^n$ and a numeric vector $\boldsymbol{p} \in (0,1)^n$. This measure can be equivalently defined as:*

$$\mathcal{L}(\boldsymbol{p},\boldsymbol{y}) = -\frac{1}{n}\left(\sum_{i=1}^{n} y_i \log(p_i) + (1-y_i)\log(1-p_i)\right).$$

Implement the above formula (using vectorised operations, but not relying on `ifelse` *this time) and compute the cross-entropy loss between, say, "*`y <- sample(c(FALSE, TRUE), n)`*" and "*`p <- runif(n)`*" for some* `n`*. Note how seamlessly we are translating between* `FALSE`*/*`TRUE`*s and* 0*/*1*s in the above equation (in particular, where we let $1-y_i$ mean the logical negation of y_i).*

4.2 Lists

Lists are *generalised* vectors. They can be comprised of R objects of any kind, also other lists. This is why we classify them as *recursive* (and not atomic) objects. They are especially useful wherever there is a need to handle some *multitude* as a single entity.

4.2.1 Creating Lists

The most straightforward way to create a list is by means of the **`list`** function:

```
list(1, 2, 3)
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

Notice that the above is not the same as "`c(1, 2, 3)`". We got a sequence that wraps three numeric vectors, each of length one. Also, how overly talkative R is when printing out lists!

```
list(c(1, 2, 3), 4, c(TRUE, FALSE, FALSE, NA, TRUE), "and so forth")
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1]  TRUE FALSE FALSE    NA  TRUE
##
## [[4]]
## [1] "and so forth"
list(list(c(TRUE, FALSE, NA, TRUE), letters), runif(5))  # a list of lists
## [[1]]
## [[1]][[1]]
## [1]  TRUE FALSE    NA  TRUE
##
## [[1]][[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

(continues on next page)

(continued from previous page)

```
##
##
## [[2]]
## [1] 0.28758 0.78831 0.40898 0.88302 0.94047
```

However, the **str** function can be used to print R objects in a more concise fashion:

```
str(list(list(c(TRUE, FALSE, NA, TRUE), letters), runif(5)))
## List of 2
##  $ :List of 2
##   ..$ : logi [1:4] TRUE FALSE NA TRUE
##   ..$ : chr [1:26] "a" "b" "c" "d" ...
##  $ : num [1:5] 0.288 0.788 0.409 0.883 0.94
```

Note In Section 4.1, we said that the **c** function, when fed with arguments of mixed types, tries to determine the common type that retains the sense of data. If a coercion to an atomic vector is not possible, the result will be a list.

```
c(1, "two", sd)  # `sd` is an object of type `function`
## [[1]]
## [1] 1
##
## [[2]]
## [1] "two"
##
## [[3]]
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x5565ec404b28>
## <environment: namespace:stats>
```

Thus, the **c** function can also be used to concatenate lists:

```
c(list(1), list(2), list(3))  # 3 lists -> 1 list
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

Lists can be repeated using **rep**:

```
rep(list(1:11, LETTERS), 2)
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11
##
## [[2]]
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
##
## [[3]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11
##
## [[4]]
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

4.2.2 Coercing to and from Lists

The conversion of an atomic vector to a list of length-1 vectors can be done via a call to **as.list**:

```
as.list(c(1, 2, 3))  # vector of length 3 -> list of 3 length-1 vectors
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

Unfortunately, calling, say **as.numeric** on a list (even if it a list comprised of numeric vectors only) will result in an error. However, we can try to flatten a list to an atomic vector, provided that it is possible, by calling **unlist**.

```
unlist(list(list(1, 2), list(3, list(4:8)), 9))
## [1] 1 2 3 4 5 6 7 8 9
unlist(list(list(1, 2), list(3, list(4:8)), "spam"))
## [1] "1"    "2"    "3"    "4"    "5"    "6"    "7"    "8"    "spam"
```

Note (*) In Chapter 11, we will mention the **simplify2array** function which generalises **unlist** in a way that can sometimes result in a matrix.

4.3 NULL

The NULL object (the one and only object of type "NULL") can be used as a placeholder for any other R object or designate the absence of such.

```
list(NULL, NULL, month.name)
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
```

NULL is different from a vector of length zero, because the latter has a type.

However, NULL sometimes *behaves* as a 0-length vector. In particular, **length**(NULL) returns 0. Also, **c** called with no arguments returns NULL.

Testing for NULL-ness can be done with a call to **is.null**.

Important NULL is not alike NA (or it is other-typed variants); the latter can be emplaced in an atomic vector.

```
c(1, NA, 3, NULL, 5)  # NULL behaves as a 0-length vector here
## [1]  1 NA  3  5
```

They both have very distinct semantics (no value vs a missing value).

Later we will see that some functions return NULL, invisibly, because they actually have nothing interesting to yield. This is the case of **print** or **plot**, which are called because of their side effects (printing and plotting).

Also, in some contexts, replacing content with NULL (e.g., when subsetting a list) will actually result in its removal.

4.4 Object Attributes

Lists can be used to wrap many objects and form a single, ordered collection thereof.

Attributes, on the other hand, give means to inject some *extra* data into an object of any type (except `NULL`).

Attributes are (unordered) `key=value` pairs, where `key` in an arbitrary single character string and `value` is any R object except `NULL`. They can be introduced by calling, amongst others[1], the **`structure`** function:

```
x_simple <- 1:10
x <- structure(
    x_simple,  # the object to be equipped with attributes
    attribute1="value1",
    attribute2=c(6, 100, 324)
)
```

4.4.1 Developing Perceptual Indifference to Most Attributes

Let us see how the above `x` is reported on the console:

```
print(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
## attr(,"attribute1")
## [1] "value1"
## attr(,"attribute2")
## [1]   6 100 324
```

Note that the object of concern, "`1:10`", was displayed first. We need to get used to that; most of the time, we should treat the "`attr`..." parts of the display as if they were printed in tiny font.

Equipping an object with attributes does not change its very nature (see, however Chapter 10 for some exceptions). For example, the above `x`, despite featuring some extra data (metadata), is still treated as an ordinary sequence of numbers by most functions:

```
sum(x)   # the same as sum(1:10), sum() does not care about any attributes
## [1] 55
typeof(x)  # just a numeric vector, but with some perks
## [1] "integer"
```

Important Attributes are generally ignored by most functions unless they have specifically been programmed to pay attention to them.

---

[1] Other ways include the replacement versions of the **`attr`** and **`attributes`** functions; see Section 9.4.6.

4.4.2 But There Are Some Use Cases

Some R functions add attributes to the return value to sneak extra information that *might* be useful, just in case.

For instance, **`na.omit`**, whose main aim is to remove missing values from an atomic vector, yields:

```
y <- c(10, 20, NA, 40, 50, NA, 70)
(y_na_free <- na.omit(y))
## [1] 10 20 40 50 70
## attr(,"na.action")
## [1] 3 6
## attr(,"class")
## [1] "omit"
```

We can enjoy the `NA`-free version of `y` in any further computations:

```
mean(y_na_free)
## [1] 38
```

However, the `na.action` attribute (we ignore the `class` part until Chapter 10) tells us *where* the missing observations were:

```
attr(y_na_free, "na.action")  # read the attribute value
## [1] 3 6
## attr(,"class")
## [1] "omit"
```

As another example, **`gregexpr`** can be used to search for a given pattern in a character vector (for more details, see Chapter 6):

```
needle <- "spam|gluten"  # pattern to search for: spam OR gluten
haystack <- c("spam, spam, bacon, and gluten-free spam", "spammer")  # text
(pos <- gregexpr(needle, haystack))
## [[1]]
## [1]  1  7 24 36
## attr(,"match.length")
## [1] 4 4 6 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] 1
## attr(,"match.length")
```

(continues on next page)

(continued from previous page)

```
## [1] 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

We sought all occurrences of the pattern within two character strings. As their number may vary from string to string, to wrap the results in a list was a good design choice. Each list element gives the starting positions where matches can be found (there are four and one match(es), respectively).

Each vector of positions also features its own `match.length` attribute (amongst others), in case we need it.

Exercise 4.2 *Create a list with EUR/AUD, EUR/GBP, and EUR/USD exchange rates read from the* `euraud-*.csv`, `eurgbp-*.csv`, *and* `eurusd-*.csv` *files in our data repository*[2]*. Each of its three elements should be a numeric vector storing the currency exchange rates. Furthermore, equip them with* `currency_from`, `currency_to`, `date_from`, *and* `date_to` *attributes, for example:*

```
##  [1]     NA 1.6006 1.6031     NA     NA 1.6119 1.6251 1.6195 1.6193 1.6132
## [11]     NA     NA 1.6117 1.6110 1.6188 1.6115 1.6122     NA
## attr(,"currency_from")
## [1] "EUR"
## attr(,"currency_to")
## [1] "AUD"
## attr(,"date_from")
## [1] "2020-01-01"
## attr(,"date_to")
## [1] "2020-06-30"
```

Note that such additional information could of course be stored in a few separate variables (other vectors), but then it would not be as convenient to use as the above representation.

4.4.3 Special Attributes

Attributes have a great potential which is somewhat wasted by R users due to their rarely knowing:

- that attributes exist (pessimistic scenario) or
- how to handle them (realistic scenario).

But we now know.

What is more, some attributes have been predestined to play a fundamental role in R. Namely, the most prevalent amongst the *special attributes* are:

[2] https://github.com/gagolews/teaching-data/tree/master/marek

- `names`, `row.names`, and `dimnames` are used to label the elements of atomic and generic vectors (see below), and also rows and columns in matrices (Chapter 11) and data frames (Chapter 12),
- `dim` allows for turning flat vectors into matrices and other tensors (Chapter 11),
- `levels` labels the underlying integer codes in factor objects (Section 10.3.3),
- `class` can be used to bring forth new complex data structures based on basic types (Chapter 10).

We call them *special*, because:

- they cannot be assigned arbitrary values; for instance, we will soon see that `names` can only be mapped to a character vector of the length equal to that of the sequence it is labelling,
- they can be accessed via designated functions, e.g., **`names`**, **`class`**, **`dim`**, **`dimnames`**, **`levels`**, etc.,
- they are widely recognised by many base and third-party R functions.

However, in spite of the above, special attributes can still be managed as any other (ordinary) ones.

Exercise 4.3 *`comment` is perhaps the most rarely used special attribute. Create an object (whatever) equipped with the `comment` attribute. Verify that assigning to it anything other than a character vector leads to an error. Read its value by calling the **`comment`** function. Display the object equipped with `comment`. Note that the **`print`** function ignores its existence whatsoever: this is how special it is.*

Important (*) The accessor functions such as **`names`** or **`class`** might return meaningful values event if the corresponding attribute is not set explicitly; see, e.g., Section 11.1.5 for an example.

4.4.4 Labelling Vector Elements with the `names` Attribute

A special attribute called `names` can be used to label the elements of atomic vectors and lists.

```
(x <- structure(c(13, 2, 6), names=c("spam", "sausage", "celery")))
##    spam sausage  celery
##      13       2       6
```

The labels may improve the expressivity and readability of our code and data.

Exercise 4.4 *Verify that the above `x` is still an ordinary numeric vector by calling **`typeof`** and **`sum`** on it.*

Note that we can ignore the `names` attribute whatsoever. If we apply any operation dis-

cussed in Chapter 2, we will still garner the same result no matter if such extra information is present or not.

It is just the **`print`** function that changed its behaviour slightly (it is a special attribute after all). Instead of reporting:

```
## [1] 13  2  6
## attr(,"names")
## [1] "spam"    "sausage" "celery"
```

we got a nicely formatted table-like display. Non-special attributes are still printed in a standard way.

```
##    spam sausage  celery
##      13       2       6
## attr(,"additional_attribute")
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Note In Chapter 5, we will also see that some operations (such as indexing) can gain extra features in the presence of the `names` attribute.

This attribute can be read by calling:

```
attr(x, "names")  # just like any other attribute
## [1] "spam"    "sausage" "celery"
names(x)  # because it is so special
## [1] "spam"    "sausage" "celery"
```

Named vectors can be easily created with the **`c`** and **`list`** functions as well:

```
c(a=1, b=2)
## a b
## 1 2
list(a=1, b=2)
## $a
## [1] 1
##
## $b
## [1] 2
c(a=c(x=1, y=2), b=3, c=c(z=4))  # this is smart
## a.x a.y   b c.z
##   1   2   3   4
```

Let us contemplate for a while how a named list looks like when printed on the console. Again, it is still a list, but with some extras.

Exercise 4.5 *A whole lot of functions return named vectors. Evaluate the following expressions and read the corresponding pages in the documentation:*

- ***`quantile(runif(100))`*** *(note that it generalises* ***`min`****,* ***`median`****, and* ***`max`****),*
- ***`hist(runif(100), plot=FALSE)`****,*
- ***`options`*** *(on a side note, take note of the* `digits`*,* `scipen`*,* `max.print`*, and* `width` *options),*
- ***`capabilities`****.*

Note (*) Most of the time, lists are used merely as *containers* for other R objects. This is a boring yet essential role. However, let us just mention here that each data frame is in fact a generic vector (see Chapter 12): each column thereof corresponds to a named list element:

```
(df <- head(iris))  # some data frame
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
typeof(df)  # it is just a list (with extras that'll be discussed later)
## [1] "list"
unclass(df)  # how it is represented exactly (without the extras)
## $Sepal.Length
## [1] 5.1 4.9 4.7 4.6 5.0 5.4
##
## $Sepal.Width
## [1] 3.5 3.0 3.2 3.1 3.6 3.9
##
## $Petal.Length
## [1] 1.4 1.4 1.3 1.5 1.4 1.7
##
## $Petal.Width
## [1] 0.2 0.2 0.2 0.2 0.2 0.4
##
## $Species
## [1] setosa setosa setosa setosa setosa setosa
## Levels: setosa versicolor virginica
##
## attr(,"row.names")
## [1] 1 2 3 4 5 6
```

Therefore, the functions we discuss in this chapter are of use in the processing of such structured data as well.

4.4.5 Altering and Removing Attributes

We know that a single attribute can be read by calling **attr**. Their whole list is generated with a call to **attributes**.

```
(x <- structure(c("some", "object"), names=c("X", "Y"),
    attribute1="value1", attribute2="value2", attribute3="value3"))
##        X        Y
##   "some" "object"
## attr(,"attribute1")
## [1] "value1"
## attr(,"attribute2")
## [1] "value2"
## attr(,"attribute3")
## [1] "value3"
attr(x, "attribute1")  # reads a single attribute, returns NULL if unset
## [1] "value1"
attributes(x)  # returns a named list with all attributes of an object
## $names
## [1] "X" "Y"
##
## $attribute1
## [1] "value1"
##
## $attribute2
## [1] "value2"
##
## $attribute3
## [1] "value3"
```

We can alter an attribute's value or add further attributes, by referring to the **structure** function once again. Moreover setting an attribute's value to NULL gets rid of it completely.

```
structure(x, attribute1=NULL, attribute4="added", attribute3="modified")
##        X        Y
##   "some" "object"
## attr(,"attribute2")
## [1] "value2"
## attr(,"attribute3")
## [1] "modified"
```

(continues on next page)

(continued from previous page)

```
## attr(,"attribute4")
## [1] "added"
```

As far as the `names` attribute is concerned, we may generated an un-named copy of an object by calling:

```
unname(x)
## [1] "some"   "object"
## attr(,"attribute1")
## [1] "value1"
## attr(,"attribute2")
## [1] "value2"
## attr(,"attribute3")
## [1] "value3"
```

In Section 9.4.6, we will discuss the so-called replacement functions which will also enable us to modify or remove an object's attribute in-place, by calling "`attr(x, "some_attribute") <- new_value`".

Moreover, in Section 5.5 we note that certain operations (such as vector indexing, elementwise arithmetic operations, coercion) might not preserve all attributes of the objects that were given as their inputs.

4.5 Exercises

Exercise 4.6 *Answer the following.*

- *That is the meaning of "`c(TRUE, FALSE) * 1:10`"?*
- *What does "`sum(as.logical(x))`" compute when `x` is a numeric vector?*
- *We said that atomic vectors of type `character` are the most general ones. Therefore, is "`as.numeric(as.character(x))`" the same as "`as.numeric(x)`", regardless of the type of `x`?*
- *What is the meaning of "`as.logical(x+y)`" if `x` and `y` are logical vectors? What about "`as.logical(x*y)`", "`as.logical(1-x)`", and "`as.logical(x!=y)`"?*
- *Let `x` be a* named *numeric vector, e.g., "`x <- quantile(runif(100))`". What is the result of "`2*x`", "`mean(x)`", and `round(x, 2)`?*
- *Give two ways to create a named character vector.*
- *Give two ways (discussed above; there are more) to remove the `names` attribute from an object.*

Exercise 4.7 *There are a few peculiarities when joining or coercing lists. Compare the results generated by the following pairs of expressions:*

```
# 1)
as.character(list(list(1, 2), list(3, list(4)), 5))
as.character(unlist(list(list(1, 2), list(3, list(4)), 5)))
# 2)
as.numeric(list(list(1, 2), list(3, list(4)), 5))
as.numeric(unlist(list(list(1, 2), list(3, list(4)), 5)))
# 3)
unlist(list(list(1, 2), sd))
list(1, 2, sd)
# 4)
c(list(c(1, 2), 3), 4, 5)
c(list(c(1, 2), 3), c(4, 5))
```

Exercise 4.8 *Given numeric vectors* `x`, `y`, `z`, *and* `w`, *how to combine* `x`, `y`, *and* `list(z, w)` *so as to obtain* `list(x, y, z, w)`? *More generally, given a set of atomic vectors and lists of atomic vectors, how to combine them to get a single list that features all atomic vectors as its elements (not a list of atomic vectors and lists, not atomic vectors unwound, etc.)?*

Exercise 4.9 *What is the meaning of the following when* `x` *is a logical vector?*

- `cummin(x)` *and* `cummin(!x)`,
- `cummax(x)` *and* `cummax(!x)`,
- `cumsum(x)` *and* `cumsum(!x)`,
- `cumprod(x)` *and* `cumprod(!x)`.

Exercise 4.10 `readRDS` *allows for serialising R objects and writing their snapshots to disk, so that they can be later restored very quickly via a call to* `saveRDS`. *Verify whether this function preserves object attributes.*

Exercise 4.11 *(*) Use* `jsonlite::fromJSON` *to read some JSON file in the form of a named list.*

In the extremely unlikely event of us finding the current chapter boring, let us rejoice: some of the exercises and remarks that we will encounter in the next part – devoted to vector indexing – will definitely be deliciously stimulating!

5

Vector Indexing

We now know plenty of ways to process vectors *in their entirety*, but how to extract and replace specific *parts* thereof? We will be referring to such activities collectively as *indexing*, because they are often performed through the *index operator*, **`[`**.

5.1 `head` and `tail`

Let us begin with something more lightweight, though. The **`head`** function can be used to fetch a few elements from the beginning of a vector.

```
x <- 1:10
head(x)  # head(x, 6)
## [1] 1 2 3 4 5 6
head(x, 3)   # get the first three
## [1] 1 2 3
head(x, -3)  # skip the last three
## [1] 1 2 3 4 5 6 7
```

Similarly, **`tail`** extracts a few elements from the end of a sequence.

```
tail(x)  # tail(x, 6)
## [1]  5  6  7  8  9 10
tail(x, 3)   # get the last three
## [1]  8  9 10
tail(x, -3)  # skip the first three
## [1]  4  5  6  7  8  9 10
```

Both functions work on lists too[1]. They are useful, e.g., when we wish to preview the contents of a *big* object.

[1] **`head`** and **`tail`** are actually S3 generics defined in the **`utils`** package. We will be able to call them on matrices and data frames as well; see Chapter 10.

5.2 Subsetting of and Extracting from Vectors

Given a vector `x`, "`x[i]`" returns its subset comprised of elements indicated by the indexer `i`, which can be a *single* vector of:

- nonnegative integers (gives the positions of elements to retrieve),
- negative integers (gives the positions to omit),
- logical values (states whether the corresponding element should be fetched or skipped),
- character strings (locates the elements with specific names).

5.2.1 Nonnegative Indexes

Consider the following example vectors:

```
(x <- seq(10, 100, 10))
##  [1]  10  20  30  40  50  60  70  80  90 100
(y <- list(1, 11:12, 21:23))
## [[1]]
## [1] 1
##
## [[2]]
## [1] 11 12
##
## [[3]]
## [1] 21 22 23
```

The first element in a vector is at index 1. Hence:

```
x[1]          # the first element
## [1] 10
x[length(x)]  # the last element
## [1] 100
```

Important We might have wondered why "`[1]`" is being displayed each time we print out an atomic vector on the console:

```
print((1:51)*10)
##  [1]  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160 170
## [18] 180 190 200 210 220 230 240 250 260 270 280 290 300 310 320 330 340
## [35] 350 360 370 380 390 400 410 420 430 440 450 460 470 480 490 500 510
```

It is merely a visual hint indicating which vector element we output first in each line.

Vectorisation is a universal feature of R. Hence, it comes as no surprise that the indexer can be also of length greater than one.

```
x[c(1, length(x))]  # the first and the last
## [1]  10 100
x[1:3]  # the first three
## [1] 10 20 30
```

Take note of some boundary cases:

```
x[c(1, 2, 1, 0, 3, NA_real_, 1, 11)]  # repeated, 0, missing, out of bound
## [1] 10 20 10 30 NA 10 NA
x[c()]  # indexing by an empty vector
## numeric(0)
```

Important Subsetting with `[` yields an object of the same type.

When applied on lists, the index operator always returns a list as well, even if we ask for a single element:

```
y[2]  # a list that includes the 2nd element
## [[1]]
## [1] 11 12
y[c(1, 3)]  # note that this is not the same as x[1, 3] (a different story)
## [[1]]
## [1] 1
##
## [[2]]
## [1] 21 22 23
```

If we wish to *extract* a component, i.e., to dig into what is inside a list at a specific location, we can refer to `[[`:

```
y[[2]]  # extract the 2nd element
## [1] 11 12
```

This is exactly why R displays "`[[1]]`", "`[[2]]`", etc. when printing out lists on the console.

Note Calling "`x[[i]]`" on an *atomic* vector, where `i` is a single value has almost[2]

[2] See also Section 5.5 for the discussion on the preservation of object attributes.

the same effect as "`x[i]`". However, "`[[`" generates an error if the subscript is out of bounds.

Note (*) "`[[`" also supports multiple indexers.

```
y[[c(1, 3)]]
## Error in y[[c(1, 3)]]: subscript out of bounds
```

Its meaning is different from `y[c(1, 3)]`, though; we are about to extract a single value, remember? Here, indexing is applied *recursively*. Namely, the above is equivalent to `y[[1]][[3]]` – we got an error because `y[[1]]` is of length smaller than three.

More examples:

```
y[[c(3, 1)]]  # y[[3]][[1]]
## [1] 21
list(list(7))[[c(1, 1)]]  # 7, not list(7)
## [1] 7
```

Important Take note of the behaviour in the case of non-existing items:

```
c(1, 2, 3)[4]
## [1] NA
list(1, 2, 3)[4]
## [[1]]
## NULL
c(1, 2, 3)[[4]]
## Error in c(1, 2, 3)[[4]]: subscript out of bounds
list(1, 2, 3)[[4]]
## Error in list(1, 2, 3)[[4]]: subscript out of bounds
```

5.2.2 Negative Indexes

The indexer can also be a vector of negative integers. This way, we can *exclude* the elements at given positions:

```
y[-1]  # all but the first
## [[1]]
## [1] 11 12
##
## [[2]]
## [1] 21 22 23
```

(continues on next page)

(continued from previous page)

```
x[-(1:3)]
## [1]  40  50  60  70  80  90 100
x[-c(1, 0, 2, 1, 1, 8:100)]  # 0, repeated, out of bound indexes
## [1] 30 40 50 60 70
```

Note Negative and positive indexes cannot be mixed.

```
x[-1:3]  # recall that -1:3 == (-1):3
## Error in x[-1:3]: only 0's may be mixed with negative subscripts
```

Also, NA indexes are not allowed amongst negative ones.

5.2.3 Logical Indexer

A vector can also be subsetted by means of a logical vector. If they both are of identical lengths, the consecutive elements in the latter indicate whether the corresponding elements of the indexed vector are supposed to be selected (TRUE) or omitted (FALSE).

```
#   1***  2      3      4      5***  6***  7      8***  9?   10***
x[c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, TRUE, NA,  TRUE)]
## [1]  10  50  60  80  NA 100
```

In other words, `x[l]`, where `l` is a logical vector, returns all `x[i]` with `i` such that `l[i]` is TRUE. Above, we extracted the elements at indexes 1, 5, 6, 8, and 10.

Important Let us be careful: if the element selector is NA, the selected element will be set to a missing value (for atomic vectors) or NULL (for lists).

```
c("one", "two", "three")[c(NA, TRUE, FALSE)]
## [1] NA    "two"
list("one", "two", "three")[c(NA, TRUE, FALSE)]
## [[1]]
## NULL
##
## [[2]]
## [1] "two"
```

This, unfortunately, comes with no warning, which might be problematic when indexers are generated programmatically.

As a remedy, we sometimes pass the logical indexer to the **`which`** function first. It returns the indexes of the elements equal to TRUE, ignoring the missing ones.

```
which(c(NA, TRUE, FALSE))
## [1] 2
c("one", "two", "three")[which(c(NA, TRUE, FALSE))]
## [1] "two"
```

Recall that in Chapter 3, we have discussed ample vectorised operations that generate logical vectors. Anything that yields a logical vector of the same length as x can be passed as an indexer.

```
x > 60  # yes, it is a perfect indexer candidate
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
x[x > 60]  # select elements in x that are greater than 60
## [1]  70  80  90 100
x[x < 30 | 70 < x]  # elements not between 30 and 70
## [1]  10  20  80  90 100
x[x < mean(x)]  # elements smaller than the mean
## [1] 10 20 30 40 50
x[x^2 > 7777 | log10(x) <= 1.6]  # indexing via a transformed version of x
## [1]  10  20  30  90 100
(z <- round(runif(length(x)), 2))  # ten pseudorandom numbers
##  [1] 0.29 0.79 0.41 0.88 0.94 0.05 0.53 0.89 0.55 0.46
x[z <= 0.5]  # indexing based on z, not x — not a problem
## [1]  10  30  60 100
```

The indexer is always evaluated first and then passed to the subsetting operation – this operation does not care how such a logical vector was generated.

Furthermore, recycling rule is of course applied when necessary:

```
x[c(FALSE, TRUE)]  # every second element
## [1]  20  40  60  80 100
y[c(TRUE, FALSE)]  # interestingly, there is no warning here
## [[1]]
## [1] 1
##
## [[2]]
## [1] 21 22 23
```

Exercise 5.1 *Consider a simple database about six people, their most favourite dishes, and birth years.*

```
name <- c("Graham", "John", "Terry", "Eric",  "Michael", "Terry")
food <- c("bacon",  "spam", "spam",  "eggs",  "spam",    "beans")
year <- c( 1941,     1939,   1942,    1943,    1943,      1940  )
```

The consecutive elements in different vectors correspond to each other, e.g., Graham was born in 1941 and his favourite food was bacon.

- *List the names of people born in 1941 or 1942.*
- *List the names of those who like spam.*
- *List the names of those who like spam and were born after 1940.*
- *Compute the average birth year of the lovers of spam.*
- *Give the average age, in 1969, of those who didn't find spam utmostly delicious.*

The answers to the above must be provided programmatically, i.e., we do not just write `"Eric"` and `"Graham"`. The code must be generic enough so that it works in the case of any other database of this kind, no matter its size.

Exercise 5.2 *Remove missing values from a given vector without referring to the* ***`na.omit`*** *function.*

5.2.4 Character Indexer

If a vector is equipped with the `names` attribute, such as this one:

```
x <- structure(x, names=letters[1:10])  # add names
print(x)
##  a  b  c  d  e  f  g  h  i  j
## 10 20 30 40 50 60 70 80 90 100
```

These labels can be referred to for the purpose of extracting the elements. To do this, we use an indexer which is a character vector:

```
x[c("a", "f", "a", "g", "z")]
##    a    f    a    g <NA>
##   10   60   10   70   NA
```

Important We have said that special object attributes add *extra* functionality on top of the existing ones. Therefore, indexing by means of positive, negative, and logical vectors is of course still available:

```
x[1:3]
##  a  b  c
## 10 20 30
x[-(1:5)]
##  f  g  h  i  j
## 60 70 80 90 100
x[x > 70]
##  h  i  j
## 80 90 100
```

Lists can also be subsetted this way.

```
(y <- structure(y, names=c("first", "second", "third")))
## $first
## [1] 1
##
## $second
## [1] 11 12
##
## $third
## [1] 21 22 23
y[c("first", "second")]
## $first
## [1] 1
##
## $second
## [1] 11 12
y["third"]    # result is a list
## $third
## [1] 21 22 23
y[["third"]]  # result is the specific element unwrapped
## [1] 21 22 23
```

Important Labels do not have to be unique. When we have repeated names, the first matching element is extracted:

```
structure(1:3, names=c("a", "b", "a"))["a"]
## a
## 1
```

There is no direct way to select all *but* given names, just like with negative integer indexers. For a workaround, see Section 5.4.1.

Exercise 5.3 *Rewrite the solution to the above spam-lovers exercise assuming that we have the three features wrapped inside a list now:*

```
(people <- list(
    Name=c("Graham", "John", "Terry", "Eric",  "Michael", "Terry", "Steve"),
    Food=c("bacon",  "spam", "spam",  "eggs",  "spam",    "beans", "spam"),
    Year=c( 1941,     1939,   1942,    1943,    1943,      1940,    NA_real_)
))
## $Name
## [1] "Graham"  "John"    "Terry"   "Eric"    "Michael" "Terry"   "Steve"
```

(continues on next page)

(continued from previous page)

```
##
## $Food
## [1] "bacon" "spam"  "spam"  "eggs"  "spam"  "beans" "spam"
##
## $Year
## [1] 1941 1939 1942 1943 1943 1940   NA
```

Do not refer to `name`, `food`, and `year` directly. Instead, use the full `people[["Name"]]` etc. accessors. There is no need to pout, it is just tiny bit of extra work. Also note that we now have Steve amongst us.

5.3 Replacing Elements

5.3.1 Modifying Atomic Vectors

There are also *replacement* versions of the above indexing schemes. They allow us to substitute some new content for the old one.

```
(x <- 1:12)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
x[length(x)] <- 42  # modify the last element
print(x)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 42
```

The principles of vectorisation, recycling rule, and implicit coercion are all in place:

```
x[c(TRUE, FALSE)] <- c("a", "b", "c")
print(x)
##  [1] "a"  "2"  "b"  "4"  "c"  "6"  "a"  "8"  "b"  "10" "c"  "42"
```

Long story long: first, to make sure that the new content can be poured into old wineskin, R needed to convert the numeric vector to a character one; compare Section 4.1. Then, every second element therein, a total of six items, was replaced by a recycled version of the replacement sequence of length 3. Finally, the name "x" was re-bound to such a brought-forth object and the previous one became forgotten.

Note For more details on replacement functions in general, see Section 9.4.6. Such operations alter the state of the object they are called on (quite a rare behaviour in functional languages).

Exercise 5.4 *Replace missing values in a given numeric vector with the arithmetic mean of well-defined observations therein.*

5.3.2 Modifying Lists

List contents can be altered as well. For modifying individual elements, the safest option is to use the replacement version of the `[[` operator:

```
y <- list(a=1, b=1:2, c=1:3)
y[[1]] <- 100:110
y[["c"]] <- -y[["c"]]
print(y)
## $a
##  [1] 100 101 102 103 104 105 106 107 108 109 110
##
## $b
## [1] 1 2
##
## $c
## [1] -1 -2 -3
```

The replacement version of `[` modifies a whole sub-list:

```
y[1:3] <- list(1, c("a", "b", "c"), c(TRUE, FALSE))
print(y)
## $a
## [1] 1
##
## $b
## [1] "a" "b" "c"
##
## $c
## [1]  TRUE FALSE
```

Moreover:

```
y[1] <- list(1:10)  # replace 1 element with 1 object
y[-1] <- 10:11      # replace 2 elements with 2 vectors of length 1
print(y)
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $b
## [1] 10
##
## $c
## [1] 11
```

Note Let `idx` be a vector of positive indexes of elements to be modified. Overall, calling "`y[idx] <- z`" behaves as if we wrote:

1. `y[[idx[1]]] <- z[[1]]`,
2. `y[[idx[2]]] <- z[[2]]`,
3. `y[[idx[3]]] <- z[[3]]`,

and so forth.

Furthermore, `z` (but not `idx`) will be recycled if necessary, i.e., we take `z[[j %% length(z)]]` for consecutive `j`s from 1 to the length of `idx`.

Exercise 5.5 *Reflect upon the results of the following expressions:*

- `y[1] <- c("a", "b", "c")`,
- `y[[1]] <- c("a", "b", "c")`,
- `y[[1]] <- list(c("a", "b", "c"))`,
- `y[1:3] <- c("a", "b", "c")`,
- `y[1:3] <- list(c("a", "b", "c"))`,
- `y[1:3] <- "a"`,
- `y[1:3] <- list("a")`,
- `y[c(1, 2, 1)] <- c("a", "b", "c")`,

Important Setting a list item to `NULL` removes it from the list completely.

```
y <- list(1, 2, 3, 4)
y[1] <- NULL       # removes the 1st element (i.e., 1)
y[[1]] <- NULL     # removes the 1st element (i.e., now 2)
y[1] <- list(NULL) # sets the 1st element (i.e., now 3) to NULL
print(y)
## [[1]]
## NULL
##
## [[2]]
## [1] 4
```

The same notation convention is used for dropping object attributes; see Section 9.4.6.

5.3.3 Inserting New Elements

New elements can be pushed at the end of the vector quite easily[3].

[3] And often cheaply; see Section 8.3.5 for some performance notes. Still, a warning can be generated on each size extension if the `"check.bounds"` flag is set; see `help("options")`.

```
(x <- 1:5)
## [1] 1 2 3 4 5
x[length(x)+1] <- 6  # insert at the end
print(x)
## [1] 1 2 3 4 5 6
x[10] <- 10  # insert at the end but add more items
print(x)
##  [1]  1  2  3  4  5  6 NA NA NA 10
```

The elements to be inserted can be named as well:

```
x["a"] <- 11  # still inserts at the end
x["z"] <- 12
x["c"] <- 13
x["z"] <- 14  # z is already there; replace
print(x)
##                               a  z  c
##  1  2  3  4  5  6 NA NA NA 10 11 14 13
```

Note that x was not equipped with the `names` attribute before – the unlabelled elements were assigned blank labels (empty strings).

Note It is not possible to insert new elements at the beginning or in the middle of a sequence, at least not with the index operator. By writing "`x[3:4] <- 1:5`" we do not replace two elements in the middle by five other ones. However, we can always use the **c** function to slice parts of the vector and intertwine them with some new content:

```
x <- seq(10, 100, 10)
x <- c(x[1:2], 1:5, x[5:7])
print(x)
##  [1] 10 20  1  2  3  4  5 50 60 70
```

5.4 Functions Related to Indexing

Let us review some operations which pinpoint interesting elements in a vector (or functions based on these).

5.4.1 Matching of Elements in Another Vector

We know that the `==` operator acts in an elementwise manner. It compares each element in a vector on the lefthand side to the *corresponding* element in a vector on the

right side. Thus, missing values and the recycling rule aside, if `z <- (x == y)`, then `z[i]` is `TRUE` if and only if `x[i] == y[i]`.

The `` `%in%` `` operator[4] is vectorised differently: it checks whether each element on the lefthand side matches *one of* the elements on the right. Given `z <- (x %in% y)`, `z[i]` is `TRUE` whenever `x[i] == y[j]` for some `j`.

```
c("spam", "bacon", "spam", "eggs", "spam") %in% c("eggs", "spam", "ham")
## [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

Example 5.6 *Here is how we can remove the elements of a vector that have been assigned specified labels.*

```
(x <- structure(1:12, names=month.abb))  # example vector
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##   1   2   3   4   5   6   7   8   9  10  11  12
x[!(names(x) %in% c("Jan", "May", "Sep", "Oct"))]  # get rid of some elements
## Feb Mar Apr Jun Jul Aug Nov Dec
##   2   3   4   6   7   8  11  12
```

More generally, **`match`**`(x, y)` gives us the index of the element in `y` that matches each `x[i]`.

```
match(c("spam", "bacon", "spam", "eggs", "spam"), c("eggs", "spam", "ham"))
## [1]  2 NA  2  1  2
match(month.abb, c("Jan", "May", "Sep", "Oct"))  # is the month on the list?
##  [1]  1 NA NA NA  2 NA NA NA  3  4 NA NA
match(c("Jan", "May", "Sep", "Oct"), month.abb)  # which month is it?
## [1]  1  5  9 10
```

`NA_real_` denotes (by default) a no-match.

Exercise 5.7 *Check out the documentation of* **`` `%in%` ``** *to see how this operator is reduced to a call to* **`match`**. *Also, verify that it treats missing values as well-defined ones.*

If the elements in `y` are not unique, the smallest index `j` such that `x[i] == y[j]` is returned. Therefore, for example, **`match`**`(TRUE, l)` can be used to fetch the index of the first occurrence of a positive value in a logical vector `l`.

```
(x <- round(runif(10), 2))  # example vector
##  [1] 0.29 0.79 0.41 0.88 0.94 0.05 0.53 0.89 0.55 0.46
match(TRUE, x>0.8)  # index of the first value > 0.8 (from the left)
## [1] 4
```

[4] A fantastic name; see Section 9.4.5.

5.4.2 Assigning Numbers into Intervals

`findInterval` can come in handy where the assigning of numeric values into real intervals is needed. Namely, `z <- findInterval(x, y)` for increasing `y` gives `z[i]` being the index `j` such that `x[i]` is between `y[j]` (by default, inclusive) and `y[j+1]` (by default, exclusive).

For example, a sequence of five knots $\boldsymbol{y} = (-\infty, 0.25, 0.5, 0.75, \infty)$ yields a division of the real line to the following four intervals:

$$\underset{(1)}{[-\infty, 0.25)} \quad \underset{(2)}{[0.25, 0.5)} \quad \underset{(3)}{[0.5, 0.75)} \quad \underset{(4)}{[0.75, \infty)}$$

Hence, for instance:

```
findInterval(c(0, 0.2, 0.25, 0.4, 0.66, 1), c(-Inf, 0.25, 0.5, 0.75, Inf))
## [1] 1 1 2 2 3 4
```

Exercise 5.8 *Refer to the manual of* ***`findInterval`*** *to verify the function's behaviour when we do not include* $\pm\infty$ *as end points and how to make* ∞ *classified as a member of the 4th interval.*

Exercise 5.9 *Using a call to* ***`findInterval`****, write a statement that generates a logical vector whose* i*-th element indicates whether* `x[i]` *is in the interval* $[0.25, 0.5]$*. Was this easier to write than an expression involving* `` `<=` `` *and* `` `>=` `` *?*

5.4.3 Splitting Vectors into Subgroups

`split``(x, z)` can take the output of **`match`** or **`findInterval`** (and many other operations) and divide the elements in a vector `x` into subgroups corresponding to identical `z`s.

For instance, we can assign people into groups determined by their favourite dish:

```
name <- c("Graham", "John", "Terry", "Eric",  "Michael", "Terry")
food <- c("bacon",  "spam", "spam",  "eggs",  "spam",    "beans")
split(name, food)  # group names with respect to food
## $bacon
## [1] "Graham"
##
## $beans
## [1] "Terry"
##
## $eggs
## [1] "Eric"
##
## $spam
## [1] "John"    "Terry"   "Michael"
```

The result is a named list with labels determined by the unique elements in the 2nd vector.

Another example: here are some numbers pigeonholed into the four previously mentioned intervals:

```
x <- c(0, 0.2, 0.25, 0.4, 0.66, 1)
split(x, findInterval(x, c(-Inf, 0.25, 0.5, 0.75, Inf)))
## $`1`
## [1] 0.0 0.2
##
## $`2`
## [1] 0.25 0.40
##
## $`3`
## [1] 0.66
##
## $`4`
## [1] 1
```

Missing values in the second argument will result in the corresponding values in the first argument ignored. Also, unsurprisingly, recycling rule is applied when necessary.

We can also split `x` into groups defined by a combination of levels of two or more variables `z1`, `z2`, etc., by calling **`split`**`(x, list(z1, z2, ...))`.

Example 5.10 *The built-in* `ToothGrowth` *is a named list (with some extra attributes that makes us rather call it a data frame; see Chapter 12) represents the results of an experimental study involving 60 guinea pigs. The experiment's aim was to measure the effect of different vitamin C supplement types and* `doses` *on the growth of the rodents' teeth* `lengths`*:*

```
ToothGrowth <- as.list(ToothGrowth)  # it is a list, but with extra attribs
ToothGrowth[["supp"]] <- as.character(ToothGrowth[["supp"]])  # was: factor
print(ToothGrowth)
## $len
##  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 16.5 16.5 15.2 17.3
## [15] 22.5 17.3 13.6 14.5 18.8 15.5 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5
## [29] 23.3 29.5 15.2 21.5 17.6  9.7 14.5 10.0  8.2  9.4 16.5  9.7 19.7 23.3
## [43] 23.6 26.4 20.0 25.2 25.8 21.2 14.5 27.3 25.5 26.4 22.4 24.5 24.8 30.9
## [57] 26.4 27.3 29.4 23.0
##
## $supp
##  [1] "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC"
## [15] "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC"
## [29] "VC" "VC" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ"
## [43] "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ"
## [57] "OJ" "OJ" "OJ" "OJ"
##
## $dose
##  [1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.0 1.0 1.0 1.0 1.0 1.0 1.0
```

(continues on next page)

(continued from previous page)

```
## [18] 1.0 1.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 0.5 0.5 0.5 0.5
## [35] 0.5 0.5 0.5 0.5 0.5 0.5 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0
## [52] 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0
```

We can split `len` *with respect to the combinations of* `supp` *and* `dose` *(also called* interactions*) by calling:*

```
split(ToothGrowth[["len"]], ToothGrowth[c("supp", "dose")], sep="_")
## $OJ_0.5
##  [1] 15.2 21.5 17.6  9.7 14.5 10.0  8.2  9.4 16.5  9.7
##
## $VC_0.5
##  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0
##
## $OJ_1
##  [1] 19.7 23.3 23.6 26.4 20.0 25.2 25.8 21.2 14.5 27.3
##
## $VC_1
##  [1] 16.5 16.5 15.2 17.3 22.5 17.3 13.6 14.5 18.8 15.5
##
## $OJ_2
##  [1] 25.5 26.4 22.4 24.5 24.8 30.9 26.4 27.3 29.4 23.0
##
## $VC_2
##  [1] 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5 23.3 29.5
```

Other synonyms are of course possible, e.g., `split(ToothGrowth[[1]], ToothGrowth[-1])`, `split(ToothGrowth[[1]], list(ToothGrowth[[2]], ToothGrowth[[3]]))`*, etc. However, we should meditate upon our conscious use of double vs single square brackets here.*

Functions such as `Map` *described in Section 7.2 will enable us to compute any summary statistics within groups (e.g., the within-group averages like with "*`SELECT AVG(len) FROM ToothGrowth GROUP BY supp, dose`*" in SQL). We are in no hurry. However, as an appetiser, let us feed the* `boxplot` *function with a list of vectors; see Figure 5.1.*

```
boxplot(split(ToothGrowth[["len"]], ToothGrowth[c("supp", "dose")], sep="_"))
```

Note `unsplit` can be used to revoke the effects of `split`. In particular, later we will get used to calling `unsplit(Map(some_transformation, split(x, z)), z)` to modify the values in `x` independently in each group defined by `z` (e.g., standardise the variables within each class separately).

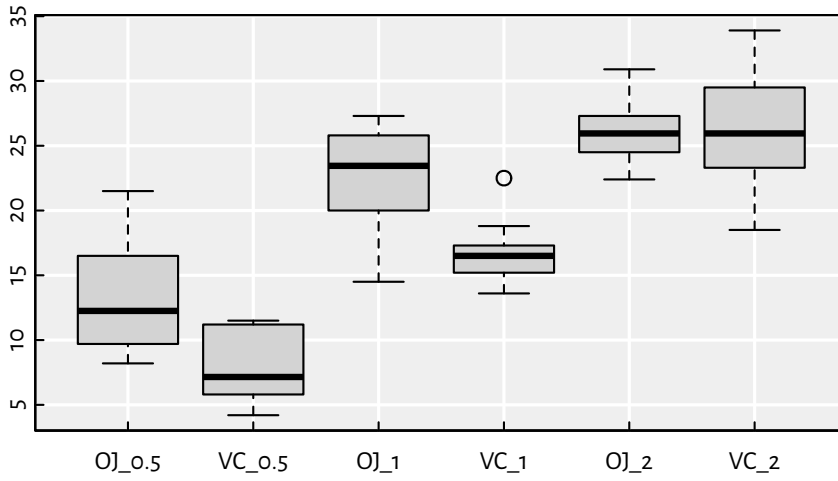

Figure 5.1: Box-and-whisker plots of `len` split by `supp` and `dose` (the `ToothGrowth` dataset)

5.4.4 Ordering Elements

The **`order`** function finds the ordering permutation of a given vector, i.e., a sequence of indexes which leads to a sorted version thereof.

```
x <- c(1024, 7, 42, 666, 0, 32787)
(o <- order(x))  # the ordering permutation of x
## [1] 5 2 3 4 1 6
x[o]  # ordered version of x
## [1]     0     7    42   666  1024 32787
```

Note that `o[1]` is the index of the smallest element in `x`, `o[2]` is the position of the 2nd smallest, …, and `o[`**`length`**`(o)]` is the index of the greatest value. Hence, e.g., `x[o[1]]` is equivalent to **`min`**`(x)`.

Another example:

```
x <- c("b", "a", "abs", "bass", "aaargh", "aargh", "aaaargh")
(o <- order(x))
## [1] 2 7 5 6 3 1 4
x[o]
## [1] "a"       "aaaargh" "aaargh"  "aargh"   "abs"     "b"       "bass"
```

Here, as `x` is a character vector, the ordering is lexicographical (like in a dictionary), because this is exactly how `` `<=` `` on strings works.

Note The ordering permutation that **`order`** returns is unique (that is why we call it *the* permutation) even for inputs containing duplicated elements. Owing to the use of a *stable* sorting algorithm, ties (repeated elements) will be listed in the order of occurrence.

```
order(c(10, 20, 40, 10, 10, 30, 20, 10, 10))
## [1] 1 4 5 8 9 2 7 6 3
```

Above we have, e.g., five 10s at positions 1, 4, 5, 6, 9. These five indexes are guaranteed to be listed in this very order.

Ordering can also be performed in a nonincreasing manner:

```
x[order(x, decreasing=TRUE)]
## [1] "bass"    "b"       "abs"     "aargh"   "aaargh"  "aaaargh" "a"
```

Note A call to **`sort`**`(x)` is equivalent to `x[`**`order`**`(x)]`, but the former function can be faster in some scenarios. For instance, one of its arguments can induce a *partially* sorted vector which can be useful if we only seek a few order statistics (e.g., the seven smallest values). Speed is rarely a bottleneck in the case of sorting (when it is, we have a problem!), this is why we will not bother ourselves with such topics until the last part of this pleasant book. Currently, we aim at expanding our repertoire of skills and abilities, so that we can implement anything we can think of (rapid prototyping with the least footprint).

Exercise 5.11 ***`is.unsorted`****`(x)`* *can be used to determine if the elements in a given vector are… not sorted with respect to* **`<=`**. *Write an R expression that generates the same result by referring to the* ***`order`*** *function. Also, assuming that* `x` *is numeric, do the same by means of a call to* ***`diff`***.

Note Looking at **`help`**`("order")`, we see that it also accepts one or more arguments via the dot-dot-dot parameter, "`...`". This way, we can sort a vector with respect to many criteria. If there are ties (equal observations) in the first variable, they will be resolved by the order of elements in the second variable. This is most useful for rearranging the rows of a data frame, which we will exercise in Chapter 12.

```
x  <- c( 10,  20,  30,  40,  50,  60)
y1 <- c("a", "b", "a", "a", "b", "b")
y2 <- c("w", "w", "v", "u", "u", "v")
x[order(y1)]
## [1] 10 30 40 20 50 60
x[order(y2)]
## [1] 40 50 30 60 10 20
x[order(y1, y2)]
```

(continues on next page)

(continued from previous page)

```
## [1] 40 30 10 50 60 20
x[order(y2, y1)]
## [1] 40 50 30 60 10 20
```

Note (*) Calling **order** on a permutation (a vector that is an arbitrary arrangement of *n* consecutive natural numbers) determines its *inverse*.

```
x <- c(10, 30, 40, 20, 10, 10, 50, 30)
order(x)
## [1] 1 5 6 4 2 8 3 7
order(order(x))  # inverse of the above permutation
## [1] 1 5 7 4 2 3 8 6
(x[order(x)])[order(order(x))]  # we get x again
## [1] 10 30 40 20 10 10 50 30
```

Note that **order**(**order**(x)) can be considered as a way to *rank* all the elements in x. For instance, the 3rd value in x, 40, is assigned rank 7: it is the 7th smallest value in this vector. Note that this breaks the ties at a first-come-first-served basis. But we can also write:

```
order(order(x, runif(length(x))))  # ranks with ties broken at random
## [1] 2 5 7 4 3 1 8 6
```

For different variations of these, see the **rank** function.

Exercise 5.12 *Recall that* ***sample****(x) returns a pseudorandom permutation of elements of a given vector unless x is a single positive number. Write an expression that always yields a proper rearrangement, regardless of the size of x.*

5.4.5 Identifying Duplicates

Whether any element in a vector was already listed in the sequence, can be verified by calling:

```
x <- c(10, 20, 30, 20, 40, 50, 50, 50, 20, 20, 60)
duplicated(x)
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

This can be used to remove repeated observations; see also **unique**. Note that the value that this function returns is not guaranteed to be sorted (unlike in some other languages/libraries).

Exercise 5.13 *What can be the use case of a call to* ***match****(x,* ***unique****(x))?*

Exercise 5.14 *Given two named lists x and y which we treat as key-value pairs, determine their set-theoretic union (with respect to the keys), for example:*

```
x <- list(a=1, b=2)
y <- list(c=3, a=4)
z <- ...to.do...  # combine x and y
str(z)
## List of 3
##  $ a: num 4
##  $ b: num 2
##  $ c: num 3
```

5.4.6 Counting Index Occurrences

`tabulate` takes a vector of values from a set of small positive integers (e.g., indexes) and determines their number of occurrences:

```
x <- c(2, 4, 6, 2, 2, 2, 3, 6, 6, 3)
tabulate(x)
## [1] 0 4 2 1 0 3
```

In other words, there are 0 ones, 4 twos, ..., and 3 sixes.

Exercise 5.15 *Using a call to **`tabulate`** (amongst others), return a named vector with the number of occurrences of each unique element in a character vector. For example:*

```
y <- c("a", "b", "a", "c", "a", "d", "e", "e", "g", "g", "c", "c", "g")
result <- ...to.do...
print(result)
## a b c d e g
## 3 1 3 1 2 3
```

5.5 Preserving and Losing Attributes

As attributes are conceived as extra data, it is up to a function's authors what they will decide to do with them. Generally, it is safe to assume that much thought has been put into the design of base R functions. Oftentimes, they behave quite reasonably. This is why we are going to spend some time now exploring their approaches to the handling of attributes.

Namely, for functions and operators that aim at transforming vectors passed as their inputs, the assumed strategy may be to:

- ignore the input attributes completely,

- equip the output object with the same set of attributes, or
- take care only of some special attributes such as `names`, if that makes sense.

Below we explore some common patterns; see also Section 1.3 in [50].

5.5.1 `c`

First, **`c`** drops[5] all attributes except `names`:

```
(x <- structure(1:4, names=c("a", "b", "c", "d"), attrib1="<3"))
## a b c d
## 1 2 3 4
## attr(,"attrib1")
## [1] "<3"
c(x)  # only `names` are preserved
## a b c d
## 1 2 3 4
```

We can therefore end up calling this function chiefly for this nice side effect. Also recall that **`unname`** drops the labels.

```
unname(x)
## [1] 1 2 3 4
## attr(,"attrib1")
## [1] "<3"
```

5.5.2 `as.something`

`as.vector`, **`as.numeric`**, and similar drop all attributes in the case where the output is an atomic vector, but it might not necessarily do so in other cases (because they are S3 generics; see Chapter 10).

```
as.vector(x)  # drops all attributes if x is atomic
## [1] 1 2 3 4
```

5.5.3 Subsetting

Subsetting with **`[`** (except where the indexer is not given) drops all attributes but `names` (as well as `dim` and `dimnames`; see Chapter 11), which is adjusted accordingly:

```
x[1]    # subset of labels
## a
## 1
```

(continues on next page)

[5] To be precise, we mean the default S3 method of **`c`** here; compare Section 10.2.4.

(continued from previous page)

```
x[[1]]  # this always drops the labels
## [1] 1
```

The replacement version of the index operator can be used to modify the values in an existing vector whilst preserving all the attributes. In particular, skipping the indexer will allow us to replace all the elements:

```
y <- x
y[] <- c("u", "v")  # note that c("u", "v") has no attributes at all
print(y)
##   a   b   c   d
## "u" "v" "u" "v"
## attr(,"attrib1")
## [1] "<3"
```

5.5.4 Vectorised Functions

Vectorised unary functions tend to copy all the attributes.

```
round(x)
## a b c d
## 1 2 3 4
## attr(,"attrib1")
## [1] "<3"
```

Binary operations should get the attributes from the longer input or both (with the first argument preferred to the second) if they are of equal sizes.

```
y <- structure(c(1, 10), names=c("f", "g"), attrib1=":|", attrib2=":O")
y * x  # x is longer
##  a  b  c  d
##  1 20  3 40
## attr(,"attrib1")
## [1] "<3"
y[c("h", "i")] <- c(100, 1000)  # add two new elements at the end
y * x
##    f    g    h    i
##    1   20  300 4000
## attr(,"attrib1")
## [1] ":|"
## attr(,"attrib2")
## [1] ":O"
x * y
##    a    b    c    d
```

(continues on next page)

(continued from previous page)

```
##    1   20  300 4000
## attr(,"attrib1")
## [1] "<3"
## attr(,"attrib2")
## [1] ":O"
```

Also, refer to Section 9.4.6 for a way to copy all the attributes from one object to another.

Important Even in base R the above rules are not enforced strictly. We consider them bugs that should be, for the time being, treated as features (with which we need to learn to live as they have not been fixed for years). But there is still hope.

As far as third-party extension packages are concerned, suffice it to say that a lot of R programmers do not know what attributes are at all! It is always best to refer to the documentation, perform some experiments, and/or manually assure the preservation of the data we care about.

5.6 Exercises

Exercise 5.16 *Answer the following questions (contemplate first, then use R to find the answer):*

- *What is the result of "`x[c()]`?" Is it the same as "`x[]`"?*
- *Is "`x[c(1, 1, 1)]`" equivalent to "`x[1]`"?*
- *Is "`x[1]`" equivalent to "`x["1"]`"?*
- *Is "`x[c(-1, -1, -1)]`" equivalent to "`x[-1]`"?*
- *What does "`x[c(0, 1, 2, NA)]`" do?*
- *What does "`x[0]`" return?*
- *What does "`x[1, 2, 3]`" do?*
- *What about "`x[c(0, -1, -2)]`" and "`x[c(-1, -2, NA)]`"?*
- *Why "`x[NA]`" is so significantly different from "`x[c(1, NA)]`"?*
- *What is "`x[c(FALSE, TRUE, 2)]`"?*
- *What will we obtain by calling "`x[x<min(x)]`"?*
- *What about "`x[length(x)+1]`"?*
- *Why "`x[min(y)]`" is probably a mistake? What could it mean? How can it be fixed?*

- *Why cannot we mix indexes of different types and write "`x[c(1, "b", "c", 4)]`"? Or can we?*
- *Why would we call "`as.vector(na.omit(x))`" instead of just `na.omit(x)`?*
- *What is the difference between `sort` and `order`?*
- *What is the type and the length of the object returned by a call to "`split(a, u)`"? What about "`split(a, c(u, v))`"?*
- *How to get rid of the 7th element from a list of ten elements?*
- *How to get rid of the 7th, 8th, and 9th element from a list of ten elements?*
- *How to get rid of the 7th element from an atomic vector of ten elements?*
- *If `y` is a list, by how many elements "`y[c(length(y)+1, length(y)+1, length(y)+1)] <- list(1, 2, 3)`" will extend it?*

Exercise 5.17 *If `x` is an atomic vector of length `n`, "`x[5:n]`" obviously extracts everything from the 5th element to the end. Does it though? Check what happens when `x` is of length less than five, including `0`. List different ways to correct this expression so that it makes (some) sense in the case of shorter vectors.*

Exercise 5.18 *Similarly, "`x[length(x) + 1 - 5:1]`" is supposed to return the last five elements in `x`. Propose a few alternatives that are correct also for short `x`s.*

Exercise 5.19 *Given a numeric vector, fetch its five largest elements. Make sure the code works for vectors of length less than five.*

Exercise 5.20 *We can compute a* trimmed *mean of some `x` by setting the `trim` argument to the `mean` function. Compute a similar robust estimator of location – the* p*-winsorised mean, $p \in [0, 0.5]$ defined as the arithmetic mean of all elements in `x` clipped to the $[Q_p, Q_{1-p}]$ interval, where Q_p is the vector's* p*-quantile; see `quantile`. For example, if `x` is (8, 5, 2, 9, 7, 4, 6, 1, 3), we have $Q_{0.25} = 3$ and $Q_{0.75} = 7$ and hence the 0.25-winsorised mean will be equal to the arithmetic mean of (7, 5, 3, 7, 7, 4, 6, 3, 3).*

Exercise 5.21 *Let `x` and `y` be two vectors of the same length,* n*, and no ties. Compute the Spearman rank correlation coefficient given by:*

$$\varrho(\mathbf{x}, \mathbf{y}) = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)},$$

where $d_i = r_i - s_i$, $i = 1, \ldots, n$, and r_i and s_i denote the rank of x_i and y_i, respectively. See also the built-in `cor`.

Exercise 5.22 *(*) Given two vectors `x` and `y` of the same length* n*, a call to `approx(x, y, ...)` can be used to interpolate linearly between the points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. We can use it whenever we wish to generate new `y`s for previously unobserved `x`s (somewhere "in-between" the data we already have). Moreover, `spline(x, y, ...)` can perform a cubic spline interpolation, which is smoother; see Figure 5.2.*

```
x <- c(1, 3,  5, 7, 10)
y <- c(1, 15, 25, 6,  0)
x_new <- seq(1, 10, by=0.25)
y_new1 <- approx(x, y, xout=x_new)[["y"]]
y_new2 <- spline(x, y, xout=x_new)[["y"]]
plot(x, y, ylim=c(-10, 30))  # the points to interpolate between
lines(x_new, y_new1, col="red", lty=2)   # linear interpolation
lines(x_new, y_new2, col="blue", lty=4)  # cubic interpolation
```

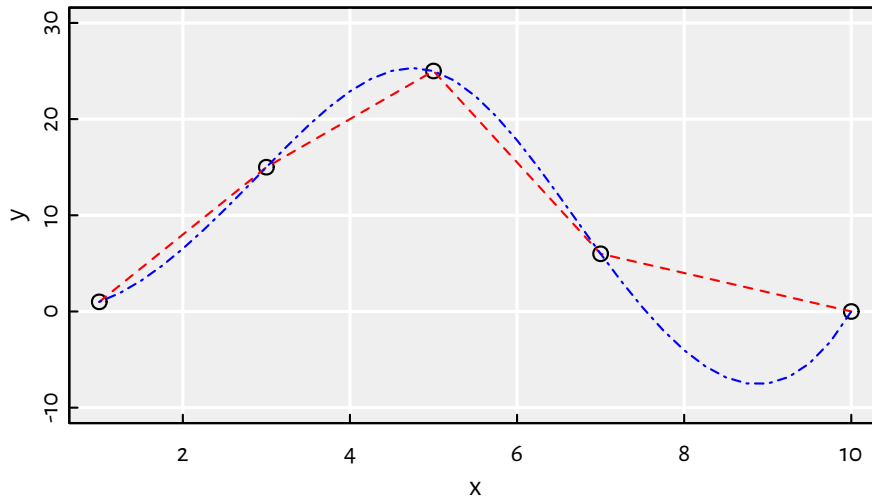

Figure 5.2: Piecewise linear and cubic spline interpolation.

Using a call to one of the above, perform the missing data imputation in the `euraud-20200101-20200630.csv`[6]*, e.g., the blanks in* `(0.60, 0.62, NA, 0.64, NA, NA, 0.58)` *should be filled so as to obtain* `(0.60, 0.62, 0.63, 0.64, 0.62, 0.60, 0.58)`*.*

Exercise 5.23 *Given some* $1 \leq$ `from` $\leq$ `to` $\leq n$*, use* **`findInterval`** *to generate a logical vector of length* n *with* `TRUE` *elements only at indexes between* `from` *and* `to`*, inclusive.*

Exercise 5.24 *Implement expressions that yield the same results as calls to* **`which`**, **`which.min`**, **`which.max`**, *and* **`rev`** *functions. What is the difference between* `x[x>y]` *and* `x[which(x>y)]`*? What about* `which.min(x)` *vs* `which(x == min(x))`*?*

Exercise 5.25 *Given two equal-length vectors* `x` *and* `y`*, fetch the value from the former that corresponds to the smallest value in the latter. Write three versions of such an expression, each dealing with potential ties in* `y` *differently, for example:*

```
x <- c("a", "b", "c", "d", "e", "f")
y <- c( 3,   1,   2,   1,   1,   4)
```

[6] https://github.com/gagolews/teaching-data/raw/master/marek/euraud-20200101-20200630.csv

should choose either the first (`"b"`), last (`"e"`), or random (`"b"`, `"d"`, `"e"` with equal probability) element from `x` fulfilling the above property. Make sure your code works for `x` being of type character or numeric as well as an empty vector.

Exercise 5.26 *Implement an expression that yields the same result as `duplicated(x)` for a numeric vector `x`, but using `diff` and `order`.*

Exercise 5.27 *Based on `match` and `unique`, implement your own versions of `union(x, y)`, `intersect(x, y)`, `setdiff(x, y)`, `is.element(x, y)`, and `setequal(x, y)` for `x` and `y` being non-empty numeric vectors.*

6

Character Vectors

Text is a universal, portable, economic, and efficient means of interacting between humans and computers as well as exchanging data between programs or APIs. This book is 99% made of text. And, wow, how much useful knowledge is in it, innit?

6.1 Creating Character Vectors

6.1.1 Inputting Individual Strings

Specific character strings are delimited either by a pair of double quotes or a pair of single quotes (apostrophes).

```
"a string"
## [1] "a string"
'another string'  # and of course neither 'like this" nor "like this'
## [1] "another string"
```

The only difference between these two lies in the fact that we cannot directly include, e.g., an apostrophe in a single quote-delimited string. On the other hand, `"'tis good ol' spam"` and `'I "love" bacon'` are both okay.

However, we may always use *escape sequences* to embrace characters whose inclusion might otherwise be difficult or impossible.

R uses the backslash, "\", as the escape character, in particular:

- `\"` inputs the double quote character,
- `\'` – single quote,
- `\\` – backslash,
- `\n` – new line.

```
(x <- "I \"love\" bacon\n\\\"/")
## [1] "I \"love\" bacon\n\\\"/"
```

The **`print`** function (which was implicitly called to display the above object) does not reveal the special meaning of the escape sequences. Rather, **`print`** outputs strings in

the very way which we ourselves would follow when inputting them. The number of characters in x is 18, and not 23:

```
nchar(x)
## [1] 18
```

To display the string as-it-really-is, we call:

```
cat(x)
## I "love" bacon
## \"/
```

Raw character constants, where the backslash character's special meaning is disabled, can be entered using the notation like `r"(...)"`, `r"{...}"`, `r"[...]"`, `r"----(...)----"`, etc.; see **`help`**`("Quotes")`. These can be useful when inputting regular expressions (see below).

```
x <- r"(spam\n\\\"maps)"
print(x)
## [1] "spam\\n\\\\\\\"maps"
cat(x)
## spam\n\\\"maps
```

... and of course the string version of the missing value marker is "`NA_character_`".

Note (*) Some output devices may support the following codes that control the position of the caret (text cursor):

- `\b` – backspace (move cursor one column to the left),
- `\t` – tab (advance to the next tab stop, e.g., a multiply of 8),
- `\r` – carriage return (move to the beginning of the current line).

```
cat("abc\bd\tef\rg\nhij")
## gbd     ef
## hij
```

These can be used on unbuffered outputs (e.g., *stderr*; see Section 8.3.5) to display the status of the current operation (a simple "animated" progress bar, the print-out of the ETA, or the % completed).

Further, certain terminals can also understand the ECMA-48/ANSI-X3.64 escape sequences[1] of the form "`\u001b[...`" to further control the cursor's position, text colour, and even style. For example, "`\u001b[1;31m`" outputs red bold text and "`\u001b[0m`" re-

[1] https://en.wikipedia.org/wiki/ANSI_escape_code

sets the settings to default. Give, e.g., "**cat**("\u001b[1;31mspam\u001b[0m")" or "**cat**("\u001b[5;36m\u001b[Abacon\u001b[Espam\u001b[0m")" a try.

Note (*) The Unicode standard 15.0 (version dated September 2022) defines over 149,186 characters, i.a., letters from different scripts, mathematical symbols, and emojis. Each of them is assigned a unique numeric identifier; see the Unicode Character Code Charts[2]. For example, the *inverted exclamation mark* (see the Latin-1 Supplement section therein) has been mapped to hexadecimal code 0xA1 (or 161 decimally). Knowing this magic number allows us to specify a Unicode code point using one of the following escape sequences:

- \uxxxx – codes using four hexadecimal digits,
- \Uxxxxxxxx – codes using eight hexadecimal digits.

For instance:

```
cat("!\u00a1!\U000000a1!")
## !¡!¡!
```

All R installations allow for working with Unicode strings (more precisely, UTF-8) – a super-encoding which is native to most Unix-like boxes (including GNU/Linux and m**OS). Other operating systems may use some 8-bit encoding as the system one (e.g., latin1 or cp1252), but they can be mixed with Unicode seamlessly. See **help**("Encoding"), **help**("iconv"), and [21] for discussion.

Nevertheless, certain output devices (web browsers, LaTeX renderers, text terminals) might be unable to display each and every Unicode character, e.g., due to some fonts missing. As far as the processing of character data is concerned, though, this does not matter: R does it with its eyes closed.

For example, in the PDF version[3] of this joyful book, none of the following Unicode glyphs are displayed properly, because yours cordially did not care about installing appropriate fonts in his XeLaTeX distribution. However, its HTML variant[4], which is generated from exactly the same source files as the former, will likely be rendered by the kind reader's web browser as intended.

```
cat("\U0001f642\u2665\u0bb8\U0001f923\U0001f60d\u2307")
## 🙂♥௸🤣😍⌇
```

---

[2] https://www.unicode.org/charts/
[3] https://deepr.gagolewski.com/deepr.pdf
[4] https://deepr.gagolewski.com

6.1.2 Many Strings, One Object

Less trivial character vectors (meaning, of length greater than one) can be constructed by means of, e.g., **c** or **rep**[5].

```
(x <- c(rep("spam", 3), "bacon", NA_character_, "spam"))
## [1] "spam"  "spam"  "spam"  "bacon" NA      "spam"
```

Thus, a character vector is in fact a sequence of sequences of characters. The total number of strings can be fetched, as usual, with the **length** function. However, the length of each individual string may be read via the vectorised **nchar**.

```
length(x)  # how many strings?
## [1] 6
nchar(x)   # the number of code points in each string
## [1]  4  4  4  5 NA  4
```

6.1.3 Concatenating Character Vectors

paste can be used to concatenate (join) the corresponding elements of two or more character vectors:

```
paste(c("a", "b", "c"), c("1", "2", "3"))  # sep=" " by default
## [1] "a 1" "b 2" "c 3"
paste(c("a", "b", "c"), c("1", "2", "3"), sep="")  # see also paste0
## [1] "a1" "b2" "c3"
```

The function is deeply vectorised:

```
paste(c("a", "b", "c"), 1:6, c("!", "?"))  # implicit coercion of numbers
## [1] "a 1 !" "b 2 ?" "c 3 !" "a 4 ?" "b 5 !" "c 6 ?"
```

We can also collapse (flatten, aggregate) a sequence of strings into a single string:

```
paste(c("a", "b", "c", "d"), collapse=",")
## [1] "a,b,c,d"
paste(c("a", "b", "c", "d"), 1:2, sep="", collapse="")
## [1] "a1b2c1d2"
```

Unfortunately (perhaps for the so-called convenience), **paste** does not treat missing values just like most other vectorised elementwise functions:

```
paste(c("A", NA_character_, "B"), "!", sep="")
## [1] "A!"  "NA!" "B!"
```

[5] Internally, there is a string cache (a hash table), so that multiple clones of the same string do not occupy more RAM than it is necessary.

6.1.4 Formatting Objects

Strings can also come into being by turning other R objects into text. For example, the quite customisable (see Chapter 10) **format** can be used for pretty-printing data in dynamically generated reports.

```
x <- c(123456.789, -pi, NaN)
format(x)
## [1] "123456.7890" "    -3.1416" "        NaN"
cat(format(x, digits=8, scientific=FALSE, drop0trailing=TRUE), sep="\n")
## 123456.789
##     -3.1415927
##            NaN
```

Moreover, **sprintf** is a workhorse for turning possibly many atomic vectors to strings. The numbers' precision, strings' widths and justification, etc., can be fully controlled. Its first argument is a format string; special escape sequences starting with percent sign, "`%`", serve as placeholders for the actual values. For instance, "`%s`" is meant to be replaced with a corresponding string and "`%f`" with a floating point value. Additional options are available, e.g., "`%10.2f`" is a number that, when converted to text, will occupy ten text columns[6], with two decimal digits of precision. Also, e.g., "`%1$s`", "`%2$s`", ... will insert the 1st, 2nd, ... argument as text.

```
sprintf("%.5f", pi)
## [1] "3.14159"
sprintf("%s%s", "a", c("X", "Y", "Z"))  # like paste(...)
## [1] "aX" "aY" "aZ"
sprintf("key=%s, value=%.1f", c("spam", "eggs"), c(100000, 0))
## [1] "key=spam, value=100000.0" "key=eggs, value=0.0"
sprintf("%.*f", 1:5, pi)  # variable precision
## [1] "3.1"     "3.14"    "3.142"   "3.1416"  "3.14159"
sprintf("%1$s, %2$s, %1$s, and %1$s", "spam", "bacon")  # numbered argument
## [1] "spam, bacon, spam, and spam"
```

See **help**("sprintf") for more details. I recommend. Marek Gagolewski.

6.1.5 Reading Text Data from Files

Given a raw text file, **readLines** can load it into memory so that it is represented as a character vector, with each line stored in a separate string.

```
f <- readLines(
    "https://github.com/gagolews/teaching-data/raw/master/README.md"
)
```

(continues on next page)

[6] Actually, this is only true for 8-bit native encodings. See also **sprintf** from the **stringx** package which takes the text width, and not the number of bytes, into account.

(continued from previous page)

```
print(head(f))
## [1] "# [Marek](https://www.gagolewski.com)'s Teaching and Training Data"
## [2] ""
## [3] "> *See the comment lines within the files themselves for"
## [4] "> a detailed description of each dataset.*"
## [5] ""
## [6] "*Good* datasets are actually hard to find!"
```

`writeLines` is its counterpart. There is also an option to read or write parts of files at a time, which me mention in Section 8.3.5. Also, **`cat`**`(..., append=TRUE)` can be used to create a text file incrementally.

6.2 Pattern Searching

6.2.1 Comparing Whole Strings

We have already reviewed a couple of ways to compare strings as a whole. For instance, the `` `==` `` operator implements elementwise testing:

```
c("spam", "spam", "bacon", "eggs") == c("spam", "eggs")  # recycling rule
## [1]  TRUE FALSE FALSE  TRUE
```

Moreover, in Section 5.4.1, we have introduced the **`match`** function and its derivative, the `` `%in%` `` operator, which are vectorised in a different way:

```
match(c("spam", "spam", "bacon", "eggs"), c("spam", "eggs"))
## [1]  1  1 NA  2
c("spam", "spam", "bacon", "eggs") %in% c("spam", "eggs")
## [1]  TRUE  TRUE FALSE  TRUE
```

Note We should stress that these are simple, bytewise comparisons of the corresponding code points and as such they might not be valid in, for example, natural language processing activities; compare [13]. In particular, German word *groß* is not deemed equal to *gross*, although we expect that should be the case, at least in a German language setting. Moreover, in the rare situations where we read Unicode-unnormalised data (say, not in the NFC form; see [12]), canonically equivalent strings may be considered as different.

6.2.2 Partial Matching

When only a consideration of the initial part of each string is required, we can call:

```
startsWith(c("s", "spam", "spamtastic", "spontaneous", "spoon"), "spam")
## [1] FALSE  TRUE  TRUE FALSE FALSE
```

Both the above and **endsWith** are applied elementwisely in case of many search prefixes/suffixes, just like in `==`.

Partial matching of strings can be performed with **charmatch**. This is a each-vs-all version of **startsWith**:

```
charmatch(c("s", "sp", "spam", "spams", "eggs", "bacon"), c("spam", "eggs"))
## [1]  1  1  1 NA  2 NA
charmatch(c("s", "sp", "spam", "spoo", "spoof"), c("spam", "spoon"))
## [1]  0  0  1  2 NA
```

Note that `0` designates that there was an ambiguity in the matching of a string to a given table.

Note (*) In `sec:to-do`, we discuss the more-advanced **match.arg** which is (unfortunately) frequently called from within other R functions, and in Section 9.4.2 and `sec:to-do`, we mention the (discouraged) partial matching of list labels and argument names in function calls.

6.2.3 Matching Anywhere Within a String

Fixed patterns can be also searched for anywhere within character strings using **grepl**:

```
x <- c("spam", "y spammite spam", "yummy SPAM", "sram")
grepl("spam", x, fixed=TRUE)  # fixed patterns, as opposed to regexes below
## [1]  TRUE  TRUE FALSE FALSE
```

Important Note that the order of arguments is like **grepl**`(needle, haystack)`, not the other way around. Also, this function is not vectorised with respect to the first argument.

Exercise 6.1 *Determine how can a call to* ***grep****`(y, x, value=FALSE)` and* ***grep****`(y, x, value=TRUE)` be implemented based on* ***grepl*** *and other operations that we are already familiar with.*

Note As a curiosity, **agrepl** performs *approximate* matching based on Levenshtein's edit distance. This can account for a small number of "typos".

```
agrepl("spam", x)
## [1]  TRUE  TRUE FALSE  TRUE
```

(continues on next page)

(continued from previous page)

```
agrepl("ham", x, ignore.case=TRUE)
## [1] TRUE TRUE TRUE TRUE
```

6.2.4 Using Regular Expressions (*)

Setting `perl=TRUE` allows for identifying occurrences of patterns specified by the PCRE2 regular expressions (regexes).

```
grepl("^spam", x, perl=TRUE)  # strings that begin with `spam`
## [1]  TRUE FALSE FALSE FALSE
grepl("(?i)^spam|spam$", x, perl=TRUE)  # begin or end; case ignored
## [1]  TRUE  TRUE  TRUE FALSE
```

Note For more details on regular expressions in general, see, e.g., [18]. The ultimate reference for PCRE2 pattern syntax is the `man`[7] page *`pcre2pattern(3)`*. R also gives access to ERE-like TRE library (see **`help("regex")`**), which is the default one. However, we discourage its use, because it is feature-poorer.

Exercise 6.2 *The **`list.files`** function generates the list of file names in a given directory that match a given regular expression. For instance, the following gives all CSV files in some directory.*

```
list.files("../../Projects/teaching-data/r/", r"(\.csv$)")  # or "\\.csv$"
## [1] "air_quality_1973.csv" "anscombe.csv"         "iris.csv"
## [4] "titanic.csv"          "tooth_growth.csv"     "trees.csv"
## [7] "world_phones.csv"
```

Write a single regular expression that matches file names ending with "`.csv`" or "`.csv.gz`". Also, write a regex that matches CSV files whose names do not begin with "`eurusd`".

6.2.5 Locating Pattern Occurrences

`regexpr` finds the first occurrence of a pattern in each string:

```
regexpr("spam", x, fixed=TRUE)
## [1]  1  3 -1 -1
## attr(,"match.length")
## [1]  4  4 -1 -1
## attr(,"index.type")
## [1] "chars"
```

(continues on next page)

[7] http://www.pcre.org/current/doc/html/pcre2pattern.html

(continued from previous page)

```
## attr(,"useBytes")
## [1] TRUE
```

In particular, there is a pattern occurrence starting at the 3th code point of the 2nd string in `x`. Moreover, there is no pattern match in the last string (denoted with -1).

The `match.length` attribute is generally more worthwhile when searching with regular expressions.

To locate all the matches, i.e., globally, we use **`gregexpr`**:

```
# `spam` followed by 0 or more letters, case insensitively
gregexpr("(?i)spam\\p{L}*", x, perl=TRUE)
## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1]  3 12
## attr(,"match.length")
## [1] 8 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1] 7
## attr(,"match.length")
## [1] 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"index.type")
```

(continues on next page)

(continued from previous page)

```
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

As we have noted in Section 4.4.2, wrapping the results in a list was a clever choice as the number of matches can obviously vary between strings.

In Section 7.2, we will take a look at the **`Map`** function, which, along with **`substring`** introduced below, can aid in getting the most out of such data. Meanwhile, let us just mention that **`regmatches`** extracts the matching substrings:

```
regmatches(x, gregexpr("(?i)spam\\p{L}*", x, perl=TRUE))
## [[1]]
## [1] "spam"
##
## [[2]]
## [1] "spammite" "spam"
##
## [[3]]
## [1] "SPAM"
##
## [[4]]
## character(0)
```

Note (*) Let us consider what happens when a regular expression contains parenthesised subexpressions (capture groups).

```
r <- "(?<basename>[^. ]+)\\.(?<extension>[^ ]*)"
```

The above regex consists of two such parts. The first one is labelled "`basename`" and is comprised of a number of arbitrary characters except for the space and the dot. The second group, named "`extension`" is a substring of anything but the space. These two are separated by a dot.

Such a pattern can be used for unpacking space-delimited lists of file names.

```
z <- "dataset.csv.gz something_else.txt spam"
regexpr(r, z, perl=TRUE)
## [1] 1
## attr(,"match.length")
## [1] 14
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

(continues on next page)

(continued from previous page)

```
## attr(,"capture.start")
##      basename extension
## [1,]        1         9
## attr(,"capture.length")
##      basename extension
## [1,]        7         6
## attr(,"capture.names")
## [1] "basename"  "extension"
gregexpr(r, z, perl=TRUE)
## [[1]]
## [1]  1 16
## attr(,"match.length")
## [1] 14 18
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
## attr(,"capture.start")
##      basename extension
## [1,]        1         9
## [2,]       16        31
## attr(,"capture.length")
##      basename extension
## [1,]        7         6
## [2,]       14         3
## attr(,"capture.names")
## [1] "basename"  "extension"
```

The `capture.*` attributes give us access to the matches to the individual capture groups, i.e., the `basename` and the `extension`.

Exercise 6.3 *(*) Check out the difference between the results generated by* ***`regexec`*** *and* ***`regexpr`*** *as well as* ***`gregexec`*** *and* ***`gregexpr`****.*

6.2.6 Replacing Pattern Occurrences

`sub` and **`gsub`** can replace first and all, respectively, matches to a pattern:

```
x <- c("spam", "y spammite spam", "yummy SPAM", "sram")
sub("spam", "ham", x, fixed=TRUE)
## [1] "ham"            "y hammite spam" "yummy SPAM"     "sram"
gsub("spam", "ham", x, fixed=TRUE)
## [1] "ham"            "y hammite ham" "yummy SPAM"    "sram"
```

Note (*) If a regex features some capture groups, matches thereto can be mentioned not only in the pattern itself, but also in the replacement string:

```
gsub("(\\p{L})\\p{L}\\1", "\\1", "aha egg gag NaN spam", perl=TRUE)
## [1] "a egg g N spam"
```

The above matches a letter (it is a capture group), another letter, and the former letter again. Each such palindrome of length 3 is replaced with just the repeated letter.

Exercise 6.4 *(*) Display the source code of* ***`glob2rx`*** *by calling* ***`print`****`(glob2rx)`* *and study how this function converts wildcards such as* `file???.*` *or* `*.csv` *to regular expressions that can be passed to, e.g.,* ***`list.files`****.*

6.2.7 Splitting Strings into Tokens

`strsplit` divides each string in a character vector into chunks. This time, though, the search pattern, specifying the token delimiter, is given as the second argument:

```
strsplit(c("spam;spam;eggs;;bacon", "spam"), ";", fixed=TRUE)
## [[1]]
## [1] "spam"  "spam"  "eggs"  ""      "bacon"
##
## [[2]]
## [1] "spam"
```

6.3 Other String Operations

6.3.1 Extracting Substrings

`substring` extracts parts of strings between given character position ranges.

```
substring("spammity spam", 1, 4)  # from 1st to 4th character
## [1] "spam"
substring("spammity spam", 10)  # from 10th to end
## [1] "spam"
substring("spammity spam", c(1, 10), c(4, 14))  # vectorisation
## [1] "spam" "spam"
substring(c("spammity spam", "bacon and eggs"), 1, c(4, 5))
## [1] "spam"  "bacon"
```

Note There is also a replacement (compare Section 9.4.6) version of the above:

```
x <- "spam, spam, bacon, and spam"
substring(x, 7, 11) <- "eggs"
print(x)
## [1] "spam, eggs, bacon, and spam"
```

Unfortunately, the number of characters in the replacement string should not exceed the length of the part being substituted (try "`chickpeas`" instead of "`eggs`"). However, substring replacement can be written as a composition of substring extraction and concatenation:

```
paste(substring(x, 1, 6), "chickpeas", substring(x, 11), sep="")
## [1] "spam, chickpeas, bacon, and spam"
```

Exercise 6.5 *Take the output generated by* ***`regexpr`*** *and apply* ***`substring`*** *to extract the pattern occurrences. If there is no match in some string, the corresponding output should be* `NA`*.*

6.3.2 Translating Characters

`tolower` and **`toupper`** can be used to convert between lower and upper case:

```
toupper("spam")
## [1] "SPAM"
```

Note Just like many other string operations in base R, these functions perform very simple character substitutions and they might not be valid in natural language processing tasks. For instance, *groß* is **not** converted to *GROSS*, which is the correct case folding in German.

Moreover, **`chartr`** translates individual characters:

```
chartr("\\", "/", "c:\\windows\\system\\cmd.exe")  # chartr(old, new, x)
## [1] "c:/windows/system/cmd.exe"
chartr("([S", ")]*", ":( :S :[")
## [1] ":) :* :]"
```

In the first line, we replace each backslash with slash. The second example replaces "`(`", "`[`", and "`S`" with "`)`", "`]`", and "`*`", respectively.

6.3.3 Ordering Strings

We have previously mentioned that operators such as `<` and `>=` as well as functions like **sort**, **order**, **rank**, but also **xtfrm**[8] are based on the lexicographic ordering of strings.

```
sort(c("chłodny", "hardy", "chladný", "hladný"))
## [1] "chladný" "chłodny" "hardy"   "hladný"
```

It is worth noting that the ordering is dependent on the currently selected locale, see **Sys.getlocale**("LC_COLLATE"). For instance, in the Slovak language setting, we would obtain "hardy" < "hladný" < "chladný" < "chłodny".

Note Many "structured" data items can be displayed or transmitted as human-readable strings. In particular, we know that **as.numeric** can be used to convert a string to a number. Moreover, in Section 10.3.1 we will discuss date-time objects such as "1970-01-01 00:00:00 GMT". We will be processing them with specialised functions such as **strptime** and **strftime**.

Important (*) As we have mentioned, many string operations in base R are not necessarily portable. The **stringx** package [22] defines drop-in, "fixed" replacements therefor. They are based on the International Components for Unicode (ICU[9]) library, which is a de facto standard for the processing of Unicode text, and the R package **stringi**; see [21].

```
# call install.packages("stringx") first
suppressPackageStartupMessages(library("stringx"))  # load the package
sort(c("chłodny", "hardy", "chladný", "hladný"), locale="sk_SK")
## [1] "hardy"   "hladný"  "chladný" "chłodny"
toupper("gro\u00DF")  # compare base::toupper("gro\u00DF")
## [1] "GROSS"
detach("package:stringx")  # unload the package
```

6.4 Other Atomic Vector Types (*)

We have discussed four vector types: logical, double, character, and list (the latter being a generic-recursive vector). To get the complete picture of the sequence-like

[8] See Section 12.3.1 for a use case.
[9] http://site.icu-project.org/

types in R, let us briefly mention `integer`, `complex`, and `raw` atomic types, so that we are not surprised when we encounter them.

6.4.1 Integer Vectors (*)

Integer scalars can be input manually by using the `L` suffix:

```
(x <- c(1L, 2L, -1L, NA_integer_))  # looks like numeric
## [1]  1  2 -1 NA
typeof(x)  # but is integer
## [1] "integer"
```

Some functions return them in certain contexts[10]:

```
typeof(1:10)  # seq(1, 10) as well, but not seq(1, 10, 1)
## [1] "integer"
as.integer(c(-1.1, 0, 1.9, 2.1))  # truncate/round towards 0
## [1] -1  0  1  2
```

In the vast majority of expressions, integer vectors behave like numeric ones, and are silently coerced to `double` if need be, so there is no real practical reason to distinguish between them (they are of *internal* interest, e.g., when writing C/C++ extensions; see `Chapter %s`). For example:

```
1L/2L  # like 1/2 == 1.0/2.0
## [1] 0.5
```

Note (*) R `integers` are 32-bit signed types. The `double` type can store more integers than them (with the maximal contiguously representable integer being 2^{53} vs $2^{31}-1$ in the former case; see Section 3.2.3):

```
as.integer(2^31-1) + 1L  # 32-bit integer overflow
## Warning in as.integer(2^31 - 1) + 1L: NAs produced by integer overflow
## [1] NA
as.integer(2^31-1) + 1 == 2^31 # integer+double == double - OK
## [1] TRUE
(2^53 - 1) + 1 == 2^53  # OK
## [1] TRUE
(2^53 + 1) - 1 == 2^53  # lost due to FP rounding, left result is 2^53 - 1
## [1] FALSE
```

---

[10] Actually, `1:10` returns an integer vector in a compact (ALTREP, see [41]) form; compare the results of the call to "`.Internal(inspect(1:10))`" and "`.Internal(inspect(seq(1, 10, 1)))`". This way, the whole vector does not have to be allocated which saves memory and time. At the R level, though, it behaves as any other integer (numeric) sequence.

Note Since R 3.0, there is support for vectors longer than $2^{31}-1$ elements. As there are no 64-bit integers in R, these are indexed by doubles anyway (as we have been doing all this time). Interestingly, `x[1.9]` is the same as `x[1]` and `x[-1.9]` means `x[-1]` (truncation of the fractional part). This is why the notation like `x[length(x)*0.2]` works regardless of whether the length of `x` is a multiple of 5 or not, which is neat.

6.4.2 Raw Vectors (*)

Vectors of type `raw` can store bytes, i.e., unsigned 8-bit integers, whose range is 0-255 (there are no raw `NA`s). For example:

```
as.raw(c(-1, 0, 1, 2, 0xc0, 254, 255, 256, NA))
## Warning: out-of-range values treated as 0 in coercion to raw
## [1] 00 00 01 02 c0 fe ff 00 00
```

They are displayed as two-digit hexadecimal (base-16) numbers. Also note that we may enter such numbers using the "`0x`" prefix.

There are only few functions that deal with such vectors: e.g., **`readBin`**, **`charToRaw`**, and **`rawToChar`**.

6.4.3 Complex Vectors (*)

We can also play with vectors of type `complex`, with "`1i`" representing the imaginary unit, $\sqrt{-1}$. Complex numbers appear in quite a few engineering or scientific applications, e.g., in physics, electronics, or signal processing and are (at least: should be) part of many university-level subjects or textbooks in mathematics[11].

```
c(0, 1i, pi+pi*1i, NA_complex_)
## [1] 0.0000+0.0000i 0.0000+1.0000i 3.1416+3.1416i             NA
```

Apart from the basic operators, mathematical and aggregation functions, procedures like **`fft`**, **`solve`**, **`qr`**, or **`svd`** can be fed with or produce such data. For more details, see **`help`**`("complex")` and some matrix examples in Chapter 11.

6.5 Exercises

Exercises marked with (*) might require tinkering with regular expressions or third-party R packages.

[11] Even the statistics/machine learning oriented ones, because of their heavy use of numerical computing, e.g., [14, 25].

Exercise 6.6 *Answer the following questions:*

- *How many characters are there in the string* `"ab\n\\\t\\\\\""`*? What about* `"-{ab\n\\\t\\\\\"-)}-"`*?*
- *What is the result of calling "*`paste(NA, 1:5, collapse="")`*"?*
- *What is the meaning of the following* `sprintf` *format strings:* `"%s"`*,* `"%20s"`*,* `"%-20s"`*,* `"%f"`*,* `"%g"`*,* `"%e"`*,* `"%5f"`*,* `"%5.2f%%"`*,* `"%.2f"`*,* `"%0+5f"`*, and* `"[%+-5.2f]"`*?*
- *What is the difference between* `regexpr` *and* `gregexpr`*? What does "g" in the latter name stand for?*
- *What is the result of a call to "*`grepl(c("spam", "spammity spam", "aubergines"), "spam")`*"?*
- *Is it always the case that "*`"Aaron" < "Zorro"`*"?*
- *If* `x` *is a character vector, is "*`x == x`*" always equal to* `TRUE`*?*
- *If* `x` *and* `y` *are character vectors of lengths* n *and* m*, respectively, what is the length of the output of "*`match(x, y)`*"?*
- *If* `x` *is a named vector, why there is a difference between "*`x[NA]`*" and "*`x[NA_character_]`*"?*
- *What is the difference between "*`x == y`*" and "*`x %in% y`*"?*

Exercise 6.7 *Let* `x`*,* `y`*, and* `z` *be atomic vectors and* `a` *and* `b` *be single strings. Generate the same results as "*`pastena(x, collapse=b)`*",* *"*`pastena(x, y, sep=a)`*",* *"*`pastena(x, y, sep=a, collapse=b)`*",* *"*`pastena(x, y, z, sep=a)`*",* *"*`pastena(x, y, z, sep=a, collapse=b)`*",* *assuming that* `pastena` *is a version of* `paste` *(which we do not have) that handles missing data in a way consistent with most other functions.*

Exercise 6.8 *Based on* `list.files` *and* `glob2rx`*, generate the list of all PDFs on your computer. Then, using* `file.size` *filter out the files smaller than* 10 MiB*.*

Exercise 6.9 *Read a text file that stores a long paragraph of some banal prose. Concatenate all the lines to form a single, long string. Using* `strwrap` *and* `cat`*, output the paragraph on the console, nicely formatted to fit an aesthetic width, say,* 60 *text columns.*

Exercise 6.10 *(*) Implement your own, simplified version of* `basename` *and* `dirname`*.*

Exercise 6.11 *(*) Implement an operation similar to* `trimws` *using the functions introduced in this chapter.*

Exercise 6.12 *(*) Write a regex that extracts all* words *from each string in a given character vector.*

Exercise 6.13 *(*) Write a regex that extracts, from each string in a character vector, all:*

- *integers numbers (signed or unsigned),*
- *floating-point numbers,*
- *numbers of any kind (including those in scientific notation),*
- *#hashtags,*

- *email@addresses,*
- *hyperlinks of the form http://... and https://....*

Exercise 6.14 *(*) What does `42i`, `42L`, and `0x42` stand for?*

Exercise 6.15 *(*) Check out **`stri_sort`** in the **`stringi`** package (or **`sort.character`** in **`stringx`**) for a way to obtain an ordering like `"a1"` < `"a2"` < `"a10"` < `"a11"` < `"a100"`.*

Exercise 6.16 *(*) In **`sprintf`**, the formatter `"%20s"` means that if a string is less than 20 bytes long, the remaining bytes will be replaced with spaces. Only for ASCII characters (English letters, digits, some punctuation marks, etc.) it is true that one character is represented by 1 byte. Other Unicode code points can take up between 2 and 4 bytes.*

```
cat(sprintf("..%6s..", c("abc", "1!<", "aßc", "qß©")), sep="\n")  # aligned?
## ..   abc..
## ..   1!<..
## ..  aßc..
## ..qß©..
```

*Use the **`stri_pad`** function from the **`stringi`** package to align the strings aesthetically. Alternatively, check out **`sprintf`** from **`stringx`**.*

Exercise 6.17 *(*) Implement an operation similar to **`stri_pad`** from **`stringi`** using the functions introduced in this chapter.*

7

Functions

R is a *functional language*, where functions play first fiddle. Each action we perform reduces itself to a call to some function, or a combination thereof.

So far we have been tinkering with dozens of available functions which are part of base R, with only few exceptions. They constitute the essential vocabulary that everyone *must* be able to speak fluently.

Any operation, be it **`sum`**, **`sqrt`**, or **`paste`**, when fed with a number of arguments, generates some (hopefully useful) return value.

```
sum(1:10)  # invoking `sum` on a specific argument
## [1] 55
```

From a user's perspective, each function is merely a tool. To achieve a goal at hand, we do not really have to care about what is going on under its hood, i.e., how the inputs are actually being transformed so that, after a couple of nanoseconds or hours, we can enjoy what has been yielded. This is very convenient: all we need to know is the function's specification which can be stated, for example, informally, in plain Polish or Malay, in its help page.

In this chapter, we will learn how to write our *own* functions. The use of this skill is a good development practice when we expect that some operations are to be executed many times but perhaps on different data.

Also, some R functions are meant to invoke other functions, for instance on every element in a list or every section of a data frame grouped by a qualitative variable, so it is good to learn know how we can specify a custom operation to be propagated thereover.

Example 7.1 *Given some objects (whatever):*

```
x1 <- runif(16)
x2 <- runif(32)
x3 <- runif(64)
```

when we want to apply the same action on different data, say, compute the root mean square, instead of re-typing almost identical expressions (or a bunch of them) over and over again:

```
sqrt(mean(x1^2))
## [1] 0.6545
```

(continues on next page)

(continued from previous page)

```
sqrt(mean(x2^2))  # the same second time - borderline okay
## [1] 0.56203
sqrt(mean(x3^2))  # tedious, barbarous, and error-prone
## [1] 0.57206
```

we can generalise *the operation to any object like* `x`*:*

```
rms <-                  # bound what follows to name `rms`
    function(x)         # a function that takes one parameter, `x`
        sqrt(mean(x^2))  # expression to transform the input to yield output
```

and then re-use it on different concrete *data instances:*

```
rms(x1)
## [1] 0.6545
rms(x2)
## [1] 0.56203
rms(x3)
## [1] 0.57206
```

or even combine it with other function calls:

```
rms(sqrt(c(x1, x2, x3)))^2
## [1] 0.50824
```

Important Does writing your own functions equal reinventing the wheel? Can everything be found on the internet these days (including on Stack Overflow, GitHub, or CRAN)?

Luckily, this is not the case. Otherwise, data analysts', researchers', and developers' lives could be considered monotonous, dreary, and uninspiring. Plus, sometimes it is much quicker to write a function from scratch than to get through the whole garbage dump from where, only occasionally, we can dig out some pearls. Not to mention the self-educative side: we become better programmers by crunching those exercises. We are advocating for minimalism here, remember?

This and many more other important issues in function design will be reflected upon in Chapter 9.

7.1 Creating and Invoking Functions

7.1.1 Anonymous Functions

Functions are usually created by means of the following notation:

```
function(args) body
```

First, `args` is a (possibly empty) list of comma-separated parameter names which are supposed to act as input variables.

Second, `body` is a *single* R expression which will be evaluated when the function is called. The value that this expression yields will constitute the function's output.

For example, here is a definition of a function which takes no inputs and generates a constant output:

```
function() 1
## function() 1
```

We thus created a *function* object. However, it has disappeared immediately thereafter, as we have not used it at all.

Any function, say, **`f`** can be invoked, i.e., evaluated on concrete data, by using the notation **`f`**`(arg1, ..., argn)`, where "`arg1, ..., argn`" are the arguments to be passed to **`f`**.

```
(function() 1)()  # invoking f like f(); here, no arguments are expected
## [1] 1
```

Only now we have obtained a return value.

Note (*) Calling **`typeof`** on a function object will report `"closure"` (for user-defined functions), `"builtin"`, or `"primitive"` (for some built-in, base ones), for the reasons that we explain in more detail[1] in Section 9.5.3:

```
typeof(function() 1)
## [1] "closure"
```

7.1.2 Named Functions

Function objects can be bound with names so that they can be referred to multiple times:

[1] In short: each function consists of a list of formal arguments, a body, an possibly (if it is a closure) an enclosing environment.

```
one <- function() 1  # one <- (function() 1)
```

We created an object named **one** (we use bold font to indicate that it is of type function, because functions are so important in R). We are very familiar with such a notation, as not since yesterday we are used to writing "`x <- 1`" etc.

Invoking **one**, which can be done by writing **one()**, will yield a return value:

```
one()  # (function() 1)()
## [1] 1
```

This output can be used in further computations, for instance:

```
0:2 - one()  # 0:2 - (function() 1)(), i.e., 0:2 - 1
## [1] -1  0  1
```

7.1.3 Passing Arguments To Functions

Functions with no arguments are kind of boring, thus let us distil a more serious operation:

```
concat <- function(x, y) paste(x, y, sep="")
```

Here we have created a mapping whose aim is to concatenate two objects by means of a specialised call to **paste**. Yours faithfully pleads guilty to multiplying entities needlessly, because it *should* not be a problem for anyone to write **paste**`(x, y, sep="")` each time. Yet, 'tis merely an illustration.

The **concat** function has two *parameters*, "x" and "y". Hence, calling it will require the provision of two *arguments*, which we put within round brackets and separate from each other by commas.

```
u <- 1:5
concat("spam", u)  # i.e., concat(x="spam", y=1:5)
## [1] "spam1" "spam2" "spam3" "spam4" "spam5"
```

Important Notice the distinction: parameters (also called formal arguments) are abstract, general, or symbolic; "something, anything that will be put in place of `x` when the function is invoked". By contrast, arguments (a.k.a. actual parameters) are concrete, specific, and real.

During the above call, `x` in the function's body is precisely `"spam"`, and nothing else. Also, the `u` object from the caller's environment is seen under the name `y` there. Most of the time (however, see Section 16.4), it is best to think of the function as being fed not with `u` per se, but the value that `u` is bound to, i.e., "`1:5`".

Also:

```
x <- 1:5
y <- "spam"
concat(y, x)  # concat(x="spam", y=1:5)
## [1] "spam1" "spam2" "spam3" "spam4" "spam5"
```

This is still a call to equivalent to **`concat(x=y, y=x)`**. The argument `x` is being assigned with the value of `y` from the calling environment, `"spam"`. Yes, one `x` is not the same as the other `x`, and which is which is unambiguously defined by the context. Understanding and being able to manipulate such abstractions is basic logic and common sense that everyone should master.

Exercise 7.2 *Write a function called* ***`standardise`*** *that takes a numeric vector `x` as argument and returns its standardised version, i.e., from each element in `x` subtract the sample arithmetic mean and then divide it by the standard deviation.*

Note Recall from Section 2.1.3 that, syntactically speaking, the following are perfectly valid alternatives to the positionally-matched call **`concat("spam", u)`**.

```
concat(x="spam", y=u)
concat(y=u, x="spam")
concat("spam", y=u)
concat(u, x="spam")
concat(x="spam", u)
concat(y=u, "spam")
```

However, the last two should particularly be avoided, for the sake of the readers' sanity. It is best to provide positionally-matched arguments before the keyword-based ones.

Also, in Section 10.5, we introduce the (overused) forward-pipe operator, **`` `|>` ``**, which enables the above to be written as "`"spam" |> concat(u)`".

7.1.4 Grouping Expressions with Curly Braces, `{`

We have been informed that a function's body is a *single* R expression whose evaluated value is passed to the user as its output. This may sound restrictive and contrast with what we have experienced so far. Rarely are we faced with such simple computing tasks and we have already seen R functions performing quite sophisticated operations.

It turns out that, grammatically, a single R expression can be arbitrarily complex (Chapter 15); we can use curly braces to group many calls that are to be evaluated one after another.

For instance:

```
{
    cat("first expression\n")
    cat("second expression\n")
    # ...
    cat("last expression\n")
}
## first expression
## second expression
## last expression
```

Note that we used four spaces to visually indent the constituents for greater readability (some developers prefer tabs over spaces, others find two or three spaces more urbane, but we do not). This single (compound) expression can now play a role of a function's body.

Important The last expression evaluated in a curly-braces delimited block will be considered its the output value.

```
x <- {
    1
    2
    3  # <--- last expression: will be taken as the output value
}
print(x)
## [1] 3
```

Note (*) The above code block can also be written more concisely by replacing newlines with semicolons, although with perhaps some loss in readability:

```
{1; 2; 3}
## [1] 3
```

In Section 9.4, we will give a few more details about `{`.

Example 7.3 *Here is a version of the above* ***concat*** *function which takes care of a more Chapter 2-style missing values' propagation:*

```
concat <- function(a, b)
{
    z <- paste(a, b, sep="")
    z[is.na(a) | is.na(b)] <- NA_character_
    z  # last expression in the block - return value
}
```

Example calls:

```
concat("a", 1:3)
## [1] "a1" "a2" "a3"
concat(NA_character_, 1:3)
## [1] NA NA NA
concat(1:6, c("a", NA_character_, "c"))
## [1] "1a" NA   "3c" "4a" NA   "6c"
```

Let us appreciate the fact that we could keep the code brief thanks to ***paste*** *and* `/` *implementing the recycling rule.*

Exercise 7.4 *Write a function called* ***normalise*** *that takes a numeric vector* `x` *and returns its version shifted and scaled to the* $[0, 1]$ *interval. To do so, from each element subtract the sample minimum and then divide it by the range, i.e., the difference between the maximum and the minimum. Avoid computing* ***min***`(x)` *twice.*

Exercise 7.5 *Write a function that applies the robust standardisation of a numeric vector: subtract the median and divide it by the median absolute deviation, 1.4826 times the median of the absolute differences between the values and their median.*

Note R is an open-source (free, libre) project – users are not only encouraged to run the software for whatever the purpose, but also study and modify its source code without any restrictions. This applies both to functions that we have authored ourselves:

```
print(concat)
## function(a, b)
## {
##     z <- paste(a, b, sep="")
##     z[is.na(a) | is.na(b)] <- NA_character_
##     z  # last expression in the block - return value
## }
## <bytecode: 0x562536c9b138>
```

and to the routines that are part of base R or any other extension packages:

```
print(union)
## function (x, y)
## {
##     u <- as.vector(x)
##     v <- as.vector(y)
##     unique(c(u, v))
## }
## <bytecode: 0x5625382f1a28>
## <environment: namespace:base>
```

Nevertheless, some functionality might be implemented in a compiled programming

language such as C, C++, or Fortran; notice a call to **`.Internal`** in the source code of **`paste`**, **`.Primitive`** in **`list`**, or **`.Call`** in **`runif`**. Therefore, we will sometimes have to dig a little bit deeper to access the underlying source code; see `Chapter %s` for more details.

7.2 Functional Programming

R is a *functional* programming language. As such, it shares a number of common features with other languages that emphasise on the role of function manipulation in software development (e.g., Common Lisp, Scheme, OCaml, Haskell, Clojure, F#). Let us explore them now.

7.2.1 Functions are Objects

R functions were given the right to a *fair go*; they are what we refer to as *first-class citizens*. In other words, our interaction with them is not limited to their invocation; we treat them as any other language objects. Namely, they can be:

- stored inside list objects:

```
list(identity, nrow, sum)  # a list with three elements of type function
## [[1]]
## function (x)
## x
## <bytecode: 0x562536828c30>
## <environment: namespace:base>
##
## [[2]]
## function (x)
## dim(x)[1L]
## <bytecode: 0x562537915320>
## <environment: namespace:base>
##
## [[3]]
## function (..., na.rm = FALSE)  .Primitive("sum")
```

  This is possible owing to the fact that lists, as we recall, can embrace R objects of *any* kind.

- created and then called inside another function's body:

```
euclidean_distance <- function(x, y)
{
    square <- function(z) z^2  # auxiliary/internal/helper function
```

(continues on next page)

(continued from previous page)

```
    sqrt(sum(square(x-y)))     # square root of the sum of squares
}

euclidean_distance(c(0, 1), c(1, 0))  # example call
## [1] 1.4142
```

This is why we tend to classify functions as representatives of *recursive* types (compare **`is.recursive`**).

- passed as arguments to other operations:

```
# Replaces missing values with a given aggregate
# of all non-missing elements:
fill_na <- function(x, filler_fun)
{
    missing_ones <- is.na(x)  # otherwise, we'd call is.na twice
    replacement_value <- filler_fun(x[!missing_ones])
    x[missing_ones] <- replacement_value
    x
}

fill_na(c(0, NA_real_, NA_real_, 2, 3, 7, NA_real_), mean)
## [1] 0 3 3 2 3 7 3
fill_na(c(0, NA_real_, NA_real_, 2, 3, 7, NA_real_), median)
## [1] 0.0 2.5 2.5 2.0 3.0 7.0 2.5
```

We call these *higher-order functions*.

Note More advanced techniques, which we will discuss later (i.e., closures, lazy evaluation, metaprogramming, etc.), will let the functions be:

- returned as other function's outputs (`sec:to-do`),
- equipped auxiliary data (`sec:to-do`),
- generated programmatically on the fly (`sec:to-do`), and
- modified at runtime (`sec:to-do`).

Below we review some noteworthy higher-order functions, in particular: **`do.call`** and **`Map`**. Many other ones will be introduced in due course or are left as an educative exercise.

7.2.2 Calling on Precomputed Arguments with `do.call`

The notation like **`f`**`(arg1, ..., argn)` has no monopoly over how we are supposed to call a function on a specific sequence of comma-delimited arguments: the latter do not have to be hardcoded.

Here is an alternative. We can first prepare a number of objects to be passed as **`f`**'s inputs, wrap them in a list `l`, and then invoke **`do.call`**`(`**`f`**`, l)` to get the same result.

```
words <- list(
    c("spam",      "bacon",  "eggs"),
    c("buckwheat", "quinoa", "barley"),
    c("ham",       "spam",   "spam")
)
do.call(paste, words)  # paste(words[[1]], words[[2]], words[[3]])
## [1] "spam buckwheat ham" "bacon quinoa spam"  "eggs barley spam"
do.call(cbind, words)  # column-bind; returns a matrix (explained later)
##      [,1]    [,2]        [,3]
## [1,] "spam"  "buckwheat" "ham"
## [2,] "bacon" "quinoa"    "spam"
## [3,] "eggs"  "barley"    "spam"
do.call(rbind, words)  # row-bind (explained later)
##      [,1]        [,2]     [,3]
## [1,] "spam"      "bacon"  "eggs"
## [2,] "buckwheat" "quinoa" "barley"
## [3,] "ham"       "spam"   "spam"
```

Note that the length and content of the list passed as the 2nd argument of **`do.call`** can be arbitrary (possibly unknown at the time of writing the code). See Section 12.1.2 for more use cases, e.g., ways to concatenate a list of data frames (perhaps produced by some complex chain of commands) into a single data frame.

If elements of the list are named, they will be matched to the corresponding keyword arguments.

```
x <- 2^(seq(-2, 2, length.out=101))
plot_opts <- list(col="red", lty="dashed", type="l")
do.call(plot, c(list(x, log2(x), xlab="x", ylab="log2(x)"), plot_opts))
## (the displaying of the plot has been suppressed)
```

Note that, e.g., `plot_opts` can now be reused in further calls to graphical functions. This is very convenient as it avoids repetitions.

7.2.3 Common Higher-Order Functions

There is an important class of higher-order functions that allow us to apply custom operations on consecutive elements of sequences without relying on loop-like statements, at least explicitly. They can be found in all functional programming languages

(e.g., Lisp, Haskell, Scala) and have been ported to various add-on libraries (**functools** in Python, more recent versions of the C++ Standard Library, etc.) or frameworks (Apache Spark and the like). Their presence reflects the obvious truth that some kinds of operations occur more frequently than other ones.

In particular:

- **Map** calls a function on each element of a sequence in order to transform:
  - their individual components (just like **sqrt**, **round**, or the unary `!` operator in R), or
  - the corresponding elements of many sequences so as to vectorise a given operation elementwisely (compare the binary `+` or **paste**),
- **Reduce** (also called accumulate) applies a binary operation to combine consecutive elements in a sequence, e.g., to generate the aggregates, like, totally (compare **sum**, **prod**, **all**, **max**) or cumulatively (compare **cumsum**, **cummin**),
- **Filter** creates a subset of a sequence that is comprised of elements that enjoy a given property (which we typically achieve in R by means of the `[` operator),
- **Find** locates the first element that fulfils some logical condition (compare **which**),

and so forth.

Below we will only focus on the **Map** function. The inspection of the remaining ones is left as an exercise. This is because, oftentimes, we can be better-off with their more R-ish versions (e.g., using the subsetting operator, `[`).

7.2.4 Vectorising Functions with Map

In data-centric computing, we are frequently faced with tasks that involve processing each and every element in a sequence independently, one after another. Such use cases can benefit from vectorised operations like those discussed in Chapter 2, Chapter 3, and Chapter 6.

Most of the functions that we have introduced in the preceding parts, unfortunately, cannot be applied on lists. For instance, if we try calling **sqrt** on a list, we will get an error, even if it is a list of numeric vectors only. One way to compute the square root of all elements would be to invoke **sqrt**(**unlist**(...)). It is a go-to approach if we wish to treat all the list's elements as one sequence. But this comes at a price of losing the list's structure.

We have also discussed some operations that are not vectorised with respect to all their arguments, even though they could have been designed this way, e.g., **grepl**.

The **Map** function[2] applies an operation on each element in a vector or the corresponding elements in a number of vectors. In many situations, it may be used as a more elegant alternative to **for** loops that we will introduce in the next chapter.

[2] Yes, the author is aware that **Map** was implemented using the slightly more primitive **mapply**, but we are not fond of the latter's having the SIMPLIFY argument set to TRUE by default.

First[3], a call to **`Map`**`(f, x)` yields a list whose *i*-th element is equal to **`f`**`(x[[i]])` (recall that `` `[[` `` works on atomic vectors too).

For example:

```
x <- list(  # an example named list
    x1=1:3,
    x2=seq(0, 1, by=0.25),
    x3=c(1, 0, NA_real_, 0, 0, 1, NA_real_)
)
Map(sqrt, x)  # x is named, hence the result will be named too
## $x1
## [1] 1.0000 1.4142 1.7321
##
## $x2
## [1] 0.00000 0.50000 0.70711 0.86603 1.00000
##
## $x3
## [1]  1  0 NA  0  0  1 NA
Map(length, x)
## $x1
## [1] 3
##
## $x2
## [1] 5
##
## $x3
## [1] 7
unlist(Map(mean, x))  # compute three aggregates, convert to an atomic vector
##  x1  x2  x3
## 2.0 0.5  NA
Map(function(n) round(runif(n, -1, 1), 1), c(2, 4, 6))  # x is atomic now
## [[1]]
## [1] 0.4 0.8
##
## [[2]]
## [1]  0.5  0.8 -0.1 -0.7
##
## [[3]]
## [1] -0.3  0.0  0.5  1.0 -0.9 -0.7
```

Next, we can vectorise a given function over a number of parameters. A call to, e.g., **`Map`**`(f, x, y, z)` results in a list whose *i*-th element is equal to **`f`**`(x[[i]], y[[i]], z[[i]])`. Just like in case of, e.g., **`paste`**, recycling rule will be applied if necessary.

[3] This use case scenario can also be programmed using **`lapply`**; **`lapply`**`(x, f, ...)` is equivalent to **`Map`**`(f, x, MoreArgs=list(...))`.

For example, the following generates `list(seq(1, 6), seq(11, 13), seq(21, 29))`:

```
Map(seq, c(1, 11, 21), c(6, 13, 29))
## [[1]]
## [1] 1 2 3 4 5 6
##
## [[2]]
## [1] 11 12 13
##
## [[3]]
## [1] 21 22 23 24 25 26 27 28 29
```

Moreover, we can get `list(seq(1, 40, length.out=10), seq(11, 40, length.out=5), seq(21, 40, length.out=10), seq(31, 40, length.out=5))` by calling:

```
Map(seq, c(1, 11, 21, 31), 40, length.out=c(10, 5))
## [[1]]
##  [1]  1.0000  5.3333  9.6667 14.0000 18.3333 22.6667 27.0000 31.3333
##  [9] 35.6667 40.0000
##
## [[2]]
## [1] 11.00 18.25 25.50 32.75 40.00
##
## [[3]]
##  [1] 21.000 23.111 25.222 27.333 29.444 31.556 33.667 35.778 37.889 40.000
##
## [[4]]
## [1] 31.00 33.25 35.50 37.75 40.00
```

Note If we have some additional arguments to be passed to the function applied (which the function does not have to be vectorised over), we can wrap them inside a separate list and toss it via the `MoreArgs` argument (à la `do.call`).

```
unlist(Map(mean, x, MoreArgs=list(na.rm=TRUE)))  # mean(..., na.rm=TRUE)
##  x1  x2  x3
## 2.0 0.5 0.4
```

Alternatively, we can always construct a custom anonymous function:

```
unlist(Map(function(xi) mean(xi, na.rm=TRUE), x))
##  x1  x2  x3
## 2.0 0.5 0.4
```

Exercise 7.6 *Here is an example list of files (see our teaching data repository[4]) with daily Forex rates:*

```
file_names <- c(
    "euraud-20200101-20200630.csv",
    "eurgbp-20200101-20200630.csv",
    "eurusd-20200101-20200630.csv"
)
```

Call **`Map`** *to read each dataset with* **`scan`** *and determine the minimal, mean, and maximal value in each series.*

Exercise 7.7 *Implement your own version of the* **`Filter`** *function based on a call to* **`Map`**.

7.3 Accessing Third-Party Functions

When we indulge in the writing of a software piece, a few questions naturally arise. Is the problem we are facing fairly complex? Has it already been successfully addressed in its entirety? If not, can it, or its parts, be split into manageable chunks? Can it be constructed based on some readily available nontrivial components?

A smart developer is independent, but knows when to stand on the shoulders to cry on. Let us explore some ways in which we can reuse the existing function libraries.

7.3.1 Using R Packages

Most contributed R extensions come in the form of the so-called *add-on packages*, which can include:

- reusable code (e.g., new functions),
- data (which we can exercise on),
- documentation (manuals, vignettes, etc.);

see Section 9.3.2 for some more and [47] for all the details.

Most packages are published in the moderated repository that is part of the *Comprehensive R Archive Network* (CRAN[5]). However, there are also other popular sources such as Bioconductor[6] which specialises in bioinformatics.

To fetch a package **`pkg`** from a repository (CRAN by default; see, however, the `repos` argument), we call **`install.packages`**`("pkg")`.

[4] https://github.com/gagolews/teaching-data/tree/master/marek
[5] https://cloud.r-project.org/
[6] https://bioconductor.org/

A call to `library("pkg")` loads an indicated package and makes its exported objects available to the user (i.e., attaches it on the search list; see `sec:to-do`).

For instance, in one of the previous chapters, we have mentioned the **gsl** package:

```
# call install.packages("gsl") first
library("gsl")  # load the package
poch(10, 3:6)   # calls gsl_sf_poch() from GNU GSL
## [1]    1320   17160  240240 3603600
```

Here, **poch** is an object exported by package **gsl**. If we did not call `library("gsl")`, trying to access the former would result in an error.

We could also have accessed the above function without attaching it onto the object search list by using the `pkg::object` syntax, i.e., `gsl::poch`.

Exercise 7.8 *Use the* ***find*** *function to determine which packages define the following objects:* ***mean***, ***var***, ***find***, *and* ***Map****. Recall from Section 1.4 where such information can be found in these objects' manual pages.*

Note For more information about any R extension, call `help(package="pkg")`. Also, it is a good idea to visit the package's CRAN entry at an address like https://CRAN.R-project.org/package=pkg to access some additional information (e.g., vignettes; see also `vignette(package="pkg")`). Why waste our time and energy by querying a web search engine that will lead us to some (usually low-quality) middleman when you can acquire authoritative knowledge directly from the source?

Moreover, it is worth exploring various CRAN Task Views[7] that group the packages into topics such as *Genetics*, *Graphics*, and *Optimisation*. These are edited by experts in their relevant fields.

Important Frequently, R packages are written in their respective authors' free time, many of whom are volunteers/public servants/enthusiasts who are neither paid for doing this nor it is part of the so-called *their job*. You can show appreciation for their generosity by, e.g., spreading the word about their software by citing them in publications (see `citation(package="pkg")`), talking about them during lunch time, or mentioning them in (un)social media. You can also help them improve the existing code base by reporting bugs, polishing documentation, proposing new features, or cleaning up the redundant fragments of their APIs. Some readers will become one of them someday (when they will come up with something useful for our community).

---

[7] https://cloud.r-project.org/web/views/

Default Packages

Note that the always-on package **base** is a must-have that provides us with the most crucial functions (vector addition, **c**, **Map**, **library**). Certain other packages are also loaded by default:

```
getOption("defaultPackages")
## [1] "datasets"  "utils"     "grDevices" "graphics"  "stats"
## [6] "methods"
```

Although this list can – technically speaking – be changed, in this book we assume that the above are always attached, because it is reasonable to do so. This is why in Section 2.4.5, there was no need to call, for example, **library**("stats") before referring to the **var** and **sd** functions.

On a side note, **grDevices** and **graphics** will be discussed in sec:to-do and **methods** will be mentioned in sec:to-do. **datasets** brings a few example R objects that we can exercise our skills on. Functions from **utils**, **graphics**, and **stats**, on the other hand, already appeared here and there.

Source vs Binary Packages (*)

R is a free and open project, therefore its packages are published primarily in the source form – so that anyone can study how they work and improve them or reuse parts thereof in different projects.

If we call **install.packages**("path", repos=NULL, type="source"), we should be able to install a package from sources: path can either be pinpointing a directory or a source tarball (see **help**("untar"), most often as a compressed pkg_version.tar.gz file).

Note that type="source" is the default unless one is on W****ws or some m**OS boxes; see **getOption**("pkgType"). This is because these two might require additional build tools to be present in the system, especially if a package features C, C++, or Fortran code; see Chapter %s and Section C.3 of [49]:

- Rtools[8] on W****ws,
- Xcode Command Line Tools[9] on m**OS.

Because of these systems' being less developer-oriented, as a courtesy to their users, CRAN also distributes the platform-specific binary versions of the packages (.zip or .tgz files). **install.packages** will try to fetch them by default.

Example 7.9 *It is very easy to fetch a package's source directly from GitLab or GitHub, which are quite popular hosting platforms these days. At the time of writing this, the relevant links were, respectively:*

- *https://gitlab.com*/user/repo/*-/archive*/branch/repo-branch.*zip*
- *https://github.com*/user/repo/*archive*/branch.*zip*

[8] https://cran.r-project.org/bin/windows/Rtools/
[9] https://developer.apple.com/xcode/resources/

For example, to download the contents of the master *branch in the repository* rpackagedemo *owned by* gagolews, *we can call:*

```
f <- tempfile()  # temporary file name - download destination
download.file("https://github.com/gagolews/rpackagedemo/archive/master.zip",
    destfile=f)
```

Next, the contents can be extracted with ***unzip****:*

```
t <- tempdir()  # temporary directory to extract the files to
(d <- unzip(f, exdir=t))  # returns extracted file paths
```

The path where the files were extracted can be passed to ***install.packages****:*

```
install.packages(dirname(d)[1], repos=NULL, type="source")
file.remove(c(f, d))  # clean up
```

Exercise 7.10 *Use the* ***git2r*** *package to clone the* ***git*** *repository located at https://github.com/gagolews/rpackagedemo.git and install the package published therein from within the current R session.*

7.3.2 Managing Dependencies (*)

The currently-installed add-on packages may be upgraded to their most recent versions available on CRAN (or other indicated repository) by calling **`update.packages`**.

As a general rule, the more experienced developers we become, the less excited we get about the *new*. Sure, bug fixes and some well-thought of additional features are usually welcome, but just we wait until an updated package API for the n-th time, $n \geq 2$, breaks our program that used to work flawlessly for so long.

Hence, when designing software projects (see Chapter 9 for more details), it is essential that we ask ourselves the ultimate question: do we really need to import that package with lots of dependencies from which we will just use only about 3–5 functions? Wouldn't it be better to write our own version of some functionality (and learn something new, exercise our brain, etc.) or call a mature terminal-based tool?

Otherwise, as all the historical versions of all the packages are archived on CRAN[10], some software dependency management can easily be conducted by storing different version of packages in different directories (only one version of a package can be loaded at a time though). This way, we can create some sort of an isolated environment for the add-ons.

To fetch the locations where packages are sought (in this very order), call:

```
.libPaths()
## [1] "/home/gagolews/R/x86_64-pc-linux-gnu-library/4.2"
```

(continues on next page)

[10] https://cran.r-project.org/src/contrib/Archive/

(continued from previous page)

```
## [2] "/usr/local/lib/R/site-library"
## [3] "/usr/lib/R/site-library"
## [4] "/usr/lib/R/library"
```

The same function can be used to add new folders to the search path; see also the environment variable `R_LIBS_USER` (e.g., **`help`**`("Sys.setenv")`). The **`install.packages`** function will honour them as target directories, see its `lib` parameter for more details.

Moreover, the packages may deposit some auxiliary data on the user's machine. Therefore, it might be a good idea to set the following directories (via the corresponding environment variables) as relative to the current project:

```
tools::R_user_dir("pkg", "data")   # R_USER_DATA_DIR
## [1] "/home/gagolews/.local/share/R/pkg"
tools::R_user_dir("pkg", "config") # R_USER_CONFIG_DIR
## [1] "/home/gagolews/.config/R/pkg"
tools::R_user_dir("pkg", "cache")  # R_USER_CACHE_DIR
## [1] "/home/gagolews/.cache/R/pkg"
```

7.3.3 Calling External Programs

Many tasks can naturally be accomplished by calling external programs. Such an approach is particularly natural on Unix-like systems, which classically follow a modular, minimalist design patterns: there are many tools at a developer's hand and each tool is specialised at solving a single, well-defined problem.

Apart from the many standard Unix commands[11], we can consider, for example:

- **`pandoc`**[12] converts documents between markup formats, e.g., Markdown, reStructuredText, LaTeX, LibreOffice Writer, EPUB;
- **`pdflatex`**, **`xelatex`**, and **`lualatex`** compile LaTeX documents to PDF;
- **`convert`** (from ImageMagick[13]) applies various operations on bitmap graphics (scaling, cropping, conversion between formats);
- **`graphviz`**[14] and **`PlantUML`**[15] can be used to create various graphs and diagrams;
- **`jupyter-nbconvert`** converts Jupyter[16] notebooks (see Section 1.2.5) to other formats such as LaTeX, HTML, Markdown, etc.;
- **`python`**, `{command}perl`, ... can be called to perform tasks that can be expressed more easily in languages other than R;

[11] https://en.wikipedia.org/wiki/List_of_Unix_commands
[12] https://pandoc.org/
[13] https://imagemagick.org/
[14] https://graphviz.org/
[15] https://plantuml.com/
[16] https://jupyter.org/

and so forth.

Good news is that R not only can be called from the shell (in an interactive or batch mode; see Section 1.2), but also it can serve well as a glue language itself.

The **system2** function can be used to invoke any system command. Communication between such programs can be done by means of, e.g., intermediate text, JSON, CSV, XML, or any other files. The `stdin`, `stdout`, and `stderr` arguments can be used to control the redirection of the standard I/O streams.

```
system2("pandoc", "-s input.md -o output.html")
system2("bash", "-c 'for i in `seq 1 2 10`; do echo $i; done'", stdout=TRUE)
## [1] "1" "3" "5" "7" "9"
system2("python3", "-", stdout=TRUE,
    input=c(
    "import numpy as np",
    "print(repr(np.arange(5)))"
    ))
## [1] "array([0, 1, 2, 3, 4])"
```

Note that the current working directory can be read and changed by means of a call to **getwd** and **setwd**, respectively. It is the directory from where the current R session was started.

Important Relying on **system2** assumes that the commands referred to are available on the target platform. Hence, it might not be portable, unless additional assumptions are made (e.g., that a user runs some Unix system, that certain libraries are installed therein). We strongly recommend GNU/Linux or FreeBSD for both software development and production use, as they are free, open, developer-friendly, user-loving, reliable, ethical, and sustainable.

7.3.4 A Note on Interfacing C, C++, Python, Java, etc. (*)

Most stand-alone data processing algorithms are implemented in compiled, slightly lower-level programming languages. This usually makes them faster and more reusable in other environments. For instance, it is often the case that an industry-standard library is written in very portable C, C++, or Fortran and has some bindings available for easier access from within R, Python, Julia, etc. This is the case with FFTW, LIBSVM, mlpack, OpenBLAS, ICU, and GNU GSL, amongst many others.

For basic ways to interact with such compiled code, see `Chapter %s`.

Also, the **rJava** package can be used to dynamically create JVM objects and access their fields and methods. Similarly, **reticulate** can be used to access Python objects, including **numpy** arrays and **pandas** data frames (but see also the **rpy2** package for Python).

Important We should not feel obliged to use R in all the parts of a data pro-

cessing pipeline. Some activities can be expressed more naturally in other languages/environments (e.g., parse raw data and create an SQL database in Python, but visualise it in R). We can use other tools as the glue language (including R, Python, or Bash) that will steer the data flow in the right direction.

7.4 Exercises

Exercise 7.11 *Answer the following questions:*

- *What is the result of "*`x <- 2; x <- function(x) x^2; x(x)`*"?*
- *How to write a function that returns two objects?*
- *What is a higher-order function?*
- *What are the use cases of* **`do.call`***?*
- *Why a call to* **`Map`** *is not necessary in the expression "*`Map(paste, x, y, z)`*"?*
- *What is the difference between* `Map(mean, x, na.rm=TRUE)` *and* `Map(mean, x, MoreArgs=list(na.rm=TRUE))`*?*
- *What do we mean when we write* **`stringx::sprintf`***?*
- *How to get access to the vignettes (tutorials, FAQs, etc.) of the* **`data.table`** *and* **`dplyr`** *packages? Why perhaps 95% of R users would just googleit and what is sub-optimal about this strategy?*
- *What is the difference between a source and a binary package?*
- *How to update the* **`base`** *package?*
- *How to assure that we will always run an R session with only specific versions of a set of packages?*

Exercise 7.12 *Write a function that computes the Gini index of a vector of positive integers* `x`*, which, assuming* $x_1 \leq x_2 \leq \ldots \leq x_n$*, is equal to:*

$$G(x_1, \ldots, x_n) = \frac{\sum_{i=1}^{n}(n - 2i + 1)x_i}{(n-1)\sum_{i=1}^{n} x_i}.$$

Exercise 7.13 *Implement a function* `between(x, a, b)` *that verifies whether each element in* `x` *is in the* `[a, b]` *interval or not. Return a logical vector of the same length as* `x`*. Make sure the function is correctly vectorised with respect to all the arguments and handles missing data correctly.*

Exercise 7.14 *Write your own version of the* **`strrep`** *function called* **`dup`***.*

```
dup <- ...to.do...
dup(c("a", "b", "c"), c(1, 3, 5))
## [1] "a"     "bbb"   "ccccc"
dup("a", 1:3)
## [1] "a"   "aa"  "aaa"
dup(c("a", "b", "c"), 4)
## [1] "aaaa" "bbbb" "cccc"
```

Exercise 7.15 *Given a list* x*, generate its sublist with all the elements equal to* `NULL` *removed.*

Exercise 7.16 *Implement your own version of the built-in* **`sequence`** *function.*

Exercise 7.17 *Using* **`Map`***, how can we generate window indexes like:*

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 2 3 4
##
## [[3]]
## [1] 3 4 5
##
## [[4]]
## [1] 4 5 6
```

Write a function **`windows`**`(k, n)` *that yields* k *index windows with elements between 1 and* n *(the above example is for k=3 and k=6).*

Exercise 7.18 *Implement a function* **`movstat`**`(f, x, k)` *that computes, using* **`Map`***, a given aggregate* **`f`** *of each* k *consecutive elements in* x*. For instance:*

```
movstat <- ...to.do...
x <- c(1, 3, 5, 10, 25, -25)  # example data
movstat(mean, x, 3)           # 3-moving mean
## [1]  3.0000  6.0000 13.3333  3.3333
movstat(median, x, 3)         # 3-moving median
## [1]  3.0000  6.0000 13.3333  3.3333
```

Exercise 7.19 *Write a function to extract all* q*-grams,* $q \geq 1$*, from a given character vector. Return a list of character vectors. For examples, 2-grams (bigrams) in* `"abcd"` *are:* `"ab"`*,* `"bc"`*, "cd"*`.

Exercise 7.20 *Recode a character vector with a small number of distinct values to a vector where each unique code is assigned a positive integer from 1 to* k*. Example calls and the corresponding expected results:*

```
recode <- ...to.do...
recode(c("a", "a", "a", "b", "b"))
## [1] 1 1 1 2 2
recode(c("x", "z", "y", "x", "y", "x"))
## [1] 1 3 2 1 2 1
```

Exercise 7.21 *Implement a function that returns the number of occurrences of each unique element in a given atomic vector. The return value should be a numeric vector equipped with a* `names` *attribute.*

```
count <- ...to.do...
count(c(5, 5, 5, 5, 42, 42, 954))
##   5  42 954
##   4   2   1
count(c("x", "z", "y", "x", "y", "x", "w", "x", "x", "y", NA_character_))
##    w    x    y    z <NA>
##    1    5    3    1    1
```

Hint: use ***match*** *and* ***tabulate****.*

Exercise 7.22 *Implement a function that extends upon the built-in* ***duplicated****, indicating which occurrence (starting from the beginning of the vector) of a repeated value a given value constitutes.*

```
duplicatedn <- ...to.do...
duplicatedn(c("a", "a", "a", "b", "b"))
## [1] 1 2 3 1 2
duplicatedn(c("x", "z", "y", "x", "y", "x", "w", "x", "x", "y", "z"))
##  [1] 1 1 1 2 2 3 1 4 5 3 2
```

Exercise 7.23 *Based on a call to* ***Map****, implement a function* ***my_split*** *such that, given a vector* `x` *and an atomic vector* `y` *of the same length as* `x`, ***my_split****(*`x, y`*)* *yields the same result as* ***split****(*`x, y`*).*

Exercise 7.24 *Extend* ***my_split*** *to handle the second argument being a list of the form* ***list****(*`y1, y2, ...`*)* *that represents the product of many levels. If the* `y`*s are of different lengths, apply the recycling rule.*

Exercise 7.25 *Implement* ***my_unsplit*** *being your own version of the built-in* ***unsplit****. Make sure it holds* ***my_unsplit****(****split****(*`x, g`*),* `g`*)* `==` `x` *for* `x` *and* `g` *of the same lengths.*

Exercise 7.26 *Write a function that takes as arguments: (a) an integer* n*, (b) a numeric vector* `x` *of length* k *and no duplicated elements, (c) a vector of probabilities* `p` *of length* k*; verify that* $p_i \geq 0$ *for all* i *and* $\sum_{i=1}^{k} p_i \simeq 1$*. Based on a random number generator from the uniform distribution on the unit interval, generate* n *independent realisations of a random variable* X *such that* $\Pr(X = x_i) = p_i$ *for* $i = 1, \ldots, k$*. Hint: to obtain a single value:*

1. *generate* $u \in [0, 1]$,

2. *find* $m \in \{1, \ldots, k\}$ *such that* $u \in \left(\sum_{j=1}^{m-1} p_j, \sum_{j=1}^{m} p_j\right]$,
3. *the result is then* x_m.

Exercise 7.27 *Write a function that takes as arguments: (a) an increasingly sorted vector* `x` *of length* n, *(b) any vector* `y` *of length* n, *(c) a vector* `z` *of length* k *and elements in* $[x_1, x_n)$. *Let* f *be the piecewise linear spline that interpolates the points* $(x_1, y_1), \ldots, (x_n, y_n)$. *Return a vector* `w` *of length* k *such that* $w_i = f(z_i)$.

Exercise 7.28 *(*) Write functions* **`dpareto`**, **`ppareto`**, **`qpareto`**, *and* **`rpareto`** *that implement the basic functions related to the Pareto distribution; compare Section 2.3.4.*

8

Flow of Execution

The **`ifelse`** and **`Map`** functions are very powerful, but they allow us to process only the consecutive elements in a vector.

Thus, let us (finally!) discuss different ways to alter a program's control flow manually, based on some criterion, and to evaluate the same expression a number of times, but perhaps on different data. Before proceeding any further, let us, however, contemplate on the fact that we have managed to do without them for such a long time – and the data processing exercises we learnt to solve were far from trivial.

8.1 Conditional Evaluation

Life is full of surprises, so we would be nice if we were able to adapt to whatever the circumstances are going to be.

The following evaluates a given `expression` *if and only if* a logical `condition` is true.

```
if (condition) expression
```

When performing some `other_expression` is preferred rather than doing nothing in the case of the `condition`'s being false, we can write:

```
if (condition) expression else other_expression
```

For instance:

```
(x <- runif(1))  # to spice things up
## [1] 0.28758
if (x > 0.5) cat("head") else cat("tail")
## tail
```

Many expressions can of course be grouped with curly braces, "{"

```
if (x > 0.5) {
    cat("head")
    x <- 1
```

(continues on next page)

(continued from previous page)

```
} else {
    cat("tail")
    x <- 0
}
## tail
print(x)
## [1] 0
```

Important At the top level, we should not put a new line before **`else`**, otherwise we will get an error like `Error: unexpected 'else' in "else"`. This is because the interpreter enthusiastically executes the statements been read line by line as soon as it regards them as stand-alone expressions. In this case, we first get an **`if`** without **`else`**, and then, separately, a *dangling* **`else`** without the preceding **`if`**.

This does not happen when a conditional statement is part of an expression group, because the latter is read in its entirety.

```
function (x)
{  # opening bracket - start
    if (x > 0.5)
        cat("head")
    else              # not dandling, because {...} is read as a whole
        cat("tail")
}  # closing bracket - expression ends
```

As an exercise, try removing the curly braces and see what happens.

8.1.1 Return Value

`if` is a function (compare Section 9.4), hence has a return value – the result of evaluating the conditional expression.

```
(x <- runif(1))
## [1] 0.28758
y <- if (x > 0.5) "head"  # no else
print(y)
## NULL
y <- if (x > 0.5) "head" else "tail"
print(y)
## [1] "tail"
```

This is particularly useful when a call to **`if`** is the last expression in the code block constituting a function's body.

```
mint <- function(x)
{
    if (x > 0.5)  # the last expression (actually, the only one)
        "head"        # this can be the return value...
    else
        "tail"        # or this one, depending on the condition
}

mint(x)
## [1] "tail"
unlist(Map(mint, runif(5)))
## [1] "tail" "head" "tail" "head" "head"
```

Example 8.1 *Add-on packages can also be loaded using* ***`requireNamespace`****. Contrary to* ***`library`****, the former does not fail when a package is not available. Also, it does not attach it on the search list; see* `sec:to-do`*.*

Instead, it returns a logical value indicating if the package is available for use. This can be useful inside other functions where the availability of some additional features depends on the user environment's configuration:

```
process_data <- function(x)
{
    if (requireNamespace("some_extension_package", quietly=TRUE))
        some_extension_package::very_fast_method(x)
    else
        normal_method(x)
}
```

8.1.2 Nested `if`s

If more than two test cases are possible, i.e., when we need to go beyond either `condition` or `!condition`, then we can use the following construction:

```
if (a) {
    expression_a
} else if (b) {
    expression_b
} else if (c) {
    expression_c
} else {
    expression_else
}
```

This evaluates all conditions `a`, `b`, ... (in this order) until the first positive case is found, and then executes the corresponding `expression`.

Note that the above is nothing else than a series of nested **`if`** statements:

```
if (a) {
    expression_a
} else {
    if (b) {
    expression_b
    } else {
        if (c) {
            expression_c
        } else {
            expression_else
        }
    }
}
```

but written in a less readable[1] manner.

Exercise 8.2 *Write a function named* ***`sign`*** *that determines if a given numeric value is* `"positive"`*,* `"negative"`*, or* `"zero"`*.*

8.1.3 Condition: Either True of False

`if` expects a `condition` that is a single, well-defined logical value, either `TRUE` or `FALSE`. Thence, problems may arise when this is not the case.

If the `condition` is of length not equal to one, we get an error:

```
if (c(TRUE, FALSE)) cat("spam")
## Error in if (c(TRUE, FALSE)) cat("spam"): the condition has length > 1
if (logical(0)) cat("bacon")
## Error in if (logical(0)) cat("bacon"): argument is of length zero
```

We cannot pass a missing value either:

```
if (NA) cat("ham")
## Error in if (NA) cat("ham"): missing value where TRUE/FALSE needed
```

Important If we think that we are absolutely immune to the writing of code violating the above constraints, just we wait until the `condition` becomes a function of data for which there is no sanity-checking in place.

```
mint <- function(x)
    if (x > 0.5) "H" else "T"
```

(continues on next page)

[1] Somewhat related is the **`switch`** function which we study in `sec:to-do`. It relies on lazy evaluation of its arguments. Still, it can always be replaced by a series of **`if`**s.

(continued from previous page)

```
mint(0.25)
## [1] "T"
mint(runif(5))
## Error in if (x > 0.5) "H" else "T": the condition has length > 1
mint(log(rnorm(1)))  # not obvious, only triggered sometimes
## Warning in log(rnorm(1)): NaNs produced
## Error in if (x > 0.5) "H" else "T": missing value where TRUE/FALSE needed
```

In Chapter 9, we will be particularly interested in ways to assure input data integrity, so that situations such as above will either fail gracefully or succeed bombastically.

Here, we should probably make sure that `x` is a single finite numeric value. Alternatively, we had rather test whether **`all`**`(x > 0.5, na.rm=TRUE)`.

Interestingly, objects other that logical are accepted: they will be coerced if needed.

```
x <- 1:5
if (length(x))  # i.e., length(x) != 0, but way less readable
    cat("length is not zero")
## length is not zero
```

Recall that coercion of numeric to logical yields `FALSE` if and only if the original value is zero.

8.1.4 Short-Circuit Evaluation

Specially for formulating logical conditions in **`if`** and **`while`** (see below), we have the *scalar* **`` `||` ``** (alternative) and **`` `&&` ``** (conjunction) operators.

```
FALSE || TRUE
## [1] TRUE
NA || TRUE
## [1] TRUE
```

Contrary to their vectorised counterparts (**`` `|` ``** and **`` `&` ``**), the scalar operators are lazy (Section 9.5.5) in the sense that they evaluate the first operand and then determine if the computing of the second one is necessary (because, e.g., `FALSE && whatever` is always `FALSE` anyway).

Therefore,

```
if (a && b)
    expression
```

is equivalent to:

```
if (a) {
    if (b) {  # compute b only if a is TRUE
        expression
    }
}
```

and:

```
if (a || b)
    expression
```

corresponds to:

```
if (a) {
    expression
} else if (b) {  # compute b only if a is FALSE
    expression
}
```

For instance, "`is.vector(x) && length(x) > 0 && x[[1]] > 0`" is a safe test that takes into account that "`x[[1]]`" has only the desired meaning for objects that are not non-empty vectors.

Some other examples (recall that the expressions within the curly braces are evaluated one after another and that the result is determined by the last value in the series):

```
{cat("spam"); FALSE} || {cat("ham"); TRUE} || {cat("cherries"); FALSE}
## spamham
## [1] TRUE
{cat("spam"); TRUE} && {cat("ham"); FALSE} && {cat("cherries"); TRUE}
## spamham
## [1] FALSE
```

Exercise 8.3 *Study the source code of* ***isTRUE*** *and* ***isFALSE*** *and determine if these functions could be useful in formulating the conditions within the* ***if*** *expressions.*

8.2 Exception Handling

Exceptions are exceptional, but they may happen and break things. For instance, when we try to download a file and the internet connection drops. Or an optimisation algorithm fails to converge. Or we just have a bug in our code. Or:

```
read.csv("/path/to/a/file/that/does/not/exist")
## Warning in file(file, "rt"): cannot open file '/path/to/a/file/that/does/
## not/exist': No such file or directory
## Error in file(file, "rt"): cannot open the connection
```

Three types of *conditions* are frequently observed:

- errors – they stop the flow of execution,
- warnings – non critical, but can be turned into errors (see `warn` in **`option`**),
- messages – they transmit some diagnostic information.

These can be manually triggered by means of **`stop`**, **`warning`**, and **`message`** functions.

Errors (but warnings too) can be handled by means of the **`tryCatch`** function, amongst others.

```
tryCatch({           # block of expressions to execute, until an error occurs
        cat("a\n")
        stop("b")    # error - breaks the linear control flow
        cat("c\n")
    },
    error = function(e) {  # executed immediately upon an error
        cat(sprintf("error: %s\n", e[["message"]]))
    },
    finally = {  # always executed at the end, regardless of error occurrence
        cat("finally!\n")
    }
)
## a
## error: b
## finally!
```

The two other conditions can be ignored by calling **`suppressWarnings`** and **`suppressMessages`**.

```
log(-1)
## Warning in log(-1): NaNs produced
## [1] NaN
suppressWarnings(log(-1))  # yeah, yeah, we know what we're doing
## [1] NaN
```

Exercise 8.4 *At the time of writing of this book, the* ***`data.table`*** *package emits a message upon attachment. Call* ***`suppressMessages`*** *to silence it. Note that consecutive calls to* ***`library`*** *do not reload an already loaded package, therefore the message will only be seen once per R session.*

Related functions include **`stopifnot`** discussed in Section 9.2 and **`on.exit`** mentioned in `sec:to-do`; see also Section 9.3.4 for some code debugging tips.

8.3 Repeated Evaluation

And now for something completely different… time for the elephant in the room!

We have been able to do without loops so far (and will be quite all right in the second part of the book too), because many data processing tasks can be written in terms of vectorised operations such as **`+`**, **`sqrt`**, **`sum`**, **`[`**, **`Map`**, and **`Reduce`**. Oftentimes, compared to their loop-based counterparts, they are not only much more readable but also more efficient. We will explore this in the exercises below.

However, at times, using an explicit **`while`** or **`for`** loop might be the only natural way of solving a problem, for instance, when processing chunks of data streams. Also, an explicitly "looped" algorithm may occasionally have better[2] time or memory complexity.

8.3.1 `while`

`if` considers a given logical condition and thus determines whether to execute a given statement. On the other hand,

```
while (condition)  # single TRUE or FALSE, as in `if`
    expression
```

evaluates a given `expression` *as long as* the logical `condition` is true. Therefore, it is advisable to make the `condition` dependent upon some variable that can be modified by the `expression`.

```
i <- 1
while (i <= 3) {
    cat(sprintf("%d, ", i))
    i <- i + 1
}
## 1, 2, 3,
```

Nested loops are of course possible too:

```
i <- 1
while (i <= 2) {
    j <- 1
    while (j <= 3) {
        cat(sprintf("%d %d, ", i, j))
        j <- j + 1
    }
```

(continues on next page)

[2] But in such cases it will often benefit from a rewrite in C or C++; see `Chapter %s`.

(continued from previous page)

```
    cat("\n")
    i <- i + 1
}
## 1 1, 1 2, 1 3,
## 2 1, 2 2, 2 3,
```

Example 8.5 *Implement a simple linear congruential pseudorandom number generator that, given some seed $X_0 \in [0, m)$, outputs a sequence $(X_1, X_2, \dots)$ defined by:*

$$X_i = (aX_{i-1} + c) \mod m,$$

with, e.g., $a = 75$, $c = 74$, and $m = 2^{16} + 1$ (here, mod is the division reminder, `%%`). Note that this generator has poor statistical properties and should not be used in practice. In particular, after some number of operations k, we will find a cycle such that $X_k = X_1, X_{k+1} = X_2, \dots$.

8.3.2 for

The for-each loop:

```
for (name in vector)
    expression
```

takes each element, from the beginning to the end, in a given `vector`, assigns it some `name`, and evaluates the `expression`.

Example:

```
fridge <- c("spam", "spam", "bacon", "eggs")
for (food in fridge)
    cat(sprintf("%s, ", food))
## spam, spam, bacon, eggs,
```

One more:

```
for (i in 1:length(fridge))  # better: seq_along(fridge), see below
    cat(sprintf("%s, ", fridge[i]))
## spam, spam, bacon, eggs,
```

Just one more, promise:

```
for (i in 1:2) {
    for (j in 1:3)
        cat(sprintf("%d %d, ", i, j))
    cat("\n")
}
## 1 1, 1 2, 1 3,
## 2 1, 2 2, 2 3,
```

Note that the iterator still exists after the loop's watch has ended:

```
print(i)
## [1] 2
print(j)
## [1] 3
```

Important Writing:

```
for (i in 1:length(x))
    print(x[i])
```

is not necessarily safe, because if `x` is an empty vector, then:

```
x <- logical(0)
for (i in 1:length(x)) print(x[i])
## [1] NA
## logical(0)
```

Recall from Chapter 5 that `x[1]` tries to access an out of bounds element here and `x[0]` returns nothing.

We generally suggest replacing `1:length(x)` with `seq_along(x)` or `seq_len(length(x))`. wherever possible.

Note The model `for` loop above is roughly equivalent to:

```
name <- NULL
tmp_vector <- vector
tmp_iter <- 1
while (tmp_iter <= length(tmp_vector)) {
    name <- tmp_vector[[tmp_iter]]
    expression
    tmp_iter <- tmp_iter + 1
}
```

Note that `tmp_vector` is determined before the loop itself. Hence, any changes to `vector` will not influence the execution flow. Also note that due to the use of `` `[[` ``, the loop can be applied on lists as well.

Example 8.6 *Let `x` be a list and `f` be a function. The following code generates the same result as `Map(f, x)`:*

```
n <- length(x)
```

(continues on next page)

(continued from previous page)

```
ret <- vector("list", n)  # a new list of length `n`
for (i in seq_len(n))
    ret[[i]] <- f(x[[i]])
```

Example 8.7 *Let x and y be two lists and* **`f`** *be a function. Here is the most basic version of* **`Map(f, x, y)`**. *Note that x and y might be of different lengths.*

```
nx <- length(x)
ny <- length(y)
n <- max(nx, ny)
ret <- vector("list", n)
for (i in seq_len(n))
    ret[[i]] <- f(x[[((i-1)%%nx)+1]], y[[((i-1)%%ny)+1]])
```

Feel free to upgrade the above by adding a warning like the longer argument is not a multiple of the length of the shorter one. *Also, rewrite it without the use of the modulo operators,* **`%%`**.

8.3.3 `break` and `next`

`break` can be used to escape the current loop. **`next`** skips the remaining expressions and advances to the next iteration (to where the testing of the logical condition occurs).

Here is a rather random example:

```
x <- runif(1000)
s <- 0
for (e in x) {
    if (e > 0.1)
        next

    print(e)
    if (e < 0.01)
        break

    s <- s + e
}
## [1] 0.045556
## [1] 0.04206
## [1] 0.024614
## [1] 0.045831
## [1] 0.094841
## [1] 0.00062477
print(s)
## [1] 0.2529
```

Computes the sum of the elements in x that are less than or equal to 0.1 from the beginning, stopping at the first element less than 0.01.

Note that we have used the frequently occurring design pattern:

```
for (e in x) {
    if (condition)
        next

    many_statements...
}
```

which is equivalent to:

```
for (e in x) {
    if (!condition) {
        many_statements...
    }
}
```

but avoids introducing a nested block of expressions.

Note (*) There is a third loop type,

```
repeat
    expression
```

which is a shorthand for

```
while (TRUE)
    expression
```

i.e., it is a possibly infinite loop. Such loops are useful when implementing situations such as *do*-stuff-*until*-a-thing-happens, e.g., when we want to execute a command at least once.

```
i <- 1
repeat {  # while (TRUE)
    # simulate dice casting until we throw "1"
    if (runif(1) < 1/6) break  # not an infinite loop after all
    i <- i+1
}
print(i)
## [1] 6
```

Exercise 8.8 *What is wrong with the following code?*

```
j <- 1
while (j <= 10) {
    if (j %% 2 == 0) next
    print(j)
    j <- j + 1
}
```

Exercise 8.9 *What about this one?*

```
j <- 1
while (j <= 10);
    j <- j + 1
```

8.3.4 `return`

`return`, when called from within a function, immediately yields a specified value and goes back to the caller.

For example, here is a simple recursive function that flattens a given list:

```
my_unlist <- function(x)
{
    if (is.atomic(x))
        return(x)

    # so if we are here, x is definitely not atomic
    out <- NULL
    for (e in x)
        out <- c(out, my_unlist(e))

    out  # or return(out); it's the last expression anyway, so not necessary
}

my_unlist(list(list(list(1, 2), 3), list(4, list(5, list(6, 7:10)))))
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Note that **`return`** is a function: the round brackets are obligatory,

8.3.5 A Note on Time and Space Complexity of Algorithms (*)

Analysis of algorithms (e.g., [9, 35]), can give us a rough estimate of their run times or memory consumption as a function of the input data size, especially for *big* data.

In scientific computing and data science, we most often deal with vectors (sequences) or matrices/data frames (tabular data). Therefore, we might be interested in determining how many *primitive operations* need to be performed as a function of their length n or the number of rows n and columns p, respectively.

The O (Big-Oh) notation, for instance, can express the upper bounds for time/resource consumption in asymptotic cases. For instance, we say that the time complexity is $O(n^2)$, if for large n, the number of operations to perform will be proportional to *at most* the square of the vector size (more precisely, there exists m and $C > 0$ such that for all $n > m$, the number of operations is $\leq Cn^2$).

Therefore, if we have two algorithms that solve the same task, one that has $O(n^2)$ time complexity, and other of $O(n^3)$, it is better to choose the former, because for large problem sizes we expect it to be faster.

Moreover, whether time grows proportionally to $\log n$, $\sqrt{n}$, n, $n \log n$, n^2, n^3, or 2^n, can be useful in predicting how big the data can be if we have a fixed deadline or not too much space left on the disk.

Exercise 8.10 *The* `hclust` *function determines a hierarchical clustering of a dataset. It is fed with an object that stores the distance between all the pairs of input points. There are* $n(n-1)/2$ *(i.e.,* $O(n^2)$*) unique point pairs for any given* `n`*. One numeric scalar (*`double` *type) takes 8 bytes of storage. If you have 16 GB or RAM, what is the largest dataset that you can cluster on your machine using this function?*

Oftentimes, we can learn about the time or memory complexity of the functions we use from their documentation; see, e.g., `help("findInterval")`.

Example 8.11 *A course in data structures in algorithms, which this one is not, will give us plenty of opportunities to implement many algorithms ourselves. This way, we can gain a lot of insights and intuitions.*

For instance, this is a $O(n)$*-time algorithm:*

```
for (i in seq_len(n))
    expression
```

and this is one runs in $O(n^2)$ *time:*

```
for (i in seq_len(n))
    for (j in seq_len(n))
        expression
```

as long as, of course, the `expression` *is rather primitive (e.g., operations on scalar variables).*

R is a very expressive language and hence quite complex and lengthy operations can look pretty innocent (it is a glue language for rapid prototyping, after all).

For example:

```
for (i in seq_len(n))
    for (j in seq_len(n))
        z <- z + x[[i]] + y[[j]]
```

can be seen as $O(n^3)$ *if each element in the lists* `x` *and* `y` *as well as* `z` *itself are atomic vectors of length* `n`.

Similarly,

```
Map(mean, x)
```

is $O(n^2)$ *if* `x` *is a list of* `n` *atomic vectors each of length* `n`.

Note A quite common statistical scenario involves the generation of a data buffer of a fixed size:

```
ret <- c()
for (i in n)
    ret[[i]] <- generate_data(i)  # here: ret[[length(ret)+1]] <- ...
```

This notation, however, involves the growing of the `ret` array in each iteration. Luckily, since R version 3.4.0, each such size extension has amortised $O(1)$ time due to the fact that some more memory is internally reserved for its prospective growth (see, e.g., Chapter 17 of [9]).

However, it would still be better to pre-allocate the output vector and grant it the desired, final size already upon creation.

Note that we can construct vectors of specific lengths and types in an efficient way (more efficient than with **`rep`**) by calling:

```
numeric(3)
## [1] 0 0 0
numeric(0)
## numeric(0)
logical(5)
## [1] FALSE FALSE FALSE FALSE FALSE
character(2)
## [1] "" ""
vector("numeric", 8)
## [1] 0 0 0 0 0 0 0 0
vector("list", 2)
## [[1]]
## NULL
##
## [[2]]
## NULL
```

Note Not all data fit into memory, but it does not mean that we should start installing Apache Hadoop and Spark immediately. Some datasets can be processed on a chunk-by-chunk basis.

R enables data stream handling (some can be of infinite length) through file connections, for example:

```
f <- file(paste0("https://raw.githubusercontent.com/gagolews/teaching-data/",
    "master/README.md"), open="r")  # a big file, the biggest file ever
i <- 0
while (TRUE) {
    few_lines <- readLines(f, n=4)  # read only four lines at a time
    if (length(few_lines) == 0) break
    i <- i + length(few_lines)
}
close(f)
print(i)  # the number of lines
## [1] 93
```

Many functions support reading from/writing to already established connections of different types, e.g., **`file`**, **`gzfile`**, **`textConnection`**, batch-by-batch.

A frequent scenario involves reading a very large CSV, JSON, or XML file only thousands of lines/records at a time, parsing and cleansing them, and exporting to SQL databases (which we will exercise in Chapter 12).

Also note that the always-open text connections **`stdout`** and **`stderr`** (for writing), and **`stdin`** (for reading) are by default mapped to the "terminal/console" and "keyboard", respectively. Call **`scan`**, **`cat`**, and **`stop`** to interact with such sources/targets.

8.4 Exercises

Note that, from now on, we should stay alert. Many, if not all, of the following tasks can still be implemented without the explicit use of the R loops, but based only on the operations covered in the previous chapters. If this is the case, try implementing both the looped and loop-free version. Use **`microbenchmark::microbenchmark`** or **`proc.time`** to compare the run-times[3].

Exercise 8.12 *Answer the following questions:*

- *Let `x` be a numeric vector. When does `if(x > 0) ...` yield a warning? When does it give an error? How to prevent this?*
- *What is the dangling **`else`**?*
- *What happens if you put **`if`** as the last expression in a curly braces block within a function's body?*

[3] It might be the case that a **`for`**-based solution is faster (e.g., for larger objects) because of the use of a more efficient algorithm. Such cases will especially benefit from a rewrite in C or C++ (`Chapter %s`).

- *Why do we say that `&&` and `||` are lazy? What are their use cases?*
- *What is the difference between `&&` and `&`?*
- *Can `while` always be replaced with `for`? What about the other way around?*

Exercise 8.13 *Verify which of the following can be safely used as logical conditions in `if` statements. If that is not the case for all `x`, `y`, ..., determine the additional conditions that should be imposed in order to make them valid.*

- `x == 0`,
- `x[y] > 0`,
- `any(x>0)`,
- `match(x, y)`,
- `any(x %in% y)`.

Exercise 8.14 *What can go wrong in the following code chunk, depending on the type and form of `x`? Consider as many scenarios as possible.*

```
count <- 0
for (i in 1:length(x))
    if (x[i] > 0)
        count <- count + 1
```

Exercise 8.15 *Implement `shift_left(x, n)` and `shift_right(x, n)`. The former function gets rid of the first* n *observations in `x` and adds* n *missing values at the end of the resulting vector, e.g., `shift_left(c(1, 2, 3, 4, 5), 2)` is `c(3, 4, 5, NA, NA)`. On the other hand, `shift_right(c(1, 2, 3, 4, 5), 2)` is `c(NA, NA, 1, 2, 3)`.*

Exercise 8.16 *Implement your own version of `diff`.*

Exercise 8.17 *Write a function that determines the longest increasing trend in a given numeric vector, i.e., the length of the longest subsequence of consecutive elements that are increasing. For example, the input `c(1, 2, 3, 2, 1, 2, 3, 4, 3)` should yield 4.*

Exercise 8.18 *Implement the functions that round down and round up, to a number of decimal digits, each element in a numeric vector.*

This concludes the first part of this magnificent book.

Part II

Deeper

9

Designing Functions

In Chapter 7, we learnt how to write our own functions. This skill is key to enforcing the good development practice of avoiding the repetition of code: running the same command sequence on different data.

This chapter is devoted to the designing of such reusable modules so that they are easier to use, test, and maintain. We also provide some more technical details which were not of the highest importance upon our first exposure to this topic, but which our crucial to our better understanding of how R works.

9.1 Principles of Sustainable Design

Good design is more art than science. As usual in real life, we will need to make many compromises. This is because improving things with regard to one criterion sometimes makes them worse with respect to other aspects[1] (also which we are not aware of). Also, not everything that counts can and will be counted. Below are some observations, ideas, and food for thought.

9.1.1 To Write or to Abstain

Functions that we write ourselves can oftentimes be considered merely creative combinations of the building blocks available in base R or a few high-quality add-on packages[2]. Some are simpler than others. Thus, there is a question if a new operation should be introduced at all: whether we are faced with the case of multiplying entities without necessity.

On the one hand, the DRY (don't repeat yourself) principle tells us that most frequently used (say, at least 3 times) code chunks should be generalised in the form of a new function. This is definitely a correct approach with regard to non-trivial operations.

On the other hand, not every generalisation is necessarily welcome. Let us say that we are lazy and tired of writing `g(f(x))` for the n-th time. Why don't we therefore introduce `h` defined as a combination of `g` and `f`? This might *seem* like a good idea, but let

[1] Compare the notion of Pareto efficiency.

[2] If some non-trivial operation is missing, we can always implement it at the C language level; see `Chapter %s`.

us not take it for granted: being tired might be an indication of our body and mind needing a rest; being lazy can be a call for more self-discipline (not an overly popular word these days, but still, a precious trait).

Example 9.1 ***paste0*** *is a specialised version of* ***paste****, but having the* `sep` *argument hardcoded to an empty string.*

- *Even if this might be the most often applied use case, is the introduction of a new function justifiable? Is it so hard to write* `paste=""` *each time?*
- *Would changing* ***paste****'s default argument be better? That of course would harm backward compatibility, but what strategies could we apply to make the transition as smooth as possible?*
- *Would it be better to introduce a new version of* ***paste*** *with* `sep` *defaulting to* `""`*, informing the users that the old version is deprecated and to be removed in, say, two years? Or maybe one year is better? Or five?*

Example 9.2 *In R 4.0,* ***deparse1*** *has been introduced: it is merely a combination of* ***deparse*** *(see below) and* ***paste****:*

```
print(deparse1)
## function (expr, collapse = " ", width.cutoff = 500L, ...)
## paste(deparse(expr, width.cutoff, ...), collapse = collapse)
## <bytecode: 0x556f21a5eef0>
## <environment: namespace:base>
```

Let us say this covers 90% of use cases: was introducing it a justified idea then? What if that number was 99%?

Overall, more functions contribute to the information overload. We do not want our users to be overwhelmed by too many choices. Luckily, nothing is cemented once and for all. Had we done bad design choices resulting in our API's being bloated, we can always clean up those that no longer spark joy.

9.1.2 To Pamper or to Challenge

Think about the kind of audience we would like to serve: is it our team only, students, professionals, certain client groups, etc.? Do they have mathematical, programming, engineering, or scientific background? Not everything that is appropriate for one cohort, will be valuable for another. And not everything that is good for some *now*, will be beneficial for them in the long run. People (their skills, attitudes, etc.) change.

Example 9.3 *Assume we are writing a friendly and inclusive package for novices who would like*

to grasp the basics of data analysis as quickly[3] *as possible. Without much effort, it would enable them to solve 80–95% of the most common, easy problems.*

Think of introducing the students to a function that returns five largest observations in a given vector. Let us call it **`nlargest`***: so pleasant to use. It makes the students feel empowered quickly.*

Still, when faced with the remaining 5–20% tasks, they will have to learn another, more advanced, generic, and powerful tool anyway (in our case, the base R itself). Are they determined and skilled enough to do that? Time will tell. The least we can do is to be explicit about it.

Recall that it took us some time to arrive at **`order`** *and subsetting via* **`[`***. Assuming that we read this book from the beginning to the end and solve all the exercises, which we should, we are now able to implement the said* **`nlargest`** *(and lots of other functions) ourselves, using a single line of code. This will also pay off in many scenarios that we will be facing in the future, e.g., when we consider matrices and data frames.*

Yes, everyone will be reinventing their own **`nlargest`** *this way. But this constitutes a great exercise: by our being too* nice*, some might have lost an opportunity to learn a new, more universal skill.*

Although most of the users would really love to minimise the effort put into all their activities, ultimately, they sometimes need to learn new things. Let us thus not be afraid to teach them stuff.

Furthermore, we do not want to discourage experts (or experts to-be) by presenting them with overly simplified solutions that keep their hands tied when something more ambitious needs to be done.

9.1.3 To Build or to Reuse

In the short term, the *fail fast* philosophy encourages us to build our applications using prefabricated components. This is fantastic at the early stage of its life cycle. If we build something really simple or whose purpose is merely to illustrate an idea, show-off how "awesome" we are, or to educate, let us be explicit about it so that other users do not feel obliged to treat our product (exercise) seriously.

In the (not so likely, probabilistically speaking) event of its becoming successful, we should start thinking about the project's long-term stability and sustainability. After all, relying on third-party functions, packages, or programs makes our software projects less… independent. This may be problematic, because:

- the dependencies might not be available on every platform or may behave differently across various system configurations,

[3] We will leave the reflection upon whether this is at all feasible for another time.

Note that this strategy is employed by many companies (and drug dealers): make the introductory experience super-smooth and fun. At the same time, do not allow your users to become independent too easily. Instead, make them rely on your product lines/proprietary solutions/payable services etc.

The free software movement with its do-it-yourself approach stresses on users' becoming autonomous. This does not contradict the user-friendliness (but that many open-source projects could benefit from becoming less exclusive is a different story, and this book tries to make a change in this area too).

- they may be huge (and can depend on other external software too),
- their APIs may change which could result in our project's not working anymore,
- their functionality can change which can lead to some unexpected behaviours.

Hence, it might be a good idea to rewrite some parts from scratch on our own.

Exercise 9.4 *Identify some R packages on CRAN with many dependencies. See what functions do they import from other packages. How often it is just a few lines of code?*

The Unix philosophy emphasises upon the building and using of minimalist yet non-trivial, single-purpose, high quality pieces of software that can work as parts of larger, custom pipelines. R serves as a glue language quite well.

In the long run, some of our software projects might converge to such a tool – it might be a good idea to standardise our API (e.g., make it available from the command-line; Section 1.2) so that the users of other languages can benefit from our work too.

Important If our project is merely a modified interface/front-end to a larger program developed by others, we should be humble about it and make sure it is not us who get all the credit for other people's work.

Also, we should state very clearly how can the original tools be used to achieve the same goals, e.g., when working from the command line.

9.2 Managing Data Flow

A function, most of the time, can and should be treated as a black box: its callers do not have to care what it hides inside. After all, they are supposed to *use* it: given some *inputs*, they expect a well-defined (read: explained in very detail in the function's manual; see Section 9.3.3) *outputs*.

9.2.1 Checking Input Data Integrity and Argument Handling

A function takes R objects of any kind as arguments, but it does not mean that feeding it with every- or any-thing is healthy for its guts.

When designing functions, it is best to handle the inputs in a manner similar to base R's behaviour. This will make our contributions easier to handle.

Unfortunately, base R functions frequently do not handle arguments of similar *kind* 100% consistently. Such variability might be due to many reasons and, in essence, is not necessarily bad. Usually, there might be many different possible behaviours and choosing one over another will make a few users unhappy anyway. Some choices might not be optimal, but they are for historical compatibility (e.g., with S). Of course, it

might also happen (but the probability is low) that there is a bug or something is not at all well designed.

This is why it is better to keep the vocabulary quite restricted (and we advocate for such minimalism in this book): even if there are exceptions to the general rules, with fewer functions, they are simply easier to remember.

Consider the following case study, illustrating that even the extremely simple scenario where we deal with a single positive integer, is not necessarily straightforward.

Exercise 9.5 *In mathematical notation, we usually denote the number of objects in a collection with the famous "*n*".*

It is implicitly assumed that such n *is a single natural number (although whether this includes* 0 *or not should be specified at some point). The functions* `runif`, `sample`, `seq`, `rep`, `strrep`, *and* `class::knn` *take it arguments. However, nothing prevents their users from trying to challenge them by passing:*

- `2.5`, `-1`, `0`, `1-1e-16` *(non-positive numbers, non-integers);*
- `NA_real_`, `Inf` *(not finite);*
- `1:5` *(not of length 1; after all, there are no scalars in R)*
- `numeric(0)` *(an empty vector);*
- `TRUE`, `NA`, `c(TRUE, FALSE, NA)`, `"1"`, `c("1", "2", "3")` *(non-numeric, but coercible to);*
- `list(1)`, `list(1, 2, 3)`, `list(1:3, 4)` *(non-atomic);*
- `"spam"` *(utter nonsense);*
- `as.matrix(1)`, `factor(7)`, `factor(c(3, 4, 2, 3))`, *etc. (compound types; see Chapter 10).*

Read the aforementioned functions' reference manuals and call them on different inputs, noting how differently they handle such atypical arguments.

Sometimes we will rely on other functions to handle the data integrity checking for us.

Example 9.6 *Let us consider the following function that generates* n *pseudorandom numbers from the unit interval rounded to* d *decimal digits. We strongly believe or hope (good faith and high competence assumption) that its authors knew what they were doing when they wrote:*

```
round_rand <- function(n, d)
{
    x <- runif(n)  # runif will check if `n` makes sense
    round(x, d)    # round will determine the appropriateness of `d`
}
```

What constitutes correct n *and* d *and how the function behaves when not provided with positive integers is determined by the two underlying functions,* `runif` *and* `round`*:*

```
round_rand(4, 1)  # the expected use case
## [1] 0.3 0.8 0.4 0.9
round_rand(4.8, 1.9)  # 4, 2
## [1] 0.94 0.05 0.53 0.89
round_rand(4, NA)
## [1] NA NA NA NA
round_rand(0, 1)
## numeric(0)
```

If well thought-out and properly documented, many such design choices can be defended. Some programmers will opt for high uniformity/compatibility across numerous tools, but there are cases where some exceptions/diversity do more good than harm.

Yet, we should keep in mind that the functions we write might be part of a more complicated data flow pipeline, where some other function generates a value that we did not expect (because of a bug therein or because we did not study its manual) and this value is used as input to our function. In our case, this would correspond to the said n or d being determined programmatically.

Example 9.7 *Continuing the previous example, the following might be somewhat challenging with regard to our being flexible and open minded:*

```
round_rand(c(100, 42, 63, 30), 1)  # length(c(...)), 1)
## [1] 0.7 0.6 0.1 0.9
round_rand("4", 1)  # as.numeric(...), 1
## [1] 0.2 0.0 0.3 1.0
```

Sure, it is quite convenient, but might lead to problems that are hard to diagnose.

Also note the not-really informative error messages in cases like:

```
round_rand(NA, 1)
## Error in runif(n): invalid arguments
round_rand(4, "1")
## Error in round(x, d): non-numeric argument to mathematical function
```

Hence, some *defensive design* mechanisms are not a bad idea, especially if they lead to generating an informative error message.

Important `stopifnot` gives a convenient means to assert the enjoyment of our expectations with regard to a function's arguments (or some intermediate values). A call to `stopifnot(cond1, cond2, ...)` is more or less equivalent to:

```
if (!(is.logical(cond1) && !any(is.na(cond1)) && all(cond1)))
    stop("`cond1` are not all TRUE")
if (!(is.logical(cond2) && !any(is.na(cond2)) && all(cond2)))
```

(continues on next page)

(continued from previous page)

```
    stop("`cond2` are not all TRUE")
...
```

Thus, if all the elements in the given logical vectors are `TRUE`, nothing happens and we can safely move on.

Example 9.8 *We can rewrite the above function as follows:*

```
round_rand2 <- function(n, d)
{
    stopifnot(
        is.numeric(n), length(n) == 1,
        is.finite(n), n > 0, n == floor(n),
        is.numeric(d), length(d) == 1,
        is.finite(d), d > 0, d == floor(d)
    )
    x <- runif(n)  # runif will check if n makes sense
    round(x, d)    # round will determine the appropriateness of d
}

round_rand2(5, 1)
## [1] 0.7 0.7 0.5 0.6 0.3
round_rand2(5.4, 1)
## Error in round_rand2(5.4, 1): n == floor(n) is not TRUE
round_rand2(5, "1")
## Error in round_rand2(5, "1"): is.numeric(d) is not TRUE
```

This implements the strictest test for "a single positive integer" possible. In the case of any violation of the underlying condition, we get a very informative error message.

Example 9.9 *At other times, we might be interested in argument checking like:*

```
if (!is.numeric(n))
    n <- as.numeric(n)
if (length(n) > 1) {
    warning("only the first element will be used")
    n <- n[1]
}
n <- floor(n)
stopifnot(is.finite(n), n > 0)
```

This way, `"4"` and `c(4.9, 100)` will all be accepted as 4[4]*.*

We see that there is always a tension between being generous/flexible and pre-

[4] Note that here we rely on S3 generics **`is.numeric`** and **`as.numeric`**; see Section 10.4.

cise/restrictive. Also, for some functions, it will be better to behave differently than the others, because of their particular use cases. Too much uniformity is as bad as chaos. Overall, we should rely on common sense, but add some lightweight foolproof mechanisms.

It is our duty to be explicit about all the assumptions we make or exceptions we allow (by writing good documentation; see Section 9.3.3).

We will revisit this topic in Section 10.4.

Note Example exercises related to the improving of the consistency of base R's handling of arguments in different domains include the **`vctrs`** and **`stringx`** packages[5]. Can these contributions be justified?

Exercise 9.10 *Reflect on how you would handle the following scenarios (and how base R and other packages or languages you know deals with them):*

- *a vectorised mathematical function (empty vectors? non-numeric inputs? what if it is equipped with the `names` attribute? what if it has other ones?);*
- *an aggregation function (what about missing values? empty vectors?);*
- *a function vectorised with regard to two arguments (elementwise vectorisation? recycling rule? only scalar vs vector or vector vs vector of the same length allowed? what if one argument is a row vector and the other is a column vector);*
- *a function vectorised with regard to all arguments (really all? maybe some exceptions are necessary?);*
- *a function vectorised with respect to the first argument but not the second (why such a restriction? when?).*

Find a few functions that match each case.

9.2.2 Putting Outputs into Context

The functions we write do not exist in a vacuum. We should put them into a much wider context: how are they going to be used when combined with other tools?

As a general rule, our functions should generate outputs of *predictable* kind, so that when we write and read the code chunks that utilise them, we can easily deduce what is going to happen.

Example 9.11 *Some base R functions do not adhere to this rule for the sake of (questionable) users' convenience. We will meet a few of them in Chapter 11 and Chapter 12. In particular,* ***`sapply`*** *and the underlying* ***`simplify2array`****, can return a list, an atomic vector, or a matrix.*

[5] Yours truly is the author of the latter and thus is guilty of multiplying entities beyond necessity.

```
simplify2array(list(1, 3:4))    # list
## [[1]]
## [1] 1
##
## [[2]]
## [1] 3 4
simplify2array(list(1, 3))      # vector
## [1] 1 3
simplify2array(list(1:2, 3:4))  # matrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Further, the index operator with `drop=TRUE`, which is the default, may output an atomic vector. But it may as well yield a matrix or a data frame.

```
(A <- matrix(1:6, nrow=3))  # an example matrix
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
A[1, ]    # vector
## [1] 1 4
A[1:2, ]  # matrix
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
A[1, , drop=FALSE]  # matrix with 1 row
##      [,1] [,2]
## [1,]    1    4
```

We proclaim that the default functions' behaviour should be to return the object of the most generic kind possible (if there are other options), and then to either have a further argument which must be explicitly set if we really wish to simplify the output, or we should ask the user to call a simplifier explicitly.

In the latter case, the simplifier should probably fail issuing an error if it is unable to neaten the object or at least apply some brute force solution (e.g., or "fill the gaps" somehow itself, possibly with a warning).

Example 9.12 *For instance:*

```
as.numeric(A[1:2, ])  # always returns a vector
## [1] 1 2 4 5
stringi::stri_list2matrix(list(1, 3:4))  # fills the gaps with NAs
##      [,1] [,2]
```

(continues on next page)

(continued from previous page)

```
## [1,] "1"  "3"
## [2,] NA   "4"
```

Ideally, a function should perform one (and only one) well-defined task. If a function tends to generate objects of different kinds, depending on the arguments provided, maybe it is better to write two functions instead?

Exercise 9.13 *Functions such as* **`rep`**, **`seq`**, *and* **`sample`** *do not perform a single task. Or do they?*

Note (*) In a purely functional programming language, we can assume the so-called *referential transparency*: a call to a *pure function* can always safely be replaced with the value it is supposed to generate. If this is true, then for the same set of argument values, the output is always the same. Furthermore, there are no side effects. In R, it is not really the case:

- a call can introduce/modify/delete the variables in other environments (see `sec:to-do`), e.g., the state of the random number generator,
- metaprogramming and lazy evaluation techniques lead to the functions' being free to interpret the argument *forms* (not only: values) freely (see `sec:to-do`),
- printing, plotting, file reading, database access have obvious consequences with regard to the state of some external resources.

Important Each function must return some value, but there are several instances (e.g., plotting, printing), where this does not make sense.

In such a case, we should consider returning **`invisible`**`(NULL)`, a `NULL` whose *first* printing will be suppressed.

Compare the following:

```
(function() NULL)()  # anonymous function, called instantly
## NULL
(function() invisible(NULL))()  # printing suppressed
x <- (function() invisible(NULL))()
print(x)  # no longer invisible
## NULL
```

Take a look at the return value of the built-on **`cat`**.

9.3 Organising and Maintaining Functions

9.3.1 Function Libraries

Definitions of frequently-used functions or datasets can be emplaced in separate source files (.R extension) for further reference.

Such libraries can be executed by calling:

```
source("path_to_file.R")
```

Exercise 9.14 *Create a source file (script) named* `mylib.R`*, where you define a function called* ***`nlargest`*** *which returns a few largest elements in a given atomic vector.*

From within another script, call ***`source`****`("mylib.R")`* *(note that relative paths to refer to the current working director; (compare Section 2.1.6) and then write a few lines of code where you test* ***`nlargest`*** *on some example inputs.*

9.3.2 Writing R Packages

When a function library grows substantially, or when there is a need for equipping it with the relevant manual pages[6] (Section 9.3.3) or compiled code (`Chapter %s`), turning it into an own R package (Section 7.3.1) might be a good idea (even if it is only for our own or small team's purpose).

A *source package* is merely a directory containing some special files and subdirectories:

- `DESCRIPTION` – a text file that gives the name of the package, its version, authors, dependencies upon other packages, license, etc.; see Section 1.1.1 of [47];
- `NAMESPACE` – a text file containing directives stating which objects are to be exported so that they are available to the package users, and which names are to be imported from other packages;
- `R` – a directory with R scripts (.R files), which define, e.g., functions, example datasets, etc.;
- `man` – a directory with R documentation files (.Rd), describing at least all the exported objects; see Section 9.3.3;
- `src` – optional; compiled code, see `Chapter %s`;
- `tests` – optional; tests to run on the package check, see Section 9.3.4;

see Section 1.5 of *Writing R Extensions* [47] for more details and other options: there is no need for us to repeat the information from the official manual as everyone can read it themself.

[6] This should read: i.e., always.

Important A source package can be built and installed from within an R session by calling:

```
install.packages("pkg_directory", repos=NULL, type="source")
```

Then it can be used as any other R package (Section 9.3.3). In particular, it can be loaded and attached via a call to:

```
library("pkg")
```

This makes all the objects listed in its `NAMESPACE` file available to the user.

Exercise 9.15 *Create your own package mypkg featuring some of the solutions to the exercises you have solved whilst studying the material in the previous chapters. When in doubt, refer to the official manual [47].*

Important Note that you *do not have to* publish your package on CRAN[7]. Many users are tempted to submit whatever they have been tinkering around with for a while. Have mercy on the busy CRAN maintainers and do not contribute to the information overload, unless you have come up with something potentially useful for other R users (make it less about you, and more about the community; thank you in advance). R packages can always be hosted on and installed from, for instance, GitLab or GitHub.

Note (*) The building and installing of packages also be done from the command line:

```
R CMD build pkg_directory
R CMD INSTALL --build pkg_directory
```

Also, some users could potentially benefit from creating own Makefiles that help automate the processes of building, testing, checking, etc.

9.3.3 Documenting R Packages

Documenting functions and commenting code thoroughly is critical, even if we just write for ourselves. Most programmers sooner or later will note that they find it hard to determine what a piece of code is doing after they took a break from it. In some sense, we always write for external audience, which incudes our future self.

The help system is one of the stronger assets of the R environment. By far we should have interacted with many man pages and got a good idea of what constitutes an informative documentation piece.

[7] And always consult the CRAN Repository Policy at https://cran.r-project.org/web/packages/policies.html.

From the technical side, R Documentation (.Rd) files should be emplaced in the `man` subdirectory of a source package. All exported objects (e.g., functions) should be described clearly. Additional topics can be covered too.

During the package install, the .Rd files are converted to various output formats, e.g., HTML or plain text, and displayed upon a call to the well-known **`help`** function.

Documentation files use a LaTeX-like syntax, which looks quite obscure to an untrained eye. The relevant commands are explained in very detail in Section 2 of *Writing R Extensions* [47].

Note The process of writing .Rd files by hand might be quite tedious, especially keeping track of the changes to the `\usage` and `\arguments` commands. Rarely do we recommend the use of third-party packages, because base R facilities are usually good enough, but **`roxygen2`** might be worth a try, because it really makes the developers' lives easier. Most importantly, it allows for documentation to be specified alongside the functions' definitions, which is much more natural.

Exercise 9.16 *Add a few manual pages to your example R package.*

9.3.4 Assuring Quality Code

Below we mention some good development practices related to maintaining quality code. This is an important topic, but writing about them is tedious to the same extent that reading about them is boring, because it is the more-artistic part of software engineering. After all, these are some heuristics that are learnt best by observing and mimicking what the others are doing (and hence the exercises below will encourage to do so).

Managing Changes and Working Collaboratively

It is a good idea to employ some source code version control system such as **`git`** to keep track of the changes made to the software.

Note It is worth investing some time and effort to learn how to use **`git`** from the command line; see https://git-scm.com/doc.

There are a few hosting providers for **`git`** repositories, with GitLab and GitHub being a popular choice amongst open-source software developers.

Not only do they support working collaboratively on the projects, but also are equipped with additional tools for reporting bugs, suggesting feature requests, etc.

Exercise 9.17 *Find where the source code of some of your most favourite R packages or other open-source projects are hosted. Explore the corresponding repositories, feature trackers, wikis, discussion boards, etc. Note that each community is different and is governed by different guidelines: after all, we are from all over the world.*

Test-driven Development and Continuous Integration

It is often hygienic to include some principles of test-driven development when writing own functions.

Exercise 9.18 *Assume that, for some reasons, we were asked to write a function to compute the root mean square (quadratic mean) of a given numeric vector. Before implementing the actual routine, it is a good idea to reflect upon what we want to achieve, especially how we want our function to behave in certain boundary cases.*

***stopifnot** gives simple means to assure a given assertion is fulfilled. If that is the case, it will move forward quietly.*

Let us say we have come up with the following set of expectations*:*

```
stopifnot(all.equal(rms(1), 1))
stopifnot(all.equal(rms(1:100), 58.167860541711519317 69))
stopifnot(all.equal(rms(rep(pi, 10)), pi))
stopifnot(all.equal(rms(numeric(0)), 0))
```

*Write a function **rms** that fulfils the above assertions.*

Exercise 9.19 *Implement your own version of the **sample** function (assuming `replace=TRUE`), using calls to **runif**. However, start by writing a few unit tests.*

There are also a couple of R packages that support writing and executing of unit tests, including **testthat**, **tinytest** (which is a lighter-weight version of the former), **RUnit**, or **realtest**. However, in the most typical use cases, relying on **stopifnot** is powerful enough.

Exercise 9.20 *(*) Consult the* Writing R Extensions *manual [47] about where and how to include unit tests in your example package.*

Note (*) R includes a built-in mechanism to check a couple of code quality areas: running `R CMD check pkg_directory` from the command line (preferably using the most recent version of R) can suggest a number of improvements.

Also, it is possible to use various continuous integration techniques that are automatically triggered when pushing changes to our software repositories; see GitLab CI or GitHub Actions. For instance, it is possible to run a package build, install, and check process upon every **git** commit. Also, CRAN features some continuous integration services, including checking the package on a range of different platforms.

Debugging

For all his life, the current author has been debugging all his programs mostly by manually printing the state of suspected variables (**printf** and the like) in different areas of the code. No shame in that.

For an interactive debugger, see the **browser** function. Also, refer to Section 9 of [51] for more details.

Some IDEs (e.g., **RStudio**) support this feature too; see their corresponding documentation.

Profiling

Typically, a program spends relatively long time executing only a small portion of code. The **Rprof** function can be a helpful tool to identify which chunks might need a rewrite, for instance using a compiled language (`Chapter %s`).

Please remember, though, that not only implementations of algorithms that have hight computational complexity can form a bottleneck, but also data input and output (such as reading files from disk, printing messages, on the console, querying Web APIs, etc.).

9.4 Special Functions: Syntactic Sugar

Some functions, such as `` `*` ``, are somewhat special. They can be referred to using an alternative syntax which for some reason most of us accepted as the default one. Below we will reveal, amongst others, that "`5 * 9`" reduces in fact to an ordinary function *call*:

```
`*`(5, 9)  # a call to `*` with 2 arguments, equivalent to 5 * 9
## [1] 45
```

9.4.1 A Note on Backticks

In Section 2.2, we have mentioned that we can assign (as in `` `<-` ``) *syntactically valid names* to our objects. Most identifiers comprised of letters, digits, dots, and underscores can be used directly in R code.

However, it is possible to label our objects however we like: non-syntactically valid (nonstandard) identifiers just need to be enclosed in backticks (back quotes, grave accents):

```
`42 a quite peculiar name :O lollolll` <- c(a=1, `b c`=2, `42`=3, `!`=4)
1/(1+exp(-`42 a quite peculiar name :O lollolll`))
##       a     b c      42       !
## 0.73106 0.88080 0.95257 0.98201
```

Of course, such names are less convenient, but still: backticks let us refer to them in any context.

9.4.2 Dollar, `$` (*)

The dollar operator, **`$`**, can be used as an alternative accessor to a single element in a named list[8].

If `label` is a syntactically valid name, then `x$label` does the same job as `x[["label"]]` (saving five keystrokes: such a burden!).

```
x <- list(spam="a", eggs="b", `eggs and spam`="c", best.spam.ever="d")
x$eggs
## [1] "b"
x$best.spam.ever  # note that a dot has no special meaning in most contexts
## [1] "d"
```

Nonstandard names must still be enclosed in backticks

```
x$`eggs and spam`  # x[["eggs and spam"]] is okay as usual
## [1] "c"
```

We are minimalist-by-design here, hence we will tend to avoid this operator, as it does not really increase the expressive power of our function repertoire. Also, it does not work on atomic vectors nor on matrices.

Furthermore, it does not work with names that are generated programmatically:

```
what <- "spam"
x$what # the same as x[["what"]] - we don't want this
## NULL
x[[what]]  # works fine
## [1] "a"
```

The support for the *partial matching* of element names has been added to provide the users working in quick-and-dirty, interactive programming sessions with some relief in the case where they find the typing of the whole label extremely problematic:

```
x$s  # x[["s"]] would return NULL; you will get no warning here!
## Warning in x$s: partial match of 's' to 'spam'
## [1] "a"
```

It is generally a bad programming practice, because the result depends on the names of other items in `x` (which might change later) and can decrease code readability. The only reason why we have obtained a warning message was because this book enforces the **`options`**`(warnPartialMatchDollar=TRUE)` setting, which – sadly – is not the default.

However, note the behaviour on ambiguous partial matches:

[8] And hence also from data frames.

```
x$egg  # ambiguous resolution
## NULL
```

9.4.3 Curly Braces, `{`

A block of statements grouped with curly braces, **`{`**, corresponds to a function call. When we write:

```
{
    print(TRUE)
    cat("two")
    3
}
## [1] TRUE
## two
## [1] 3
```

The parser translates it to a call to:

```
`{`(print(TRUE), cat("two"), 3)
## [1] TRUE
## two
## [1] 3
```

When the above is executed, every argument, one by one, is evaluated and then the last value is returned in result of that call.

9.4.4 `if`

`if` is a function too; as mentioned in Section 8.1, it returns the value corresponding to the expression evaluated conditionally. Hence, we may write:

```
if (runif(1) < 0.5) "head" else "tail"
## [1] "head"
```

but also:

```
`if`(runif(1) < 0.5, "head", "tail")
## [1] "head"
```

Note A call like **`if`**`(test, what_if_true, what_if_false)` can only work properly because of the lazy evaluation of function arguments; see Section 9.5.5.

On a side note, **`while`**, **`for`**, **`repeat`** can also be called that way, but they return **`invisible`**`(NULL)`.

9.4.5 Operators are Functions Too

Calling Built-in Operators as Functions

Every arithmetic, logical, and comparison operator is translated to a call to the corresponding function. For instance:

```
`<`(`+`(`*`(`-`(3), 4)), 5)  # 2+(-3)*4 < 5
## [1] TRUE
```

Also, `x[i]` is equivalent to `` `[`(x, i) `` and `x[[i]]` maps to `` `[[`(x, i) ``.

Knowing this will not only enable us to manipulate unevaluated R code (Chapter 15) or access the corresponding manual pages (see, e.g., `help("[")`), but also write some expressions in a more concise manner. For instance,

```
x <- list(1:5, 11:17, 21:23)
unlist(Map(`[`, x, 1))  # 1 is a further argument passed to `[`
## [1]  1 11 21
```

is equivalent to a call to `Map(function(e) e[1], x)`.

Note Unsurprisingly, the assignment operator, `` `<-` ``, is a function too. It returns the assigned value, invisibly.

Knowing that `` `<-` `` binds right to left (compare `help("Syntax")`), this is why the expression "`a <- b <- 1`" results in both `a` and `b` being assigned 1: it is equivalent to "`` `<-`("a", `<-`("b", 1)) ``" and "`` `<-`("b", 1) ``" returns 1.

Owing to the pass-by-value semantics (Section 9.5.1) we can also expect that we will always be (with the exception of environments, Chapter 16) assigning a *copy* of the value on the righthand side.

```
x <- 1:6
y <- x  # makes a copy (but delayed, on demand, for performance reasons)
y[c(TRUE, FALSE)] <- NA_real_  # modify every 2nd element
print(y)
## [1] NA  2 NA  4 NA  6
print(x)  # state of x has not changed – x and y are different objects
## [1] 1 2 3 4 5 6
```

This is especially worth pointing out to Python (amongst others) programmers, where the above assignment would mean that `x` and `y` both refer to the same (shared) object in the computer's memory.

However, with no harm done to semantics, the actual copying of `x` is postponed until absolutely necessary (Section 16.1.4). This is efficient both time- and memory-wise.

Creating Own Binary Operators

We can also introduce our own binary operators named like `%myopname%`:

```
`%:)%` <- function(e1, e2) (e1+e2)/2
5 %:)% 1:10
##  [1] 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5
```

Recall that `%%` and `%/%` are built-in operators denoting division remainder and integer division. Rarely will we be defining our own operators, but when we encounter a similar one next time, we will no longer be surprised. For instance, in Chapter 11 we will learn about `%*%` which implements matrix multiplication.

Note In Chapter 10, we will note that most existing operators can be overloaded for objects of different types.

9.4.6 Replacement Functions

Functions generally do not change the state of their arguments. However, there is some syntactic sugar that allows us to replace objects or parts thereof with new content. We call them *replacement functions*.

For instance, three of the following calls *replace* the input `x` with its modified version:

```
x <- 1:5  # example input
x[3] <- 0  # replace the 3rd element with 0
length(x) <- 7  # "replace" length
names(x) <- LETTERS[seq_along(x)]  # replace the names attribute
print(x)  # x is different than before
##  A  B  C  D  E  F  G
##  1  2  0  4  5 NA NA
```

Creating Own Replacement Functions

A replacement function is a mapping named like `name<-` with at least two parameters:

- `x` (the object to be modified),
- `...` (possible further arguments),
- `value` (as the last parameter; the object on the righthand side of the `<-` operator).

Most often, we will be interacting with existing replacement functions, not creating our own ones. However, knowing how to do the latter is key to understanding this language feature.

For example:

```
`add<-` <- function(x, where=TRUE, value)
{
    x[where] <- x[where] + value
    x  # the modified object that will replace the original one
}
```

The above aims to add some `value` to a subset of the input vector `x` (by default, to each element therein) and return its altered version that will replace the object it has been called upon.

```
y <- 1:5           # example vector
add(y) <- 10       # calls `add<-`(y, value=10)
print(y)           # y has changed
## [1] 11 12 13 14 15
add(y, 3) <- 1000  # calls `add<-`(y, 3, value=1000)
print(y)           # y has changed again
## [1]   11   12 1013   14   15
```

We see that calling "**`add`**`(y, w) <- v`" works as if we have called "`y <-` **`` `add<-` ``**`(y, w, value=v)`".

Note (*) According to [51], a call "**`add`**`(y, 3) <- 1000`" is a syntactic sugar precisely for:

```
`*tmp*` <- y  # temporary substitution
y <- `add<-`(`*tmp*`, 3, value=1000)
rm("*tmp*")  # remove the named object from the current scope
```

This has at least two implications. First, in the unlikely event that a variable `` `*tmp*` `` existed before the call to the replacement function, it will be no more, it will cease to be. It will be an ex-variable. Second, the temporary substitution guarantees that `y` must exist before the call (a function's body does not have to refer to all the arguments passed; because of lazy evaluation, see Section 9.5.5, we could get away with it otherwise).

Substituting Parts of Vectors

The replacement versions of the subsetting operators are named as follows:

- **`` `[<-` ``** is used in substitutions like "`x[i] <- value`",
- **`` `[[<-` ``** is called when we perform "`x[[i]] <- value`",
- **`` `$<-` ``** is used whilst calling "`x$i <- value`".

Here is a use case:

```
x <- 1:5
`[<-`(x, c(3, 5), NA_real_)  # returns a new object
```

(continues on next page)

(continued from previous page)

```
## [1]  1  2 NA  4 NA
print(x)  # does not change the original input
## [1] 1 2 3 4 5
```

On a side note, `` `length<-` `` can be used to expand or shorten a given vector by calling "`length(x) <- new_length`", see also Section 5.3.3.

```
x <- 1:5
x[7] <- 7
length(x) <- 10
print(x)
##  [1]  1  2  3  4  5 NA  7 NA NA NA
length(x) <- 3
print(x)
## [1] 1 2 3
```

Despite the fact that, semantically speaking, calling `` `[<-` `` results in the creation of a new vector (a modified version of the original one), we may luckily expect some performance optimisations happening behind our back (reference counting, modification in-place; see `sec:to-do`).

Exercise 9.21 *Write a function `` `extend<-` `` which pushes new elements at the end of a given vector, modifying it in place.*

```
`extend<-` <- function(x, value) ...to.do...
```

Example use:

```
x <- 1
extend(x) <- 2     # push 2 at the back
extend(x) <- 3:10  # add 3, 4, ..., 10
print(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Replacing Attributes

Many replacement functions deal with the re-setting of objects' attributes (Section 4.4).

In particular, for each special attribute, there is also its replacement version, e.g., `` `names<-` ``, `` `class<-` ``, `` `dim<-` ``, `` `levels<-` ``, etc.

```
x <- 1:3
names(x) <- c("a", "b", "c")  # change the `names` attribute
print(x)  # x has been altered
## a b c
## 1 2 3
```

Individual (arbitrary, including non-special ones) attributes can be set using `attr<-` and all of them can be established by means of a single call to `attributes<-`.

```
x <- "spam"
attributes(x) <- list(shape="oval", smell="meaty")
attributes(x) <- c(attributes(x), taste="umami")
attr(x, "colour") <- "rose"
print(x)
## [1] "spam"
## attr(,"shape")
## [1] "oval"
## attr(,"smell")
## [1] "meaty"
## attr(,"taste")
## [1] "umami"
## attr(,"colour")
## [1] "rose"
```

Also note that setting an attribute to NULL results, by convention, in its removal:

```
attr(x, "taste") <- NULL  # this is tasteless now
print(x)
## [1] "spam"
## attr(,"shape")
## [1] "oval"
## attr(,"smell")
## [1] "meaty"
## attr(,"colour")
## [1] "rose"
attributes(x) <- NULL  # remove all
print(x)
## [1] "spam"
```

Which can be useful in contexts such as:

```
x <- structure(c(a=1, b=2, c=3), some_attrib="value")
y <- `attributes<-`(x, NULL)
```

Here, x retains its attributes and y is a version of x with metadata removed.

Compositions of Replacement Functions

Updating only selected names like:

```
x <- c(a=1, b=2, c=3)
names(x)[2] <- "spam"
print(x)
```

(continues on next page)

(continued from previous page)

```
##    a spam    c
##    1    2    3
```

is possible due to the fact that "`names(x)[i] <- v`" is equivalent to:

```
old_names <- names(x)
new_names <- `[<-`(old_names, i, value=v)
x <- `names<-`(x, value=new_names)
```

Important More generally, a composition of replacement calls "`g(f(x, a), b) <- y`" yields a result equivalent to "`x <- `f<-`(x, a, value=`g<-`(f(x, a), b, value=y))`". Note that both `f` and `` `f<-` `` need to be defined, but having `g` is not necessary.

Exercise 9.22 *(*) What is "`h(g(f(x, a), b), c) <- y`" equivalent to?*

Exercise 9.23 *Write a (actually very useful!) function* `` `recode<-` `` *which replaces specific elements in a character vector with some other ones, allowing the following interface:*

```
`recode<-` <- function(x, value) ...to.do...
x <- c("spam", "bacon", "eggs", "spam", "eggs")
recode(x) <- c(eggs="best spam", bacon="yummy spam")
print(x)
## [1] "spam"       "yummy spam" "best spam"  "spam"       "best spam"
```

We see that the named character vector gives a few `from="to"` *pairs, e.g., all* `eggs` *are to be replaced by* `best spam`*.*

Now, determine which calls are equivalent to the following:

```
x <- c(a=1, b=2, c=3)
recode(names(x)) <- c(c="z", b="y")  # or equivalently = ... ?
print(x)
## a y z
## 1 2 3
y <- list(c("spam", "bacon", "spam"), c("spam", "eggs", "cauliflower"))
recode(y[[2]]) <- c(cauliflower="broccoli")  # or = ... ?
print(y)
## [[1]]
## [1] "spam"  "bacon" "spam"
##
## [[2]]
## [1] "spam"     "eggs"     "broccoli"
```

Exercise 9.24 *(*) Consider the* `` `recode<-` `` *function from the previous exercise.*

Here is an example matrix with the `dimnames` attribute whose `names` attribute is set (more details in Chapter 11):

```
(x <- Titanic["Crew", "Male", , ])
##        Survived
## Age      No Yes
##   Child   0   0
##   Adult 670 192
recode(names(dimnames(x))) <- c(Age="age", Survived="survived")
print(x)
##        survived
## age      No Yes
##   Child   0   0
##   Adult 670 192
```

This changes the `x` object. For each of the following subtasks, write a single call which alters `names(dimnames(x))` without modifying `x` in-place but returning a recoded copy of:

- *`names(dimnames(x))`,*
- *`dimnames(x)`,*
- *`x`.*

Exercise 9.25 *(*) Consider the `` `recode<-` `` function once again but now let an example object be a data frame featuring a column of class `factor`:*

```
x <- iris[c(1, 2, 51, 101), ]
recode(levels(x[["Species"]])) <- c(
    setosa="SET", versicolor="VER", virginica="VIR"
)
print(x)
##     Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1            5.1         3.5          1.4         0.2     SET
## 2            4.9         3.0          1.4         0.2     SET
## 51           7.0         3.2          4.7         1.4     VER
## 101          6.3         3.3          6.0         2.5     VIR
```

Without modifying `x` in-place, how to change `levels(x[["Species"]])` and return an altered copy of:

- *`levels(x[["Species"]])`,*
- *`x[["Species"]]`,*
- *`x`?*

9.5 Arguments and Local Variables

9.5.1 Pass by "Value"

As a general rule, functions cannot change the state of their arguments[9]. We can think of them as being passed by *value*, i.e., as if their copy was made.

```
test_change <- function(y)
{
    y[1] <- 7
    y
}

x <- 1:5
test_change(x)
## [1] 7 2 3 4 5
print(x)  # same
## [1] 1 2 3 4 5
```

If the above was not the case, the state of `x` would have been changed after the call.

9.5.2 Variable Scope

Function arguments as well as any other variables we create inside a function's body are *relative* to each call to that function.

```
test_change <- function(x)
{
    x <- x+1
    z <- -x
    z
}

x <- 1:5
test_change(x*10)
## [1] -11 -21 -31 -41 -51
print(x)  # x in the function's body was a different x
## [1] 1 2 3 4 5
print(z)  # z was local
## Error in print(z): object 'z' not found
```

Both `x` and `z` are local variables and live only whilst our function is being executed.

[9] With the exception of objects of type `environment`, which are passed by reference; see Chapter 16. Also, the fact that we have access to unevaluated R expressions can cause further deviations to this rule (see below).

The former temporarily "overshadows"[10] the object of the same name from the caller's context.

Important It is a good development practice to refrain from referring to objects not created within the current function, especially to "global" variables. We can always pass an object as an argument explicitly.

Note It is a function call as such, not curly braces per se that form a local scope.

Writing "`x <- { y <- 1; y + 1 }`", `y` is not an auxiliary variable; it is an ordinary named object created alongside `x`.

On the other hand, in "`x <- (function() { z <- 1; z + 1 })()`", `z` will not be available thereafter.

9.5.3 Closures (*)

Most user-defined functions are in fact representatives of the so-called *closures*; see `sec:to-do` and [1]. They not only consist of an R expression to evaluate, but also can carry some auxiliary data.

For instance, given two equal-length numeric vectors `x` and `y`, a call to **`approxfun`**`(x, y)` returns a *function* that linearly interpolates between the consecutive points (x_1, y_1), (x_2, y_2), and so forth, so that a corresponding y can be determined for any x.

```
x <- seq(0, 1, length.out=11)
f1 <- approxfun(x, x^2)
f2 <- approxfun(x, x^3)
f1(0.75)  # check that it is quite close to the true 0.75^2
## [1] 0.565
f2(0.75)  # compare with 0.75^3
## [1] 0.4275
```

Inspecting, however, the source codes of the above functions:

```
print(f1)
## function (v)
## .approxfun(x, y, v, method, yleft, yright, f, na.rm)
## <bytecode: 0x556f1f6ea350>
## <environment: 0x556f1f6eac80>
print(f2)
```

(continues on next page)

[10] In `sec:to-do`, we will discuss this topic in-depth; objects are bound to their names within environments. Moreover, R uses lexical (static) scoping, which is not necessarily intuitive, especially taking into account that a function's environment can always be changed.

(continued from previous page)

```
## function (v)
## .approxfun(x, y, v, method, yleft, yright, f, na.rm)
## <bytecode: 0x556f1f6ea350>
## <environment: 0x556f1f275c68>
```

we might wonder how can they produce different results — it is evident that they are identical. It turns out, however, that they internally store some additional data that is referred to upon their calls:

```
environment(f1)[["y"]]
##  [1] 0.00 0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81 1.00
environment(f2)[["y"]]
##  [1] 0.000 0.001 0.008 0.027 0.064 0.125 0.216 0.343 0.512 0.729 1.000
```

This and many more we will explore in great detail in the third part of this book.

9.5.4 Default Arguments

We have already mentioned above that, when designing functions that perform complex tasks, we will sometimes be faced with a design problem: how to find a sweet spot between being generous/mindful of the diverse needs of our users and making the API neither overwhelming nor oversimplistic.

We know that it is best if a function performs one, well-specified task, but also allows its behaviour be tuned-up if one wishes to do so. This principle can be facilitated by the use of *default arguments*.

For instance, **`log`** computes logarithms, by default the natural ones.

```
log(2.718)  # the same as log(2.78, base=exp(1)) – default base
## [1] 0.9999
log(4, base=2)  # different base
## [1] 2
```

Exercise 9.26 *Study the documentation of the following functions and note the default values that they define:* ***`round`****,* ***`hist`****,* ***`grep`****, and* ***`download.file`****.*

We can easily define our own functions equipped with such *recommended* settings:

```
test_default <- function(x=1) x

test_default()   # use default
## [1] 1
test_default(2)  # use something else
## [1] 2
```

Most often, default arguments are just constants, e.g., `1`. However, they can be any R

expressions, also including a reference to other arguments passed to the same function; see more in Section 16.4.1.

Note that default arguments will most often appear and the end of the parameter list, but see Section 9.4.6 (on replacement functions) for a well-justified exception.

9.5.5 Lazy Evaluation

In some languages, function arguments are always evaluated prior to a call. In R, though, they are only computed when actually needed. We call it *lazy* or *delayed* evaluation. Recall that in Section 8.1.4 we introduced the short-circuit evaluation operators `` `||` `` (or) and `` `&&` `` (and). They are able to do their job precisely thanks to this mechanism.

Example 9.27 *In the following example, we do not use the function's argument at all:*

```
lazy_test1 <- function(x) 1  # x not used at all

lazy_test1({cat("and now for something completely different!"); 7})
## [1] 1
```

Otherwise, we would see a message being printed out on the console.

Example 9.28 *Next, let us use x amidst other expressions in the body:*

```
lazy_test2 <- function(x)
{
    cat("it's... ")
    y <- x+x  # using x twice
    cat(" a man with two noses")
    y
}

lazy_test2({cat("and now for something completely different!"); 7})
## it's... and now for something completely different! a man with two noses
## [1] 14
```

Note that an argument is evaluated once and its value is stored for further reference. If that was not the case, we would see two messages like `and now...`.

9.5.6 Ellipsis, `...`

Let us start with an exercise.

Exercise 9.29 *Note the presence of* `...` *in the parameter list of* **`c`**, **`list`**, **`structure`**, **`cbind`**, **`rbind`**, **`cat`**, **`Map`** *(and the underlying* **`mapply`***),* **`lapply`** *(a specialised version of* **`Map`***),* **`optimise`**, **`optim`**, **`uniroot`**, **`integrate`**, **`outer`**, **`aggregate`**. *What purpose does it serve, according to these functions manual pages?*

We can create a *variadic function* by placing a dot-dot-dot (ellipsis; see **`help("dots")`**), `...`, somewhere in its parameter list. The ellipsis serves as placeholder for all objects passed to the function but not matched by any formal (named) parameters.

The easiest way to process arguments passed via `...` programmatically (see also Section 16.4.4) is by redirecting them to **`list`**.

```
test_dots <- function(...)
    list(...)

test_dots(1, a=2)
## [[1]]
## [1] 1
##
## $a
## [1] 2
```

Such a list can be processed just like… any other R list. What we can do with these arguments is only limited by our creativity (in particular, recall from Section 7.2.2 the very powerful **`do.call`** function). Still, there are two major use cases of the ellipsis[11]:

- create a new object by combining an arbitrary number of other objects:

```
c(1, 2, 3)   # 3 arguments
## [1] 1 2 3
c(1:5, 6:7)  # 2 arguments
## [1] 1 2 3 4 5 6 7
structure("spam")  # 0 additional arguments
## [1] "spam"
structure("spam", color="rose", taste="umami")  # 2 further arguments
## [1] "spam"
## attr(,"color")
## [1] "rose"
## attr(,"taste")
## [1] "umami"
cbind(1:2, 3:4)
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
cbind(1:2, 3:4, 5:6, 7:8)
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 42)
## [1] 108
```

[11] Which is somewhat similar to Python's `*args` and `**kwargs` in a function's parameter list.

- pass further arguments (as-is) to other methods :

```
lapply(list(c(1, NA, 3), 4:9), mean, na.rm=TRUE)  # mean(x, na.rm=TRUE)
## [[1]]
## [1] 2
##
## [[2]]
## [1] 6.5
integrate(dbeta, 0, 1,
    shape1=2.5, shape2=0.5)  # dbeta(x, shape1=2.5, shape2=0.5)
## 1 with absolute error < 1.2e-05
```

Example 9.30 *The documentation of* ***lapply*** *(let us call* ***help****("lapply") now) states that this function is defined as* ***lapply****(X, FUN, ...). Here, the ellipsis is a placeholder for a number of optional arguments that can be passed to* ***FUN****. Hence, if we denote the* i*-th element of a vector* `X` *by* `X[[i]]`*, calling* ***lapply****(X, FUN, ...) will return a list whose* i*-th element will be equal to* ***FUN****(X[[i]], ...).*

Exercise 9.31 *Using a single call to* ***lapply****, generate a list with three numeric vectors of lengths 3, 9, and 7, respectively, drawn from the uniform distribution on the unit interval. Then, upgrade your code to get numbers sampled form the interval* $[-1, 1]$.

9.5.7 Metaprogramming (*)

Under the hood, lazy evaluation is a quite complicated mechanism that relies upon the storing of unevaluated R expressions and special *promises* to instantiate them[12].

It turns out that we have access to such expressions programmatically. In particular, a call to the composition of **deparse** and **substitute** can convert them to a character vector:

```
test_deparse_substitute <- function(x)
    deparse(substitute(x))

test_deparse_substitute(testing+1+2+3)
## [1] "testing + 1 + 2 + 3"
test_deparse_substitute(spam & spam^2 & bacon | grilled(spam))
## [1] "spam & spam^2 & bacon | grilled(spam)"
```

Exercise 9.32 *Check out the y-axis label generated by* ***plot.default****((1:100)^2). Inspect its source code and note a call to the two aforementioned functions.*

Similarly, call ***shapiro.test****(****log****(****rlnorm****(100))) and take note of the* `data:` *field.*

A function is free to do with such an expression whatever it likes. For instance, it can manipulate it and evaluate it in a different context. Thanks to such a language feature,

[12] Such an evaluation model has been heavily inspired by Scheme [31]. It will be explained in more detail in `sec:to-do`.

certain operations can be designed so that their users can express them much more compactly. This is certainly (in theory) a very powerful tool but from practice we know many instances where it has been over/misused and made the use of R confusing.

Example 9.33 *(*) The built-in* ***`subset`*** *and* ***`transform`*** *use metaprogramming techniques to specify basic data frame transformation techniques (see Section 12.3.9). For instance:*

```
transform(
    subset(
        iris,
        Sepal.Length>=7.7 & Sepal.Width >= 3.0,
        select=c(Species, Sepal.Length:Sepal.Width)
    ),
    Sepal.Length.mm=Sepal.Length/10
)
##       Species Sepal.Length Sepal.Width Sepal.Length.mm
## 118 virginica          7.7         3.8            0.77
## 132 virginica          7.9         3.8            0.79
## 136 virginica          7.7         3.0            0.77
```

Note that none of the arguments – except `iris` *– makes sense outside of the function call contexts. In particular, neither* `Sepal.Length` *nor* `Sepal.Width` *variables exist.*

The two functions took the liberty to interpret the arguments passed as they felt like. They have created their own virtual reality within our well-defined world. The reader must refer to their documentation to discover the meaning of the special syntax used therein.

Note (*) Some functions have rather peculiar default arguments. For instance, in the manual page of **`prop.test`**, we read that the `alternative` parameter defaults to **`c`**`("two.sided", "less", "greater")` but that `"two.sided"` is actually the default one.

If we call **`print`**`(`**`prop.test`**`)`, we will find the code line responsible for this behaviour: "`alternative <-` **`match.arg`**`(alternative)`". Consider the following example:

```
test_match_arg <- function(x=c("a", "b", "c")) match.arg(x)

test_match_arg()  # missing argument – choose 1st
## [1] "a"
test_match_arg("c")  # one of the predefined options
## [1] "c"
test_match_arg("d")  # unexpected setting
## Error in match.arg(x): 'arg' should be one of "a", "b", "c"
```

In this setting, **`match.arg`** allows only an actual parameter amongst a given set of choices, but selects the first option if the argument is missing.

Unfortunately, we have to learn this behaviour by heart, because actually looking at the above source code gives us no clue about this being possible whatsoever. If such an ex-

pression was normally evaluated, we would either be passing the default argument or whatever the user passed as x (but then the function would not know about the range of possible choices). A call to "`match.arg(x, c("a", "b", "c"))`" could guarantee the desired functionality and would be much more readable. Instead, metaprogramming techniques allowed `match.arg` to access the enclosing function's default argument list without our explicitly referring to them.

One may ask "why is it so" and the only sensible answer to this will be "because its programmer decided it must be this way". Let us contemplate this for a while. In cases like this, we are dealing not with some base R language design choice that we might like or dislike, but which we should normally just accept as an inherent feature. Rather, we are struggling intellectually because of some R programmer's (mis)use (in good faith…) of R's flexibility itself. They have introduced a slang/dialect on top of our mother tongue, whose meaning is valid only within this function. Blame the middle-man, not the environment, please.

We generally advocate for avoiding metaprogramming wherever possible (and will elaborate on this later on, including formulas (`~`), built-in functions like `subset` or `transform`, etc.).

9.6 Exercises

Exercise 9.34 *Answer the following questions:*

- *Will "`stopifnot(1)`" stop? What about "`stopifnot(NA)`", "`stopifnot(TRUE, FALSE)`", and "`stopifnot(c(TRUE, TRUE, NA))`"?*
- *What does the `` `if` `` function return?*
- *Does `` `attributes<-`(x, NULL) `` modify x?*
- *When can we be interested in calling `` `[` `` and `` `[<-` `` as functions (and not as operators) directly?*
- *How to define our own binary operator? Can it have some default arguments?*
- *What are the main use cases of `` `...` ``?*
- *What is wrong with `transform`, `subset`, and `match.arg`?*
- *When a call like "`f(-1, do_something_that_takes_a_million_years())`" does not necessarily have to be a bad idea?*

Exercise 9.35 *What is the return value of a call to "`f(list(1, 2, 3))`"?*

```
f <- function(x)
{
```

(continues on next page)

(continued from previous page)

```
    for (e in x) {
        print(e)
    }
}
```

Is it `NULL`*,* `invisible(NULL)`*,* `x[[length(x)]]`*, or* `invisible(x[[length(x)]])`*?*

Exercise 9.36 *The* `split` *function also has its replacement version. Study its documentation to learn how it works.*

Exercise 9.37 *A call to* `ls(envir=baseenv())` *returns all objects defined in package* `base` *(see Chapter 16). List the names corresponding to some replacement functions.*

Important Apply the principle of test-driven development when solving the remaining exercises (or those which you have skipped intentionally).

Exercise 9.38 *Implement your own version of the* `Position` *and* `Find` *functions. Evaluation should stop as soon as the first element fulfilling a given predicate has been found.*

Exercise 9.39 *Implement your own version of the* `Reduce` *function.*

Exercise 9.40 *Write a function* `slide(f, x, k, ...)` *which returns a list* `y` *of size* `length(x)-k+1` *such that* `y[[i]] = f(x[i:(i+k-1)], ...)`

```
unlist(slide(sum, 1:5, 1))
## [1] 1 2 3 4 5
unlist(slide(sum, 1:5, 3))
## [1]  6  9 12
unlist(slide(sum, 1:5, 5))
## [1] 15
```

Exercise 9.41 *Using* `slide` *defined above, write another function that counts how many increasing pairs of numbers are featured in a given numeric vector. For instance, in* `c(0,2,1,1,0,1,6,0)` *there are three such pairs:* (0,2)*,* (0,1)*,* (1,6)*.*

Exercise 9.42 *(*) Write your own version of* `tools::package_dependencies` *with* `reverse=TRUE` *based on information extracted by calling* `utils::available.packages`*.*

10

S3 Classes

Let `x` be a randomly generated matrix with 1,000,000 rows and 1,000 columns, `y` be a data frame with results from the latest survey indicating that things are not the way most people (no matter the side of the many political spectra) think they are, and and `z` be another matrix, this time with many zeroes.

Human brain is not capable of handling too much information which is too specific. This is why we have a natural tendency to group different entities based on their similarities so as to form some more abstract classes thereof.

Also, many of us are inherently lazy. Thus, oftentimes we will take shortcuts to minimise energy (at a price to be paid later).

Printing out a matrix, a data frame, and a time series are all still instances of the displaying of things, although they surely differ in detail. Now that ad probably forgotten which objects are hidden behind `x`, `y`, and `z`, being able to simply call "**`print`**`(y)`" without having to recall that, yes, `y` is a data frame, might seem quite appealing.

This chapter introduces the so-called S3 classes [8], which provide a lightweight object oriented programming (OOP) approach for automated dispatching of calls to *generics* of the type "**`print`**`(y)`" to concrete *methods* such as "**`print.data.frame`**`(y)`", based on the *class* of the object they are invoked upon.

S3 classes in their essence are beautifully simple[1]. They are inspired[2] by the well-thought-through concepts present in other functional programming languages (such as the Common Lisp Object System; see below). Ultimately, those *generics* and *methods* are ordinary R functions (Chapter 7) and *classes* are merely additional object attributes (Section 4.4).

Of course this does not mean that wrapping our heads around them will be effortless. However, unlike other "class systems"[3], S3 is ubiquitous in R programming: suffice it

[1] However, some classes, even the built-in ones that we describe here, can be poorly designed (e.g, some crucial methods might be missing, they can be not-well-interoperable with other classes, etc.). Do not blame this messenger. Remember that the R environment is still very reliable. Also, there are cases where changing the current behaviour in one place could lead to undesirable consequences elsewhere.

[2] They were built on top of the ordinary ("old S") R, hence have certain limitations what we discuss in the sequel: classes cannot be formally defined (often we will use named lists for representing objects, and we know we cannot be any more flexible than this), and the dispatching can only be based on the class of one (usually the first, but, e.g., binary operators take both types into account) of the arguments.

[3] Other class systems may give an impression that they are alien implants that were forcefully added to our language to solve a specific, rather narrow class of problems; e.g., S4 (Section 11.5), Reference Classes (Section 16.1.5), and other ones proposed by third-party packages

to say that factors, matrices, and data frames discussed in the next chapters are quite straightforward, S3-based extensions of the concepts we have introduced so far.

10.1 Object Type vs Class

Recall that **`typeof`** (introduced in Section 4.1) returns the *internal* type of any R object. Even though there are only few admissible cases thereof[4], they open the world of endless possibilities[5].

The basic types we covered so far (mostly atomic and generic vectors; compare Figure 1) provide a basis for more complex data structures. This is thanks to the fact that they can be equipped with arbitrary attributes (Section 4.4).

```
typeof(NULL)
## [1] "NULL"
typeof(c(TRUE, FALSE, NA))
## [1] "logical"
typeof(c(1, 2, 3, NA_real_))
## [1] "double"
typeof(c("a", "b", NA_character_))
## [1] "character"
typeof(list(list(1, 2, 3), LETTERS))
## [1] "list"
typeof(function(x) x)
## [1] "closure"
```

The interesting fact is that most *compound types*, whose most prevalent instances are constructed using the mechanisms discussed in this chapter[6], only pretend they are something different from what they actually are. They are often quite good at doing their job, though, and hence might be useful. By knowing what is under their hood we will demystify them and become able to manipulate their state outside of the prescribed use cases.

Important Setting the `class` attribute might make some objects behave differently in certain scenarios.

Example 10.1 *Let us consider two identical objects equipped with different `class` attributes.*

[4] Their list is hardcoded at the C language level; compare the list of `SEXPTYPE`s in [50] and see also `Chapter %s`.

[5] In particular, later we mention `externalptrs` which are simply pointers to memory allocated on the heap, so these might be any instances of C `structs` or C++ `classes`, etc. This makes R a very extensible language.

[6] But of course there is more; see the S4 and other systems discussed in Section 11.5.

```
xt <- structure(123, class="POSIXct")  # POSIX calendar time
xd <- structure(123, class="Date")
```

Despite that both objects are being represented *using numeric vectors:*

```
c(typeof(xt), typeof(xd))
## [1] "double" "double"
```

When printed, they are decoded *quite differently:*

```
print(xt)
## [1] "1970-01-01 10:02:03 AEST"
print(xd)
## [1] "1970-05-04"
```

In the former case, 123 is treated as the number of seconds since the so-called UNIX epoch, 1970-01-01T00:00:00+0000. The latter is deciphered as the number of days since the said (quite widely used in computer systems by the way) timestamp.

We may hence suspect, and we are absolutely right, that there exists some underlying mechanism that actually calls a version *of* **`print`** *that is dependent on an object's virtual class.*

That this only depends on the `class` *attribute, which might be set, unset, or reset quite freely, is emphasised below:*

```
attr(xt, "class") <- "Date"  # change class from POSIXct to Date
print(xt)  # same 123, but now interpreted as Date
## [1] "1970-05-04"
as.numeric(xt)  # drops all attributes
## [1] 123
unclass(xd)  # drops the class attribute; `attr<-`(xd, "class", NULL)
## [1] 123
```

We are having so much fun that one more illustration can only add to joy.

Example 10.2 *Consider an example data frame:*

```
x <- iris[1:3, 1:2]  # a subset of a built-in example data frame
print(x)
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
```

This is an object of the following class (an object whose `class` *attribute is set to):*

```
attr(x, "class")
## [1] "data.frame"
```

Some may say, and they are absolutely right, that we have not covered data frames yet: this is the topic of Chapter 12, which is still ahead of us. However, from the current perspective, we are interested in the fact that an R data frame is merely a list (Chapter 4) of vectors of the same lengths equipped with `names` and `row.names` attributes.

```
typeof(x)
## [1] "list"
attr(x, "class") <- NULL  # or x <- unclass(x)
print(x)
## $Sepal.Length
## [1] 5.1 4.9 4.7
##
## $Sepal.Width
## [1] 3.5 3.0 3.2
##
## attr(,"row.names")
## [1] 1 2 3
```

Important Revealing how `x` is *actually* represented, enables us to process it (although perhaps not in the most convenient or efficient manner) using the extensive skill set that we have already[7] developed by studying the material covered in the previous part of our book (including solving all the exercises). This can be particularly useful, especially bearing in mind that some (built-in or third-party) data types are not particularly well-designed.

Note again that attributes are simple additions to R objects. However, as we said in Section 4.4.3, certain attributes are special, and `class` is one of them.

In particular, we can set `class` to be only a character vector (possibly of length greater than one; see Section 10.2.5).

```
x <- 12345
attr(x, "class") <- 1  # character vectors only
## Error in attr(x, "class") <- 1: attempt to set invalid 'class' attribute
```

Furthermore, there exists the **`class`** function that can read the value of the `class` attribute. Its replacement version is also available.

```
class(x) <- "Date"  # set; the same as attr(x, "class") <- "Date"
class(x)  # get; the same as attr(x, "class")
## [1] "Date"
```

Important The **`class`** function always yields a value, even if the corresponding at-

[7] For instance, consider once again the example from Section 5.4.3 that applies the **`split`** function on a data frame reduced to a list.

tribute is not set. We call it an *implicit* class. Compare between the following and the outputs generated by **typeof**:

```
class(NULL)  # no `class` set, because NULL cannot have attributes at all
## [1] "NULL"
class(c(TRUE, FALSE, NA))  # no attributes, so class is implicit (= typeof)
## [1] "logical"
class(c(1, 2, 3, NA_real_))  # typeof yields "double"
## [1] "numeric"
class(c("a", "b", NA_character_))
## [1] "character"
class(list(list(1, 2, 3), LETTERS))
## [1] "list"
class(function(x) x)  # typeof yields "closure"
## [1] "function"
```

Also, in Chapter 11, we will note that any object equipped with the `dim` attribute also has an implicit class:

```
(x <- as.matrix(c(1, 2, 3)))
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
attributes(x)  # `class` is not amongst the attributes
## $dim
## [1] 3 1
class(x)  # implicit class
## [1] "matrix" "array"
typeof(x)  # it is still a numeric vector (under the hood)
## [1] "double"
```

10.2 Generics and Method Dispatching

10.2.1 Generics, Default, and Custom Methods

Let us inspect the source code of the **print** function:

```
print(print)  # sic!
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x557a34c3edc8>
## <environment: namespace:base>
```

Any function like the above[8] we will call from now on an S3 (S version 3) *generic*. Its only job is to invoke **`UseMethod`**`("print")`[9]. This dispatches the control flow to another function, referred to as *method*, based on the class of the first argument.

For example, let us define an object of class `categorical` (a name that we have just come up with; we could have called it `cat`, `CtGrCl`, or `SpanishInquisition` as well), which will be our own version of the famous built-in `factor` type that we discuss later.

```
x <- structure(
    c(1, 3, 2, 1, 1, 1, 3),
    levels=c("a", "b", "c"),
    class="categorical"
)
```

We assume that such an object is a vector of small positive integers (codes) equipped with the `levels` attribute being a character vector of length no less than the maximum of the said integers. The first category will be used to decipher the meaning of code "1", for example. Hence, the above vector represents a sequence *a, c, b, a, a, a, c*.

We have not defined any special method for the printing of objects of class `categorical`. Hence, when we call **`print`**, the *default* (fallback) method will be called:

```
print(x)
## [1] 1 3 2 1 1 1 3
## attr(,"levels")
## [1] "a" "b" "c"
## attr(,"class")
## [1] "categorical"
```

This is the standard function for displaying numeric vectors that we are all well familiar with. Its name is **`print.default`**, and we can always call it directly:

```
print.default(x)  # the default print method
## [1] 1 3 2 1 1 1 3
## attr(,"levels")
## [1] "a" "b" "c"
## attr(,"class")
## [1] "categorical"
```

We can, however, introduce our own method for the custom printing of objects of class `categorical`, whose name must precisely be **`print.categorical`**:

```
print.categorical <- function(x, ...)
```

(continues on next page)

[8] Note that some functions can have a version of **`UseMethod`** hidden at the C language level (internally); see Section 10.2.3.

[9] Which in this context is equivalent to **`UseMethod`**`("print", x)`, with `x` being the first argument to the function.

(continued from previous page)

```
{
    x_character <- attr(x, "levels")[unclass(x)]
    print(x_character)  # calls `print.default`
    cat(sprintf("Categories: %s\n",
        paste(attr(x, "levels"), collapse=", ")))
    invisible(x)  # this is what all print methods do; see help("print")
}
```

Now, calling **print** automatically dispatches the control flow to the above method:

```
print(x)
## [1] "a" "c" "b" "a" "a" "a" "c"
## Categories: a, b, c
```

Of course, the default method can still be called; calling **print.default**(x) directly will output the same result as before.

Note **print.categorical** has been equipped with the dot-dot-dot attribute, because the generic **print** had one too; we should always ensure consistency ourselves[10].

10.2.2 Creating Own Generics

Introducing new S3 generics is as straightforward as defining a function that calls **UseMethod**.

For instance, here is a dispatcher which allows for creating new objects of class `categorical` based on other objects:

```
as.categorical <- function(x, ...)
    UseMethod("as.categorical")
```

We always need to define the default method:

```
as.categorical.default <- function(x, ...)
{
    x <- as.character(x)
    xu <- unique(sort(x))  # drops NAs
    structure(
        match(x, xu),
        class="categorical",
        levels=xu
    )
}
```

[10] In particular, the checking of S3 generic/method consistency is part of R package check.

Testing:

```
as.categorical(c("a", "c", "a", "a", "d", "c"))
## [1] "a" "c" "a" "a" "d" "c"
## Categories: a, c, d
as.categorical(c(3, 6, 4, NA, 9, 9, 6, NA, 3))
## [1] "3" "6" "4" NA  "9" "9" "6" NA  "3"
## Categories: 3, 4, 6, 9
```

Note that **`print.categorical`** has been invoked twice here. The above is quite flexible already, because it relies on the *generic* (Section 10.2.3) **`as.character`**, which handles a wide variety of data types. Of course, it does not mean we cannot be more precise about some particular ones.

Example 10.3 *For instance, we might want to forbid the conversion from lists, because this does not necessarily make sense:*

```
as.categorical.list <- function(x, ...)
    stop("conversion of lists to categorical is not supported")
```

The users can always be instructed in the method's documentation that they are the ones responsible for an explicit conversion of list objects to something different prior to a call to ***`as.categorical`****. Whether this was a good design choice, time will tell.*

Example 10.4 *Note that the default method deals with logical vectors perfectly fine:*

```
as.categorical(c(TRUE, FALSE, NA, NA, FALSE))  # as.categorical.default
## [1] "TRUE"  "FALSE" NA      NA      "FALSE"
## Categories: FALSE, TRUE
```

However, we might still want to introduce a specialised version, because we know a slightly more efficient algorithm (and we have nothing better to do) based on the fact that `FALSE` *and* `TRUE` *converted to numeric yield* `0` *and* `1`*, respectively:*

```
as.categorical.logical <- function(x, ...)
{
    x <- as.logical(x)  # or stopifnot(is.logical(x)) ?
    structure(
        x + 1,  # only 1, 2, and NAs will be generated
        class="categorical",
        levels=c("FALSE", "TRUE")
    )
}
```

This yields the same result, but is a bit faster:

```
as.categorical(c(TRUE, FALSE, NA, NA, FALSE))  # as.categorical.logical
```

(continues on next page)

(continued from previous page)

```
## [1] "TRUE"  "FALSE" NA      NA      "FALSE"
## Categories: FALSE, TRUE
```

Note that we have performed some argument validation at the beginning, because a user is always able to call a method directly on an R object of any kind (which is a good thing!; see Section 10.2.4). In other words, there is no guarantee that the argument x must be of type logical.

10.2.3 Built-in Generics

Many functions and operators we have introduced so far are in fact S3 generics: **`print`**, **`head`**, **`[`**, **`+`**, **`<=`**, **`as.character`**, **`as.list`**, **`round`**, **`log`**, **`sum`**, **`c`**, and **`na.omit`**, to name a few.

Some of them might not even call **`UseMethod`** explicitly; dispatching can be done internally, at the C language level[11]. Overall, the list of all S3 generics is somewhat difficult to generate[12], but at least the internal ones are enumerated in **`help("InternalMethods")`** and **`help("groupGeneric")`**.

Example 10.5 *Let us overload the **`as.character`** method. The default one does not make much sense for the objects of our custom type:*

```
as.character(x)
## [1] "1" "3" "2" "1" "1" "1" "3"
```

So:

```
as.character.categorical <- function(x, ...)
    attr(x, "levels")[unclass(x)]
```

And now:

```
as.character(x)
## [1] "a" "c" "b" "a" "a" "a" "c"
```

Exercise 10.6 *Overload the **`unique`** method for objects of class `categorical`.*

Exercise 10.7 *Overload the **`rep`** method for objects of class `categorical`.*

Example 10.8 *New types should be designed carefully. For instance, if we forget to consider overloading the to-numeric converter, we might end up with some users being puzzled[13] when they see:*

[11] Which is quite unfortunate because it decreases transparency; we need to look this information up somewhere in the documentation (instead of simply inspecting a function's source code; see, e.g., **`cbind`**). Also, it allows for some methods to be hardcoded at the C language level too and thus be unoverloadable. Some of such design choices can somewhat be defended, though, as they increase execution speed or memory consumption, but still: we are not fans thereof.

[12] See also `.knownS3Generics` and `.S3_methods_table` which are related to the advanced topics we cover in Section 16.4.5.

[13] It is a different story if this is our conscious design choice and that this is the behaviour we really

```
(x <- as.categorical(c(4, 9, 100, 9, 9, 100, 42, 666, 4)))
## [1] "4"   "9"   "100" "9"   "9"   "100" "42"  "666" "4"
## Categories: 100, 4, 42, 666, 9
as.double(x)  # as.double.default(x)
## [1] 2 5 1 5 5 1 3 4 2
```

Hence, we might want to introduce:

```
as.double.categorical <- function(x, ...)
{
    # as.double.default(as.character.categorical(x))
    as.double(as.character(x))
}
```

Which now yields:

```
as.double(x)  # as.double.categorical(x)
## [1]   4   9 100   9   9 100  42 666   4
```

Note We can still use **`unclass`** to fetch the codes:

```
unclass(x)
## [1] 2 5 1 5 5 1 3 4 2
## attr(,"levels")
## [1] "100" "4"   "42"  "666" "9"
```

This is because the above returns a class-free object, which is now guaranteed to be handled by the default methods (**`print`**, subsetting, **`as.character`**, etc.).

Exercise 10.9 *What would happen if we used* ***`as.numeric`*** *instead of* ***`unclass`*** *in* ***`print.categorical`*** *and* ***`as.character.categorical`****?*

Exercise 10.10 *Update the above methods in such a way that we can also create* named *objects of class* `categorical` *(i.e., equipped with the* `names` *attribute).*

Exercise 10.11 *Note that the levels of x are sorted lexicographically, not numerically. Introduce a single method that would make the above code (when re-run without any alterations) generate a more natural result.*

---

want. If we document this thoroughly (see how **`help`**`("factor")` discusses the behaviour of a to-numeric conversion), only a user's ignorance will there be to blame when they still are confused about this behaviour. Remember that we can never make an API totally foolproof and that there will always be someone who will challenge/stress-test our ideas. Bad design is always bad, but being overprotective has its cons as well. Choose your audience wisely.

10.2.4 Dispatching Only on One Argument and Calling S3 Methods Directly

With S3, the dispatching is done based on the class of only one[14] argument: by default, the first one from the parameter list.

For example, the **c** function is a generic which dispatches on the class of the first argument. Let us overload it for `categorical` objects (or, more precisely, create a function that will be dispatched to when the generic is called upon a series of objects such that the first element is of the said class).

```
c.categorical <- function(...)
    as.categorical(
        unlist(
            lapply(list(...), as.character)
        )
    )
```

It converts each argument to a character vector (relying on the generic **`as.character`** to take care of the details) and makes use of the fact that **`unlist`** converts a list of such atomic vectors to a single sequence of strings.

Calling **c** with the first argument being of class `categorical` dispatches to the above method:

```
x <- c(9, 5, 7, 7, 2)
xc <- as.categorical(x)
c(xc, x)  # c.categorical
##  [1] "9" "5" "7" "7" "2" "9" "5" "7" "7" "2"
## Categories: 2, 5, 7, 9
```

However, if the first argument is, say, unclassed, the default method will be consulted:

```
c(x, xc)  # default c
##  [1] 9 5 7 7 2 4 2 3 3 1
```

This method ignores the `class` attribute and sees `xc` as-it-is, a barebone numeric vector:

```
`attributes<-`(xc, NULL)  # the underlying codes
## [1] 4 2 3 3 1
```

This is not a bug! This is a well-documented (and now explained) behaviour. After all, compound types (classed objects) are merely emulated through the basic ones.

[14] This is R, so there are of course many exceptions to this rule which were made for the (debatable) sake of the R users' convenience. In particular, in Section 12.1.2 we mention that **`cbind`** and **`rbind`** will dispatch to the `data.frame` method if at least one argument is a data frame (and other ones are unclassed). Also it is worth noting that the S4 class system that we discuss in Section 11.5 allows for dispatching based on the classes many arguments.

Important In most cases, S3 methods can be called directly to get the desired outcome:

```
c.categorical(x, xc)  # force a call to the specific method
##  [1] "9" "5" "7" "7" "2" "9" "5" "7" "7" "2"
## Categories: 2, 5, 7, 9
```

Note We said "in most cases", because some methods can be:

- hardcoded at the C language level (for instance, there is no **`c.default`** defined at all[15]),
- hidden (defined in a package namespace but not exported); see Section 16.4.5,
- overloaded as a group; see Section 16.4.6.

Example 10.12 *Just for fun, let us find a partition of the* `iris` *dataset into three clusters using the* k*-means algorithm:*

```
res <- kmeans(iris[-5], centers=3, nstart=10)
print(res)
## K-means clustering with 3 clusters of sizes 50, 62, 38
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1       5.0060      3.4280       1.4620      0.2460
## 2       5.9016      2.7484       4.3935      1.4339
## 3       6.8500      3.0737       5.7421      2.0711
##
## Clustering vector:
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [71] 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [ reached getOption("max.print") -- omitted 51 entries ]
##
## Within cluster sum of squares by cluster:
## [1] 15.151 39.821 23.879
##  (between_SS / total_SS =  88.4 %)
##
## Available components:
```

(continues on next page)

[15] Also, dispatching can be done internally to internal methods: overloading **`` `<.character` ``** will have no effect unless the base **`` `<` ``** is replaced with a custom one that makes an explicit call to **`UseMethod`**. Most often, we can expect that the built-in types (e.g., atomic vectors), factors, data frames, and matrices and other arrays might be treated specially.

(continued from previous page)

```
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

The above is an object of class:

```
class(res)
## [1] "kmeans"
```

which in fact is a:

```
typeof(res)
## [1] "list"
```

The underlying list looks like:

```
unclass(res)
## $cluster
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [71] 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [ reached getOption("max.print") -- omitted 51 entries ]
##
## $centers
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1       5.0060      3.4280       1.4620      0.2460
## 2       5.9016      2.7484       4.3935      1.4339
## 3       6.8500      3.0737       5.7421      2.0711
##
## $totss
## [1] 681.37
##
## $withinss
## [1] 15.151 39.821 23.879
##
## $tot.withinss
## [1] 78.851
##
## $betweenss
## [1] 602.52
##
## $size
## [1] 50 62 38
##
```

(continues on next page)

(continued from previous page)

```
## $iter
## [1] 2
##
## $ifault
## [1] 0
```

and we already know that the above is displayed in a fancy way only because there is a ***`print`*** *method overloaded for objects of class* `kmeans`*.*

But is there really?

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

Even though the method is hidden in the ***`stats`*** *package's namespace, from Section 16.4.5 we will learn that it can be accessed by calling* ***`getS3method`****`("print", "kmeans")`* *or referring to* ***`stats:::print.kmeans`*** *(note the triple colon).*

10.2.5 Multi-class-ness

The `class` attribute can be instantiated as a character vector of any length. For example:

```
(t1 <- Sys.time())
## [1] "2023-01-15 14:21:12 AEDT"
(t2 <- strptime("2021-08-15T12:59:59+1000", "%Y-%m-%dT%H:%M:%S%z"))
## [1] "2021-08-15 12:59:59"
```

Let us inspect the two objects' classes:

```
class(t1)
## [1] "POSIXct" "POSIXt"
class(t2)
## [1] "POSIXlt" "POSIXt"
```

When we discuss date-time classes in more detail later, we will take note that the former is represented as a numeric vector, whilst the latter is a list. Hence, primarily, these two should be seen as instances of two distinct types. However, both of them have a lot in common, hence it was a wise design choice to also allow them to be seen as the representatives of the same generic category of *POSIX time* objects.

Important When calling a generic function[16] **`f`** on an object `x` of classes[17] `class1`,

[16] The case of binary operators is handled differently; see Section 10.2.6.

[17] **`UseMethod`** dispatches on the implicit class as determined by the **`class`** function (note that the `class` attribute does not necessarily have to be set in order for **`class`** to return a sensible answer).

`class2`, ..., `classK` (in this order), **`UseMethod`**`(f, x)` dispatches to the method determined as follows:

1. if `f.class1` is available[18], call it;
2. otherwise, if `f.class2` is available, call this one;
3. ...;
4. otherwise, if `f.classK` is available, invoke it;
5. otherwise, refer to the fallback `f.default`.

Example 10.13 *There is a method* ***`diff`*** *for objects of class* `POSIXt` *featuring a statement:*

```
r <- if (inherits(x, "POSIXlt")) as.POSIXct(x) else x
```

This way, we can be handling both `POSIXct` *and* `POSIXlt` *instances via the same procedure.*

Let us see in this simple scheme any magic. It is nothing more than what was described above: a way of determining which method should be called for a particular R object. It can of course be used as a mechanism to mimic (and certainly it was inspired by) the idea of inheritance in object-oriented programming languages, but note that the S3 system does not allow for defining classes in any formal manner.

For example, we cannot say that objects of class `POSIXct` inherit from `POSIXt` or each object of class `POSIXct` is also an instance of `POSIXt`. The class attribute can still be set arbitrarily on an per-object basis: we can create ones whose class is simply `POSIXct` (without the `POSIXt` part) or even **`c`**`("POSIXt", "POSIXct")` (in this very order).

10.2.6 Operator Overloading

Operators are ordinary functions (Section 9.4.5). Even though what follows can partially be implied by what we have said above, as usual in R, there will be some oddities.

For example, let us overload the index operator for objects of class `categorical`. Looking at **`help`**`("[")`, we see that the default method[19] has two arguments: `x` (the `categorical` object being sliced) and `i` (the indexer). Ours will have the same interface then:

```
`[.categorical` <- function(x, i)
{
    structure(
        unclass(x)[i],  # `[`(unclass(x), i)
        class="categorical",
        levels=attr(x, "levels")  # the same levels as input
```

(continues on next page)

[18] For more details on S3 method lookup; see Section 16.4.5.

[19] Note that the default S3 method, **`[.default`**, is hardcoded at the C language level. Therefore, we cannot refer to it directly (but **`unclass`** does the trick). Also note that we can also call **`NextMethod`** here; see Section 16.4.5.

(continued from previous page)

```
    )
}
```

We can also introduce the replacement version of this operator:

```
`[<-.categorical` <- function(x, i, value)
{
    levels <- attr(x, "levels")
    codes <- match(value, levels)  # integer codes corresponding to levels
    x <- unclass(x)
    x[i] <- codes  # default method for the replacement version of `[`
    structure(
        x,
        class="categorical",
        levels=levels  # same levels as input
    )

    # # or, equivalently:
    # structure(
    #     `[<-`(unclass(x), i, value=match(value, attr(x, "levels"))),
    #     class="categorical",
    #     levels=attr(x, "levels")
    # )
}
```

Testing:

```
x <- as.categorical(c(3, 6, 4, NA, 9, 9, 6, NA, 3))
x[1:4]
## [1] "3" "6" "4" NA
## Categories: 3, 4, 6, 9
x[1:4] <- c("6", "7")
print(x)
## [1] "6" NA  "6" NA  "9" "9" "6" NA  "3"
## Categories: 3, 4, 6, 9
```

Note how we handled the case of non-existing levels and that the recycling rule has been automagically inherited (amongst other features) from the default index operator.

Exercise 10.14 *Do these two operators preserve the* `names` *attribute of* `x`*? Is indexing with negative integers or logical vectors supported as well? Why is that/is that not the case?*

Furthermore, let us overload the `` `==` `` operator. Assume[20] that we would like two `cat-`

[20] There are of course many possible ways to implement the `` `==` `` operator. For instance, it may return either a single `TRUE` or `FALSE` depending if two objects are identical (although probably overloading **`all.equal`**

egorical objects be compared based on the actual labels they encode, in an element-wise manner:

```
`==.categorical` <- function(e1, e2)
    as.character(e1) == as.character(e2)
```

We are feeling lucky: by not performing any type checking, we rely on the particular **as.character** methods corresponding to the types of e1 and e2. Also, assuming that **as.character** always[21] returns an object of type character, we dispatch to the default method for `==` (which handles atomic vectors).

Some examples:

```
as.categorical(c(1, 3, 5, 1)) == as.categorical(c(1, 3, 1, 1))
## [1]  TRUE  TRUE FALSE  TRUE
as.categorical(c(1, 3, 5, 1)) == c(1, 3, 1, 1)
## [1]  TRUE  TRUE FALSE  TRUE
c(1, 3, 5, 1) == as.categorical(c(1, 3, 1, 1))
## [1]  TRUE  TRUE FALSE  TRUE
```

Important In the case of binary operators, dispatching is done based on the classes of both arguments. In all three example calls above, we call **`==.categorical`**, regardless of whether the classed object is the first or the second operand. If two operands are classed and different methods are overloaded for both of them, a warning will be generated and the default internal method will be called.

```
`==.A` <- function(e1, e2) "A"
`==.B` <- function(e1, e2) "B"
structure(c(1, 2, 3), class="A") == structure(c(2, NA, 3), class="B")
## Warning: Incompatible methods ("==.A", "==.B") for "=="
## [1] FALSE    NA  TRUE
```

Note In Section 16.4.6, we will mention that operators as well as certain groups of functions (including **min**, **sum**, and **all** or **abs**, **log**, and **round**) can be overloaded all at once; see also **help**("groupGeneric").

---

would be a better idea). We could also be comparing the corresponding underlying integer codes instead of the labels, etc.

[21] Which of course does not have to be the case; it is merely an assumption based on our belief in the common sense of other developers.

10.3 Common Built-in S3 Classes

Let us discuss some noteworthy built-in classes, including the ones that represent date/time information and factors (ordered or not).

Classes for representing tabular data will be dealt with in separate parts of this textbook, owing to their importance and ubiquity. Namely, matrices and other arrays are covered in Chapter 11, and data frames in Chapter 12.

The inspecting of other[22] interesting classes is left as a simple exercise to the kind reader.

10.3.1 Date, Time, etc.

The `Date` class can be used to represent... dates.

```
(x <- c(Sys.Date(), as.Date(c("1969-12-31", "1970-01-01", "2023-02-29"))))
## [1] "2023-01-15" "1969-12-31" "1970-01-01" NA
class(x)
## [1] "Date"
```

Complex types are built upon basic ones; underneath, what we deal with is:

```
typeof(x)
## [1] "double"
unclass(x)
## [1] 19372    -1     0    NA
```

which is the number of days since the so called UNIX epoch, 1970-01-01T00:00:00+0000 (midnight GMT/UTC).

The `POSIXct` (calendar time) class can be used to represent date-time objects:

```
(x <- Sys.time())
## [1] "2023-01-15 14:21:12 AEDT"
class(x)
## [1] "POSIXct" "POSIXt"
typeof(x)
## [1] "double"
unclass(x)
## [1] 1673752873
```

Underneath, it is the number of seconds since the UNIX epoch. By default, whilst

[22] An (incomprehensive) approximation to the list of available classes can be generated by calling `unique(.S3_methods_table[, 2])`.

printing, the current default timezone is used (see **`Sys.timezone`**). However, such objects can be equipped with the `tzone` attribute.

```
structure(1, class=c("POSIXct", "POSIXt"))  # using current default timezone
## [1] "1970-01-01 10:00:01 AEST"
structure(1, class=c("POSIXct", "POSIXt"), tzone="UTC")
## [1] "1970-01-01 00:00:01 UTC"
```

In both cases, the time is 1 second after the beginning of UNIX epoch. In the former, it is displayed in the current local timezone, though (on the author's PC).

Exercise 10.15 *Use **`ISOdatetime`** to inspect how midnights are displayed in different timezones.*

There is also the `POSIXlt` (local time) class, which is represented using a list of atomic vectors[23].

```
(x <- as.POSIXlt(c(a="1970-01-01 00:00:00", b="2030-12-31 23:59:59")))
##                          a                          b
## "1970-01-01 00:00:00 AEST" "2030-12-31 23:59:59 AEDT"
class(x)
## [1] "POSIXlt" "POSIXt"
typeof(x)
## [1] "list"
str(unclass(x))  # calling str instead of print to make display more compact
## List of 11
##  $ sec   : num [1:2] 0 59
##  $ min   : int [1:2] 0 59
##  $ hour  : int [1:2] 0 23
##  $ mday  : int [1:2] 1 31
##  $ mon   : int [1:2] 0 11
##  $ year  : Named int [1:2] 70 130
##   ..- attr(*, "names")= chr [1:2] "a" "b"
##  $ wday  : int [1:2] 4 2
##  $ yday  : int [1:2] 0 364
##  $ isdst : int [1:2] 0 1
##  $ zone  : chr [1:2] "AEST" "AEDT"
##  $ gmtoff: int [1:2] NA NA
```

Exercise 10.16 *Read about the meaning of each named element, especially `mon` and `year`; see **`help`**`("DateTimeClasses")`.*

The manual states that `POSIXlt` is supposedly *closer to human-readable forms* than `POSIXct`, but it is a matter of taste. Some R functions return the former, and some yield the latter type.

Exercise 10.17 *The two main functions for date formatting and parsing, **`strftime`** and **`strp-`***

[23] Which was inspired by C's `tm` structure defined in `<time.h>`.

***time**, use special field formatters (similar to those used by **sprintf**). Read about them in the R manual. What type of inputs do they accept? What outputs do they produce?*

There is a number of methods overloaded for objects of the said classes. In fact, the first call in this section already involved the use of **c.Date**.

Exercise 10.18 *Play around with the overloaded versions of **seq**, **rep**, and **as.character**.*

Note that a specific number of days or seconds can be added to or subtracted from a date or time, respectively. However, **-** (see also **diff**) can also be applied on two date-time objects, which yields an object of class `difftime`.

```
Sys.Date() - (Sys.Date() - 1)
## Time difference of 1 days
Sys.time() - (Sys.time() - 1)
## Time difference of 1 secs
```

Exercise 10.19 *Check out how objects of class `difftime` are internally represented.*

Applying other arithmetic operations on date-time objects yields an error. Also note that because date-time objects are just numbers, they can be compared to each other using binary operators[24] and methods such as **sort** and **order**[25].

Exercise 10.20 *Check out the **stringx** package [22] which replaces the base R date-time processing functions with their more portable counterparts.*

Exercise 10.21 ***system.time** can be used to measure the time to execute a given expression:*

```
system.time({
    sum(runif(1e7))  # whatever, just testing
})
##    user  system elapsed
##   0.212   0.032   0.245
```

The function returns an object of class `proc_time`. Inspect how it is represented internally.

10.3.2 Formulas (*)

Formulas (created by means of `` `~` ``) are quite advanced language constructs and hence they will be discussed much further: in Section 16.5.

Some R users refer to them in functions such as **lm**, **aggregate**, **t.test**, **boxplot**, or **plot** to specify models or queries such as "`y` as a function of `x1`, `x2`, and `x3`" and "`y` grouped/split by a combination of `x1` and `x2`" where `y`, `x1`, etc. are for example column names in a data frame or named items in a list.

There is no single standard governing how a function should interpret a formula's

[24] The overloaded group generic `Ops` prevents us from adding or multiplying two dates and defines the meaning of the comparison operators; see Section 16.4.6.

[25] See an exercise below on the use of **xtfrm**.

terms. In fact, each procedure is free to introduce its own meaning (a micro-language built on top of R). Due to this, yours truly discourages[26] their use (especially by beginners).

10.3.3 Factors

The `factor` class is often used to represent categorical (qualitative) data, e.g., species, groups, types. In fact, the example `categorical` class that we played with above has been inspired by the built-in `factor`.

```
(x <- c("spam", "spam", "bacon", "sausage", "spam", "bacon"))
## [1] "spam"    "spam"    "bacon"   "sausage" "spam"    "bacon"
(f <- factor(x))
## [1] spam    spam    bacon   sausage spam    bacon
## Levels: bacon sausage spam
```

Take note of how factors are printed: there are no double quote characters around the labels and the list of levels is given at the end.

Internally, such objects are represented as integer vectors (Section 6.4.1) with elements between 1 and k with the special (as in Section 4.4.3) `levels` attribute being a character vector of length k[27].

```
class(f)
## [1] "factor"
typeof(f)
## [1] "integer"
unclass(f)
## [1] 3 3 1 2 3 1
## attr(,"levels")
## [1] "bacon"   "sausage" "spam"
attr(f, "levels")  # also: levels(f)
## [1] "bacon"   "sausage" "spam"
```

Factors are often used instead of character vectors defined over a small number of unique labels[28], where there is a need to manipulate their levels easily.

```
attr(f, "levels") <- c("a", "b", "c")  # also levels(f) <- c(...new...)
print(f)
```

(continues on next page)

[26] For example, **`lm.fit`** can be used instead of **`lm`**. It is slightly more difficult to learn, surely, but has the added benefit of making sure the user knows that all model variables are not magical (especially the nonlinear/mixed effect terms).

[27] [51] states: *Factors are currently implemented using an integer array to specify the actual levels and a second array of names that are mapped to the integers. Rather unfortunately users often make use of the implementation in order to make some calculations easier. This, however, is an implementation issue and is not guaranteed to hold in all implementations of R.* Still, *fortunately*, this has been a de facto standard for factors for a very long time.

[28] Recall that there is a global (internal) string cache, hence having many duplicated strings is not an issue, memory-use-wisely.

(continued from previous page)

```
## [1] c c a b c a
## Levels: a b c
```

The underlying codes remain the same.

Certain operations on vectors of small integers are relatively easy to implement, especially those concerning element grouping: splitting, counting, plotting (e.g., Figure 13.1). It is because the integer codes can naturally be used whilst indexing other vectors. In Section 5.4, we mentioned a few functions related to this, such as **`match`**, **`split`**, **`findInterval`**, and **`tabulate`**. Specifically, the latter can be implemented like "for each `i`, increase `count[factor_codes[i]]` by one".

Exercise 10.22 *Study the source code of the* ***`factor`*** *function. Note the use of* ***`as.character`***, ***`unique`***, ***`order`***, *and* ***`match`***.

Exercise 10.23 *Implement a simplified version of* ***`table`*** *based on* ***`tabulate`***. *It should work for objects of class* `factor` *and return a named numeric vector.*

Exercise 10.24 *Implement your own version of* ***`cut`*** *based on* ***`findInterval`***.

Important The **`as.numeric`** method has not been overloaded for factors. Therefore, when we call the generic, the default method is used: it returns the underlying integer codes as-is. This can surprise the unaware users when they play with factors that feature levels consisting of strings representing integer numbers:

```
(g <- factor(c(11, 15, 16, 11, 13, 4, 15)))  # converts numbers to strings
## [1] 11 15 16 11 13 4  15
## Levels: 4 11 13 15 16
as.numeric(g)  # the underlying codes
## [1] 2 4 5 2 3 1 4
as.numeric(as.character(g))  # to get the numbers en-coded
## [1] 11 15 16 11 13  4 15
```

Unfortunately, support for factors is often hardcoded at the C language level, which will make this class behave less predictably (from the R perspective). In particular, the manual overloading of methods for factor objects might have no effect.

Important If `f` is a factor, then `x[f]` does not behave like `x[`**`as.character`**`(f)]` (indexing by labels, using the `names` attribute). Instead, we get `x[`**`as.numeric`**`(f)]` (the underlying codes will determine the positions).

```
h <- factor(c("a", "b", "a", "c", "a", "c"))
levels(h)[h]  # the same as c("a", "b", "c")[c(1, 2, 1, 3, 1, 3)]
## [1] "a" "b" "a" "c" "a" "c"
c(b="x", c="y", a="z")[h]  # names are not used whilst indexing
```

(continues on next page)

(continued from previous page)

```
##   b   c   b   a   b   a
## "x" "y" "x" "z" "x" "z"
c(b="x", c="y", a="z")[as.character(h)]  # names are used now
##   a   b   a   c   a   c
## "z" "x" "z" "y" "z" "y"
```

More often than not, indexing by factors will happen "accidentally", leading to our being slightly puzzled. In particular, factors look much like character vectors when they are featured in data frames:

```
(df <- data.frame(A=c("x", "y", "z"), B=factor(c("x", "y", "z"))))
##   A B
## 1 x x
## 2 y y
## 3 z z
class(df[["A"]])
## [1] "character"
class(df[["B"]])
## [1] "factor"
```

(*) Up until R 4.0, many functions (including **`data.frame`** and **`read.csv`**) had the `stringsAsFactors` option (see **`help`**`("options")`) set to `TRUE`, which resulted in all character vectors' being automatically converted to factors when, e.g., creating data frames (compare Chapter 12). Luckily, this is no longer the case, but they can still be encountered sporadically: for instance, the built-in `iris` dataset has the fifth column of class:

```
class(iris[["Species"]])
## [1] "factor"
```

Important Be careful when combining factors and not-factors:

```
x <- factor(c("A", "B", "A"))
c(x, "C")
## [1] "1" "2" "1" "C"
c(x, factor("C"))
## [1] A B A C
## Levels: A B C
```

Exercise 10.25 *Note that when subsetting a factor object, the result will have the `levels` attribute inherited as-is.*

```
f[c(1, 2)]
## [1] c c
## Levels: a b c
```

Implement your own version of the ***droplevels*** *function which removes the unused attributes.*

Exercise 10.26 *The replacement version of the index operator does not automatically add new levels to the modified object:*

```
x <- factor(c("A", "B", "A"))
`[<-`(x, 4, value="C")  # like in x[4] <- "C"
## Warning in `[<-.factor`(x, 4, value = "C"): invalid factor level, NA
## generated
## [1] A    B    A    <NA>
## Levels: A B
```

Implement your own version of ***`[<-.factor`*** *which is capable of doing so.*

10.3.4 Ordered Factors

Note that when creating factors, we can enforce a particular ordering and the number of levels:

```
x <- c("spam", "spam", "bacon", "sausage", "spam", "bacon")
factor(x, levels=c("eggs", "bacon", "sausage", "spam"))
## [1] spam    spam    bacon   sausage spam    bacon
## Levels: eggs bacon sausage spam
```

If we want the arrangement of the levels to define a linear ordering relation over set of the labels, we can call:

```
(f <- factor(x, levels=c("eggs", "bacon", "sausage", "spam"), ordered=TRUE))
## [1] spam    spam    bacon   sausage spam    bacon
## Levels: eggs < bacon < sausage < spam
class(f)
## [1] "ordered" "factor"
```

This yields an ordered factor, which enables comparisons like:

```
f[f >= "bacon"]  # what's not worse than bacon?
## [1] spam    spam    bacon   sausage spam    bacon
## Levels: eggs < bacon < sausage < spam
```

How is that possible? Well, based on information provided in this chapter it will come as no surprise that it is because... someone has implemented a comparison operator for objects of class `ordered`.

10.4 Argument Checking Revisited

Recall that anything can be passed as a function's input. Here are some additions to the topic we touched upon in Section 9.2.1.

Despite that compound objects are internally represented through basic types (such as numeric vectors, lists, or combinations thereof) and attributes, unless we really know better (which, by the way, this book is all about), we should be relying on the hopefully well-thought-out methods developed by the class' designer.

Ideally, when checking arguments passed to a function, determining if an object is of a desired type should be solely done by means of the generics like **`is.class`**. If that is not the case, a call to **`as.class`** should be used to make sure we will be dealing with an object of the desired type.

If a conversion is not possible, either because a specific method is unavailable or because its designer decided that this must be the case, whatever the consequences are is not necessarily our problem anymore.

We should explain to the user that the input type assurance is done via this very mechanism and, in case they get any surprising results, they should check/redefine the underlying **`is.class`** or **`as.class`** themselves.

This is of course not watertight, and there will be users complaining that they get *unexpected* or *confusing* (in their opinion) outputs. With infinitely many potential types, however, we cannot respond to every possible situation.

Example 10.27 *As an illustration, here is a function that counts the number of occurrences of items in a* numerised *(digitised?) version of a given object:*

```
numtable <- function(x)
{
    if (!is.numeric(x)) x <- as.numeric(x)  # two generics!

    u <- unique(x)
    structure(
        tabulate(match(x, u)),
        names=as.character(u)
    )
}
```

Let us assume that the user has been informed (in the corresponding documentation page) that `x` *must be a numeric vector (as in* `is.numeric`*) or an object coercible to (by means of* `as.numeric`*).*

The callers will be stress-testing our function in many different ways:

```
numtable(c(1, 3, 5, 5, 1, 5))
```

(continues on next page)

(continued from previous page)

```
## 1 3 5
## 2 1 3
```

This is an intended behaviour.

```
numtable(c("1", "3", "5", "5", "1", "5"))
## 1 3 5
## 2 1 3
```

This makes sense too, a character vector consisting of number-strings has been fed on input.

```
numtable(c("a", "e", "z", "z", "a", "z"))
## Warning in numtable(c("a", "e", "z", "z", "a", "z")): NAs introduced by
## coercion
## <NA>
##    6
```

Does the output make no sense? Of course, it does, they have just passed something not easily coercible to a numeric vector. Note the warning that suggests there is something wrong. The user needs to correct their possible mistake by themself.

```
numtable(list(1, 2, 3:10, 2))
## Error in numtable(list(1, 2, 3:10, 2)): 'list' object cannot be coerced to type 'doubl
```

Again, makes sense. 'But I think that this function should apply **`unlist`** *automatically' – well, if you want such a behaviour, why don't you call* **`numtable(unlist(...))`** *yourself? It is not so difficult.*

```
numtable(factor(c(1, 3, 5, 5, 1, 5)))
## 1 2 3
## 2 1 3
```

Is this confusing? No; this is a well-documented behaviour of **`as.numeric`** *on objects of type* `factor` *(which was designed by another developer). A user should know (but we can remind them about it in the documentation) that in this case,* **`as.character`** *should rather be called first.*

Of course, sometimes users might discover bugs or unexpected behaviours, especially related to boundary cases we have not been considerate enough to inspect. We are of course the ones to blame for the following:

```
numtable(numeric(0))  # bug: this should be corrected
## <NA>
##    0
```

10.5 (Over)using the Forward-pipe Operator, `|>` (*)

The object-oriented programming paradigm is useful when we wish to define a new data type, perhaps even a hierarchy of types. Many development teams find it an efficient tool to organise larger pieces of software. Yet, in the broad data science and numerical computing domains, we are more often consumers of OOP rather than class designers.

Thanks to the discussed method dispatch mechanism, our language is easily extensible and something that mimics a new data type can easily be introduced. Most importantly, methods can be added or removed during run-time, e.g., when importing external packages.

However, R is still a functional programming language, where functions not only are first-class citizens; they are privileged. Of course, there are some inherent limitations stemming from the ingenious simplicity of S3: method dispatch is usually based only on the type of the first function argument, classes cannot be defined formally (but see Section 11.5) and that there is no real encapsulation (we cannot actually hide data from a user[29]). However, overall the whole concept has proven quite versatile.

In functional programming, emphasis is on operations (verbs), not data (nouns). This leads to a very readable syntax, for example (assuming that **`square`**, `x`, and `y` are sensibly defined), the mean squared error can be written as:

```
mean(square(x-y))
```

This is very user-centric. However, when implementing more complex data processing pipelines, a programmer thinks "first, I need to do this, then I need to do that, and afterwards...". When they write it down, there can be some pressing of HOME and END keys on the keyboard involved. This should not be a problem for most programmers.

```
finally(thereafter(then(first(x))))
```

However, some people are inherently lazy, always complaining and/or always trying to "optimise"[30] things.

Example 10.28 *Base R is of course extremely flexible and we can introduce new vocabulary as we please. In Chapter 12, we study an example, where we define:*

[29] Which can be good, right?

[30] Do not get yours truly wrong, improving things is generally good, but overall, in the long run, as a compulsive habit ("this is what (some) stakeholders want", "we need to be agile and responsive", etc.), it is not really sustainable (also for the environment!). Less is better, even though a little harder. By introducing a new, parallel syntax, we not only duplicate the existing features and cause some divide in the community (some users will be introduced to the system through the new interface and not know the old one, others will have to learn the new syntax to be able to communicate with the former group) but also introduce a whole new set of issues (how to make the new functions interoperable with each other in a seamless manner, etc.).

- ***group_by*** *(a function that splits a data frame with respect to a combination of levels in given named columns and returns a list of data frames with class* `list_dfs`*),*
- ***aggregate.list_dfs*** *(which applies a given aggregation function on each column of each data frame in a given list), and*
- ***mean.list_dfs*** *(a specialised version of the former that calls* ***mean****).*

The specifics do not really matter now, let us just consider the notation we use when the operations are chained:

```
# select a few rows and columns from the `iris` data frame:
iris_subset <- iris[51:150, c("Sepal.Width", "Petal.Length", "Species")]
# compute the averages of all variables grouped by Species:
mean(group_by(iris_subset, "Species"))
##      Species            x  Mean
## 1 versicolor  Sepal.Width 2.770
## 2 versicolor Petal.Length 4.260
## 3  virginica  Sepal.Width 2.974
## 4  virginica Petal.Length 5.552
```

This is quite readable: we compute the mean in groups defined by `Species` *in a subset of the* `iris` *data frame. All* verbs *appear on the lefthand side of the expression, with the last (the most important?) operation being listed first.*

By the way, self-explanatory variable names and rich comments are priceless.

In more traditional object-oriented programming languages, either the method list is sealed inside[31] the class' definition (like in C++), or some peculiar patches must be applied to inject a method (like in Python)[32]. There, it is the objects that are *told* what to do: they are treated as black boxes.

Some popular languages rely on the message-passing syntax, where operations are propagated (and written) left-to-right instead of inside-out. For instance, in C++ and Python (amongst many others), "`obj.method1().method2()`" means "call **`method1`** on `obj` and then call **`method2`** on the result.

Since R 4.1.0, there is a *pipe operator*[33], `` `|>` ``, which is merely a syntactic sugar for translating between the message-passing and function-centric notion. In a nutshell, writing:

```
x |> f() |> g(y) |> h()
(x-y) |> square() |> mean()
```

is equivalent to:

[31] When methods are parts of particular classes, there can be a lot of duplicated code. Functional OOP can be more developer-friendly as we can implement all methods related to roughly the same functionality in one spot.

[32] See also the concept of extension methods in C# or Kotlin or, to some extent, class inheritance.

[33] It was inspired by `` `|` `` in Bash and `` `|>@` `` in F# and Julia (which are part of the language specification). Also, there is a `` `%>%` `` operator (and related ones) in the R package **`magrittr`**.

```
h(g(f(x), y))
mean(square(x-y))
```

This syntax is developer-centric: it emphasises on the order in which the operations are executed, something that could always be achieved with the function-centric form and perhaps a few auxiliary variables.

Example 10.29 *In the above example, a pipe operator version of the* `iris` *aggregation exercise would look like:*

```
iris_subset |> group_by("Species") |> mean()
```

This book is minimalist by design and there is nothing that cannot be achieved without the pipe operator, hence we will be refraining[34] ourselves from using it.

Note When writing code interactively, we may sometimes benefit from the use of the the rightward `` `->` `` operator. Suffice to say that "`name <- value`" and "`value -> name`" are synonymous.

This way we can write some lengthy code, store the result in an intermediate variable, and then continue on in the next line (possibly referring to that auxiliary value more than once). In the long run, multiplying entities without necessity is unsustainable.

For instance:

```
iris[, c("Sepal.Width", "Petal.Length", "Species")] -> .
.[ .[, "Species"] %in% c("versicolor", "virginica"), ] -> .
mean(group_by(., "Species"))
##      Species            x  Mean
## 1 versicolor  Sepal.Width 2.770
## 2 versicolor Petal.Length 4.260
## 3  virginica  Sepal.Width 2.974
## 4  virginica Petal.Length 5.552
```

"." is as good a variable name as any other one.

10.6 Exercises

Exercise 10.30 *Answer the following questions:*

- *How to display the source code of the default methods for* ***`head`*** *and* ***`tail`****?*

[34] Which some readers would name an *uncool* (old-school) approach, but we do not care. Remember that the functional syntax is the native one and we have to be able to understand it anyway.

- *Can there be, at the same time, one object of class* `c("A", "B")` *and another one of class* `c("B", "A")`*?*
- *If* `f` *is a factor, what are the relationships between* `as.character(f)`*,* `as.numeric(f)`*,* `as.character(as.numeric(f))`*, and* `as.numeric(as.character(f))`*?*
- *If* `x` *is a named vector and* `f` *is a factor, is* `x[f]` *equivalent to* `x[as.character(f)]` *or rather* `x[as.numeric(f)]`*?*

Exercise 10.31 *A user calls:*

```
plot(x, y, col="red", ylim=c(1, max(x)), log="y")
```

where `x` *and* `y` *are numeric vectors. Consult* `help("plot")` *for the meaning of the* `ylim` *and* `log` *arguments. Was that straightforward?*

Exercise 10.32 *Explain why the two following calls yield significantly different results and present a workaround:*

```
c(Sys.Date(), "1970-01-01")
## [1] "2023-01-15" "1970-01-01"
c("1970-01-01", Sys.Date())
## [1] "1970-01-01" "19372"
```

Exercise 10.33 *Write methods* `head` *and* `tail` *for our example* `categorical` *class.*

Exercise 10.34 *(*) Write an R package that defines S3 class* `categorical` *and a couple of methods therefor. Note the need for the use of the* `S3method` *directive* `NAMESPACE`*; see [47].*

Exercise 10.35 *Inspect the result of a call to* `binom.test(79, 100)`*. Find the method responsible for the pretty-printing of such objects.*

Exercise 10.36 *Inspect the result of a call to* `rle(c(1, 1, 1, 4, 3, 3, 3, 3, 3,, 2, 2))`*. Find the method responsible for the pretty-printing of such objects.*

Exercise 10.37 *Read more about the* `connection` *class; see the* Value *section in* `help("connections")`*.*

Exercise 10.38 *Read about the subsetting operators overloaded for the* `package_version` *class; see* `help("numeric_version")`*.*

Exercise 10.39 *There are* `xtfrm` *methods overloaded for classes such as* `numeric_version`*,* `difftime`*,* `Date`*, and* `factor`*. Find out how they work and where they might be useful (especially in relation to* `order` *and* `sort`*; see also Section 12.3.1).*

Exercise 10.40 *Give an example where* `split(x, list(y1, y2))` *(with default arguments) will fail to generate the correct result.*

Exercise 10.41 *Write a function that determines the mode, i.e., the most frequently occurring value in a given object of class* `factor`*. If the mode is not unique, return a randomly chosen one (each with the same probability).*

Exercise 10.42 *Implement your own version of the* `gl` *function.*

Exercise 10.43 *Check out which built-in date-time functions the* ***`stringx`*** *package replaces with more portable ones.*

11

Matrices and Other Arrays

When we equip an atomic or generic vector with the `dim` attribute, it automatically becomes an object of S3 class `array`. In particular, two-dimensional arrays (primary S3 class `matrix`) allow us to represent *tabular* data where items are aligned into rows and columns:

```
structure(1:6, dim=c(2, 3))  # a matrix with 2 rows and 3 columns
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

This (combined with the fact that there are many built-in functions overloaded for the `matrix` class) opens up a range of new possibilities, which we explore in this chapter. In particular, we discuss how to perform basic algebraic operations such as matrix multiplication, transpose, finding eigenvalues, and performing various decompositions. We also cover data wrangling operations such as array subsetting and column- and rowwise aggregation.

Important Oftentimes, a numeric matrix with n rows and m will be used to represent n points (samples) in an m-dimensional (with m features or variables) space, $\mathbb{R}^m$.

Furthermore, in the next chapter, we will introduce data frames: matrix-like objects whose columns can be of any (not necessarily the same) type.

11.1 Creating Arrays

11.1.1 `matrix` and `array`

A matrix can be conveniently created by means of the **`matrix`** function.

```
(A <- matrix(1:6, byrow=TRUE, nrow=2))
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

The above converted an atomic vector of length six into a matrix with two rows. The number of columns was determined automatically (`ncol=3` could have been passed to get the same result).

Important By default, the elements of the input vector are read columnwise:

```
matrix(1:6, ncol=3)  # byrow=FALSE
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

A matrix can be equipped with dimension names, being a list of two character vectors of appropriate sizes, labelling each row and column, in this order:

```
matrix(1:6, byrow=TRUE, nrow=2, dimnames=list(c("x", "y"), c("a", "b", "c")))
##   a b c
## x 1 2 3
## y 4 5 6
```

Alternatively, to create a matrix, we can use the **`array`** function, which requires the number of rows and columns be specified explicitly.

```
array(1:6, dim=c(2, 3))
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Note that the elements are consumed in a column-major manner.

Arrays of dimensionality other than 2 are also possible. Here is a one-dimensional array. When printed, it is indistinguishable from an atomic vector (but still the `class` attribute is set to `array`):

```
array(1:6, dim=6)
## [1] 1 2 3 4 5 6
```

And now for something completely different: a three-dimensional array of size 3-by-4-by-2

```
array(1:24, dim=c(3, 4, 2))
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

(continues on next page)

(continued from previous page)

```
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

which can be thought of as two matrices of size 3-by-4 (because how else can we print out a 3D object on a 2D console?).

The **array** function can be fed with the `dimnames` argument too. For instance, the above three-dimensional hypertable would require a list of three character vectors, of sizes 3, 4, and 2, respectively.

Exercise 11.1 *That 10-dimensional arrays are also possible the reader is encouraged to try out themself.*

11.1.2 Promoting and Stacking Vectors

We can promote an ordinary vector to a column vector, i.e., a matrix with one column by calling:

```
as.matrix(1:2)
##      [,1]
## [1,]    1
## [2,]    2
cbind(1:2)
##      [,1]
## [1,]    1
## [2,]    2
```

and to a row vector:

```
t(1:3)  # transpose
##      [,1] [,2] [,3]
## [1,]    1    2    3
rbind(1:3)
##      [,1] [,2] [,3]
## [1,]    1    2    3
```

Actually, **cbind** and **rbind** stand for column- and row-bind; they allow multiple vectors and matrices be stacked one after/below another:

```
rbind(1:4, 5:8, 9:10, 11)  # row bind
##      [,1] [,2] [,3] [,4]
```

(continues on next page)

(continued from previous page)

```
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10    9   10
## [4,]   11   11   11   11
cbind(1:4, 5:8, 9:10, 11)  # column bind
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   11
## [2,]    2    6   10   11
## [3,]    3    7    9   11
## [4,]    4    8   10   11
cbind(1:2, 3:4, rbind(11:13, 21:23))  # vector, vector, 2x3 matrix
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3   11   12   13
## [2,]    2    4   21   22   23
```

and so forth.

Unfortunately, the *generalised* recycling rule is not implemented in full:

```
cbind(1:4, 5:8, cbind(9:10, 11))  # different from cbind(1:4, 5:8, 9:10, 11)
## Warning in cbind(1:4, 5:8, cbind(9:10, 11)): number of rows of result is
## not a multiple of vector length (arg 1)
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   11
## [2,]    2    6   10   11
```

Note that the first two arguments are of length four.

11.1.3 Simplifying Lists

`simplify2array` is an extension of the **`unlist`** function. Given a list of atomic vectors, each of length one, it will return a flat atomic vector. However, if a list of equisized vectors of greater lengths is given, these will be converted to a matrix.

```
simplify2array(list(1, 11, 21))  # each of length 1
## [1]  1 11 21
simplify2array(list(1:3, 11:13, 21:23, 31:33))  # each of length 3
##      [,1] [,2] [,3] [,4]
## [1,]    1   11   21   31
## [2,]    2   12   22   32
## [3,]    3   13   23   33
simplify2array(list(1, 11:12, 21:23))  # no can do
## [[1]]
## [1] 1
##
```

(continues on next page)

(continued from previous page)

```
## [[2]]
## [1] 11 12
##
## [[3]]
## [1] 21 22 23
```

Note that in the second example, each vector becomes a separate column of the resulting matrix[1].

See Section 12.3.7 for a few more examples.

Example 11.2 *There are quite a few functions that call the above automatically by default (compare the `simplify` or `SIMPLIFY` (sic!) argument in **`sapply`**, **`tapply`**, **`mapply`**, **`replicate`**, etc.).*

For instance:

```
min_mean_max <- function(x) c(Min=min(x), Mean=mean(x), Max=max(x))
sapply(split(iris[["Sepal.Length"]], iris[["Species"]]), min_mean_max)
##      setosa versicolor virginica
## Min   4.300      4.900     4.900
## Mean  5.006      5.936     6.588
## Max   5.800      7.000     7.900
```

Take note of what constitutes the columns of the return matrix.

Exercise 11.3 *Note the behaviour of **`as.matrix`** on list arguments. Write your own version of **`simplify2array`** named **`as.matrix.list`** that* always *returns a matrix. If a list of non-equisized vectors is given, fill the missing cells with `NA`s.*

Important Sometimes a call to **`do.call(cbind, x)`** might be a better idea than a referral to **`simplify2array`**. Provided that `x` is a list of atomic vectors, it *always* returns a matrix: shorter vectors are recycled (which might be welcome, but not necessarily).

```
do.call(cbind, list(a=c(u=1), b=c(v=2, w=3), c=c(i=4, j=5, k=6)))
## Warning in (function (..., deparse.level = 1) : number of rows of result
## is not a multiple of vector length (arg 2)
##   a b c
## i 1 2 4
## j 1 3 5
## k 1 2 6
```

Example 11.4 *Consider a named toy list of numeric vectors:*

[1] Which can easily be explained by the fact that matrix elements are stored in a columnwise order.

```
x <- list(a=runif(10), b=rnorm(15))
```

Compare the results generated by ***sapply*** *(which calls* ***simplify2array****):*

```
sapply(x, function(e) c(Mean=mean(e)))
##  a.Mean  b.Mean
## 0.57825 0.12431
sapply(x, function(e) c(Min=min(e), Max=max(e)))
##            a       b
## Min 0.045556 -1.9666
## Max 0.940467  1.7869
```

with its version based on ***do.call*** *and* ***cbind****:*

```
sapply2 <- function(...)
    do.call(cbind, lapply(...))

sapply2(x, function(e) c(Mean=mean(e)))
##            a       b
## Mean 0.57825 0.12431
sapply2(x, function(e) c(Min=min(e), Max=max(e)))
##            a       b
## Min 0.045556 -1.9666
## Max 0.940467  1.7869
```

Note that ***sapply*** *may return an atomic vector with somewhat surprising* `names`*.*

11.1.4 Beyond Numeric Arrays

Arrays built upon atomic vectors other than numeric ones are possible too. For instance, later we will stress that comparisons featuring matrices are performed elementwisely, which results in logical matrices:

```
A >= 3
##       [,1]  [,2] [,3]
## [1,] FALSE FALSE TRUE
## [2,]  TRUE  TRUE TRUE
```

Furthermore, matrices of character strings can be useful too:

```
matrix(strrep(LETTERS[1:6], 1:6), ncol=3)
##      [,1] [,2]   [,3]
## [1,] "A"  "CCC"  "EEEEE"
## [2,] "BB" "DDDD" "FFFFFF"
```

And of course complex matrices:

```
A + 1i
##      [,1] [,2] [,3]
## [1,] 1+1i 2+1i 3+1i
## [2,] 4+1i 5+1i 6+1i
```

We are not limited to *atomic* vectors: lists can be a basis for arrays as well:

```
matrix(list(1, 11:21, "A", list(1, 2, 3)), nrow=2)
##      [,1]       [,2]
## [1,] 1          "A"
## [2,] integer,11 list,3
```

Some elements are not displayed properly, but they are still there.

11.1.5 Internal Representation

An object of S3 class `array` is an atomic vector or a list equipped with the `dims` attribute, which is a vector of nonnegative integers. Interestingly, we do not have to set the `class` attribute explicitly: the accessor function **`class`** will return an implicit[2] class anyway (compare Section 4.4.3).

```
class(1)  # atomic vector
## [1] "numeric"
class(structure(1, dim=rep(1, 1)))  # 1D array (vector)
## [1] "array"
class(structure(1, dim=rep(1, 2)))  # 2D array (matrix)
## [1] "matrix" "array"
class(structure(1, dim=rep(1, 3)))  # 3D array
## [1] "array"
```

Note that a 2-dimensional array is additionally of class `matrix`.

Optional dimension names are represented by means of the `dimnames` attribute, which is a list of d character vectors, where d is the array's dimensionality.

```
(A <- structure(1:6, dim=c(2, 3), dimnames=list(letters[1:2], LETTERS[1:3])))
##   A B C
## a 1 3 5
## b 2 4 6
dim(A)  # or attr(A, "dim")
## [1] 2 3
dimnames(A)  # or attr(A, "dimnames")
## [[1]]
## [1] "a" "b"
```

(continues on next page)

[2] Also, note that calling **`unclass`** on a matrix has no effect.

(continued from previous page)

```
##
## [[2]]
## [1] "A" "B" "C"
```

Important Internally, elements in an array are always stored in the columnwise (column-major, Fortran) order:

```
as.numeric(A)  # drop all attributes to reveal the underlying numeric vector
## [1] 1 2 3 4 5 6
```

Setting `byrow=TRUE` in a call to the **`matrix`** only affects the order in which this function reads a given source vector, not the column/row-majorness.

```
(B <- matrix(1:6, ncol=3, byrow=TRUE))
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
as.numeric(B)
## [1] 1 4 2 5 3 6
```

The two said special attributes can be modified through the replacement functions **`` `dim<-` ``** and **`` `dimnames<-` ``** (and of course **`` `attr<-` ``** as well). In particular, changing `dim` does not alter the underlying atomic vector; it only affects how other functions, including the corresponding **`print`** method, interpret their placement on a virtual grid:

```
`dim<-`(A, c(3, 2))  # not the same as transpose of A
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

What we have obtained is a different *view* on the same *flat* data vector. Also, `dimnames` were dropped because its size became incompatible with the newly requested dimensionality.

Exercise 11.5 *Study the source code of the* **`nrow`**, **`NROW`**, **`ncol`**, **`NCOL`**, **`rownames`**, **`row.names`**, *and* **`colnames`** *functions.*

Interestingly, for one-dimensional arrays, the **`names`** function returns a sensible value (based on the `dimnames` attribute which is a list featuring one character vector), despite the `names` attribute's not being set.

What is more, `dimnames` itself can be named:

```
names(dimnames(A)) <- c("ROWS", "COLUMNS")
print(A)
##      COLUMNS
## ROWS A B C
##    a 1 3 5
##    b 2 4 6
```

It is still a numeric matrix, but the presentation thereof is slightly prettified.

Exercise 11.6 ***outer*** *applies a given (vectorised elementwise) function on each pair of elements from two vectors, forming a two-dimensional result grid. Based on two calls to* ***rep****, implement your own version thereof.*

Some examples:

```
outer(c(x=1, y=10, z=100), c(a=1, b=2, c=3, d=4), "*")  # multiplication
##     a   b   c   d
## x   1   2   3   4
## y  10  20  30  40
## z 100 200 300 400
outer(c("A", "B"), 1:8, paste, sep="-")  # concatenate strings
##      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]
## [1,] "A-1" "A-2" "A-3" "A-4" "A-5" "A-6" "A-7" "A-8"
## [2,] "B-1" "B-2" "B-3" "B-4" "B-5" "B-6" "B-7" "B-8"
```

Exercise 11.7 *Show how* ***match****(y, z) can be implemented with* ***outer****. Is its time and memory complexity optimal, though?*

Exercise 11.8 ***table*** *creates a contingency matrix/array that counts the number of unique pairs of corresponding elements from one or more vectors of equal lengths. Implement its one- and two-argument version based on* ***tabulate****.*

For example:

```
tips <- read.csv(paste0("https://github.com/gagolews/teaching-data/raw/",
    "master/other/tips.csv"), comment.char="#")  # a data.frame (list)
table(tips[["day"]])
##
##  Fri  Sat  Sun Thur
##   19   87   76   62
table(tips[["smoker"]], tips[["day"]])
##
##       Fri Sat Sun Thur
##   No    4  45  57   45
##   Yes  15  42  19   17
```

11.2 Array Indexing

Array subsetting can be performed by means of an overloaded[3] `[` method, which we will usually provide with many indexers – two in the matrix case; see `help("[")`.

In this section, we will be referring to the two following example matrices.

```
(A <- matrix(1:12, byrow=TRUE, nrow=3))
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
B <- A
dimnames(B) <- list(
    c("a", "b", "c"),     # row labels
    c("x", "y", "z", "w") # column labels
)
B
##   x  y  z  w
## a 1  2  3  4
## b 5  6  7  8
## c 9 10 11 12
```

Subsetting higher-dimensional arrays will be covered at the end.

11.2.1 Arrays Are Built upon Basic Vectors

Firstly, let us note, though, that subsetting based on one indexer (as in Chapter 5) will refer to the underlying flat vector.

For instance:

```
A[6]
## [1] 10
```

This is the element in the third row, second column: recall that values are stored in a column-major order.

11.2.2 Selecting Individual Elements

Mathematically, we say that our example 3-by-4 real matrix $\mathbf{A} \in \mathbb{R}^{3\times 4}$ is like:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}.$$

[3] Hidden deeply at the C language level.

Matrix elements are aligned in a two-dimensional grid. They are organised into rows and columns. Hence, we can pinpoint a cell using two indexes: $a_{i,j}$ refers to the i-th row and the j-th column.

Similarly in R:

```
A[3, 2]  # 3rd row, 2nd column
## [1] 10
B["c", "y"]  # using dimnames == B[3, 2]
## [1] 10
```

11.2.3 Selecting Rows and Columns

Some textbooks, and we are fond of this notation here as well, mark with $\mathbf{a}_{i,\cdot}$ a vector that consists of all the elements in the i-th row and with $\mathbf{a}_{\cdot,j}$ all items in the j-th column.

In R, these will correspond to one of the indexers being left out.

```
A[3, ]  # 3rd row
## [1]  9 10 11 12
A[, 2]  # 2nd column
## [1]  2  6 10
B["c", ]  # or B[3, ]
##  x  y  z  w
##  9 10 11 12
B[, "y"]  # or B[, 2]
##  a  b  c
##  2  6 10
```

Let us stress that `A[1]`, `A[1, ]`, and `A[, 1]` have all different meanings. Also, we see that the results' `dimnames` are adjusted accordingly; see also **`unname`** which can take care of them once and for all.

Exercise 11.9 *Use **`duplicated`** to remove repeating rows in a given numeric matrix (see also **`unique`**).*

11.2.4 Dropping Dimensions

Extracting an individual element or a single row/column from a matrix yields an atomic vector. If the `dim` attribute consists of 1s only, it will be removed whatsoever.

In order to obtain proper row and column vectors, we can request the preservation of the dimensionality of the output object (and, more precisely, the length of `dim`) by passing `drop=FALSE` to `` `[` ``.

```
A[1, 2, drop=FALSE]  # 1st row, 2nd columns
##      [,1]
## [1,]    2
```

(continues on next page)

(continued from previous page)

```
A[1,  , drop=FALSE]  # 1st row
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
A[ , 2, drop=FALSE]  # 2nd column
##      [,1]
## [1,]    2
## [2,]    6
## [3,]   10
```

Important The `drop` argument unfortunately defaults to `TRUE`. Many bugs could be avoided more easily otherwise, especially when the indexers are generated programmatically.

See also the **`drop`** function which gets rid of the dimensions that have only one level.

Note For list-based matrices, we can also use a multi-argument version of **`[[`** to extract the individual elements.

```
C <- matrix(list(1, 11:12, 21:23, 31:34), nrow=2)
C[1, 2]  # for `[`, input type is the same as the output type, hence a list
## [[1]]
## [1] 21 22 23
C[1, 2, drop=FALSE]
##      [,1]
## [1,] integer,3
C[[1, 2]]  # extract
## [1] 21 22 23
```

11.2.5 Selecting Submatrices

Indexing based on two vectors, both of length two or more, extracts a sub-block of a given matrix:

```
A[1:2, c(1, 2, 4)]  # rows 1,2 columns 1,2,4
##      [,1] [,2] [,3]
## [1,]    1    2    4
## [2,]    5    6    8
B[c("a", "b"), -3]
##   x y w
## a 1 2 4
## b 5 6 8
```

Note again that `drop=TRUE` is the default, which affects the behaviour if one of the indexers is a scalar.

```
A[c(1, 3), 3]
## [1]  3 11
A[c(1, 3), 3, drop=FALSE]
##      [,1]
## [1,]    3
## [2,]   11
```

Exercise 11.10 *Overload the* ***`split`*** *function for the* `matrix` *class in such a way that, given a matrix with* n *rows and an object of class* `factor` *of length* n *(or a list of such objects), a list of* n *matrices is returned. For example:*

```
split.matrix <- ...to.do...
A <- matrix(1:12, nrow=3)  # matrix whose rows are to be split
s <- factor(c("a", "b", "a"))  # determines the grouping of rows
split(A, s)
## $a
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    3    6    9   12
##
## $b
##      [,1] [,2] [,3] [,4]
## [1,]    2    5    8   11
```

11.2.6 Selecting Elements Based on Logical Vectors

Logical vectors can also be used as indexers, with consequences that are not hard to guess:

```
A[c(TRUE, FALSE, TRUE), -1]  # select 1st and 3rd row, all but 1st column
##      [,1] [,2] [,3]
## [1,]    4    7   10
## [2,]    6    9   12
B[B[, "x"]>1 & B[, "x"]<=9, ]  # all rows where x is in (1, 9]
##   x  y  z  w
## b 5  6  7  8
## c 9 10 11 12
A[2, colMeans(A)>6, drop=FALSE]  # 2nd row of the columns with means > 6
##      [,1] [,2]
## [1,]    8   11
```

Note In Section 11.3, we note that comparisons involving matrices are performed in an elementwise manner, for example:

```
A>7
##       [,1]  [,2]  [,3] [,4]
## [1,] FALSE FALSE FALSE TRUE
## [2,] FALSE FALSE  TRUE TRUE
## [3,] FALSE FALSE  TRUE TRUE
```

Such logical matrices can be used to index other matrices of the same size. This always yields a (flat) vector in return.

```
A[A>7]
## [1]  8  9 10 11 12
```

This nothing else than the single-indexer subsetting involving two flat vectors (a numeric and a logical one); the `dim` attributes are not considered here.

Exercise 11.11 *Implement your own versions of* ***`max.col`****,* ***`lower.tri`****, and* ***`upper.tri`****.*

11.2.7 Selecting Based on Two-Column Numeric Matrices

We can also index a matrix `A` with a two-column matrix of positive integers `I`, for instance:

```
(I <- cbind(
    c(1, 3, 2, 1, 2),
    c(2, 3, 2, 1, 4)
))
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    3
## [3,]    2    2
## [4,]    1    1
## [5,]    2    4
```

Now `A[I]` gives an easy access to:

- `A[I[1, 1], I[1, 2]]`,
- `A[I[2, 1], I[2, 2]]`,
- `A[I[3, 1], I[3, 2]]`,
- ...

and so forth. In other words, each row of `I` gives the coordinates of the elements to extract.

```
A[I]
## [1]  4  9  5  1 11
```

This is exactly `A[1, 2], A[3, 3], A[2, 2], A[1, 1], A[2, 4]`. The result is always a flat vector.

Note **`which`** can also return a list of index matrices:

```
which(A>7, arr.ind=TRUE)
##      row col
## [1,]   2   3
## [2,]   3   3
## [3,]   1   4
## [4,]   2   4
## [5,]   3   4
```

Moreover, **`arrayInd`** can be used to convert flat indexes to multidimensional ones.

Exercise 11.12 *Implement your own version of* ***`arrayInd`*** *and a function performing the inverse operation.*

Exercise 11.13 *Implement your own version of* ***`diag`****.*

11.2.8 Higher-Dimensional Arrays

For d-dimensional arrays, indexing can involve up to d indexes.

This is particularly useful for dim-named arrays that represent contingency tables over a Cartesian product of multiple factors. The built-in **`datasets`**`::Titanic` object is an example of this:

```
str(dimnames(Titanic))  # for reference (note that dimnames are named)
## List of 4
##  $ Class   : chr [1:4] "1st" "2nd" "3rd" "Crew"
##  $ Sex     : chr [1:2] "Male" "Female"
##  $ Age     : chr [1:2] "Child" "Adult"
##  $ Survived: chr [1:2] "No" "Yes"
Titanic["Crew", "Male", "Adult", "Yes"]
## [1] 192
```

gives the number of adult male members of the crew who survived the accident. Also:

```
Titanic["Crew", , "Adult", ]
##         Survived
## Sex       No Yes
##   Male   670 192
##   Female   3  20
```

and so on.

Exercise 11.14 *Check if the above four-dimensional array can be indexed by means of matrices with four columns.*

11.2.9 Replacing Elements

There is of course also a multidimensional version of the replacement subsetting function, `` `[<-` ``.

Generally, subsetting drops all attributes except `names`, `dim`, and `dimnames` (unless it does not make sense otherwise). The replacement variant of the index operator modifies vector values but generally preserves all the attributes.

This enables transforming matrix elements like:

```
B[B<10] <- A[B<10]^2
print(B)
##   x  y  z   w
## a 1 16 49 100
## b 4 25 64 121
## c 9 10 11  12
B[] <- rep(seq_len(NROW(B)), NCOL(B))  # NOT the same as B <- ...
print(B)
##   x y z w
## a 1 1 1 1
## b 2 2 2 2
## c 3 3 3 3
```

Take note of the preservation of `dim` and `dimnames`.

Exercise 11.15 *Given a character matrix with entities that can be interpreted as numbers like:*

```
(X <- rbind(x=c(a="1", b="2"), y=c("3", "4")))
##   a   b
## x "1" "2"
## y "3" "4"
```

convert it to a numeric matrix with a single line of code.

11.3 Common Operations

11.3.1 Matrix Transpose

The matrix *transpose*, mathematically denoted with $\mathbf{A}^T$, is available via a call to **t**:

```
(A <- matrix(1:6, byrow=TRUE, nrow=2))
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
t(A)
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Hence, if $\mathbf{B} = \mathbf{A}^T$, then it is a matrix such that $b_{i,j} = a_{j,i}$. In other words, in the transposed matrix, rows become columns and columns become rows.

For higher-dimensional arrays, a generalised transpose can be achieved with **`aperm`** (try permuting the dimensions of `Titanic`). Also note that the conjugate transpose of a complex matrix $\mathbf{A}$ is done via **`Conj(t(A))`**.

11.3.2 Vectorised Mathematical Functions

Vectorised functions such as **`sqrt`**, **`abs`**, **`round`**, **`log`**, **`exp`**, **`cos`**, **`sin`**, etc., operate on each element of a given array[4].

```
A <- matrix(1/(1:6), nrow=2)
round(A, 2)  # rounds every element in A
##      [,1] [,2] [,3]
## [1,]  1.0 0.33 0.20
## [2,]  0.5 0.25 0.17
```

Exercise 11.16 *Using a single call to* **`matplot`**, *which accepts the* `y` *argument be a matrix, draw a plot of* $\sin(x)$, $\cos(x)$, $|\sin(x)|$, *and* $|\cos(x)|$ *for* $x \in [-2\pi, 6\pi]$.

11.3.3 Aggregating Rows and Columns

When we call an aggregation function on an array, it will reduce all elements to a single number:

```
(A <- matrix(1:12, byrow=TRUE, nrow=3))
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
mean(A)
## [1] 6.5
```

[4] They are simply applied on each element of the underlying flat vector. In Section 5.5, we have mentioned that unary functions preserve *all* attributes of their inputs, hence also `dim` and `dimnames`.

The **apply** function may be used to summarise individual rows or columns in a matrix:

- **apply**(A, 1, f) applies a given function **f** on each *row* of a matrix A;
- **apply**(A, 2, f) applies **f** on each *column* of A.

For instance:

```
apply(A, 1, mean)  # synonym: rowMeans(A)
## [1]  2.5  6.5 10.5
apply(A, 2, mean)  # synonym: colMeans(A)
## [1] 5 6 7 8
```

Note that the function being applied does not have to return a single number:

```
apply(A, 2, range)  # min and max
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    9   10   11   12
apply(A, 1, function(row) c(Min=min(row), Mean=mean(row), Max=max(row)))
##      [,1] [,2] [,3]
## Min   1.0  5.0  9.0
## Mean  2.5  6.5 10.5
## Max   4.0  8.0 12.0
```

Take note of the columnwise order of the output values.

apply works on higher-dimensional arrays too:

```
apply(Titanic, 1, mean)  # 1st dimension - Class
##     1st     2nd     3rd    Crew
##  40.625  35.625  88.250 110.625
apply(Titanic, c(1, 3), mean)  # w.r.t. Class (1st) and Age (3rd)
##       Age
## Class  Child  Adult
##   1st   1.50  79.75
##   2nd   6.00  65.25
##   3rd  19.75 156.75
##   Crew  0.00 221.25
```

11.3.4 Binary Operators

In Section 5.5, we have stated that binary elementwise operations, such as addition or multiplication, preserve the attributes of the longer input or both (with the first argument preferred to the second) if they are of equal sizes.

Taking into account that:

- an array is simply a flat vector equipped with the `dim` attribute, and

- we refer to the respective *default* methods when applying binary operators

allows us to deduce how `+`, `<=`, `&`, etc. behave in a number of different contexts.

Array-Array. First, let us note what happens when we operate on two arrays of identical dimensionalities.

```
(A <- rbind(c(1, 10, 100), c(-1, -10, -100)))
##      [,1] [,2] [,3]
## [1,]    1   10  100
## [2,]   -1  -10 -100
(B <- matrix(1:6, byrow=TRUE, nrow=2))
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
A + B  # elementwise addition
##      [,1] [,2] [,3]
## [1,]    2   12  103
## [2,]    3   -5  -94
A * B  # elementwise multiplication (not: algebraic matrix multiply)
##      [,1] [,2] [,3]
## [1,]    1   20  300
## [2,]   -4  -50 -600
```

This is simply the addition and multiplication of the corresponding elements of two given matrices.

Array-Scalar. Second, we can apply scalar-matrix operations:

```
(-1)*B
##      [,1] [,2] [,3]
## [1,]   -1   -2   -3
## [2,]   -4   -5   -6
A^2
##      [,1] [,2]  [,3]
## [1,]    1  100 10000
## [2,]    1  100 10000
```

These multiplied each element in `B` by -1 and squared every element in `A`, respectively.

Also note that the behaviour of comparison operators is similar:

```
A >= 1 & A <= 100
##       [,1]  [,2]  [,3]
## [1,]  TRUE  TRUE  TRUE
## [2,] FALSE FALSE FALSE
```

Array-Vector. Next, based on the recycling rule and the fact that elements are ordered columnwisely, we get that:

```
B * c(10, 100)
##      [,1] [,2] [,3]
## [1,]   10   20   30
## [2,]  400  500  600
```

multiplied every element in the first *row* by 10 and each element in the second row by 100.

Note that if wish to multiply each element in the first, second, ..., etc. *column* by the first, second, ..., etc. value in a vector, we should *not* call:

```
B * c(1, 100, 1000)
##      [,1] [,2] [,3]
## [1,]    1 2000  300
## [2,]  400    5 6000
```

but rather:

```
t(t(B) * c(1, 100, 1000))
##      [,1] [,2] [,3]
## [1,]    1  200 3000
## [2,]    4  500 6000
```

or:

```
t(apply(B, 1, `*`, c(1, 100, 1000)))
##      [,1] [,2] [,3]
## [1,]    1  200 3000
## [2,]    4  500 6000
```

Exercise 11.17 *Write a function which standardises the values in each column of a given matrix: for each column, from every element, subtract the mean and then divide it by the standard deviation. Try to do it in a few different ways, including via a call to* ***`apply`***, ***`sweep`***, ***`scale`***, *or based solely on arithmetic operators.*

Note Some sanity checks are being done on the `dim` attributes, so not every configuration is possible. Notice the following peculiarities:

```
getOption("error")
## NULL
A + t(B)  # dim==c(2, 3) vs dim==c(3, 2)
## Error in A + t(B): non-conformable arrays
A * cbind(1, 10, 100)  # this is too good to be true
## Error in A * cbind(1, 10, 100): non-conformable arrays
```

(continues on next page)

(continued from previous page)

```
A * rbind(1, 10)  # but A * c(1, 10) works...
## Error in A * rbind(1, 10): non-conformable arrays
A + 1:12
## Error in eval(expr, envir, enclos): dims [product 6] do not match the length of ob
A + 1:5  # partial recycling is okay
## Warning in A + 1:5: longer object length is not a multiple of shorter
## object length
##      [,1] [,2] [,3]
## [1,]    2   13  105
## [2,]    1   -6  -99
```

11.4 Numerical Matrix Algebra (*)

Many data analysis and machine learning algorithms, in their essence, involve quite simple matrix algebra and numerical mathematics. Suffice to say that anyone serious about data science and scientific computing should learn the necessary theory; see, for example, [25] and [26].

R is a convenient interface to the well-tested and stable algorithms from, amongst others, **LAPACK** and **BLAS**[5]. Below we mention only a few of them. Note that there are many third-party packages featuring hundreds of algorithms tackling differential equations, constrained and unconstrained optimisation, etc.; exploring the relevant CRAN Task Views[6] can give a good overview.

11.4.1 Matrix Multiplication

`*` performs elementwise multiplication. For what we call (algebraic) matrix multiplication, we should use the `%*%` operator.

Refreshing from a basic linear algebra course, matrix multiplication can only be performed on two matrices of *compatible sizes*: the number of columns in the left matrix must match the number of rows in the right operand.

Given $\mathbf{A} \in \mathbb{R}^{n \times p}$ and $\mathbf{B} \in \mathbb{R}^{p \times m}$, their multiply is a matrix $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{n \times m}$ such that $c_{i,j}$ is the dot product of the i-th row in $\mathbf{A}$ and the j-th column in $\mathbf{B}$:

$$c_{i,j} = \mathbf{a}_{i,\cdot} \cdot \mathbf{b}_{\cdot,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j},$$

[5] (*) Note that we can select the underlying implementation of **BLAS** at R's compile time; see Section A.3 in [49]. Some of them are faster than others.

[6] https://cran.r-project.org/web/views/

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

For instance:

```
(A <- rbind(c(1, 1, 1), c(-1, 1, 0)))
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]   -1    1    0
(B <- rbind(c(3, -1), c(1, 2), c(6, 1)))
##      [,1] [,2]
## [1,]    3   -1
## [2,]    1    2
## [3,]    6    1
A %*% B
##      [,1] [,2]
## [1,]   10    2
## [2,]   -2    3
```

Note When applying `%*%` on one or more flat vectors, their dimensionality will be promoted automatically to make the operation possible. Note that, however, `c(a, b) %*% c(c, d)` gives a scalar $ac + bd$, and not a 2-by-2 matrix.

Further, `crossprod(A, B)` yields $\mathbf{A}^T\mathbf{B}$ and `tcrossprod(A, B)` determines $\mathbf{A}\mathbf{B}^T$ more efficiently than relying on `%*%`. Note that we can omit the second argument and get $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}$, respectively

```
crossprod(c(1, 1))  # Euclidean norm squared
##      [,1]
## [1,]    2
crossprod(c(1, 1), c(-1, 1))  # dot product of two vectors
##      [,1]
## [1,]    0
crossprod(A)  # same as t(A) %*% A, i.e., dot products of all column pairs
##      [,1] [,2] [,3]
## [1,]    2    0    1
## [2,]    0    2    1
## [3,]    1    1    1
```

Recall that if the dot product of two vectors is equal to 0, we say that they are orthogonal (perpendicular).

Exercise 11.18 *(*) Write your own versions of `cov` and `cor`: functions to compute the covariance and correlation matrices. Make use of the fact that the former can be determined with `crossprod` based on a centred version of an input matrix.*

11.4.2 Solving Systems of Linear Equations

The `solve` function can be used to solve m systems of n linear equations of the form $\mathbf{AX} = \mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{X}, \mathbf{B} \in \mathbb{R}^{n \times m}$ (via the LU decomposition with partial pivoting and row interchanges).

11.4.3 Norms and Metrics

Given an n-by-m matrix $\mathbf{A}$, calling `norm(A, "1")`, `norm(A, "2")`, and `norm(A, "I")`, we can compute the operator norms:

$$\begin{aligned} \|\mathbf{A}\|_1 &= \max_{j=1,\ldots,m} \sum_{i=1}^{n} |a_{i,j}|, \\ \|\mathbf{A}\|_2 &= \sigma_1(\mathbf{A}) = \sup_{\mathbf{0} \neq \mathbf{x} \in \mathbb{R}^m} \frac{\|\mathbf{Ax}\|_2}{\|\mathbf{x}\|_2} \\ \|\mathbf{A}\|_I &= \max_{i=1,\ldots,n} \sum_{j=1}^{m} |a_{i,j}|, \end{aligned}$$

where σ_1 gives the largest singular value (see below).

Also, passing `"F"` as the second argument yields the Frobenius norm, $\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} a_{i,j}^2}$, and `"M"` computes the max norm, $\|\mathbf{A}\|_M = \max_{\substack{i=1,\ldots,n \\ j=1,\ldots,m}} |a_{i,j}|$.

Note that if $\mathbf{A}$ is a column vector, then $\|\mathbf{A}\|_F$ and $\|\mathbf{A}\|_2$ are equivalent and are referred to as the Euclidean norm. Moreover, $\|\mathbf{A}\|_M = \|\mathbf{A}\|_I$ give the supremum norm and outputs $\|\mathbf{A}\|_1$ the Manhattan (taxicab) one.

Exercise 11.19 *Given an* n*-by-*m *matrix* $\mathbf{A}$ *representing* m *vectors in* $\mathbb{R}^n$*, normalise each column so that you obtain* m *unit vectors, i.e., whose Euclidean norm is 1.*

Further, `dist` determines all pairwise distances between a set of n vectors in $\mathbb{R}^m$, written as a n by m matrix.

For example, let us consider three vectors in $\mathbb{R}^2$:

```
(X <- rbind(c(1, 1), c(1, -2), c(0, 0)))
##      [,1] [,2]
## [1,]    1    1
## [2,]    1   -2
## [3,]    0    0
as.matrix(dist(X, "euclidean"))
##        1      2      3
## 1 0.0000 3.0000 1.4142
## 2 3.0000 0.0000 2.2361
## 3 1.4142 2.2361 0.0000
```

From that we see that the distance between the 1st and the 3rd vector is ca. 1.41421. Euclidean, maximum, Manhattan, and Canberra distances/metrics are available, amongst others.

Exercise 11.20 *`dist` returns an object of S3 class `dist`. Inspect how it is represented.*

Example 11.21 *`adist` implements a couple of string metrics. For example:*

```
x <- c("spam", "bacon", "eggs", "spa", "spams", "legs")
names(x) <- x
(d <- adist(x))
##       spam bacon eggs spa spams legs
## spam     0     5    4   1     1    4
## bacon    5     0    5   5     5    5
## eggs     4     5    0   4     4    2
## spa      1     5    4   0     2    4
## spams    1     5    4   2     0    4
## legs     4     5    2   4     4    0
```

gives the Levenshtein distances between each pair of strings. In particular, we need two edit operations (character insertions, deletions, or replacements) to turn `"eggs"` *into* `"legs"` *(add* `l` *and remove* `g`*).*

Example 11.22 *Objects of class* `dist` *can be used to perform hierarchical clusterings of datasets. For example:*

```
h <- hclust(as.dist(d), method="average")  # see also: plot(h, labels=x)
cutree(h, 3)
##  spam bacon  eggs   spa spams  legs
##     1     2     3     1     1     3
```

yields a grouping into 3 clusters determined by the average linkage (`"legs"` *and* `"eggs"` *are grouped together,* `"spam"`, `"spa"`, `"spams"` *form another cluster, and* `"bacon"` *is a singleton).*

11.4.4 Eigenvalues and Eigenvectors

`eigen` returns a sequence of eigenvalues $(\lambda_1, \ldots, \lambda_n)$ (ordered nondecreasingly w.r.t. $|\lambda_i|$) and a matrix $\mathbf{V}$ whose columns define the corresponding eigenvectors (scaled to unit length) of a given matrix $\mathbf{X}$. To recall, by definition it holds that $\mathbf{X}\mathbf{v}_{\cdot,i} = \lambda_i \mathbf{v}_{\cdot,i}$.

Here are the eigenvalues and the corresponding eigenvectors of an example matrix (defining rotation in 2D by $\pi/3$):

```
(R <- rbind(c(cos(pi/3), -sin(pi/3)), c(sin(pi/3), cos(pi/3))))
##         [,1]    [,2]
## [1,] 0.50000 -0.86603
## [2,] 0.86603  0.50000
eigen(R)
## eigen() decomposition
## $values
## [1] 0.5+0.86603i 0.5-0.86603i
##
## $vectors
##                  [,1]             [,2]
```

(continues on next page)

(continued from previous page)

```
## [1,] 0.70711+0.00000i 0.70711+0.00000i
## [2,] 0.00000-0.70711i 0.00000+0.70711i
```

Example 11.23 *Consider a pseudorandom sample from a bivariate[7] normal distribution; see Figure 11.1.*

```
Z <- matrix(rnorm(2000), ncol=2)  # independent N(0, 1)
Z <- Z %*% rbind(c(1, 0), c(0, sqrt(5)))  # scaling
Z <- Z %*% R  # rotation
Z <- t(c(10, -5) + t(Z))  # translation
plot(Z, asp=1)
```

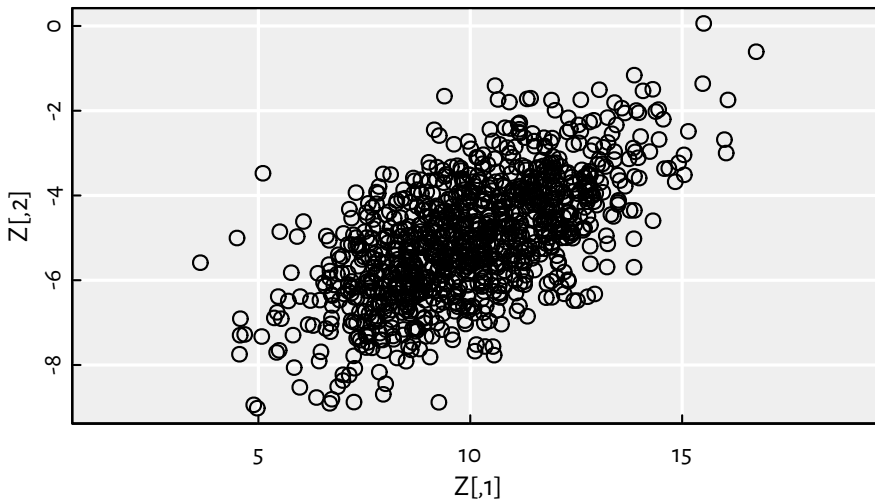

Figure 11.1: Example bivariate normal sample

It is known that eigenvectors of the covariance matrix correspond to the principal components of the original dataset and the eigenvalues give the variance explained by them:

```
eigen(cov(Z))
## eigen() decomposition
## $values
## [1] 5.18609 0.98386
##
## $vectors
##          [,1]     [,2]
```

(continues on next page)

[7] For drawing random samples from any multivariate distribution, refer to the theory of copulas, e.g., [38]. There are a few R packages on CRAN that implement the most popular models.

(continued from previous page)

```
## [1,] -0.86715  0.49804
## [2,] -0.49804 -0.86715
```

this roughly corresponds to the principal directions $[\sin(\pi/3), \cos(\pi/3)]$ *and the thereto-orthogonal* $[\cos(\pi/3), -\sin(\pi/3)]$ *with variances of 5 and 1, respectively. Still, this method of performing a PCA is not particularly numerically stable; see below for an alternative.*

11.4.5 QR Decomposition

We say that a real n-by-m matrix $\mathbf{Q}$, $n \geq m$, is orthogonal, whenever $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ (identity matrix) which is equivalent to its columns being orthogonal unit vectors (note that if $\mathbf{Q}$ is a square matrix, then $\mathbf{Q}^T = \mathbf{Q}^{-1}$ if and only if $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$).

Let $\mathbf{A}$ be a real[8] n-by-m matrix with $n \geq m$. Then $\mathbf{A} = \mathbf{QR}$ is its QR decomposition (in the so-called narrow form), if $\mathbf{Q}$ is an orthogonal n-by-m matrix and $\mathbf{R}$ is an upper triangular m-by-m one.

The `qr` function returns an object of S3 class `qr` from which we can extract the two components; see the `qr.Q` and `qr.R` functions.

Example 11.24 *Let* $\mathbf{X}$ *be an* n*-by-*m *data matrix, representing* n *points in* $\mathbb{R}^m$*, and a vector* $\mathbf{y} \in \mathbb{R}^n$ *of the* desired *outputs corresponding to each input. For fitting a linear model* $\mathbf{x}^T\boldsymbol{\theta}$*, where* $\boldsymbol{\theta}$ *is a vector of* m *parameters, we can use the method of least squares, which minimises*

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(\mathbf{x}_{i,\cdot}^T\boldsymbol{\theta} - y_i\right)^2 = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$$

It might be shown that if $\mathbf{X} = \mathbf{QR}$*, then* $\boldsymbol{\theta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{y}$*, which can conveniently be determined via a call to* `qr.coef`*.*

In particular, we can fit a simple linear regression model $y = ax + b$ *by considering* $\mathbf{X} = [x, 1]$ *and* $\boldsymbol{\theta} = [a, b]$*, for example (see Figure 11.2):*

```
x <- cars[["speed"]]
y <- cars[["dist"]]
X <- cbind(x, 1)  # the model is theta[1]*x + theta[2]*1
qrX <- qr(X)
(theta <- solve(qr.R(qrX)) %*% t(qr.Q(qrX)) %*% y)   # or: qr.coef(qrX, y)
##       [,1]
## x   3.9324
##   -17.5791
plot(x, y, xlab="speed", ylab="dist")  # scatter plot
abline(theta[2], theta[1], lty=2)  # add the regression line
```

[8] $\mathbf{A}$ can also be a complex matrix, which results in its QR decomposition's being such that $\mathbf{Q}$ is a unitary matrix.

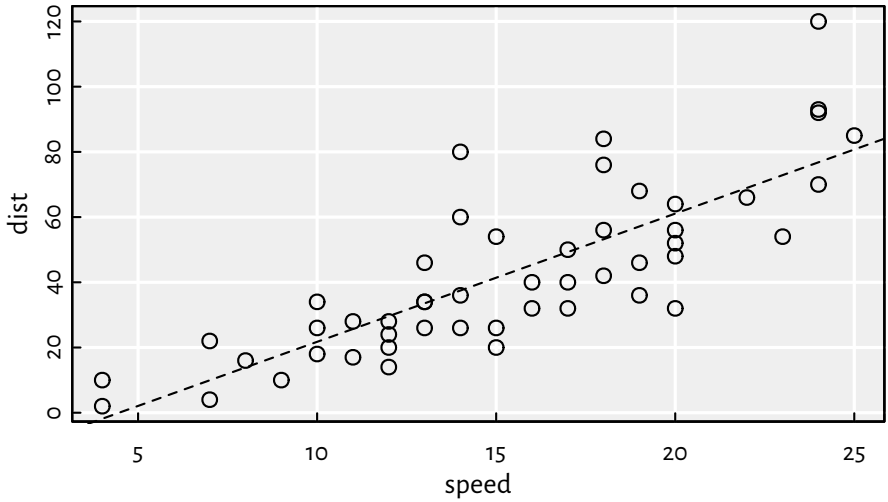

Figure 11.2: The built-in `cars` dataset and the fitted regression line

Note that ***`solve`*** *with one argument determines the inverse of a given matrix. The fitted model is* $y = 3.93241x - 17.5791$.

The same approach is used by ***`lm.fit`****, which is the workhorse behind the* ***`lm`*** *method accepting an R formula (which some readers might be familiar with; compare Section 16.5).*

```
lm.fit(cbind(x, 1), y)[["coefficients"]]  # also: lm(dist~speed, data=cars)
##         x
##    3.9324 -17.5791
```

11.4.6 SVD Decomposition

Given a real n-by-m matrix $\mathbf{X}$, its singular value decomposition (SVD) is given by $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$, where $\mathbf{D}$ is a p-by-p diagonal matrix (featuring the so-called singular values of $\mathbf{X}$, $d_{1,1} \geq \ldots \geq d_{p,p} \geq 0$, $p = \min\{n, m\}$) and $\mathbf{U}$, $\mathbf{V}$ are orthogonal matrices of size n-by-p and m-by-p, respectively.

`svd` may not only be used to determine the solution to linear regression[9] but also to perform the principal component analysis[10]. Namely, $\mathbf{V}$ gives the eigenvectors of $\mathbf{X}^T\mathbf{X}$. Assuming that $\mathbf{X}$ is centred at 0, the latter is precisely its scaled covariance matrix.

Example 11.25 *Continuing the PCA example above, we can determine the principal directions also by calling:*

[9] As the pseudoinverse $\mathbf{X}^+ = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T = \mathbf{V}\mathbf{D}^+\mathbf{U}^T = \mathbf{R}^{-1}\mathbf{Q}^T$, with $\mathbf{X}^+\mathbf{X} = \mathbf{I}$. Here $\mathbf{D}^+$ is a transposed version of $\mathbf{D}$ featuring the reciprocals of its non-zero elements.

[10] See the source code of `getS3method("prcomp", "default")`.

```
Zc <- apply(Z, 2, function(x) x-mean(x))  # centred version of Z
svd(Zc)[["v"]]
##          [,1]     [,2]
## [1,] -0.86715  0.49804
## [2,] -0.49804 -0.86715
```

11.5 S4 Classes (*)

The concept of the S3-style object oriented programming is based on a brilliantly simple idea (see Chapter 10): calling a generic **f**(x) automatically dispatches to a method **f.class_of_x**(x) or **f.default**(x) in the case where the former does not exist. Naturally, it has some inherent limitations:

- classes cannot be formally defined; the `class` attribute may be assigned arbitrarily onto any object[11],
- argument dispatch is performed only[12] with regard to one data type[13].

In most cases, and with appropriate level of mindfulness, this is not a problem at all. However, it is a typical condition of programmers who come to our world from more mainstream languages (e.g., C++; yours truly included) until they appreciate the true beauty of R's being somewhat different. Before they fully develop such an acquired taste, though, they grow restless as "R is not a real object oriented system because it lacks polymorphism, encapsulation, formal inheritance, and so on and so forth, and something must be done about it". The truth is that it had not have to, but with high probability it would have anyway in one way or another.

And so when the fourth version of the S language was introduced in 1998 (see [4]), it brought a new object oriented system which we are used to referring to as S4. Its R version has been implemented in the **methods** package. Below we discuss it briefly; for more details, see **help**("Classes_Details") and **help**("Methods_Details") as well as [5] and [6].

Note (*) S4 was loosely inspired by the Common Lisp Object System (with its `defclass`, `defmethod`, etc.; see, e.g., [15]). In the current author's opinion, the S4 system is somewhat an afterthought. Due to appendages like this, R seems like a patchwork

[11] A partial solution to this could involve defining a method like **validate.class_name**, that is called frequently and which checks whether a given object enjoys some desired constraints.

[12] Although there are functions featuring some workarounds (see, e.g., **cbind** which dispatches to **cbind.data.frame** if one argument is a data frame and the remaining ones are vectors or matrices). Also, binary operators overloaded via group generics consider the classes of both operands; see Section 16.4.6.

[13] Hypothetically, we can imagine an OOP system relying on methods named like **method.class_name1.class_name2** where dispatching is based on two argument types.

language; suffice it to say that it was not the last attempt to do a somewhat more real OOP in the overall functional R: the story will continue in Section 16.1.5.

The main problem with all the OOP approaches is that each of them is parallel to S3 which never lost its popularity and is still at the very core of our language. We are thus covering them for the sake of completeness, because that's what must be done. After all, with non-zero probability, the reader will sooner or later come across such objects (e.g., below we explain the meaning of notation like `x@slot`). Also, yours truly rebelliously suggests taking statements such as "for new projects, it is recommended to use the more flexible and robust S4 scheme provided in the **`methods`** package" (see **`help`**`("UseMethod")`) with a pinch of salt.

11.5.1 Defining S4 Classes

An S4 class can formally be registered by means of a call to **`setClass`**.

For instance:

```
library("methods")  # in the case where it is not auto-loaded
setClass("categorical", slots=c(data="integer", levels="character"))
```

defines a class named `categorical` with two slots `data` and `levels` being integer and character vectors, respectively. Note that this notation is already quite peculiar: there is no assignment which would suggest that we have introduced something novel.

An object of the above class can be instantiated by calling **`new`**:

```
z <- new("categorical", data=c(1L, 2L, 2L, 1L, 1L), levels=c("a", "b"))
print(z)
## An object of class "categorical"
## Slot "data":
## [1] 1 2 2 1 1
##
## Slot "levels":
## [1] "a" "b"
```

That z is of the recently-introduced class can be verified as follows:

```
is(z, "categorical")
## [1] TRUE
class(z)  # also: attr(z, "class")
## [1] "categorical"
## attr(,"package")
## [1] ".GlobalEnv"
```

Important Some R packages will be importing from the **`methods`** only for the sake of

being able to access the convenient **is** function – it does not mean they are defining new S4 classes.

Note S4 objects are marked as being of the following basic type:

```
typeof(z)
## [1] "S4"
```

For technical details on how they are internally represented, see Section 1.12 in [50]. In particular, in our case, all the slots are simply stored as object attributes:

```
attributes(z)
## $data
## [1] 1 2 2 1 1
##
## $levels
## [1] "a" "b"
##
## $class
## [1] "categorical"
## attr(,"package")
## [1] ".GlobalEnv"
```

11.5.2 Accessing Slots

Reading or writing slot contents can be done by means of the `@` operator and the **slot** function or their replacement versions.

```
z@data  # or slot(z, "data")
## [1] 1 2 2 1 1
z@levels <- c("A", "B")
```

Note The `@` operator can only be used on S4 objects and some sanity checks are automatically performed:

```
z@unknown <- "spam"
## Error in (function (cl, name, valueClass) : 'unknown' is not a slot in class "categori
z@data <- "spam"
## Error in (function (cl, name, valueClass) : assignment of an object of class "characte
```

11.5.3 Defining Methods

For the S4 counterparts of the S3 generics (Section 10.2), see **`help("setGeneric")`**. Luckily, there is a good degree of interoperability between the S3 and S4 systems.

Let us start by introducing a new method for the well-known **`as.character`** generic. Instead of defining **`as.character.categorical`**, we need to register a new routine with **`setMethod`**.

```
setMethod(
    "as.character",     # name of the generic
    "categorical",      # class of 1st arg; or: signature=c(x="categorical")
    function(x, ...)    # method definition
        x@levels[x@data]
)
```

Testing:

```
as.character(z)
## [1] "A" "B" "B" "A" "A"
```

The S4 counterpart of **`print`** is **`show`**:

```
setMethod(
    "show",
    "categorical",
    function(object) {
        x_character <- as.character(object)
        print(x_character)  # calls `print.default`
        cat(sprintf("Categories: %s\n",
            paste(object@levels, collapse=", ")))
    }
)
```

Interestingly, it is involved automatically upon a call to **`print`**:

```
print(z)  # calls `show` for `categorical`
## [1] "A" "B" "B" "A" "A"
## Categories: A, B
```

Methods that dispatch on the type of multiple arguments are possible too, for example:

```
setMethod(
    "split",
    c(x="ANY", f="categorical"),
    function (x, f, drop=FALSE, ...)
        split(x, as.character(f), drop=drop, ...)
)
```

allows the first argument to be of any type (like a default method), and:

```
setMethod(
    "split",
    c(x="matrix", f="categorical"),
    function (x, f, drop=FALSE, ...)
        lapply(
            split(seq_len(NROW(x)), f, drop=drop, ...),  # calls the above
            function(i) x[i, , drop=FALSE])
)
```

is a version tailored for matrices. Testing:

```
A <- matrix(1:35, nrow=5)  # whatever
split(A, z)    # matrix,categorical
## $A
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    6   11   16   21   26   31
## [2,]    4    9   14   19   24   29   34
## [3,]    5   10   15   20   25   30   35
##
## $B
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    2    7   12   17   22   27   32
## [2,]    3    8   13   18   23   28   33
split(1:5, z)  # ANY,categorical
## $A
## [1] 1 4 5
##
## $B
## [1] 2 3
```

Exercise 11.26 *Overload the `[` operator for the `categorical` class*

11.5.4 Defining Constructors

We can also overload the **`initialize`** method which is automatically called by **`new`**:

```
setMethod(
    "initialize",   # class name
    "categorical",  # method name
    function(.Object, x) {  # the method itself
        x <- as.character(x)  # see above
        xu <- unique(sort(x))  # drops NAs

        .Object@data <- match(x, xu)
```

(continues on next page)

(continued from previous page)

```
        .Object@levels <- xu

        .Object  # return value - a modified object
    }
)
```

This allows for constructing new objects of class `categorical` based on an object like `x` above, for instance:

```
w <- new("categorical", c("a", "c", "a", "a", "d", "c"))
print(w)
## [1] "a" "c" "a" "a" "d" "c"
## Categories: a, c, d
```

Note that we have not set the two slots directly. They were automatically taken care of by **`initialize`**.

Exercise 11.27 *Set up a validating method for our class; see* ***`help`****`("setValidity")`*.

11.5.5 Inheritance

New S4 classes can be derived from existing ones, for instance:

```
setClass("binary", contains="categorical")
```

is a child class inhering all slots from its parent. We can, for example, overload the initialisation method for it:

```
setMethod(
    "initialize",
    "binary",
    function(.Object, x)
    {
        x <- as.character(as.integer(as.logical(x)))
        xu <- c("0", "1")
        .Object@data <- match(x, xu)
        .Object@levels <- xu
        .Object
    }
)
```

Testing:

```
new("binary", c(TRUE, FALSE, TRUE, FALSE, NA, TRUE))
## [1] "1" "0" "1" "0" NA  "1"
## Categories: 0, 1
```

Note that we are still using the **show** method of the parent class.

11.5.6 A Note on the Matrix Package

The **Matrix** package is perhaps the most widely known showcase of the S4 object-orientation (and that is the reason why we cover S4 in this very chapter). It defines classes and methods for dense and sparse matrices, including rectangular, symmetric, triangular, band, and diagonal ones.

For instance, large graph (e.g., in network sciences) or preference (e.g., in recommender systems) data can be represented using sparse matrices (those which feature many 0s; after all, it is extremely more common for two vertices in a network to *not* be joined by an edge than to be connected).

For example:

```
library("Matrix")
(A <- Diagonal(x=1:5))
## 5 x 5 diagonal matrix of class "ddiMatrix"
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    .    .    .    .
## [2,]    .    2    .    .    .
## [3,]    .    .    3    .    .
## [4,]    .    .    .    4    .
## [5,]    .    .    .    .    5
```

created a real diagonal matrix. Moreover:

```
B <- as(A, "sparseMatrix")
B[1, 2] <- 7
B[4, 1] <- 42
print(B)
## 5 x 5 sparse Matrix of class "dgCMatrix"
##
## [1,]  1 7 . . .
## [2,]  . 2 . . .
## [3,]  . . 3 . .
## [4,] 42 . . 4 .
## [5,]  . . . . 5
```

yields a general sparse real matrix in the CRC (compressed, sparse, column-oriented) format.

For more information on the package, see **vignette**(package="Matrix").

11.6 Exercises

Exercise 11.28 *Let `X` be a matrix with `dimnames` set, e.g.:*

```
X <- matrix(1:12, byrow=TRUE, nrow=3)      # example matrix
dimnames(X)[[2]] <- c("a", "b", "c", "d")  # set column names
print(X)
##      a  b  c  d
## [1,] 1  2  3  4
## [2,] 5  6  7  8
## [3,] 9 10 11 12
```

Explain (in your own words) the meaning of the following expressions involving matrix subsetting. Note that not each of them is valid.

- `X[1, ]`,
- `X[, 3]`,
- `X[, 3, drop=FALSE]`,
- `X[3]`,
- `X[, "a"]`,
- `X[, c("a", "b", "c")]`,
- `X[, -2]`,
- `X[X[,1] > 5, ]`,
- `X[X[,1]>5, c("a", "b", "c")]`,
- `X[X[,1]>=5 & X[,1]<=10, ]`,
- `X[X[,1]>=5 & X[,1]<=10, c("a", "b", "c")]`,
- `X[, c(1, "b", "d")]`.

Exercise 11.29 *Assuming that `X` is an array, what are the differences between the following indexing schemes?*

- `X["1", ]` vs `X[1, ]`,
- `X[, "a", "b", "c"]` vs `X["a", "b", "c"]` vs `X[, c("a", "b", "c")]` vs `X[c("a", "b", "c")]`,
- `X[1]` vs `X[, 1]` vs `X[1, ]`,
- `X[X>0]` vs `X[X>0, ]` vs `X[, X>0]`,
- `X[X[, 1]>0]` vs `X[X[, 1]>0,]` vs `X[,X[,1]>0]`,
- `X[X[, 1]>5, X[1, ]<10]` vs `X[X[1, ]>5, X[, 1]<10]`.

Exercise 11.30 *For a given real* n*-by-*m *matrix* $\mathbf{X}$*, determine the bounding hyperrectangle of thusly encoded* n *input points in an* m*-dimensional space. Return a* 2*-by-*m *matrix* $\mathbf{B}$ *with* $b_{1,j} = \min_i x_{i,j}$ *and* $b_{2,j} = \max_i x_{i,j}$.

Exercise 11.31 *Let* $\mathbf{t}$ *be vector of* n *integers in* $\{1, \ldots, k\}$*. Write a function to one-hot-encode each* t_i*: return a* 0*-*1 *matrix* $\mathbf{R}$ *of size* n*-by-*k *such that* $r_{i,j} = 1$ *if and only if* $j = t_i$*. For example, if* $\mathbf{t} = [1, 2, 3, 2, 4]$ *and* $k = 4$*, then:*

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

On a side note, such a representation is useful when solving, e.g., a multiclass classification problem by means of k *binary classifiers.*

Then, write another function, but this time setting $r_{i,j} = 1$ *if and only if* $j \geq t_i$*, e.g.:*

$$R = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Important Kind reminder: as usual, try to solve all the exercises without the use of explicit **`for`** and **`while`** loops (provided that it is possible).

Exercise 11.32 *Given an* n*-by-*k *real matrix, apply the softmax function on each row, i.e., map* $x_{i,j}$ *to* $\frac{\exp(x_{i,j})}{\sum_{l=1}^{k} \exp(x_{i,l})}$*. Then, one-hot decode the values in each row, i.e., find the column number with the greatest value. Return a vector of size* n *with elements in* $\{1, \ldots, k\}$.

Exercise 11.33 *Assume that an* n*-by-*d *real matrix* $\mathbf{X}$ *represents* n *points in* $\mathbb{R}^d$*. Write a function (but do not refer to* **`dist`***) that determines the pairwise distances between all the* n *points and a given* $\mathbf{y} \in \mathbb{R}^d$*. Return a vector* $\mathbf{d}$ *of length* n *with* $d_i = \|\mathbf{x}_{i,\cdot} - \mathbf{y}\|_2$.

Exercise 11.34 *Let* $\mathbf{X}$ *and* $\mathbf{Y}$ *be two real-valued matrices of sizes* n*-by-*d *and* m*-by-*d*, respectively, representing two sets of points in* $\mathbb{R}^d$*. Return an integer vector* $\mathbf{r}$ *of length* m *such that* r_i *indicates the index of the point in* $\mathbf{X}$ *with the least distance to (the closest to) the* i*-th point in* $\mathbf{Y}$*, i.e.,* $r_i = \arg\min_j \|\mathbf{x}_{j,\cdot} - \mathbf{y}_{i,\cdot}\|_2$.

Exercise 11.35 *Write your own version of the built-in* **`utils::combn`**.

Exercise 11.36 *Time series are vectors or matrices of class* `ts` *equipped with the* `tsp` *attribute, amongst others. Refer to* **`help("ts")`** *for more information about how they are represented and what S3 methods have been overloaded for them.*

Exercise 11.37 *(*) Numeric matrices can be stored in a CSV file, amongst others. Usually, we will be loading them via* **`read.csv`***, which returns a data frame (see Chapter 12), for example:*

```
X <- as.matrix(read.csv(
    paste0(
        "https://github.com/gagolews/teaching-data/",
        "raw/master/marek/eurxxx-20200101-20200630.csv"
    ),
    comment.char="#",
    sep=","
))
```

Write your own function **`read_numeric_matrix(file_name, comment, sep)`** *which is instead based on a few calls to* **`scan`**. *Use* **`file`** *to establish a file connection to be able to ignore the comment lines and fetch the column names before reading the actual numeric values.*

Exercise 11.38 *(*) Using* **`readBin`**, *read the* `t10k-images-idx3-ubyte.gz` *from the MNIST database homepage*[14]. *The output object should be a three-dimensional, 10000-by-28-by-28 array with real elements between 0 and 255. Refer to the File Formats section therein for more details.*

Exercise 11.39 *(**) Circular convolution of discrete-valued multidimensional signals can be performed by means of* **`fft`** *and matrix multiplication, whereas affine transformations require only the latter. Apply various image transformations such as sharpening, shearing, and rotating on the MNIST digits and plot the results using the* **`image`** *function.*

Exercise 11.40 *(*) Using* **`constrOptim`**, *find the minimum of the Constrained Betts Function* $f(x_1, x_2) = 0.01x_1^2 + x_2^2 - 100$ *with linear constraints* $2 \leq x_1 \leq 50$, $-50 \leq x_2 \leq 50$, *and* $10x_1 \geq 10 + x_2$. *(**) Also, use* **`solve.QP`** *from the* **`quadprog`** *package of find the minimum.*

---

[14] https://web.archive.org/web/20211107114045/http://yann.lecun.com/exdb/mnist/

12

Data Frames

Most matrices are built on top of atomic vectors and hence allow items of the same type to be arranged into rows and columns. Data frames (objects of S3 class `data.frame`, first introduced in [8]), on the other hand, are collections of vectors of identical lengths or matrices with identical row counts, hence allowing to represent structured[1] data of possibly heterogeneous types, for instance:

```
class(iris)  # `iris` is an example built-in data frame
## [1] "data.frame"
iris[c(1, 51, 101), ]  # 3 chosen rows from `iris`
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 51           7.0         3.2          4.7         1.4 versicolor
## 101          6.3         3.3          6.0         2.5  virginica
```

is a mix of numeric and factor-type data.

The good news is that not only data frames are built upon named lists (e.g., to extract a column we can refer to `[[`), but also many functions recognise them to be matrix-like, (e.g., to select specific rows and columns, two indexes can be passed to `[` like in the example above). Hence, it will soon turn out that we already know a lot about how to perform basic data wrangling activities, even if we do not full realise it now.

Important Some of us will approach this chapter biased by what we know from elsewhere, including our experience with some popular third-party packages for data frame processing. The art is to filter out that information as noise (at least, for the time being). We will show how powerful base R vocabulary is and how much can be implied from the material covered in the preceding chapters. And yes, this book is like a good thriller/drama/love story: it is meant to be read from the beginning to end, so please go back to the start of this comprehensive course if you happened to pop in here by accident or driven by "but I need to know *now*". Good morning.

---

[1] We are already highly skilled in handling unstructured data and turning it to something that is much more regular: the numerous functions for processing numeric and character vectors as well as lists that we have covered in the first part of this book allow us to extract meaningful data from text, handle missing values, engineer features, and so forth.

12.1 Creating Data Frames

12.1.1 `data.frame` and `as.data.frame`

Most frequently, we create data frames based on a series of logical, numeric, or characters vectors of identical lengths. The **`data.frame`** function is particularly useful in such a scenario:

```
(x <- data.frame(
    a=c(TRUE, FALSE),
    b=1:6,
    c=runif(6),
    d=c("spam", "spam", "eggs")
))
##       a b       c    d
## 1  TRUE 1 0.77437 spam
## 2 FALSE 2 0.19722 spam
## 3  TRUE 3 0.97801 eggs
## 4 FALSE 4 0.20133 spam
## 5  TRUE 5 0.36124 spam
## 6 FALSE 6 0.74261 eggs
```

Note that shorter vectors were recycled. That the diverse column types were retained and no coercion has been made, can be verified, e.g., by calling:

```
str(x)
## 'data.frame':        6 obs. of  4 variables:
##  $ a: logi  TRUE FALSE TRUE FALSE TRUE FALSE
##  $ b: int  1 2 3 4 5 6
##  $ c: num  0.774 0.197 0.978 0.201 0.361 ...
##  $ d: chr  "spam" "spam" "eggs" "spam" ...
```

We can also fetch the class of each column directly by calling (compare Section 12.3):

```
sapply(x, class)  # the same as unlist(Map(class, x))
##           a           b           c           d
##   "logical"   "integer"   "numeric" "character"
```

Important For many reasons (see, e.g., Section 12.1.5 and Section 12.1.6), we recommend to have the type of each column always checked, for instance by calling the **`str`** function.

Many objects, such as matrices, can easily be coerced to data frames using particular **`as.data.frame`** methods.

Here is an example matrix:

```
(A <- matrix(1:6, nrow=3,
    dimnames=list(
        NULL,         # no row labels
        c("u", "v")  # some column labels
)))
##      u v
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

Let us convert it to a data frame:

```
as.data.frame(A)  # as.data.frame.matrix
##   u v
## 1 1 4
## 2 2 5
## 3 3 6
```

Note that a matrix with no row labels is printed slightly differently than a data frame with (as it will soon turn out) the default `row.names`.

Named lists are amongst other candidates for a meaningful conversion. Consider an example list, where each element is a vector of the same length as the other ones:

```
(l <- Map(
    function(x) {
        c(Min=min(x), Median=median(x), Mean=mean(x), Max=max(x))
    },
    split(iris[["Sepal.Length"]], iris[["Species"]])
))
## $setosa
##    Min Median   Mean    Max
##  4.300  5.000  5.006  5.800
##
## $versicolor
##    Min Median   Mean    Max
##  4.900  5.900  5.936  7.000
##
## $virginica
##    Min Median   Mean    Max
##  4.900  6.500  6.588  7.900
```

Each list element will be turned to a separate column:

```
as.data.frame(l)  # as.data.frame.list
##        setosa versicolor virginica
## Min     4.300      4.900     4.900
## Median  5.000      5.900     6.500
## Mean    5.006      5.936     6.588
## Max     5.800      7.000     7.900
```

Sadly, **`as.data.frame.list`** is not particularly fond of lists of vectors of incompatible lengths:

```
as.data.frame(list(a=1, b=11:12, c=21:23))
## Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE, : a
```

The above vectors could have been recycled with a warning, but they were not.

```
as.data.frame(list(a=1:4, b=11:12, c=21))  # recycling rule okay
##   a  b  c
## 1 1 11 21
## 2 2 12 21
## 3 3 11 21
## 4 4 12 21
```

The method for the S3 class `table` (mentioned in Chapter 11) can be helpful as well. Here is an example contingency table together with its *unstacked* version.

```
(t <- table(mtcars[["vs"]], mtcars[["cyl"]]))
##
##      4  6  8
##   0  1  3 14
##   1 10  4  0
as.data.frame(t)  # as.data.frame.table; see the stringsAsFactors note below!
##   Var1 Var2 Freq
## 1    0    4    1
## 2    1    4   10
## 3    0    6    3
## 4    1    6    4
## 5    0    8   14
## 6    1    8    0
```

Actually, **`as.data.frame.table`** is so useful that we might want to call it directly on any array. This way, we can convert it from the so-called *wide* format to the *long* one; see Section 12.3.6 for more details.

Note The above method is based on **`expand.grid`**, which determines all combinations of a given series of vectors.

```
expand.grid(1:2, c("a", "b", "c"))  # see the stringsAsFactors note below!
##   Var1 Var2
## 1    1    a
## 2    2    a
## 3    1    b
## 4    2    b
## 5    1    c
## 6    2    c
```

Overall, many classes of objects can be included[2] in a data frame; the popular choices include `Date`, `POSIXct`, and `factor`. It is worth noting that the **`data.frame`** function calls the corresponding **`as.data.frame`** method, and **`format`** is used on printing.

Example 12.1 *Here are two custom methods for what we would like to call from now on an S3 class `spam`:*

```
as.data.frame.spam <- function(x, ...)
    structure(
        list(x),
        class="data.frame",
        row.names=seq_along(x)
    )
format.spam <- function(x, ...)
    paste0("*", x, "*")
```

Testing data frame creation and printing:

```
data.frame(
    a=structure(c("a", "b", "c"), class="spam"),
    b=factor(c("spam", "bacon", "spam")),
    c=Sys.Date()+1:3
)
##     a     b          c
## 1 *a*  spam 2023-01-15
## 2 *b* bacon 2023-01-16
## 3 *c*  spam 2023-01-17
```

12.1.2 `cbind.data.frame` and `rbind.data.frame`

There are data frame-specific versions of **`cbind`** or **`rbind`** (which we discussed in the context of stacking matrices in Section 11.1.2). They are used quite eagerly:

[2] Also, the attributes of objects stored as matrix columns will generally be preserved (even if they are not displayed by **`print`**; see **`str`** though).

help("cbind") states that they will be referred to if at least[3] one of its arguments is a data frame and the other arguments are atomic vectors or lists (possibly with the dim attribute).

For example:

```
x <- iris[c(1, 51, 101), c("Sepal.Length", "Species")]  # whatever
cbind(Yummy=c(TRUE, FALSE, TRUE), x)
##     Yummy Sepal.Length    Species
## 1    TRUE          5.1     setosa
## 51  FALSE          7.0 versicolor
## 101  TRUE          6.3  virginica
```

added a new column to a data frame x. Moreover:

```
rbind(x, list(42, "virginica"))
##     Sepal.Length    Species
## 1            5.1     setosa
## 51           7.0 versicolor
## 101          6.3  virginica
## 11          42.0  virginica
```

added a new row. Note that columns are of different types. Hence, the values to row-bind were provided as a generic vector. The list can also be named. It can consist of vectors of length greater than one, given in any order:

```
rbind(x, list(
    Species=c("virginica", "setosa"),
    Sepal.Length=c(42, 7)
))
##     Sepal.Length    Species
## 1            5.1     setosa
## 51           7.0 versicolor
## 101          6.3  virginica
## 11          42.0  virginica
## 2            7.0     setosa
```

Sometimes referring to these methods directly will be necessary. Consider an example list of atomic vectors:

```
x <- list(a=1:3, b=11:13, c=21:23)
```

First, we call the generic which dispatches to the default method:

[3] This is a clear violation of the rule that an S3 generic dispatches on the type of only one (usually: first) argument; an exception made for the sake of the questionable user *convenience*. Also, note that there is no **cbind.default** method available: it is hardcoded at the C language level.

```
do.call(cbind, x)
##      a  b  c
## [1,] 1 11 21
## [2,] 2 12 22
## [3,] 3 13 23
```

If we want to make sure we garner a data frame in result, we need to write:

```
do.call(cbind.data.frame, x)
##   a  b  c
## 1 1 11 21
## 2 2 12 22
## 3 3 13 23
```

This is particularly useful in the context of fetching outputs from **`Map`** and its friends, which are wrapped inside a list. For instance:

```
l <- unname(Map(
    function(x) list(
        Sepal.Length=mean(x[["Sepal.Length"]]),
        Sepal.Width=mean(x[["Sepal.Width"]]),
        Species=x[["Species"]][1]
    ),
    split(iris, iris[["Species"]])  # split.data.frame; see below
))
str(l)
## List of 3
##  $ :List of 3
##   ..$ Sepal.Length: num 5.01
##   ..$ Sepal.Width : num 3.43
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1
##  $ :List of 3
##   ..$ Sepal.Length: num 5.94
##   ..$ Sepal.Width : num 2.77
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2
##  $ :List of 3
##   ..$ Sepal.Length: num 6.59
##   ..$ Sepal.Width : num 2.97
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3
```

This was nothing more than a fancy way to obtain an illustrative list, which we may now turn into a data frame by calling:

```
do.call(rbind.data.frame, l)
##   Sepal.Length Sepal.Width    Species
## 1        5.006       3.428     setosa
```

(continues on next page)

(continued from previous page)

```
## 2        5.936       2.770 versicolor
## 3        6.588       2.974  virginica
```

On the other hand, **`do.call(rbind, l)`** does not return a particularly friendly object type:

```
do.call(rbind, l)
##      Sepal.Length Sepal.Width Species
## [1,] 5.006        3.428       setosa
## [2,] 5.936        2.77        versicolor
## [3,] 6.588        2.974       virginica
```

Despite the pretty face, it is a matrix... over a list:

```
str(do.call(rbind, l))
## List of 9
##  $ : num 5.01
##  $ : num 5.94
##  $ : num 6.59
##  $ : num 3.43
##  $ : num 2.77
##  $ : num 2.97
##  $ : Factor w/ 3 levels "setosa","versicolor",..: 1
##  $ : Factor w/ 3 levels "setosa","versicolor",..: 2
##  $ : Factor w/ 3 levels "setosa","versicolor",..: 3
##  - attr(*, "dim")= int [1:2] 3 3
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:3] "Sepal.Length" "Sepal.Width" "Species"
```

12.1.3 Reading Data Frames

Structured data can be imported from external sources, such as CSV/TSV (comma/tab-separated values) or HDF5 files, relational databases supporting SQL (see Section 12.1.4) web APIs (e.g., through the **`curl`** and **`jsonlite`** packages), spreadsheets [48], and so on.

In particular, **`read.csv`** and the like fetch data from plain text files consisting of records where fields are separated by commas, semicolons, tabs, etc.

For instance:

```
x <- data.frame(a=runif(3), b=c(TRUE, FALSE, TRUE))  # example data frame
f <- tempfile()  # temporary file name
write.csv(x, f, row.names=FALSE)  # export
```

This created a CSV file which looks like:

```
cat(readLines(f), sep="\n")  # print file contents
## "a","b"
## 0.287577520124614,TRUE
## 0.788305135443807,FALSE
## 0.4089769218117,TRUE
```

The above can be read by calling:

```
read.csv(f)
##         a     b
## 1 0.28758  TRUE
## 2 0.78831 FALSE
## 3 0.40898  TRUE
```

Exercise 12.2 *Check out* `help("read.table")` *for a long list of tunable parameters, especially:* `sep`*,* `dec`*,* `quote`*,* `header`*,* `comment.char`*, and* `row.names`*. Further, note that reading from compressed files is supported directly.*

Important CSV is by far the most portable and user-friendly format for exchanging matrix-like objects between different programs and computing languages (e.g., Python, Julia, LibreOffice Calc, etc.). Such files can be opened in any text editor.

Note As mentioned in Section 8.3.5, it is possible to process data frames on a chunk-by-chunk basis, which is beneficial especially when data do not fit into memory (compare the `nrows` argument to **`read.csv`**).

12.1.4 Interfacing Relational Databases and Querying with SQL (*)

The **`DBI`** package provides a universal interface for particular database management systems whose drivers are implemented in additional add-ons such as **`RSQLite`**, **`RMariaDB`**, **`RPostgreSQL`**, etc., or, more generally, **`RODBC`** or **`odbc`**. For more details, see Section 4 of [48].

Example 12.3 *Let us play with an in-memory (volatile) instance of an SQLite database.*

```
library("DBI")
con <- dbConnect(RSQLite::SQLite(), ":memory:")
```

This returns an object representing a database connection which we can refer to in further communication.

An easy way to create a database table is to call:

```
dbWriteTable(con, "mtcars", mtcars)  # `mtcars` is a toy built-in data frame
```

*Alternatively, **`dbExecute`** could have been referred to in order to send SQL statements such as `CREATE TABLE ...` followed by a series of `INSERT INTO ...`.*

Some data retrieval can now follow:

```
dbGetQuery(con, "
    SELECT cyl, vs, AVG(mpg) AS mpg_ave, AVG(hp) AS hp_ave
    FROM mtcars
    GROUP BY cyl, vs
")
##   cyl vs mpg_ave hp_ave
## 1   4  0  26.000  91.00
## 2   4  1  26.730  81.80
## 3   6  0  20.567 131.67
## 4   6  1  19.125 115.25
## 5   8  0  15.100 209.21
```

This gives us an ordinary R data frame which we can process in the same fashion as any other object of this kind.

At the end, the database connection must be closed.

```
dbDisconnect(con)
```

Exercise 12.4 *Database passwords should never be stored in plain text files, let alone in R scripts in version-controlled repositories. Consider a few ways for fetching credentials programmatically:*

- *using environment variables (see **`help("Sys.getenv")`**),*
- *using the **`keyring`** package,*
- *calling **`system2`** (Section 7.3.3) to retrieve it from the system keyring (e.g., the **`keyring`** package for Python provides a platform-independent command-line utility).*

12.1.5 Strings as Factors?

The following is so critical that we will devote a separate subsection to discuss it, so that we always remain vigilant (such is life: maintaining some level of mindfulness is often a good idea).

Important Some functions related to data frames automatically convert character vectors to factors. This behaviour is frequently controlled by the `stringsAsFactors` argument thereto.

This is particularly problematic due to the fact that, when printed, factor and character columns look identical:

```
(x <- data.frame(a=factor(c("U", "V")), b=c("U", "V")))
##   a b
## 1 U U
## 2 V V
```

We recall from Section 10.3.3 that factors can be nasty. For example, passing factors as indexers in `[` or converting them with **as.numeric** might give counterintuitive (for the uninformed) results. Also, new factor levels must be added manually when we want to extend them with more diverse data. This can cause some unexpected behaviour in contexts such as:

```
rbind(x, c("W", "W"))
## Warning in `[<-.factor`(`*tmp*`, ri, value = "W"): invalid factor level,
## NA generated
##      a b
## 1    U U
## 2    V V
## 3 <NA> W
```

It is therefore a good habit to have the data types always checked, for instance:

```
str(x)
## 'data.frame':        2 obs. of  2 variables:
##  $ a: Factor w/ 2 levels "U","V": 1 2
##  $ b: chr  "U" "V"
```

Before R 4.0, a number of functions, including **data.frame** and **read.csv** had the `stringsAsFactors` argument defaulting to `TRUE`. This is no longer the case for many of them.

However, exceptions to this rule still exist, e.g., including **as.data.frame.table** and **expand.grid**. Besides, some built-in example data frames have factor-typed columns inherited from the old days, e.g.:

```
class(iris[["Species"]])
## [1] "factor"
```

We observe that the `Species` column in `iris` is not of type `character`. Thence, adding a new variety might be oblique:

```
iris2 <- iris[c(1, 51, 101), ]  # example subset
levels(iris2[["Species"]]) <- c(levels(iris2[["Species"]]), "croatica")
rbind(iris2, c(6, 3, 3, 2, "croatica"))
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
```

(continues on next page)

(continued from previous page)

```
## 1           5.1         3.5          1.4         0.2     setosa
## 51            7         3.2          4.7         1.4 versicolor
## 101         6.3         3.3            6         2.5  virginica
## 4             6           3            3           2   croatica
```

Alternatively, we could have simply converted the `Species` column to `character`.

12.1.6 Internal Representation

Objects of S3 class `data.frame` are built upon lists of vectors of the same length or matrices with identical row counts, which define consecutive columns thereof. Apart from `class`, they must be equipped with the following special attributes:

- `names` – a character vector (as usual in any named list) labelling the columns or their groups,
- `row.names` – a character or integer vector with no duplicates nor missing values, doing what advertised.

Therefore, a data frame can be created from scratch by calling, for example:

```
structure(
    list(a=11:13, b=21:23),  # sets the `names` attribute already
    row.names=1:3,
    class="data.frame"
)
##    a  b
## 1 11 21
## 2 12 22
## 3 13 23
```

Here is a data frame based on a length-5 list, a matrix with five rows, and a length-5 numeric vector, with some fancy row names on top:

```
structure(
    list(
        a=list(1, 1:2, 1:3, numeric(0), -(4:1)),
        b=cbind(u=11:15, v=21:25),
        c=runif(5)
    ),
    row.names=c("spam", "bacon", "eggs", "ham", "aubergine"),
    class="data.frame"
)
##                       a b.u b.v       c
## spam                  1  11  21 0.28758
## bacon              1, 2  12  22 0.78831
```

(continues on next page)

(continued from previous page)

```
## eggs             1, 2, 3  13  23 0.40898
## ham                       14  24 0.88302
## aubergine -4, -3, -2, -1  15  25 0.94047
```

In general, the columns of type `list` can contain anything, e.g., other lists or R functions. Including atomic vectors of varying lengths just like above allows for creating something à la *ragged arrays* – a pretty handy scenario.

The issue with matrix entries, on the other hand, is that they appear as if they were many, but – as it will turn out in the sequel – they are often treated as a single complex column, e.g., by the index operator (see Section 12.2). Therefore, from this perspective, the above data frame has three columns, not four. Such objects can be output by **`aggregate`** (see Section 12.3), amongst others. Nevertheless, they can be very useful too, forming natural *column groups* which can be easily accessed and batch-processed in the same way.

Important Unfortunately, data frames with list or matrix columns cannot be normally created with the **`data.frame`** nor **`cbind`** functions which might explain why they are less popular. This behaviour is dictated by the particular underlying **`as.data.frame`** methods which are called by both of them. As a curiosity, see **`help`**`("I")` though.

Exercise 12.5 *Verify that for a data frame featuring a matrix column, the latter does not require column names (the second `dimnames`) set.*

The `names` and `row.names` attributes are special in the sense of Section 4.4.3. In particular, they can be accessed or modified by the corresponding functions.

It is worth noting that **`row.names`**`(df)` always returns a character vector, even when **`attr`**`(df, "row.names")` is an integer vector. Further, setting **`row.names`**`(df) <- NULL` will re-set[4] this attribute to the most commonly desired case of consecutive natural numbers, for example:

```
(x <- iris[c(1, 51, 101), ])  # comes with some sad row names
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 51           7.0         3.2          4.7         1.4 versicolor
## 101          6.3         3.3          6.0         2.5  virginica
row.names(x) <- NULL  # reset to seq_len(NROW(x))
print(x)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          5.1         3.5          1.4         0.2     setosa
```

(continues on next page)

[4] `` **`attr<-`**`(df, "row.names")`` does not feature the same sanity checks as `` **`row.names<-`**`(df)`` does. For instance, it is easy to corrupt a data frame by setting a too-short `row.names` attribute.

(continued from previous page)

```
## 2          7.0         3.2          4.7         1.4 versicolor
## 3          6.3         3.3          6.0         2.5  virginica
```

Exercise 12.6 *What is the name of the replacement version of the* `row.names` *method for the* `data.frame` *class?*

Exercise 12.7 *Implement your own version of* ***`expand.grid`****.*

Exercise 12.8 *Implement your own version of* ***`xtabs`****, but which does not rely on a formula interface. Allow three parameters: a data frame, the name of the "counts" column and the names of the cross-classifying variables. Hence,* `my_xtabs(x, "Freq", c("Var1", "Var2"))` *should be equivalent to* `xtabs(Freq~Var1+Var2, x)`*.*

12.2 Data Frame Subsetting

12.2.1 Data Frames are Lists

Data frames are named lists, where each element represents an individual column. Therefore[5], **`length`** yields the number of columns and **`names`** gives their respective labels.

Let us play with the following data frame:

```
(x <- data.frame(
    a=runif(6),
    b=rnorm(6),
    c=LETTERS[1:6],
    d1=c(FALSE, TRUE, FALSE, NA, FALSE, NA),
    d2=c(FALSE, TRUE, FALSE, TRUE, FALSE, TRUE)
))
##          a         b c    d1    d2
## 1 0.287578  0.070508 A FALSE FALSE
## 2 0.788305  0.129288 B  TRUE  TRUE
## 3 0.408977  1.715065 C FALSE FALSE
## 4 0.883017  0.460916 D    NA  TRUE
## 5 0.940467 -1.265061 E FALSE FALSE
## 6 0.045556 -0.686853 F    NA  TRUE
typeof(x)  # each data frame is a list
## [1] "list"
```

(continues on next page)

[5] This is a strong word. This implication relies on an implicit assumption that the primitive functions **`length`** and **`names`** have not be contaminated by treating data frames differently than named lists. Luckily, that is indeed not the case. Also, despite the fact that we have the index operators specially overloaded for the `data.frame` class, they behave quite reasonably and, as we will see, they allow for a mix of list- and matrix-like behaviours.

(continued from previous page)

```
length(x)  # the number of columns
## [1] 5
names(x)   # column labels
## [1] "a"  "b"  "c"  "d1" "d2"
```

The one-argument versions of extract and index operators behave as expected. `[[` fetches (looks inside) the contents of a given column:

```
x[["a"]]  # or x[[1]]
## [1] 0.287578 0.788305 0.408977 0.883017 0.940467 0.045556
```

and `[` returns a data frame (a list with extras) comprised of the specified elements:

```
x["a"]  # or x[1]
##          a
## 1 0.287578
## 2 0.788305
## 3 0.408977
## 4 0.883017
## 5 0.940467
## 6 0.045556
x[c(TRUE, TRUE, FALSE, TRUE, FALSE)]
##          a         b    d1
## 1 0.287578  0.070508 FALSE
## 2 0.788305  0.129288  TRUE
## 3 0.408977  1.715065 FALSE
## 4 0.883017  0.460916    NA
## 5 0.940467 -1.265061 FALSE
## 6 0.045556 -0.686853    NA
```

Just like with lists, the replacement versions of the said operators can be used to add new or replace existing columns.

```
y <- head(x, 1)  # for a more compact display
y[["a"]] <- round(y[["a"]], 1)  # replaces the column with new content
y[["b"]] <- NULL  # removes the column, like, totally
y[["e"]] <- 10*y[["a"]]^2  # adds a new column at the end
print(y)
##     a c    d1    d2   e
## 1 0.3 A FALSE FALSE 0.9
```

Example 12.9 *Some spam for thought to show how much we already know: some common use cases of indexing and vectorised functions:*

```
y <- head(x, 1)  # for a more compact display
```

Move column a to the end:

```
y[unique(c(names(y), "a"), fromLast=TRUE)]
##          b c    d1    d2       a
## 1 0.070508 A FALSE FALSE 0.28758
```

Remove column a and c:

```
y[-match(c("a", "c"), names(y))]
##          b    d1    d2
## 1 0.070508 FALSE FALSE
```

All columns between a and c:

```
y[match("a", names(y)):match("c", names(y))]
##         a        b c
## 1 0.28758 0.070508 A
```

Names starting with d:

```
y[grep("^d", names(y))]
##      d1    d2
## 1 FALSE FALSE
```

Change name of column c to z:

```
names(y)[names(y) == "c"] <- "z"  # in-place
print(y)
##         a        b z    d1    d2
## 1 0.28758 0.070508 A FALSE FALSE
```

Change names: d2 to u and d1 to v:

```
names(y)[match(c("d2", "d1"), names(y))] <- c("v", "u")  # in-place
print(y)
##         a        b z     u     v
## 1 0.28758 0.070508 A FALSE FALSE
```

Note Some R users might prefer the `$` operator over `[[`, but we do not. By default, the former supports partial matching of column names which might be appealing when R is used interactively. However, it does not work on matrices, nor it allows for programmatically generated names. It is also trickier to use on non-syntactically valid labels; compare Section 9.4.1.

Exercise 12.10 *Write a function* ***`names_replace`*** *that changes the name of a data frame columns based on a translation table given in a* `from=to` *fashion, for instance:*

```
names_replace <- function(x, ...) ...to.do...
x <- data.frame(a=1, b=2, c=3)
names_replace(x, c="new_c", a="new_a")
##   new_a b new_c
## 1     1 2     3
```

12.2.2 Data Frames are Matrix-like

Data frames can be considered "generalised" matrices. They store data of any kind (possibly mixed) organised in a tabular fashion. Some functions mentioned in the previous chapter will hence be overloaded for the data frame case. These include: **`dim`** (despite the lack of the `dim` attribute), **`NROW`**, **`NCOL`**, and **`dimnames`** (which is of course based on `row.names` and `names`).

For example:

```
(x <- data.frame(
    a=runif(6),
    b=rnorm(6),
    c=LETTERS[1:6],
    d1=c(FALSE, TRUE, FALSE, NA, FALSE, NA),
    d2=c(FALSE, TRUE, FALSE, TRUE, FALSE, TRUE)
))
##          a         b c    d1    d2
## 1 0.287578  0.070508 A FALSE FALSE
## 2 0.788305  0.129288 B  TRUE  TRUE
## 3 0.408977  1.715065 C FALSE FALSE
## 4 0.883017  0.460916 D    NA  TRUE
## 5 0.940467 -1.265061 E FALSE FALSE
## 6 0.045556 -0.686853 F    NA  TRUE
dim(x)  # the number of rows and columns
## [1] 6 5
dimnames(x)  # it is not a matrix, but a matrix-like object
## [[1]]
## [1] "1" "2" "3" "4" "5" "6"
##
## [[2]]
## [1] "a"  "b"  "c"  "d1" "d2"
```

In addition to the list-like behaviour, which only allows for dealing with particular columns or groups thereof, the `` `[` `` operator was also equipped with the ability to take two indexers:

```
x[1:2, ]  # first two rows
##         a        b c    d1    d2
## 1 0.28758 0.070508 A FALSE FALSE
## 2 0.78831 0.129288 B  TRUE  TRUE
x[x[["a"]] >= 0.3  &  x[["a"]] <= 0.8, -2]  # or use x[, "a"]
##         a c    d1    d2
## 2 0.78831 B  TRUE  TRUE
## 3 0.40898 C FALSE FALSE
```

Recall the `drop` argument to `` `[` `` and its effects on matrix indexing. It the current case, its behaviour will be similar with regard to the operations on individual columns:

```
x[, 1]              # synonym: x[[1]], because drop=TRUE
## [1] 0.287578 0.788305 0.408977 0.883017 0.940467 0.045556
x[, 1, drop=FALSE]  # synonym: x[1]
##          a
## 1 0.287578
## 2 0.788305
## 3 0.408977
## 4 0.883017
## 5 0.940467
## 6 0.045556
```

Also, note that when we extract a single row and more than one column, `drop` does not really apply. It is because columns (unlike in matrices) can potentially be of different types:

```
x[1, 1:2]  # two numeric columns but the result is still a numeric
##         a        b
## 1 0.28758 0.070508
```

However:

```
x[1, 1]
## [1] 0.28758
x[1, 1, drop=FALSE]
##         a
## 1 0.28758
```

Note Once again let us take note of logical indexing featuring missing values:

```
x[x[["d1"]], ]
##            a       b    c   d1   d2
## 2    0.78831 0.12929    B TRUE TRUE
```

(continues on next page)

(continued from previous page)

```
## NA        NA      NA <NA>   NA   NA
## NA.1      NA      NA <NA>   NA   NA
x[which(x[["d1"]]), ]  # drops missing values
##         a       b c   d1   d2
## 2 0.78831 0.12929 B TRUE TRUE
```

The default behaviour is consistent with many other R functions: it explicitly indicates that something is missing (we are selecting a "don't know"; hence, the result is "don't know" as well). Unfortunately, this comes with no warning. As we rarely check manually for missing values in the outputs, our absent-mindedness can lead to code bugs.

By far, we might have already noted that the index operator adjusts (not: resets) the `row.names` attribute. For instance:

```
(xs <- x[head(order(x[["a"]], decreasing=TRUE), 3), ])
##         a        b c    d1    d2
## 5 0.94047 -1.26506 E FALSE FALSE
## 4 0.88302  0.46092 D    NA  TRUE
## 2 0.78831  0.12929 B  TRUE  TRUE
```

It is a version of `x` comprised of only top three values in the `u` column. Indexing by means of character vectors will refer to `row.names` and `names`:

```
xs["5", c("a", "b")]
##         a       b
## 5 0.94047 -1.2651
```

Note that this is not the same as "`xs[5, c("a", "b")]`", despite the fact that `row.names` is formally an integer vector here.

Note If a data frame features a matrix, we need to use the index/extract operator twice in order to access a specific sub-column:

```
(x <- aggregate(iris[1], iris[5], function(x) c(Min=min(x), Max=max(x))))
##      Species Sepal.Length.Min Sepal.Length.Max
## 1     setosa              4.3              5.8
## 2 versicolor              4.9              7.0
## 3  virginica              4.9              7.9
x[["Sepal.Length"]][, "Min"]
## [1] 4.3 4.9 4.9
```

In other words, neither "`x[["Sepal.Length.Min"]]`" nor "`x[, "Sepal.Length.Min"]`" works.

As far as the replacement version of the index operator is concerned, it is a quite flexible tool, allowing the new content to be a vector, a data frame, a list, or even a matrix.

Exercise 12.11 *Write two replacement functions*[6]*. First,* ***`set_row_names`*** *which replaces the* `row.names` *of a data frame with the contents of a specific column, for example:*

```
(x <- aggregate(iris[1], iris[5], mean))  # some data frame
##      Species Sepal.Length
## 1     setosa        5.006
## 2 versicolor        5.936
## 3  virginica        6.588
set_row_names(x) <- "Species"
print(x)
##            Sepal.Length
## setosa            5.006
## versicolor        5.936
## virginica         6.588
```

Second, ***`reset_row_names`*** *which converts* `row.names` *to a standalone column of a given name, for instance:*

```
reset_row_names(x) <- "Type"
print(x)
##   Sepal.Length       Type
## 1        5.006     setosa
## 2        5.936 versicolor
## 3        6.588  virginica
```

These two functions may be handy as they allow for writing "`x[something, ]`*" instead of "*`x[x[["column"]] %in% something, ]`*".*

12.3 Common Operations

Below we review the most commonly applied operations related to data frame wrangling. We have a few dedicated functions or methods overloaded for the `data.frame` class. However, we have already mastered the necessary skills to deal with this kind of objects through our hard work, in particular involving the solving of the ex-

[6] (*) Compare **`pandas.DataFrame.set_index`** and **`pandas.DataFrame.reset_index`** in Python.

ercises in the preceding chapters. Let us repeat: data frames are just lists exhibiting matrix-like behaviour.

12.3.1 Ordering Rows

Ordering rows in a data frame with respect to different criteria can be easily achieved by means of the **order** function and the two-argument version of `[`.

For instance, here are the top six cars in terms of the time (in seconds) to complete a 402-metre race:

```
mtcars6 <- mtcars[order(mtcars[["qsec"]])[1:6], ]
mtcars6[["model"]] <- row.names(mtcars6)
row.names(mtcars6) <- NULL
print(mtcars6)
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
```

order uses a stable sorting algorithm, therefore sorting with respect to a different criterion will not break the *relative* ordering of `qsec` in row groups with ties:

```
mtcars6[order(mtcars6[["cyl"]]), ]
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
```

Example 12.12 *Notice the difference between ordering by `cyl` and `gear` vs `gear` and `cyl`:*

```
mtcars6[order(mtcars6[["cyl"]], mtcars6[["gear"]]), ]
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
mtcars6[order(mtcars6[["gear"]], mtcars6[["cyl"]]), ]
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
```

(continues on next page)

(continued from previous page)

```
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
```

Note Mixing a increasing and decreasing ordering is tricky as the `decreasing` argument to **`order`** currently does not accept multiple flags in all the contexts. Perhaps the easiest way to change the ordering direction is to use the unary minus operator on the column(s) to be sorted decreasingly.

```
mtcars6[order(mtcars6[["gear"]], -mtcars6[["cyl"]]), ]
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
```

For factor and character columns, **`xtfrm`** can be used to convert them to sort keys first.

```
mtcars6[order(mtcars6[["cyl"]], -xtfrm(mtcars6[["model"]])), ]
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
```

Both of the above behave like `decreasing=c(FALSE, TRUE)`.

Exercise 12.13 *Write a method* ***`sort.data.frame`*** *that orders a data frame with respect to a given set of columns.*

```
sort.data.frame <- function(x, decreasing=FALSE, cols) ...to.do...
sort(mtcars6, cols=c("cyl", "model"))
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb          model
## 4 19.7   6  145 175 3.62 2.77 15.50  0  1    5    6   Ferrari Dino
## 6 21.0   6  160 110 3.90 2.62 16.46  0  1    4    4      Mazda RX4
## 3 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4     Camaro Z28
```

(continues on next page)

(continued from previous page)

```
## 5 14.3   8  360 245 3.21 3.57 15.84  0  0    3    4     Duster 360
## 1 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4 Ford Pantera L
## 2 15.0   8  301 335 3.54 3.57 14.60  0  1    5    8  Maserati Bora
```

*Unfortunately, that `decreasing` must be of length one and be placed as the second method argument is imposed by the **`sort`** S3 generic.*

12.3.2 Handling Duplicated Rows

`duplicated`, **`anyDuplicated`**, and **`unique`** have methods overloaded for the `data.frame` class. They can be used to indicate, get rid of, or replace the repeating rows.

```
sum(duplicated(iris))  # how many duplicated rows are there?
## [1] 1
iris[duplicated(iris), ]  # show the duplicated rows
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 143          5.8         2.7          5.1         1.9 virginica
```

12.3.3 Joining (Merging) Data Frames

The **`merge`** function can perform the JOIN operation that some readers might know from SQL[7]. It matches the items in the columns that two given data frames somewhat share, and then returns their combination.

Example 12.14 *Two calls to **`merge`** could be used to match data on programmers (each identified by `developer_id` and giving such details as their name, location, main skill, etc.) with the information about the open-source projects (each identified by `project_id` and informing us about its title, scope, web site, and so forth) they are engaged in (based on a third data frame featuring `developer_id` and `project_id` pairs).*

As an simple illustration, consider the two following objects:

```
A <- data.frame(
    u=c("b0", "b1", "b2", "b3"),
    v=c("a0", "a1", "a2", "a3")
)

B <- data.frame(
    v=c("a0", "a2", "a2", "a4"),
    w=c("c0", "c1", "c2", "c3")
)
```

[7] JOIN is the reverse operation to data normalisation known from theory of relational databases, which itself reduces data redundancy and increases their integrity. What data scientists need for succeeding with their daily activities (analysis, visualisation, processing) is thus the opposite of what the art of data management focuses on (efficient collection and storage). Readers are encouraged to learn about various normalisation forms from, e.g., [11] or any other course covering this topic.

The two *common* columns, i.e., storing data of similar nature (`a`-something strings), are both named `v`.

First, the *inner (natural) join*, where we list only the matching pairs:

```
merge(A, B)  # x=A, y=B, by="v", all.x=FALSE, all.y=FALSE
##    v  u  w
## 1 a0 b0 c0
## 2 a2 b2 c1
## 3 a2 b2 c2
```

Note that the common column (or, more generally, columns) is included only once in the result.

The *left join* guarantees that all elements in the first data frame will be included in the result:

```
merge(A, B, all.x=TRUE)  # by="v", all.y=FALSE
##    v  u    w
## 1 a0 b0   c0
## 2 a1 b1 <NA>
## 3 a2 b2   c1
## 4 a2 b2   c2
## 5 a3 b3 <NA>
```

The *right join* includes all records in the second argument:

```
merge(A, B, all.y=TRUE)  # by="v", all.x=FALSE
##    v    u  w
## 1 a0   b0 c0
## 2 a2   b2 c1
## 3 a2   b2 c2
## 4 a4 <NA> c3
```

And the *full outer join* is their set-theoretic union:

```
merge(A, B, all.x=TRUE, all.y=TRUE)  # by="v"
##    v    u    w
## 1 a0   b0   c0
## 2 a1   b1 <NA>
## 3 a2   b2   c1
## 4 a2   b2   c2
## 5 a3   b3 <NA>
## 6 a4 <NA>   c3
```

Exercise 12.15 *Show how* ***`match`*** *(Section 5.4.1) can be used to implement a very basic version of* ***`merge`****.*

12.3.4 Aggregating and Transforming Columns

Let us discuss how to perform data aggregation or engineer features. Despite the fact that we already know how to access individual columns with `[` and process them using the many vectorised functions, we still have something interesting to add about the said matter.

It would be tempting to try implementing such operations with **apply**. Unfortunately, currently this function coerces its argument to a matrix. Hence, we should refrain from applying it on data frames whose columns are of mixed types[8].

However, taking into account that data frames are special lists, we can always call **Map** and its relatives.

Example 12.16 *Given an example data frame:*

```
(iris_sample <- iris[sample(NROW(iris), 6), ])
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 28           5.2         3.5          1.5         0.2     setosa
## 80           5.7         2.6          3.5         1.0 versicolor
## 101          6.3         3.3          6.0         2.5  virginica
## 111          6.5         3.2          5.1         2.0  virginica
## 137          6.3         3.4          5.6         2.4  virginica
## 133          6.4         2.8          5.6         2.2  virginica
```

To get the class of each column, we can call:

```
sapply(iris_sample, class)  # or unlist(Map(class, iris))
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##    "numeric"    "numeric"    "numeric"    "numeric"     "factor"
```

Next, here is a way to compute some aggregates of the numeric columns:

```
unlist(Map(mean, Filter(is.numeric, iris_sample)))
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##       6.0667       3.1333       4.5500       1.7167
```

or:

```
sapply(iris_sample[sapply(iris_sample, is.numeric)], mean)
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##       6.0667       3.1333       4.5500       1.7167
```

We can also fetch more than a single summary of each column:

```
as.data.frame(Map(
    function(x) c(Min=min(x), Max=max(x)),
```

(continues on next page)

[8] Due to this, storing data as matrix columns inside data frames is not such a bad idea.

(continued from previous page)

```
    Filter(is.numeric, iris_sample)
))
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## Min          5.2         2.6          1.5         0.2
## Max          6.5         3.5          6.0         2.5
```

or:

```
sapply(iris_sample[sapply(iris_sample, is.numeric)], quantile, c(0, 1))
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## 0%            5.2         2.6          1.5         0.2
## 100%          6.5         3.5          6.0         2.5
```

Note that the latter called ***simplify2array*** *automatically, thus the result is a matrix.*

On the other hand, standardisation of all the numeric features can be performed, e.g., via a call:

```
iris_sample[] <- Map(function(x) {
    if (!is.numeric(x)) x else (x-mean(x))/sd(x)
}, iris_sample)
print(iris_sample)
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 28       -1.70405     1.03024     -1.76004    -1.65318     setosa
## 80       -0.72094    -1.49854     -0.60591    -0.78117 versicolor
## 101       0.45878     0.46829      0.83674     0.85384  virginica
## 111       0.85202     0.18732      0.31738     0.30884  virginica
## 137       0.45878     0.74927      0.60591     0.74484  virginica
## 133       0.65540    -0.93659      0.60591     0.52684  virginica
```

12.3.5 Handling Missing Values

The **is.na** method for objects of class `data.frame` returns a logical matrix of the same dimensionality[9] indicating whether the corresponding items are missing or not. Of course, this function can still be called on individual columns as well.

Further, **na.omit** can be used to get rid of rows with missing values.

Exercise 12.17 *Given a data frame, use* ***is.na*** *and other functions such as* ***apply***, ***approx***, *etc., to:*

1. *remove all rows that feature at least one missing value,*
2. *remove all rows that only consist of missing values,*
3. *remove all columns that feature at least one missing value,*
4. *for each column, replace all missing values with the column averages,*

[9] Provided that a data frame does not feature a matrix column.

5. *for each column, replace all missing values with values that linearly interpolate between the preceding and succeeding well-defined observations (which is useful on time series), e.g., the blanks in* `c(0.60, 0.62, NA, 0.64, NA, NA, 0.58)` *should be filled so as to obtain* `c(0.60, 0.62, 0.63, 0.64, 0.62, 0.60, 0.58)`.

12.3.6 Reshaping Data Frames

Consider an example matrix:

```
A <- matrix(round(runif(6), 2), nrow=3,
    dimnames=list(
        c("X", "Y", "Z"),  # row labels
        c("u", "v")        # column labels
))
names(dimnames(A)) <- c("Row", "Col")
print(A)
##    Col
## Row    u    v
##   X 0.29 0.88
##   Y 0.79 0.94
##   Z 0.41 0.05
```

The **`as.data.frame`** method for the `table` class can be called directly on any array:

```
as.data.frame.table(A, responseName="Val")
##   Row Col  Val
## 1   X   u 0.29
## 2   Y   u 0.79
## 3   Z   u 0.41
## 4   X   v 0.88
## 5   Y   v 0.94
## 6   Z   v 0.05
```

This is an instance of reshaping an array, and more precisely, *stacking*: converting from a *wide* (okay, in this example, not so wide, as we have only two columns) to a *long* format.

This can be also achieved by means of the **`reshape`** function which is more flexible and operates directly on data frames (but is harder to use):

```
(df <- `names<-`(
    data.frame(row.names(A), A, row.names=NULL),
    c("Row", "Col.u", "Col.v")))
##   Row Col.u Col.v
## 1   X  0.29  0.88
## 2   Y  0.79  0.94
## 3   Z  0.41  0.05
```

(continues on next page)

(continued from previous page)

```
(stacked <- reshape(df, varying=2:3, direction="long"))
##     Row time  Col id
## 1.u   X    u 0.29  1
## 2.u   Y    u 0.79  2
## 3.u   Z    u 0.41  3
## 1.v   X    v 0.88  1
## 2.v   Y    v 0.94  2
## 3.v   Z    v 0.05  3
```

Maybe the default column names are not superb, but we can always adjust them manually afterwards.

The reverse operation is called *unstacking*:

```
reshape(stacked, idvar="Row", timevar="time", drop="id", direction="wide")
##     Row Col.u Col.v
## 1.u   X  0.29  0.88
## 2.u   Y  0.79  0.94
## 3.u   Z  0.41  0.05
```

Exercise 12.18 *Given a named numeric vector, convert it to a data frame with two columns, for instance:*

```
convert <- function(x) ...to.do...
x <- c(spam=42, eggs=7, bacon=3)
convert(x)
##     key value
## 1  spam    42
## 2  eggs     7
## 3 bacon     3
```

Exercise 12.19 *Reshape (stack) the built-in* `WorldPhones` *dataset. Then, reshape (unstack) the stacked* `WorldPhones` *dataset. Further, unstack the stacked set but first remove*[10] *five random rows from it, and then randomly permute all the remaining rows. Fill the missing entries with* `NA`*s.*

Exercise 12.20 *Implement a basic version of* ***`as.data.frame.table`*** *manually (using* ***`rep`*** *etc.). Also, write a function* ***`as.table.data.frame`*** *that implements its reverse. Make sure both functions are compatible with each other.*

Exercise 12.21 *The built-in* `Titanic` *is a four-dimensional array. Convert it to a long data frame.*

Exercise 12.22 *Perform what follows on the data frame defined below:*

1. *convert the second column from character to a list of character vectors (split at* `", "`*);*

[10] The original dataset can be thought of as representing a fully crossed design experiment (all combinations of two grouping variables are present). Its truncated version is like an incomplete cross design.

2. *extract first elements from each of the vectors;*
3. *extract last elements;*
4. *(*) unstack the data frame;*
5. *(*) stack it back to a data frame featuring a list;*
6. *convert the list back to a character column (concatenate with `","` as separator).*

```
(x <- data.frame(
    name=c("Kat", "Ron", "Jo", "Mary"),
    food=c("buckwheat", "spam,bacon,spam", "", "eggs,spam,spam,lollipops")
))
##   name                     food
## 1  Kat                 buckwheat
## 2  Ron           spam,bacon,spam
## 3   Jo
## 4 Mary eggs,spam,spam,lollipops
```

Exercise 12.23 *Write a function that converts all matrix-based columns in a given data frame to separate, atomic columns. Also, write a function to that does the opposite: one that groups all columns with similar prefixes and turns them into matrices.*

12.3.7 Aggregating Data in Groups

We can straightforwardly apply various transforms on data groups determined by a factor-like variable or a combination thereof thanks to the **`split.data.frame`** method, which returns a list of data frames.

For example:

```
x <- data.frame(
    a=c(    10,     20,     30,     40,     50),
    u=c("spam", "spam", "eggs", "spam", "eggs"),
    v=c(     1,      2,      1,      1,      1)
)
split(x, x["u"])  # i.e., split.data.frame(x, x["u"]) or x[["u"]]
## $eggs
##    a    u v
## 3 30 eggs 1
## 5 50 eggs 1
##
## $spam
##    a    u v
## 1 10 spam 1
## 2 20 spam 2
## 4 40 spam 1
```

This split x with respect to the u column serving as the grouping variable. On the other hand:

```
split(x, x[c("u", "v")])  # sep="."
## $eggs.1
##    a    u v
## 3 30 eggs 1
## 5 50 eggs 1
##
## $spam.1
##    a    u v
## 1 10 spam 1
## 4 40 spam 1
##
## $eggs.2
## [1] a u v
## <0 rows> (or 0-length row.names)
##
## $spam.2
##    a    u v
## 2 20 spam 2
```

partitioned with respect to a combination of two factor-like sequences. Note that a non-existing level pair (eggs, 2) results in an empty data frame.

Exercise 12.24 ***split.data.frame*** *(when called explicitly) can also be used to break a matrix into a list of matrices (rowwisely). Given a matrix, perform its train-test split: allocate, say, 70% of the rows at random into one matrix and the remaining 30% into another one.*

If the aggregation of grouped data in numeric columns is needed, **sapply** is quite convenient. To recall, it is a combination of **lapply** (one-vector version of **Map**) and **simplify2array** (Section 11.1.3).

```
sapply(split(iris[1:2], iris[5]), sapply, mean)
##              setosa versicolor virginica
## Sepal.Length  5.006      5.936     6.588
## Sepal.Width   3.428      2.770     2.974
```

If the function being to apply returns more than a single value, **sapply** will not return a too-informative result by default: the list of matrices converted to a matrix will not have the `row.names` argument set. As a workaround, we either call **simplify2array** explicitly or pass `simplify="array"` to **sapply**:

```
(res <- sapply(
    split(iris[1:2], iris[5]),
    sapply,
    function(x) c(Min=min(x), Max=max(x)),
```

(continues on next page)

(continued from previous page)

```
    simplify="array"
))  # or simplify2array(lapply or Map etc.)
## , , setosa
##
##     Sepal.Length Sepal.Width
## Min          4.3         2.3
## Max          5.8         4.4
##
## , , versicolor
##
##     Sepal.Length Sepal.Width
## Min          4.9         2.0
## Max          7.0         3.4
##
## , , virginica
##
##     Sepal.Length Sepal.Width
## Min          4.9         2.2
## Max          7.9         3.8
```

This yields a three-dimensional array which is particularly handy if we now would like to access specific results by name:

```
res[, "Sepal.Length", "setosa"]
## Min Max
## 4.3 5.8
```

Also, the previously mentioned **`as.data.frame.table`** method works like a charm on it (up to the column names):

```
as.data.frame.table(res)
##    Var1         Var2       Var3 Freq
## 1   Min Sepal.Length     setosa  4.3
## 2   Max Sepal.Length     setosa  5.8
## 3   Min  Sepal.Width     setosa  2.3
## 4   Max  Sepal.Width     setosa  4.4
## 5   Min Sepal.Length versicolor  4.9
## 6   Max Sepal.Length versicolor  7.0
## 7   Min  Sepal.Width versicolor  2.0
## 8   Max  Sepal.Width versicolor  3.4
## 9   Min Sepal.Length  virginica  4.9
## 10  Max Sepal.Length  virginica  7.9
## 11  Min  Sepal.Width  virginica  2.2
## 12  Max  Sepal.Width  virginica  3.8
```

Note If the grouping (`by`) variable is a list of two or more factors, the combined levels will be concatenated to a single string:

```
as.data.frame.table(as.array(sapply(
    split(ToothGrowth["len"], ToothGrowth[c("supp", "dose")]),
    sapply,
    mean
)))
##         Var1  Freq
## 1 OJ.0.5.len 13.23
## 2 VC.0.5.len  7.98
## 3   OJ.1.len 22.70
## 4   VC.1.len 16.77
## 5   OJ.2.len 26.06
## 6   VC.2.len 26.14
```

Also, the name of the aggregated column (`len`) has been included. This behaviour yields a result that may be deemed convenient in some contexts, but not necessarily so in other ones.

Exercise 12.25 *Many aggregation functions are idempotent, which means that when they are fed with a vector with all the elements being identical, the result is exactly that unique element:* ***min***, ***mean***, ***median***, *and* ***max*** *behave exactly this way.*

Overload the ***mean*** *and* ***median*** *methods for character vectors and factors so that they return* `NA` *when they are fed with a sequence of not all elements being the same and the unique value otherwise.*

```
mean.character <- function(x, na.rm=FALSE, ...) ...to.do...
mean.factor <- function(x, na.rm=FALSE, ...) ...to.do...
```

This way, we can also aggregate the grouping variables in a convenient way:

```
do.call(rbind.data.frame,
    lapply(split(ToothGrowth, ToothGrowth[c("supp", "dose")]), lapply, mean))
##          len supp dose
## OJ.0.5 13.23   OJ  0.5
## VC.0.5  7.98   VC  0.5
## OJ.1   22.70   OJ  1.0
## VC.1   16.77   VC  1.0
## OJ.2   26.06   OJ  2.0
## VC.2   26.14   VC  2.0
```

The built-in **`aggregate`** method can assist us in a situation where a single function is to be applied on all columns in a data frame.

```
aggregate(iris[-5], iris[5], mean)  # not: ...[[5]]
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
aggregate(ToothGrowth["len"], ToothGrowth[c("supp", "dose")], mean)
##   supp dose   len
## 1   OJ  0.5 13.23
## 2   VC  0.5  7.98
## 3   OJ  1.0 22.70
## 4   VC  1.0 16.77
## 5   OJ  2.0 26.06
## 6   VC  2.0 26.14
```

Note that the second argument, `by`, must be list-like (therefore also a data frame is accepted), not a factor nor an atomic vector. Also, if the function being applied returns many values, they will be wrapped into a matrix column:

```
(x <- aggregate(iris[2], iris[5], function(x) c(Min=min(x), Max=max(x))))
##      Species Sepal.Width.Min Sepal.Width.Max
## 1     setosa             2.3             4.4
## 2 versicolor             2.0             3.4
## 3  virginica             2.2             3.8
class(x[["Sepal.Width"]])
## [1] "matrix" "array"
x[["Sepal.Width"]]  # not: Sepal.Width.Max, etc.
##      Min Max
## [1,] 2.3 4.4
## [2,] 2.0 3.4
## [3,] 2.2 3.8
```

It is actually handy, because by referring to `x[["Sepal.Width"]]` we have access to all the stats for this column. Further, if many columns are being aggregated at the same time, we can process all the summaries in the same way.

Exercise 12.26 *Check out the built-in* ***by*** *function which supports some basic split-apply-bind use cases. Note the particularly peculiar behaviour of the* ***print*** *method for the* `by` *class.*

The most flexible scenario involves applying a custom function returning any set of aggregates in the form of a list and then row-binding the results to obtain a data frame.

Example 12.27 *The following implements an R version of what we would express in SQL as:*

```
SELECT supp, dose, AVG(len) AS ave_len, COUNT(*) AS count
FROM ToothGrowth
GROUP BY supp, dose
```

Ad rem:

```
do.call(rbind.data.frame, lapply(
    split(ToothGrowth, ToothGrowth[c("supp", "dose")]),
    function(df) list(
        supp=df[1, "supp"],
        dose=df[1, "dose"],
        ave_len=mean(df[["len"]]),
        count=NROW(df)
    )
))
##        supp dose ave_len count
## OJ.0.5   OJ  0.5   13.23    10
## VC.0.5   VC  0.5    7.98    10
## OJ.1     OJ  1.0   22.70    10
## VC.1     VC  1.0   16.77    10
## OJ.2     OJ  2.0   26.06    10
## VC.2     VC  2.0   26.14    10
```

Example 12.28 *As an exercise, let us study a function that takes a named list `x` (can be a data frame) and a sequence of `col=f` pairs and applies the function `f` (or each function from a list of functions `f`) on the named element `col` in `x`:*

```
napply <- function(x, ...)
{
    fs <- list(...)
    stopifnot(is.list(x), !is.null(names(x)))
    stopifnot(all(names(fs) %in% names(x)))
    do.call(
        c,  # concatenates lists
        lapply(
            structure(seq_along(fs), names=names(fs)),
            function(i) {  # always returns a list
                y <- x[[ names(fs)[i] ]]
                if (is.function(fs[[i]]))
                    list(fs[[i]](y))
                else
                    lapply(fs[[i]], function(f) f(y))
            }
        )
    )
}
```

For example:

```
first <- function(x, ...) head(x, n=1L, ...)  # we use it below
```

(continues on next page)

(continued from previous page)

```
napply(ToothGrowth,
    supp=first, dose=first, len=list(ave=mean, count=length)
)
## $supp
## [1] VC
## Levels: OJ VC
##
## $dose
## [1] 0.5
##
## $len.ave
## [1] 18.813
##
## $len.count
## [1] 60
```

applies ***first*** *on both* `ToothGrowth[["supp"]]` *and* `ToothGrowth[["dose"]]` *as well as* ***mean*** *and* ***length*** *on* `ToothGrowth[["len"]]`*. List names are there for some dramatic effects.*

And now:

```
do.call(
    rbind.data.frame,
    lapply(
        split(ToothGrowth, ToothGrowth[c("supp", "dose")]),
        napply,
        supp=first, dose=first, len=list(ave=mean, count=length)
    )
)
##        supp dose len.ave len.count
## OJ.0.5   OJ  0.5   13.23        10
## VC.0.5   VC  0.5    7.98        10
## OJ.1     OJ  1.0   22.70        10
## VC.1     VC  1.0   16.77        10
## OJ.2     OJ  2.0   26.06        10
## VC.2     VC  2.0   26.14        10
```

or even:

```
aaaggg <- function(x, by, ...)
    do.call(rbind.data.frame, lapply(split(x, x[by]), napply, ...))
```

so that:

```
aaaggg(iris, "Species", Species=first, Sepal.Length=mean)
```

(continues on next page)

(continued from previous page)

```
##               Species Sepal.Length
## setosa         setosa        5.006
## versicolor versicolor        5.936
## virginica   virginica        6.588
```

This brings fun back to R programming in the sad times when many things are given to us on a plate. And by the way, the above has not been tested thoroughly, it is a proof of concept; as usual, testing, debugging, and extending is left as an exercise to the reader.

Example 12.29 *In Section 10.5, we have considered an example where we have used our own* ***group_by*** *function and an aggregation method overloaded for the object's class it returns.*

Here is the function that splits a data frame into a list of data frames with respect to a combination of levels in given named columns:

```
group_by <- function(df, by)
{
    stopifnot(is.character(by), is.data.frame(df))
    df <- droplevels(df)  # in case there are factors with empty levels
    structure(
        split(df, df[names(df) %in% by]),
        class="list_dfs",
        by=by
    )
}
```

The next function applies a set of aggregates on every column of each data frame in a given list (two nested ***lapply****s plus some cosmetic additions):*

```
aggregate.list_dfs <- function(x, FUN, ...)
{
    aggregates <- lapply(x, function(df) {
        is_by <- names(df) %in% attr(x, "by")
        res <- lapply(df[!is_by], FUN, ...)
        res_mat <- do.call(rbind, res)
        if (is.null(dimnames(res_mat)[[2]]))
            dimnames(res_mat)[[2]] <- paste0("f", seq_len(NCOL(res_mat)))
        cbind(
            `row.names<-`(df[1, is_by, drop=FALSE], NULL),
            x=row.names(res_mat),
            `row.names<-`(res_mat, NULL)
        )
    })
    combined_aggregates <- do.call(rbind.data.frame, aggregates)
    `row.names<-`(combined_aggregates, NULL)
}
```

(continues on next page)

(continued from previous page)

```
aggregate(group_by(ToothGrowth, c("supp", "dose")), range)
##   supp dose   x   f1   f2
## 1   OJ  0.5 len  8.2 21.5
## 2   VC  0.5 len  4.2 11.5
## 3   OJ  1.0 len 14.5 27.3
## 4   VC  1.0 len 13.6 22.5
## 5   OJ  2.0 len 22.4 30.9
## 6   VC  2.0 len 18.5 33.9
```

We really want our API be bloated, hence let us introduce a convenience *function being a specialised version of the above:*

```
mean.list_dfs <- function(x, ...)
    aggregate.list_dfs(x, function(y) c(Mean=mean(y, ...)))
mean(group_by(iris[51:150, c(2, 3, 5)], "Species"))
##      Species            x  Mean
## 1 versicolor  Sepal.Width 2.770
## 2 versicolor Petal.Length 4.260
## 3  virginica  Sepal.Width 2.974
## 4  virginica Petal.Length 5.552
```

12.3.8 Transforming Data in Groups

Some variables will sometimes need to be transformed relative to what is happening in subsets of a dataset. This is the case, e.g., where we decide that missing values should be replaced by the corresponding within-group averages, or want to compute the relative ranks or z-scores.

If the losing of the original ordering of rows is not an issue, the standard split-apply-bind will suffice.

An example data frame:

```
(x <- data.frame(
    a=c( 10,   1,  NA,  NA,  NA,   4),
    b=c( -1,  10,  40,  30,   1,  20),
    c=runif(6),
    d=c("v", "u", "u", "u", "v", "u")
))
##    a  b       c d
## 1 10 -1 0.52811 v
## 2  1 10 0.89242 u
## 3 NA 40 0.55144 u
## 4 NA 30 0.45661 u
```

(continues on next page)

(continued from previous page)

```
## 5 NA  1 0.95683 v
## 6  4 20 0.45333 u
```

Some operations:

```
fill_na <- function(x) `[<-`(x, is.na(x), value=mean(x[!is.na(x)]))
standardise <- function(x) (x-mean(x))/sd(x)
```

And now:

```
do.call(rbind.data.frame, lapply(
    split(x, x["d"]),
    function(df) {
        df[["a"]] <- fill_na(df[["a"]])
        df[["b"]] <- rank(df[["b"]])
        df[["c"]] <- standardise(df[["c"]])
        df
    }
))
##        a b        c d
## u.2  1.0 1  1.46357 u
## u.3  2.5 4 -0.17823 u
## u.4  2.5 3 -0.63478 u
## u.6  4.0 2 -0.65057 u
## v.1 10.0 1 -0.70711 v
## v.5 10.0 2  0.70711 v
```

Note that only the *relative* ordering of rows within groups has been retained. Overall, the rows are in a different order.

If this is an issue, we can use the **`unsplit`** function:

```
unsplit(
    lapply(
        split(x, x["d"]),
        function(df) {
            df[["a"]] <- fill_na(df[["a"]])
            df[["b"]] <- rank(df[["b"]])
            df[["c"]] <- standardise(df[["c"]])
            df
        }
    ),
    x["d"]
)
##      a b        c d
```

(continues on next page)

(continued from previous page)

```
## 1 10.0 1 -0.70711 v
## 2  1.0 1  1.46357 u
## 3  2.5 4 -0.17823 u
## 4  2.5 3 -0.63478 u
## 5 10.0 2  0.70711 v
## 6  4.0 2 -0.65057 u
```

Exercise 12.30 *Show how we can do the above also via the replacement version of* ***`split`****.*

Example 12.31 *Reverting to the previous ordering can be done manually too. It is because the split operation behaves as if we first ordered the data frame with respect to the grouping variable(s) (using a stable sorting algorithm).*

Here is some transformation of a sample data frame split by a combination of two factors:

```
(x <- `row.names<-`(ToothGrowth[sample(NROW(ToothGrowth), 10), ], NULL))
##     len supp dose
## 1  23.0   OJ  2.0
## 2  23.3   OJ  1.0
## 3  29.4   OJ  2.0
## 4  14.5   OJ  1.0
## 5  11.2   VC  0.5
## 6  20.0   OJ  1.0
## 7  24.5   OJ  2.0
## 8  10.0   OJ  0.5
## 9   9.4   OJ  0.5
## 10  7.0   VC  0.5
(y <- do.call(rbind.data.frame, lapply(
    split(x, x[c("dose", "supp")]),  # two grouping variables
    function(df) {
        df[["len"]] <- df[["len"]] * 100^df[["dose"]] *  # whatever
            ifelse(df[["supp"]] == "OJ", -1, 1)          # do not overthink it
        df
    }
)))
##               len supp dose
## 0.5.OJ.8     -100   OJ  0.5
## 0.5.OJ.9      -94   OJ  0.5
## 1.OJ.2      -2330   OJ  1.0
## 1.OJ.4      -1450   OJ  1.0
## 1.OJ.6      -2000   OJ  1.0
## 2.OJ.1    -230000   OJ  2.0
## 2.OJ.3    -294000   OJ  2.0
## 2.OJ.7    -245000   OJ  2.0
## 0.5.VC.5      112   VC  0.5
## 0.5.VC.10      70   VC  0.5
```

In Section 5.4.4, we have mentioned that by calling **`order`**, *we ca determine the inverse of a given permutation. Hence, we can call:*

```
y[order(order(x[["supp"]], x[["dose"]])), ]  # not: dose, supp
##               len supp dose
## 2.OJ.1    -230000   OJ  2.0
## 1.OJ.2      -2330   OJ  1.0
## 2.OJ.3    -294000   OJ  2.0
## 1.OJ.4      -1450   OJ  1.0
## 0.5.VC.5      112   VC  0.5
## 1.OJ.6      -2000   OJ  1.0
## 2.OJ.7    -245000   OJ  2.0
## 0.5.OJ.8     -100   OJ  0.5
## 0.5.OJ.9      -94   OJ  0.5
## 0.5.VC.10      70   VC  0.5
```

Additionally, we can manually restore the original `row.names`, *et voilà.*

12.3.9 Metaprogramming-Based Techniques (*)

In Section 9.5.7, we have mentioned that due to R's being equipped with the ability to write programs that manipulate unevaluated expressions, some functions can provide us with quite weird interfaces to a few common operations. These include **`transform`**, **`subset`**, **`with`**, and basically every procedure accepting a formula. Also, the popular **`data.table`** and **`dplyr`** packages that we briefly mention in Section 12.3.10 fall into this class.

In some contexts, they all may be found convenient[11].

However, overall, each of these methods must be carefully studied separately. This is because they can arbitrarily interpret the *form* of the arguments passed thereto, without taking into account their *real* meaning.

We try to avoid[12] them in this course, as we can do perfectly without them. However, they are not only interesting on their own, but also quite popular in other users' code, hence the honourable mention. Learning them in more detail is left to the kind reader as an optional exercise. In `sec:to-do`, we will return to these functions as they will serve as a very interesting illustration of how to implement our own procedures that rely on metaprogramming techniques.

[11] And, in the case of third-party packages, sometimes faster and more memory efficient (on larger datasets), as it is usually the case with more specialised tools. However, in many daily programming contexts, speed of the data wrangling operations is not that often an issue. Remember that we always have SQL-supporting relational databases at our disposal too.

[12] We are not alone in our calling to refrain from using them. **`help`**`("subset")` warns (and **`help`**`("transform")` quite similarly): *This is a convenience function intended for use interactively. For programming, it is better to use the standard subsetting functions like* **`[`**, *and in particular the non-standard evaluation of argument* **`subset`** *can have unanticipated consequences.* The same in **`help`**`("with")`: *For* interactive *use, this is very effective and nice to read. For programming however, i.e., in one's functions, more care is needed, and typically one should refrain from using* **`with`**, *as, e.g., variables in data may accidentally override local variables.*

Example 12.32 *For instance, let us consider an example call to the* ***`subset`*** *function:*

```
subset(iris, Sepal.Length>7.5, -(Sepal.Width:Petal.Width))
##     Sepal.Length   Species
## 106          7.6 virginica
## 118          7.7 virginica
## 119          7.7 virginica
## 123          7.7 virginica
## 132          7.9 virginica
## 136          7.7 virginica
```

Neither `Sepal.Length>7.5` *nor* `-(Sepal.Width:Petal.Width)` *make sense as standalone R expressions, because we have not defined the named variables used therein:*

```
Sepal.Length>7.5            # utter nonsense
## Error in eval(expr, envir, enclos): object 'Sepal.Length' not found
-(Sepal.Width:Petal.Width)  # gibberish
## Error in eval(expr, envir, enclos): object 'Sepal.Width' not found
```

Only from ***`help`***`("subset")`*, we can learn that this tool generously decides that the second expression plays the role of a row selector and the third one removes all the columns between the two given ones.*

In our course, we pay attention to developing transferable skills. Assuming that R is not the only language we are going to learn during of our long and happy lives, it is much more likely that in the next environment, we will rather be writing something more of the more basic form:

```
between <- function(x, from, to) (which(from == x):which(to == x))
iris[iris[["Sepal.Length"]]>7.5,
    -between(names(iris), "Sepal.Width", "Petal.Width")]
##     Sepal.Length   Species
## 106          7.6 virginica
## 118          7.7 virginica
## 119          7.7 virginica
## 123          7.7 virginica
## 132          7.9 virginica
## 136          7.7 virginica
```

Let us stress again that this is a book on how to become a great chef who proudly uses produce from sustainable sources, and not how to order ultra-processed food from DeliverNoodles.com.

Example 12.33 ***`transform`*** *can be used to add, modify, and remove columns in a data frame with the possibility of referring to existing features as if they were ordinary variables:*

```
head(transform(mtcars, log_hp=log(hp), am=2*am-1, hp=NULL))
##                   mpg cyl disp drat    wt  qsec vs am gear carb log_hp
## Mazda RX4         21.0   6  160 3.90 2.620 16.46  0  1    4    4 4.7005
```

(continues on next page)

(continued from previous page)

```
## Mazda RX4 Wag     21.0   6  160 3.90 2.875 17.02  0  1    4    4 4.7005
## Datsun 710        22.8   4  108 3.85 2.320 18.61  1  1    4    1 4.5326
## Hornet 4 Drive    21.4   6  258 3.08 3.215 19.44  1 -1    3    1 4.7005
## Hornet Sportabout 18.7   8  360 3.15 3.440 17.02  0 -1    3    2 5.1648
## Valiant           18.1   6  225 2.76 3.460 20.22  1 -1    3    1 4.6540
```

*Similarly, **`attach`** adds any named list to the search path (see Chapter 16) so that the columns can be accessed by name. This, however, does not allow any alterations thereof to be performed. As an alternative, **`with`** and **`within`** may be referred to if writing `df[["..."]]` each time is so difficult to us (it should not be):*

```
within(head(mtcars), {
    log_hp <- log(hp)
    fuel_economy <- 235/mpg
    am <- factor(am, levels=c(0, 1), labels=c("no", "yes"))
    rm(list=c("mpg", "hp", "vs", "qsec"))
})
##                   cyl disp drat    wt  am gear carb fuel_economy log_hp
## Mazda RX4           6  160 3.90 2.620 yes    4    4       11.190 4.7005
## Mazda RX4 Wag       6  160 3.90 2.875 yes    4    4       11.190 4.7005
## Datsun 710          4  108 3.85 2.320 yes    4    1       10.307 4.5326
## Hornet 4 Drive      6  258 3.08 3.215  no    3    1       10.981 4.7005
## Hornet Sportabout   8  360 3.15 3.440  no    3    2       12.567 5.1648
## Valiant             6  225 2.76 3.460  no    3    1       12.983 4.6540
```

Example 12.34 *As mentioned in Section 10.3.2, see Section 16.5 for more details, formulas are special objects that consist of two unevaluated expressions separated by a tilde (`~`).*

Functions can support formulas and do what they please with them, but a popular approach is to allow them to express "something grouped by something else" or "one thing as a function of other things".

```
do.call(rbind.data.frame, lapply(split(ToothGrowth, ~supp+dose), head, 1))
##         len supp dose
## OJ.0.5 15.2   OJ  0.5
## VC.0.5  4.2   VC  0.5
## OJ.1   19.7   OJ  1.0
## VC.1   16.5   VC  1.0
## OJ.2   25.5   OJ  2.0
## VC.2   23.6   VC  2.0
aggregate(cbind(mpg, log_hp=log(hp))~am:cyl, mtcars, mean)
##   am cyl    mpg log_hp
## 1  0   4 22.900 4.4186
## 2  1   4 28.075 4.3709
## 3  0   6 19.125 4.7447
```

(continues on next page)

(continued from previous page)

```
## 4  1   6 20.567 4.8552
## 5  0   8 15.050 5.2553
## 6  1   8 15.400 5.6950
head(model.frame(mpg~hp~log(hp)+I(1/qsec), mtcars))
##                   mpg + hp log(hp)    I(1/qsec)
## Mazda RX4            131.0  4.7005 0.060753....
## Mazda RX4 Wag        131.0  4.7005 0.058754....
## Datsun 710           115.8  4.5326 0.053734....
## Hornet 4 Drive       131.4  4.7005 0.051440....
## Hornet Sportabout    193.7  5.1648 0.058754....
## Valiant              123.1  4.6540 0.049455....
```

If these seem esoteric, it is because that is exactly the case. We need to consult the corresponding functions' manuals to be able to understand what they do. And, as we do not recommend their use, we are not going to explain them here.

Exercise 12.35 *In the last example, the peculiar printing of the last column is due to which method being overloaded?*

12.3.10 A Note on the `dplyr` (`tidyverse`) and `data.table` Packages (*)

The popular third-party packages **`data.table`** and **`dplyr`** implement the most common data frame wrangling procedures. Moreover, some of the operations may be much faster for larger data sets.

They both introduce a completely new API for the operations we already know well how to perform. Furthermore, they are heavily based on metaprogramming (nonstandard evaluation). A good way to learn them is by solving some of the exercises listed below.

Note that **`dplyr`** is part of a huge system of interdependent packages called **`tidyverse`** which tend to do things their own way and which became quite invasive over the last years. Nevertheless, R programmers should remember that they are not only able to do without them; they also need to when the processing of other prominent data structures is required, e.g., of fancy lists and matrices. Base R always comes first as the more fundamental layer.

Important Some functions we may find useful will (annoyingly to base R users) return objects of class `tibble` (`tbl_df`) (e.g., **`haven::read.xpt`** that reads **`SAS`** data files). However, those are in fact `data.frame` subclasses and we can always use **`as.data.frame`** to get our favourite objects back.

Also, we cannot stress enough that it is SQL that we recommend to learn as perhaps the most powerful interface to more considerable amounts of data, and also one that gives skills which can be used at a later time in other programming environments.

We should remember that base R has already proven long time ago to be a versatile

tool for rapid prototyping, calling specialised procedures written in C or Java, and wrangling data that *fit into memory*. For larger problems, techniques for working with batches of data, sampling methods, or aggregating data stored elsewhere is often the way to go, especially when building machine learning models or visualisation[13] is required. Usually, the most recent data will be stored in normalised databases and you will need to join a few tables in order to fetch something of interest in the current analysis context.

12.4 Exercises

Exercise 12.36 *Answer the following questions:*

- *What attributes a data frame must be equipped with?*
- *If `row.names` is an integer vector, how to access rows labelled `1`, `7`, and `42`?*
- *How to create a data frame that features a column that is a list of character vectors of different lengths?*
- *How to create a data frame that includes a matrix column?*
- *How to convert all numeric columns in a data frame to a numeric matrix?*
- *Assuming that `x` is an atomic vector, what is the difference between "`as.data.frame(x)`" vs "`as.data.frame(as.list(x))`" vs "`as.data.frame(list(a=x))`" vs "`data.frame(a=x)`"?*

Exercise 12.37 *Assuming that `x` is a data frame, what is the meaning of/difference between the following:*

- *"`x["u"]`" vs "`x[["u"]]`" vs "`x[, "u"]`"?*
- *"`x["u"][1]`" vs "`x[["u"]][1]`" vs "`x[1, "u"]`" vs "`x[1, "u", drop=FALSE]`"?*
- *"`x[which(x[[1]] > 0), ]`" vs "`x[x[[1]] > 0, ]`"?*
- *"`x[grep("^foo", names(x))]`"?*

Exercise 12.38 *Assume we have a data frame with columns named like: `ID` (character), `checked` (logical, possibly with missing values), `category` (factor), `x0`, … `x9` (ten separate numeric columns), `y0`, … `y9` (ten separate numeric columns), `coords` (numeric matrix with two columns named `lat` and `long`), and `features` (list of character vectors of different lengths).*

- *How to extract the rows where `checked` is `TRUE`?*
- *How to extract a subset comprised only of `ID` and `x`-something columns?*
- *How to extract the rows for which `ID` is like 3 letters and then 5 digits (e.g., `XYZ12345`)?*
- *How to select all the numeric columns in one go?*

[13] For example, drawing a scatter plot of one billion points barely makes sense and may result in unreadable images of large file sizes. They need to be sampled or summarised (e.g., binned) somehow first.

- *Assuming that the `ID`s are like three letters and then five digits, how to add two columns: `ID3` (the letters) and `ID5` (the five digits).*
- *How to get rid of all the columns between `x3` and `y7`?*
- *How to check where both `lat` and `long` in `coords` are positive?*
- *How to add the row indicating the number of `features`?*
- *How to extract the rows where `"spam"` is amongst the `features`?*
- *How to convert it to a long data frame with two columns: `ID` and `feature` (individual strings)?*
- *How to change the name of the `ID` column to `id`?*
- *How to make the `y`-foo columns appear before the `x`-bar ones?*
- *How to order the rows with respect to `checked` (`FALSE` first, then `TRUE`) and `ID`s (decreasingly)?*
- *How to remove rows with duplicate IDs?*
- *How to determine how many entries correspond to each `category`?*
- *How to compute the average `lat` and `long` in each `category`?*
- *How to compute the average `lat` and `long` for each `category` and `checked` combined?*

Exercise 12.39 *Consider the `flights`[14] dataset. Give some ways to select all rows between March and October (regardless of the year).*

Exercise 12.40 *In this task, you will be working with a version of a dataset on 70k+ Melbourne trees (`urban_forest`[15]). Before proceeding any further, read the dataset's description available here[16].*

1. *Load the downloaded dataset by calling the **`read.csv`** function.*
2. *Fetch the IDs (`CoM.ID`) and trunk diameters (`Diameter.Breast.Height`) of five horse chestnuts with the smallest diameters at breast height. The output data frame must be sorted with respect to `Diameter.Breast.Height`, decreasingly.*
3. *Create a new data frame that gives the number of trees planted in each year.*
4. *Compute the average age (in years, based on `Year.Planted`; using **`aggregate`**) of the trees of genera (each genus separately): Eucalyptus, Platanus, Ficus, Acer, and Quercus. Depict the sorted data with **`barplot`**.*

Exercise 12.41 *(*) Consider the historic data dumps of https://travel.stackexchange.com/ available at https://github.com/gagolews/teaching-data/tree/master/travel_stackexchange_com_2017.*

[14] https://github.com/gagolews/teaching-data/blob/master/other/flights.csv
[15] https://github.com/gagolews/teaching-data/raw/master/marek/urban_forest.csv.gz
[16] https://data.melbourne.vic.gov.au/Environment/Trees-with-species-and-dimensions-Urban-Forest-/fp38-wiyy

Export the CSV files located therein to an SQLite database. Then, write some R code that correspond to the following SQL queries (use ***dbGetQuery*** *to verify your results):*

```
--- 1)
SELECT
    Users.DisplayName,
    Users.Age,
    Users.Location,
    SUM(Posts.FavoriteCount) AS FavoriteTotal,
    Posts.Title AS MostFavoriteQuestion,
    MAX(Posts.FavoriteCount) AS MostFavoriteQuestionLikes
FROM Posts
JOIN Users ON Users.Id=Posts.OwnerUserId
WHERE Posts.PostTypeId=1
GROUP BY OwnerUserId
ORDER BY FavoriteTotal DESC
LIMIT 10
--- 2)
SELECT
    Posts.ID,
    Posts.Title,
    Posts2.PositiveAnswerCount
FROM Posts
JOIN (
        SELECT
            Posts.ParentID,
            COUNT(*) AS PositiveAnswerCount
        FROM Posts
        WHERE Posts.PostTypeID=2 AND Posts.Score>0
        GROUP BY Posts.ParentID
    ) AS Posts2
    ON Posts.ID=Posts2.ParentID
ORDER BY Posts2.PositiveAnswerCount DESC
LIMIT 10
--- 3)
SELECT
    Posts.Title,
    UpVotesPerYear.Year,
    MAX(UpVotesPerYear.Count) AS Count
FROM (
        SELECT
            PostId,
            COUNT(*) AS Count,
            STRFTIME('%Y', Votes.CreationDate) AS Year
        FROM Votes
        WHERE VoteTypeId=2
```

(continues on next page)

(continued from previous page)

```
        GROUP BY PostId, Year
    ) AS UpVotesPerYear
JOIN Posts ON Posts.Id=UpVotesPerYear.PostId
WHERE Posts.PostTypeId=1
GROUP BY Year
--- 4)
SELECT
    Questions.Id,
    Questions.Title,
    BestAnswers.MaxScore,
    Posts.Score AS AcceptedScore,
    BestAnswers.MaxScore-Posts.Score AS Difference
FROM (
        SELECT Id, ParentId, MAX(Score) AS MaxScore
        FROM Posts
        WHERE PostTypeId==2
        GROUP BY ParentId
    ) AS BestAnswers
JOIN (
        SELECT * FROM Posts
        WHERE PostTypeId==1
    ) AS Questions
    ON Questions.Id=BestAnswers.ParentId
JOIN Posts ON Questions.AcceptedAnswerId=Posts.Id
WHERE Difference>50
ORDER BY Difference DESC
--- 5)
SELECT
    Posts.Title,
    CmtTotScr.CommentsTotalScore
FROM (
        SELECT
            PostID,
            UserID,
            SUM(Score) AS CommentsTotalScore
        FROM Comments
        GROUP BY PostID, UserID
) AS CmtTotScr
JOIN Posts ON Posts.ID=CmtTotScr.PostID
    AND Posts.OwnerUserId=CmtTotScr.UserID
WHERE Posts.PostTypeId=1
ORDER BY CmtTotScr.CommentsTotalScore DESC
LIMIT 10
--- 6)
```

(continues on next page)

(continued from previous page)

```
SELECT DISTINCT
    Users.Id,
    Users.DisplayName,
    Users.Reputation,
    Users.Age,
    Users.Location
FROM (
        SELECT
            Name, UserID
        FROM Badges
        WHERE Name IN (
            SELECT
                Name
            FROM Badges
            WHERE Class=1
            GROUP BY Name
            HAVING COUNT(*) BETWEEN 2 AND 10
        )
        AND Class=1
    ) AS ValuableBadges
JOIN Users ON ValuableBadges.UserId=Users.Id
--- 7)
SELECT
    Posts.Title,
    VotesByAge2.OldVotes
FROM Posts
JOIN (
    SELECT
        PostId,
        MAX(CASE WHEN VoteDate = 'new' THEN Total ELSE 0 END) NewVotes,
        MAX(CASE WHEN VoteDate = 'old' THEN Total ELSE 0 END) OldVotes,
        SUM(Total) AS Votes
    FROM (
        SELECT
            PostId,
            CASE STRFTIME('%Y', CreationDate)
                WHEN '2017' THEN 'new'
                WHEN '2016' THEN 'new'
                ELSE 'old'
                END VoteDate,
            COUNT(*) AS Total
        FROM Votes
        WHERE VoteTypeId=2
        GROUP BY PostId, VoteDate
```

(continues on next page)

(continued from previous page)

```
    ) AS VotesByAge
    GROUP BY VotesByAge.PostId
    HAVING NewVotes=0
) AS VotesByAge2 ON VotesByAge2.PostId=Posts.ID
WHERE Posts.PostTypeId=1
ORDER BY VotesByAge2.OldVotes DESC
LIMIT 10
```

Exercise 12.42 *(*) Generate a CSV file featuring some random data arranged in a few columns of the size at least two times larger than your available RAM. Then, export the CSV file to an SQLite database. Use file connections (Section 8.3.5) and the* `nrow` *argument to* ***`read.table`*** *to be able to process it on a chunk-by-chunk basis.*

Determine whether setting `colClasses` *in* ***`read.table`*** *speeds up the reading of large CSV files significantly or not.*

Exercise 12.43 *(*) Export the whole XML data dump of StackOverflow*[17] *published at https://archive.org/details/stackexchange (see also https://data.stackexchange.com/) to an SQLite database.*

Now the second part of our course is ended.

---

[17] https://stackoverflow.com

Part III

Deepest

13

⨯ *Graphics*

The R Project homepage[1] advertises our free software as an *environment for statistical computing and graphics*. Hence, had we not dealt with the latter use case, our course would have been incomplete.

R is equipped with two independent systems for graphics generation.

1. The (historically) newer one, **`grid`**, is quite complicated. Some readers might have come across the **`lattice`** and **`ggplot2`** packages before: they are built on top of **`grid`**.
2. On the other hand, its traditional (S-style) counterpart, **`graphics`**, is much easier to master. Still, it gives their users full control over the drawing process. Its being both simple, fast, and low-level makes it very attractive from the perspective of this course's philosophy.

This is why, in this chapter, we will only cover the second approach. Note that *all* figures in this book were generated using **`graphics`** and its dependants. They are sufficiently aesthetic, aren't they?

⨯ **This chapter is under construction. Please come back later.**

13.1 ⨯ Placeholders for Plots Referred to Elsewhere

⨯ Plotting and factors; see Figure 13.1.

```
plot(iris[["Sepal.Length"]],  # x (it is a vector)
     iris[["Petal.Width"]],   # y (it is a vector)
     col=as.numeric(iris[["Species"]]),  # colours
     pch=as.numeric(iris[["Species"]])
)
```

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

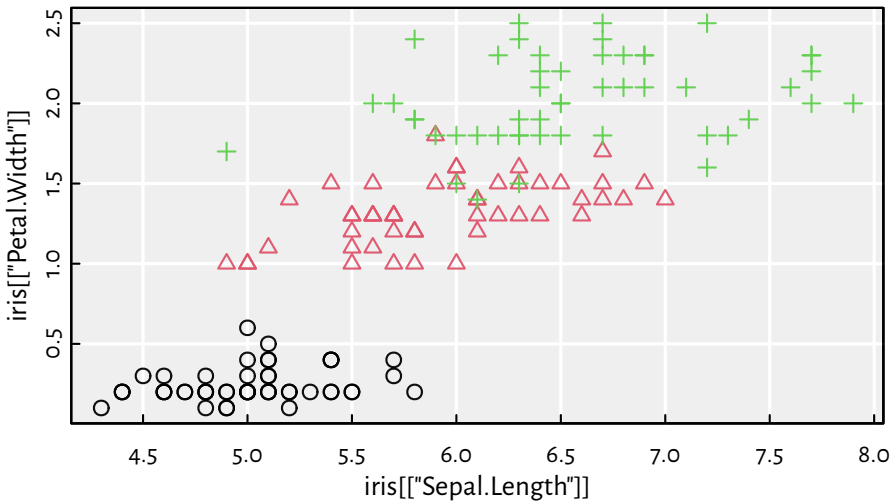

Figure 13.1: **`as.numeric`** on factors can be used to create different plotting styles

14

⨯⨯ *Interfacing Compiled Code (*)*

R is a nice *glue* language: it is perfect for implementing data wrangling pipelines, visualisation, and developing prototypes of data analysis algorithms. In other words, it makes connecting larger *building blocks* very easy. Still, the more computing-intensive tasks should be done at the C, C++, or Fortran level.

⨯ **This chapter is under construction. Please come back later.**

14.1 ⨯ R/C API

14.2 ⨯ External Pointers

14.3 ⨯ RCpp

14.4 ⨯ Memory Management

See `help("Memory")`, `help("Memory-limits")`

gc

15

⨯ *Unevaluated Expressions (**)*

In this and the next chapter, we will learn some hocus-pocus that should only be of interest to advanced-to-be[1] and open-minded R programmers who would really like to understand what is going on under our language's hood. In particular, we will inspect the mechanisms behind why certain functions do something very different from what we would expect them to do, if a *standard* evaluation scheme was followed (compare **`subset`** and **`transform`** mentioned in Section 12.3.9).

Namely, in *normal* programming languages, when we write something like:

```
plot(x, exp(x))
```

the expression **`exp`**`(x)`, is evaluated *first* and its value[2] (in this case: a numeric vector) is only then passed to the **`plot`** function as the actual parameter. Thus, if `x` was set to be **`seq`**`(0, 10, length.out=1001)`, the above never means anything else than:

```
plot(c(0.00, 0.01, 0.02, 0.03, ...), c(1.0000, 1.0101, 1.0202, 1.0305, ...))
```

But R was heavily inspired by a Lisp language dialect called Scheme[3], from whom it has inherited a quite disturbing ability to apply a set of techniques referred to as *meta-programming* (computing on the language). Namely, we may define functions that can peek outside their small world and clearly see the code used to generate the arguments passed thereto. Having access to such *unevaluated expressions*, we can do to them whatever we please: print, modify, subset, re-interpret, evaluate on different data, or ignore whatsoever.

In theory, this enables the implementing of many *potentially helpful* beginner-friendly features, which allow us to express certain requests in a more concise manner. For instance, that the y-axis labels in Figure 2.2 could be generated automatically is exactly due to the fact that **`plot`** was able to see not only a vector like **`c`**`(1.0000, 1.0101, 1.0202, 1.0305, ...)`, but also the expression that generated it, **`exp`**`(x)`.

It also can, and did, cause chaos, confusion, and division.

[1] Remember that this book is supposed to be read from the beginning to the end. Also, if you have not tested yourself against all the 250-odd exercises suggested so far, please do it before proceeding with the material presented here. Only practice makes perfect, and nothing is built in a day. Give yourself time: you can always come back later.

[2] Or a reference/pointer to an object that stores the said value, whatever.

[3] That is why everyone *serious* (not exactly in a good way) about R programming should add the *Structure and Interpretation of Computer Programs* [1] to their reading list. Also note that R is not the only environment that marries statistics and Lisp-like languages; see also LISP-STAT [40].

In the current author's opinion, R (as a whole, in the sense of *R as a Language and an Environment*) would be better-off if metaprogramming techniques were not exposed to an ordinary programmer[4]. A healthy R user (also yours truly 99% of the time) can perfectly do without and thus refrain from using them. The fact that we call them "advanced" will not make us "cool" if we start horsing around with nonstandard evaluation. "Perverse" is perhaps a better label.

Cursed be us, as we are about to start eating from tree of the knowledge of good and evil. But remember: with great power comes great responsibility.

15.1 Expressions at a Glance

At the most general level, expressions in a language like R can be classified into two groups:

- *simple expressions*:
  - *constants* (e.g., `3.14`, `2i`, `NA_real_`, `TRUE`, `"character string"`),
  - *names* (symbols, identifiers),
- *compound expressions*: combinations of $n + 1$ expressions (simple or compound) of the form $(f, e_1, e_2, \ldots, e_n)$.

As we will soon see, compound expressions are used to represent a *call* to f (an *operator*) on a sequence of arguments $e_1, e_2, \ldots, e_n$ (*operands*). This is why we will also be denoting them simply with $f(e_1, e_2, \ldots, e_n)$.

On the other hand, *names* such as `x`, `iris`, `sum`, and `spam`, have no meaning without an explicitly provided context, which will be a topic that we explore in `sec:to-do`. Prior to that, we treat them as meaning-less.

Hence, for the time being, we are now only interested in the syntax or grammar of our language, not the semantics. We are abstract in the sense that in the expression "`mean(rates)+2`"[5] neither `` `mean` ``, `` `x` ``, nor even `` `+` `` have the "usual" sense. We should therefore treat them as equivalent to, say, `f(g(x), 2)` or `spam(bacon(spanish_inquisition), 2)`.

15.2 Language Objects

There are three types of *language objects* in R:

[4] So yes, no formulas; but possibly retaining the ability to postpone argument evaluation possibly forever; see Section 16.4.2.

[5] Which we know that we can equivalently express as "`` `+`(mean(rates), 2) ``"; see Section 9.4.5.

- `name` (`symbol`) – stores object names in the sense of "simple expressions: names" in Section 15.1;
- `call` – represents unevaluated function calls in the sense of "compound expressions" above;
- `expression` – quite confusingly, represents a *sequence* of simple or compound expressions (constants, names, or calls).

One way to create a simple or compound expression is is by *quoting*, where we ask R to refrain itself from evaluating a given command:

```
quote(spam)  # name (symbol)
## spam
quote(f(x))  # call
## f(x)
quote(1+2+3*pi)  # another call
## 1 + 2 + 3 * pi
```

Note that none of the above was executed.

Single strings can be converted to names by calling:

```
as.name("spam")
## spam
```

And calls can be built programmatically by invoking:

```
call("sin", pi/2)
## sin(1.5707963267949)
```

Sometimes we might rather wish to quote the arguments passed:

```
call("sin", quote(pi/2))
## sin(pi/2)
call("c", 1, exp(1), quote(exp(1)), pi, quote(pi))
## c(1, 2.71828182845905, exp(1), 3.14159265358979, pi)
```

Objects of type `expression` can be thought of as lists of simple or compound expressions.

```
(exprs <- expression(1, spam, mean(x)+2))
## expression(1, spam, mean(x) + 2)
```

Note that all the arguments were quoted.

We can access the individual components using the index or extraction operators:

```
exprs[-1]
## expression(spam, mean(x) + 2)
exprs[[3]]
## mean(x) + 2
```

Exercise 15.1 *Check the type of the object returned by a call to* ***c****(1, "two",* ***sd****,* ***list****(3, 4:5),* ***expression****(3+3)).*

Note Calling **class** on the aforementioned language objects yields `name`, `call`, and `expression`, whereas **typeof** returns `symbol`, `language`, and `expression`, respectively.

There is also an option to *parse* a given text fragment or a whole R script:

```
parse(text="mean(x)+2")
## expression(mean(x) + 2)
parse(text="  # two code lines (a comment to be ignored by the parser)
    x <- runif(5, -1, 1)
    print(mean(x)+2)
")
## expression(x <- runif(5, -1, 1), print(mean(x) + 2))
parse(text="2+")  # syntax error - unfinished business
## Error in parse(text = "2+"): <text>:2:0: unexpected end of input
## 1: 2+
##    ^
```

Important **deparse** can be used to convert language objects to character vectors. For instance:

```
deparse(quote(mean(x+2)))
## [1] "mean(x + 2)"
```

This function has the nice side effect of tidying up the code formatting:

```
exprs <- parse(text=
    "`+`(x, 2)->y; if(y>0) print(y**10|>log()) else { y<--y; print(y)}")
for (e in exprs)
    cat(deparse(e), sep="\n", end="\n")
## y <- x + 2
##
##
## if (y > 0) print(log(y^10)) else {
##     y <- -y
##     print(y)
## }
```

15.3 Calls as Combinations of Expressions

We have mentioned that calls (compound expressions) are combinations of simple or compound expressions of the form $(f, e_1, \ldots, e_n)$.

That the first expression on the list, denoted above with f, plays a special role, is exactly seen in the following examples:

```
as.call(expression(f, x))
## f(x)
as.call(expression(`+`, 1, x))
## 1 + x
as.call(expression(`while`, i < 10, i <- i + 1))
## while (i < 10) i <- i + 1
as.call(expression(function(x) x**2, log(exp(1))))
## (function(x) x^2)(log(exp(1)))
as.call(expression(1, x, y, z))  # utter nonsense, but syntactically valid
## 1(x, y, z)
```

Recall from Section 9.4 that operators and language constructs such as `if` and `while` are ordinary functions.

Furthermore:

```
expr <- quote(f(1+2, a=1, b=2))
length(expr)
## [1] 4
names(expr)  # NULL if no arguments are named
## [1] ""  ""  "a" "b"
```

15.3.1 Browsing Parse Trees

We can access the individual expressions constituting an object of type `call` using square brackets. For example,

```
expr <- quote(1+x)
expr[[1]]
## `+`
expr[2:3]
## 1(x)
```

A compound expression was defined recursively: it can consist of other compound expressions.

For instance, the following expression:

```
expr <- quote(
    while (i < 10) {
        cat("i =", i, "\n")
        i <- i+1
    }
)
```

can be rewritten using the $f(\ldots)$ notation like:

```
`while`(`<`(i, 10), `{`( cat("i =", i, "\n"), `<-`(i, `+`(i, 1))))
```

Equivalently, in the Polish (prefix; $(f, \ldots)$; traditionally used in Lisp) notation it will look like:

```
(
    `while`,
    (`<`, i, 10),
    (
        `{`,
        (cat, "i =", i, "\n"),
        (
            `<-`,
            i,
            (`+`, i, 1)
        )
    )
)
```

Thus, for example, we can dig into the sub-expressions using a series of extractions:

```
expr[[2]][[1]]  # or expr[[c(2, 1)]]
## `<`
expr[[3]][[2]][[4]]  # or expr[[c(3, 2, 4)]]
## [1] "\n"
```

Example 15.2 *We can even write a recursive function to traverse the whole parse tree:*

```
recapply <- function(expr)
{
    if (is.call(expr)) lapply(expr, recapply)
    else expr
}

str(recapply(expr))
## List of 3
```

(continues on next page)

(continued from previous page)

```
##  $ : symbol while
##  $ :List of 3
##   ..$ : symbol <
##   ..$ : symbol i
##   ..$ : num 10
##  $ :List of 3
##   ..$ : symbol {
##   ..$ :List of 4
##   .. ..$ : symbol cat
##   .. ..$ : chr "i ="
##   .. ..$ : symbol i
##   .. ..$ : chr "\n"
##   ..$ :List of 3
##   .. ..$ : symbol <-
##   .. ..$ : symbol i
##   .. ..$ :List of 3
##   .. .. ..$ : symbol +
##   .. .. ..$ : symbol i
##   .. .. ..$ : num 1
```

15.3.2 Manipulating Calls

The R language is *homoiconic*: it can treat code as data. This includes the ability to arbitrarily manipulate it on the fly.

Just like on lists: we can freely use the replacement versions of `` `[` `` and `` `[[` ``.

```
expr[[2]][[1]] <- as.name("<=")
expr[[3]] <- quote(i <- i + 2)
print(expr)
## while (i <= 10) i <- i + 2
```

We are only limited by our imagination.

15.4 Inspecting Function Definitions and Arguments Thereto

15.4.1 Getting Formal Arguments and Body

Consider the following function:

```
test <- function(x, y=1)
    x+y  # whatever
```

We know from the first part of this book that calling **`print`** on the above will reveal its source code.

It turns out that we can easily get access to the list of parameters in the form of a named list[6]:

```
formals(test)
## $x
##
##
## $y
## [1] 1
```

Note that the expressions generating the values of the default arguments (compare Section 16.4.1) are stored as ordinary list elements.

Furthermore, we can get access to its body:

```
body(test)
## x + y
```

It is an object of the now-well-known class `call`.

Thus, we can manipulate it arbitrarily:

```
body(test)[[1]] <- as.name("*")  # change from `+` to `*`
body(test) <- as.call(list(as.name("{"), quote(cat("spam")), body(test)))
test
## function (x, y = 1)
## {
##     cat("spam")
##     x * y
## }
```

15.4.2 Getting the Expression Passed as an Argument

A call to **`substitute`** allows us to reveal the expression used to generate a function's argument:

```
test <- function(x) substitute(x)

test(1)
## [1] 1
test(2+spam)
```

(continues on next page)

[6] Actually, a special internal datatype called `pairlist` which is rarely seen in R; see [50] and [47] for information how to deal with them at the C level. From this course's perspective, seeing pairlists as named lists is perfectly fine.

(continued from previous page)

```
## 2 + spam
test(test(test(!!7)))
## test(test(!!7))
test()  # it is not an error
```

In Section 16.4.2 we note that arguments are evaluated only on demand – **substitute** does not trigger that. Therefore, we are able to write functions that accept gobbledegook (as long as it is syntactically correct) and programmatically reinterpret it in whichever way we like. Please, do not do that. Have mercy on other R users.

Exercise 15.3 *It is quite common to see a call like* ***deparse(substitute(arg))****. in many R functions. Study the source code of* ***plot.default***, ***hist.default***, ***prop.test***, *and* ***wilcox.test.default****. Explain why they do that. Propose a solution to to achieve the same functionality without the use of reflection techniques.*

15.4.3 Checking if an Argument is Missing

There is an easy way to check whether an argument was provided at all:

```
test <- function(x) missing(x)

test(1)
## [1] FALSE
test()
## [1] TRUE
```

Exercise 15.4 *Study the source code of* ***sample***, ***seq.default***, ***plot.default***, ***matplot***, *and* ***t.test.default****. Determine the role of a call to* ***missing****. Would introducing a default argument* `NULL` *and testing its value with* ***is.null*** *be a good alternative?*

15.4.4 Determining How a Function was Called

Even though this somewhat already touches the topic of the environment model of evaluation that we discuss in the next chapter, it is worth knowing that **sys.call** can take a look at the call stack and determine how the current function was invoked.

Moreover, **match.call** takes a step further: it returns a call with argument names matched to the list of a function's formal parameters.

For instance:

```
test <- function(x, y, ..., a="yes", b="no")
{
    print(sys.call())  # sys.call(0)
    print(match.call())
}
```

(continues on next page)

(continued from previous page)

```
x <- "maybe"
test("spam", "bacon", "eggs", u = "ham"<"jam", b=x)
## test("spam", "bacon", "eggs", u = "ham" < "jam", b = x)
## test(x = "spam", y = "bacon", "eggs", u = "ham" < "jam", b = x)
```

Another example, where we see that we can access the call stack much more deeply:

```
f <- function(x)
{
    g <- function(y)
    {
        cat("g:\n")
        print(sys.call(0))
        print(sys.call(-1))  # go back one frame
        y
    }

    cat("f:\n")
    print(sys.call(0))
    g(x+1)
}

f(1)
## f:
## f(1)
## g:
## g(x+1)
## f(1)
## [1] 2
```

Exercise 15.5 *A function can*[7] *see how it has been defined by its maker. Call* ***`sys.function`*** *inside its body to reveal that.*

Exercise 15.6 *Call* ***`match.call(sys.function(-1), sys.call(-1))`*** *in the* ***`g`*** *function above.*

15.5 Exercises

Exercise 15.7 *Answer the following questions:*

- *What is a simple expression? What is a compound expressions? Give a few examples.*

[7] Therefore, it is possible to write a function that returns a modified version of itself.

- *What is the difference between a `call` and an `expression` object?*
- *What does **`formals`** and **`body`** return when fed with a function object?*
- *How to test if an argument to a function was given at all? Provide a use case for such a verification.*
- *Give two ways to create an unevaluated expression by quoting.*
- *What is the purpose of **`deparse(substitute(...))`**? Give a few examples of functions that use this technique.*
- *What is the difference between **`sys.call`** and **`match.call`**?*

Exercise 15.8 *Write a function that takes a dot-dot-dot argument (Section 9.5.6). Using **`match.call`** (amongst others), determine a list of all the expressions passed via `` `...` ``. Note that some of them might be named (just like in one of the above examples).*

Exercise 15.9 *Write a function **`check_if_calls(f, fun_list)`** that takes another function `f` on input and check if it calls any of the functions (by name) from a character vector `fun_list`.*

16

⨳ *Environments and Evaluation (**)*

In the first part of our book, we have discussed quite a few *basic* object types: numeric, logical, and character vectors, lists (generic vectors), and functions.

Environments, which we introduce in this chapter, just like lists, are instances of recursive types (compare the diagram in Figure 16.1).

Even though we rarely interact with them directly (unless we need a hash table-like data structure with quick by-name element look-up), they are crucial for R itself: they form the basis of the *environment model of evaluation* which governs how expressions are computed; see Section 16.2.

Important Each object of type *environment* consists of:

- a *frame*[1] (Section 16.1) – stores a set of *bindings*, which associate variable names with their corresponding values; it can be thought of as a container of named R objects of any type;
- a pointer to an *enclosing environment*[2] (Section 16.3).

⨳ **This chapter is under construction. Please come back later.**

16.1 Frames: Environments as Object Containers

To create a new, empty environment, we can call the **`new.env`** function:

[1] Not to be confused with a "data frame", i.e., an object of S3 class `data.frame`; see Chapter 12.

[2] Some also call it a *parent* environment, but we will not. We will try following the nomenclature established in Section 3.2 in [1]. Note that there is a bit of a mess in the R documentation regarding the way enclosing environments are referred to as.

```
e1 <- new.env()
typeof(e1)
## [1] "environment"
```

In this section, we treat environments merely as containers for named objects of any kind, i.e., we deal with the *frame* part thereof.

Let us insert some elements into `e1`:

```
e1[["x"]] <- "x in e1"
e1[["y"]] <- 1:3
```

The `` `[[` `` operator provides some named list-like look-and-feel also in the case of element extraction:

```
e1[["x"]]
## [1] "x in e1"
e1[["spam"]]  # does not exist
## NULL
e1[["y"]] <- e1[["y"]]*10  # replace with new content
e1[["z"]] <- NULL  # unlike in the case of lists, creates a new element
```

16.1.1 Printing

Let us note that the printing of an environment is quite awkward:

```
print(e1)  # same with str(e1)
## <environment: 0x557639a86e58>
```

Later we will mention that this is the address where `e1` is stored in computer's memory.

As we have said, these objects are rather of *internal* interest, so nobody cared[3] about making the interaction therewith particularly smooth.

However, we can easily get the list of objects stored within the container using **`names`**[4]:

```
names(e1)  # compare attr(e1, "names") - it is not set
## [1] "x" "y" "z"
```

Also:

```
length(e1)
## [1] 3
```

gives the number of objects in the container.

[3] Which might be by design, to discourage the novices from playing with fire.

[4] Even though the `names` attribute is not set explicitly, the corresponding function returns a sensible result.

16.1.2 Environments vs Named Lists

Environment frames, in some sense, can be thought of as named lists, but the set of admissible operations is severely restricted. In particular, we cannot extract more than one element at the same time with the index operator:

```
e1[c("x", "y")]  # but see the mget function
## Error in e1[c("x", "y")]: object of type 'environment' is not subsettable
```

nor can we refer to the elements by position:

```
e1[[1]] <- "bad key"
## Error in e1[[1]] <- "bad key": wrong args for environment subassignment
```

Exercise 16.1 *Check if* ***`lapply`*** *and* ***`Map`*** *can be applied directly on environments. Also, can we iterate over their elements using a* ***`for`*** *loop?*

Actually, named lists can be converted to environments and vice versa using **`as.list`** and **`as.environment`**.

```
as.list(e1)
## $x
## [1] "x in e1"
##
## $y
## [1] 10 20 30
##
## $z
## NULL
as.environment(list(u=42, whatever="it's not gonna be printed anyway"))
## <environment: 0x557639ad5fd8>
as.list(as.environment(list(x=1, y=2, x=3)))  # no duplicates allowed
## $y
## [1] 2
##
## $x
## [1] 3
```

16.1.3 Hash Maps: Fast Element Look-up by Name

Environment frames are internally implemented using hash tables (hash maps; see, e.g., [9, 33]) with string keys.

Important A *hash table* is a data structure that allows for a very quick[5] lookup and insertion of individual elements *by name*.

This comes at a price, including what we have already observed above:

- the elements are not particularly unordered: they cannot be referred to by a numeric index;
- all element names must be unique.

Note A list may be considered a *sequence*, but an environment frame is only *set* (a bag) of key–value pairs. In the majority of numerical computing applications, we would rather store, iterate over, and process all the elements *in order*, hence the greater popularity of the former. Lists still allow for an element look-up by name, even though this is slightly slower[6]. Hence, they are much more universal.

Example 16.2 *A natural use case of manually-created environment frames deals with grouping a series of objects identified by character string keys.*

Consider a simple pseudocode for counting the number of occurrences of objects in a given container:

```
for (key in some_container) {
    if (!is.null(counter[["key"]]))
        counter[["key"]] <- counter[["key"]]+1
    else
        counter[["key"]] <- 1
}
```

If `some_container` *is large, say, of size* n *(let us say it is generated on the fly by reading some data stream), then the run-time of the above will depend on the type of the data structure used. If* `counter` *is a list, then, theoretically, the worst-case performance will be* $O(n^2)$ *(if all keys are unique). However, for environments, it is an order of magnitude faster: down to amortised* $O(n)$*.*

Exercise 16.3 *(*) Implement the above pseudocode and benchmark the two data structures using* **`proc.time`** *on some example data:*

```
t0 <- proc.time()  # timer start
# ... to do - something time-consuming ...
print(proc.time() - t0)  # elapsed time
```

Exercise 16.4 *(*) Determine the number of unique text lines in a very large file (assuming that*

[5] Element lookup, insertion, and deletion in hash tables takes amortised $O(1)$ time.

[6] Accessing elements by position (numeric index) in lists takes $O(1)$ time. Worst-case scenario for the element look-up by name (non-existence) is linear with respect to the container size. Also, inserting new elements at the end takes amortised $O(1)$ time.

the set of unique text lines fits into memory, but the file itself does not). Also, determine the five most frequently occurring text lines.

16.1.4 Pass-by-Value, Copy on Demand – Not for Environments

Given any R object, say, `x`, when we issue:

```
y <- x
```

its copy[7] is made so that `y` and `x` are independent of each other. In other words, any change in the state of `x` (or `y`) is not reflected in the state of `y` (or `x`).

For instance:

```
x <- list(a=1)
y <- x
y[["a"]] <- y[["a"]]+1
print(y)
## $a
## [1] 2
print(x)  # not affected - `x` and `y` are independent
## $a
## [1] 1
```

The same happens with arguments that we feed to the functions:

```
mod <- function(y, key)  # it's like: local_y <- passed_argument
{
    y[[key]] <- y[[key]]+1
    y
}

mod(x, "a")  # returns a modified copy of `x`
## $a
## [1] 2
print(x)
## $a
## [1] 1
```

We thus say that R has *pass-by-value* semantics.

Important Environments are the only[8] R objects that have an assign- and pass-by-reference semantics.

---

[7] Delayed (on demand); see below.

[8] Tricks that we can do at the C language level do not count (`Chapter %s`).

In other words, if we perform:

```
x <- as.environment(x)
y <- x
```

then the names x and y are bound with exactly the same objects in computer's memory.

```
y[["a"]] <- y[["a"]]+1
print(y[["a"]])
## [1] 2
print(x[["a"]])  # `x` is `y`, `y` is `x`
## [1] 2
```

This is exactly seen when we print the address where the environments are located:

```
print(x)
## <environment: 0x557638e9cac0>
print(y)
## <environment: 0x557638e9cac0>
```

The same when we pass them to a function:

```
mod(y, "a")  # pass-by-reference (`y` is `x`, remember?)
## <environment: 0x557638e9cac0>
x[["a"]]   # `x` has changed (names `y` and `x` are bound to the same object)
## [1] 3
```

Thus, any changes we make to an environment passed as an argument to a function will be visible "outside" the call. This minimises time and memory use in certain situations. If this sounds complicated, we should rather be avoiding these data structures whatsoever. As we have said, an R user can live without them.

Note (*) Actually, for efficiency reasons, when we write "`y <- x`", a copy of `x` (unless it is an environment) is only created if absolutely necessary.

Here is some benchmarking of the *copy-on-demand* mechanism.

```
n <- 100000000  # like, a lot
```

Creation of a new large numeric vector:

```
t0 <- proc.time();          x <- numeric(n);              proc.time() - t0
##    user  system elapsed
##   0.135   0.175   0.309
```

Creation of a (delayed) copy:

```
t0 <- proc.time();          y <- x;                      proc.time() - t0
##    user  system elapsed
##       0       0       0
```

This was instant. Thus, we definitely did not clone the `n` data cells.

Copy-on-demand is implemented using some quite simple *reference counting*; compare Section 14.4. That – temporarily – `x` and `y` point to the same address in memory can be inspected by calling:

```
.Internal(inspect(x))
## @7f39ca5df010 14 REALSXP g0c7 [REF(4)] (len=100000000, tl=0) 0,0,0,0,0,...
.Internal(inspect(y))
## @7f39ca5df010 14 REALSXP g0c7 [REF(5)] (len=100000000, tl=0) 0,0,0,0,0,...
```

The real copying is only triggered when we try to modify `x` or `y`. This is when they need to be separated.

```
t0 <- proc.time();          y[1] <- 1;                   proc.time() - t0
##    user  system elapsed
##   0.182   0.187   0.369
```

Now `x` and `y` are different objects.

```
.Internal(inspect(x))
## @7f39ca5df010 14 REALSXP g0c7 [MARK,REF(5)] (len=100000000, tl=0) 0,0,0,0,0,...
.Internal(inspect(y))
## @7f399aaee010 14 REALSXP g0c7 [MARK,REF(1)] (len=100000000, tl=0) 1,0,0,0,0,...
```

Note that the elapsed time is similar to that needed to create `x` from scratch.

Further modifications can already be fast:

```
t0 <- proc.time();          y[2] <- 2;                   proc.time() - t0
##    user  system elapsed
##   0.163   0.239   0.402
```

16.1.5 ×A Note on Reference Classes (*)

16.2 ×The Environment Model of Evaluation

16.3 ×Enclosing Environments

16.4 ×Evaluating Functions

16.4.1 ×Evaluation of Default Arguments

16.4.2 ×Not All Arguments Need to Be Evaluated

Calls such as `` `if`(test, ifyes, ifno) `` or `` `&&`(mustbe, maybe) `` do not have to evaluate all their arguments.

```
{cat(" first "); FALSE} && {cat(" second "); FALSE}
##  first
## [1] FALSE
{cat(" first "); TRUE} && {cat(" Spanish Inquisition "); FALSE}
##  first  Spanish Inquisition
## [1] FALSE
```

This is also a kind of nonstandard evaluation.

We can write such functions too.

```
test <- function(a, b, c) a + c  # b is unused

test({cat(" spam "); 1}, {cat(" eggs "); 10}, {cat(" bacon "); 100})
##  spam  bacon
## [1] 101
```

Here, the second argument was not used, therefore it did not have to be evaluated. And it was not.

A more advanced example:

```
test <- function(a, b, c)
{
    cat("Arguments passed to the functions (expressions): \n")
    cat("a = ", deparse(substitute(a)), "\n")
    cat("b = ", deparse(substitute(b)), "\n")
    cat("c = ", deparse(substitute(c)), "\n")
    cat("Using a: ")
```

(continues on next page)

(continued from previous page)

```
    a  # we don't even have to be particularly creative
    cat("Using a and c: ")
    retval <- a + c  # a is reused, b is unused
    cat("Cheers! ")
    retval
}

test({cat(" spam "); 1}, {cat(" eggs "); MeAn(egGs)}, {cat(" bacon "); 100})
## Arguments passed to the functions (expressions):
## a =  {     cat(" spam ")     1 }
## b =  {     cat(" eggs ")     MeAn(egGs) }
## c =  {     cat(" bacon ")     100 }
## Using a:  spam Using a and c:  bacon Cheers!
## [1] 101
```

Notice that:

- either the evaluation of an argument does not happen or it is triggered only once (in which case the result is cached);
- evaluation is delayed until the very first request for the underlying value;
- regardless whether we request it or not, we still have access to the underlying unevaluated expression passed as an argument.

As a consequence, we can pass arbitrary gibberish as the second argument (and we did above).

In other words, the evaluation of arguments can be postponed arbitrarily, possibly forever.

16.4.3 ×Matching of Argument Names (TODO: MOVE)

16.4.4 ×Ellipsis Revisited

16.4.5 ×S3 Method Lookup by `UseMethod`

16.4.6 ×Overloading S3 Group Generics

16.4.7 ×Package Namespaces

16.5 ×Formulas, `~` (*)

16.6 ×Exercises

Exercise 16.5 *Answer the following questions:*

- *What is the role of a frame in an environment?*
- *What is the role of an enclosing environment?*
- *What is the difference between a named list and an environment?*
- *What functions and operators work on named lists but cannot be applied on environments?*
- *What do we mean by saying that environments are not passed by value to R functions?*
- *What do we mean by saying that sometimes objects are copied on demand?*
- ⨯ *TODO...*

Exercise 16.6 *(*) ⨯ TODO: assign consecutive, unique numeric IDs for a large list of items*

```
pkg <- available.packages()
pkg[, "Package"]  # a list of the names of available packages
pkg[, "Depends"]  # dependencies
```

Convert the dependency lists to a list of character vectors (preferably using a regular expression).

Then, generate a list of reverse dependencies:

What packages depend on each given package.

This will be much faster using an environment.

16.7 ⨯ Outro

This is the end.

Recall our first approximation to the classification of R Data Types that we presented in the *Preface*.

As a summary of what we have covered, Figure 16.1 gives a much broader picture.

Now that we have reached the end of this course, we might be interested in reading the following guides:

- *R Language Definition* [51],
- *R Internals* [50],
- *Writing R Extensions* [47].

Also, the NEWS files available at https://cran.r-project.org/doc/manuals/r-release/ will keep us up to date with new features, bug fixes, and deprecated functionality.

Good luck.

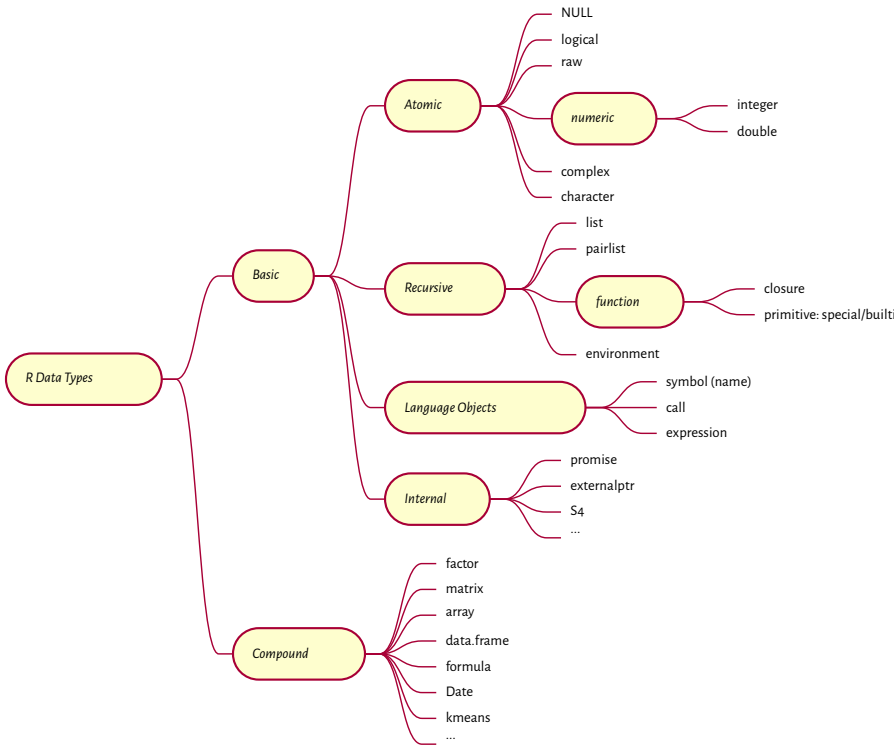

Figure 16.1: R data types

Changelog

Important This book is still a work in progress. The first twelve chapters are already quite readable, but there will be more. Stay tuned.

Any bug/typos reports/fixes[9] are appreciated.

Below is the list of the most noteworthy changes.

- **under development (v0.1.13.9xxx)**:
  - (...) to do (...) work in progress (...)
- **2023-01-15 (v0.1.13)**:
  - Alpha version of Chapter 15.
- **2022-12-29 (v0.1.12)**:
  - First public release at https://deepr.gagolewski.com.
  - Beta (complete) versions of Chapters 1–12 (basic and compound types, functions, etc.) published.
  - Preface drafted (alpha version).
  - ISBN 978-0-6455719-2-9 reserved.
  - Cover.

[9] https://github.com/gagolews/deepr/issues

References

[1] Abelson, H., Sussman, G.J., Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press.

[2] Abramowitz, M., Stegun, I.A. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover. URL: https://people.math.sfu.ca/~cbm/aands/.

[3] Becker, R.A., Chambers, J.M., Wilks, A.R. (1988). *The New S Language*. Chapman & Hall.

[4] Chambers, J.M. (1998). *Programming with Data. A Guide to the S Language*. Springer-Verlag.

[5] Chambers, J.M. (2008). *Software for Data Analysis. Programming with R*. Springer.

[6] Chambers, J.M. (2016). *Extending R*. Chapman & Hall.

[7] Chambers, J.M. (2020). S, R, and data science. *The R Journal*, 12(1):462–476. DOI: 10.32614/RJ-2020-028.

[8] Chambers, J.M., Hastie, T.J. (1991). *Statistical Models in S*. Chapman & Hall.

[9] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms*. MIT Press and McGraw-Hill.

[10] Crawley, M.J. (2007). *The R Book*. John Wiley & Sons.

[11] Date, C.J. (2003). *An Introduction to Database Systems*. Pearson.

[12] Davis, M., Whistler, K. (2021). Unicode standard annex #15: Unicode normalization forms. URL: http://www.unicode.org/reports/tr15/.

[13] Davis, M., Whistler, K., Scherer, M. (2021). Unicode technical standard #10: Unicode collation algorithm. URL: http://www.unicode.org/reports/tr10/.

[14] Deisenroth, M.P., Faisal, A.A., Ong, C.S. (2020). *Mathematics for Machine Learning*. Cambridge University Press. URL: https://mml-book.com/.

[15] DeMichiel, L.G., Gabriel, R.P. (1987). The Common Lisp Object System: An overview. *ECOOP*. URL: https://www.dreamsongs.com/Files/ECOOP.pdf.

[16] Devroye, L. (1986). *Non-Uniform Random Variate Generation*. Springer-Verlag. URL: http://luc.devroye.org/rnbookindex.html.

[17] Forbes, C., Evans, M., Hastings, N., Peacock, B. (2010). *Statistical Distributions*. Wiley.

[18] Friedl, J.E.F. (2006). *Mastering Regular Expressions*. O'Reilly.

[19] Gagolewski, M. (2016). *Programowanie w języku R. Analiza danych, obliczenia, symulacje (R Programming. Data Analysis, Computing, Simulations)*. Wydawnictwo Naukowe PWN, 2nd edition. in Polish (1st edition published in 2014).

[20] Gagolewski, M. (2022). *Minimalist Data Wrangling with Python*. Zenodo, Melbourne. URL: https://datawranglingpy.gagolewski.com/, DOI: 10.5281/zenodo.6451068.

[21] Gagolewski, M. (2022). stringi: Fast and portable character string processing in R. *Journal of Statistical Software*, 103(2):1–59. URL: https://stringi.gagolewski.com, DOI: 10.18637/jss.v103.i02.

[22] Gagolewski, M. (2023). *stringx: Drop-in replacements for base R string functions powered by stringi*. URL: https://stringx.gagolewski.com.

[23] Galassi, M., Theiler, J., et al. (2021). *GNU Scientific Library Reference Manual*. URL: http://www.gnu.org/software/gsl/.

[24] Gentle, J.E. (2003). *Random Number Generation and Monte Carlo methods*. Springer.

[25] Gentle, J.E. (2007). *Matrix Algebra*. Springer.

[26] Gentle, J.E. (2009). *Computational Statistics*. Springer.

[27] Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 21(1):5–48. URL: https://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf.

[28] Hankin, R.K.S. (2006). Special functions in R: Introducing the gsl package. *R News*, 6:24–26. URL: https://cran.r-project.org/web/packages/gsl/vignettes/gslpaper.pdf.

[29] Harris, C.R., et al. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362. DOI: 10.1038/s41586-020-2649-2.

[30] Higham, N.J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA. DOI: 10.1137/1.9780898718027.

[31] Ihaka, R., Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314. DOI: 10.1080/10618600.1996.10474713.

[32] Knuth, D.E. (1992). *Literate Programming*. CSLI.

[33] Knuth, D.E. (1997). *The Art of Computer Programming III: Sorting and Searching*. Addison-Wesley.

[34] Knuth, D.E. (1997). *The Art of Computer Programming II: Seminumerical Algorithms*. Addison-Wesley.

[35] Knuth, D.E. (1997). *The Art of Computer Programming I: Fundamental Algorithms*. Addison-Wesley.

[36] Matloff, N.S. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press.

[37] Matsumoto, M., Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30.

[38] Nelsen, R.B. (1999). *An Introduction to Copulas*. Springer-Verlag.

[39] Olver, F.W.J., et al. (2021). *NIST Digital Library of Mathematical Functions*. NIST. URL: https://dlmf.nist.gov/.

[40] Tierney, L. (1990). *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. Wiley.

[41] Tierney, L., Becker, G., Kalibera, T. (2018). *ALTREP: Alternative Representations for R Objects*. URL: https://svn.r-project.org/R/branches/ALTREP/ALTREP.html.

[42] Venables, W.N., Ripley, B.D. (2000). *S Programming*. Springer.

[43] Venables, W.N., Smith, D.M., R Development Core Team. (2023). *An Introduction to R*. URL: https://CRAN.R-project.org/doc/manuals/r-release/R-intro.html.

[44] Wickham, H. (2014). *Advanced R*. Chapman & Hall/CRC.

[45] Wickham, H., Grolemund, G. (2017). *R for Data Science*. O'Reilly. URL: https://r4ds.had.co.nz/.

[46] Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC.

[47] R Development Core Team. (2023). *Writing R Extensions*. URL: https://CRAN.R-project.org/doc/manuals/r-release/R-exts.html.

[48] R Development Core Team. (2023). *R Data Import/Export*. URL: https://CRAN.R-project.org/doc/manuals/r-release/R-data.html.

[49] R Development Core Team. (2023). *R Installation and Administration*. URL: https://CRAN.R-project.org/doc/manuals/r-release/R-admin.html.

[50] R Development Core Team. (2023). *R Internals*. URL: https://CRAN.R-project.org/doc/manuals/r-release/R-ints.html.

[51] R Development Core Team. (2023). *R Language Definition*. URL: https://CRAN.R-project.org/doc/manuals/r-release/R-lang.html.

[52] R Development Core Team. (2023). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL: http://www.R-project.org.