

Analyzing Financial and Economic Data with R - Online Edition

Marcelo S. Perlin (marcelo.perlin@ufrgs.br)

2022-11-23

Analyzing Financial and Economic Data with R

by **Marcelo Scherer Perlin**

© 2020 Marcelo S. Perlin. All rights reserved.

Independent publication. Printed on demand by Amazon.com.

Online edition with first six chapters available at:

<https://www.msperlin.com/afedR/>

Cover: Rubens Lima - <https://capista.com.br>

Proofreader: Various

ISBN (paperback): 9781710627312

ISBN (hardcover): 9798714799266

ISBN (ebook): -

History of editions:

2017-05-01 First edition

2020-02-15 Second edition

2021-03-15 Second edition revised

While the author has used good faith efforts to ensure that the instructions and code contained in this work are accurate, the author disclaims all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work and its resulting code. The use of the information contained in this work is at your own risk. If any code in this book is subject to open source licenses or the intellectual property rights of others, complying with such rights and licenses is your responsibility as a user.

Contents

About New Edition	11
Revision 2021	12
Preface	13
Supplement Material	15
Content for Instructors	16
1 Introduction	17
1.1 What is R	18
1.2 Why Choose R	18
1.3 What Can You Do With R and RStudio?	19
1.4 Installing R	21
1.5 Installing RStudio	24
1.6 Resources in the Web	25
1.7 Structure and Organization	25
1.8 Exercises	27
2 Basic Operations in R	31
2.1 Working With R	31
2.2 Objects in R	33
2.3 International and Local Formats	34
2.4 Types of Files in R	35
2.5 Explaining the RStudio Screen	36
2.6 R Packages	39
2.6.1 Installing Packages from CRAN	42
2.6.2 Installing Packages from Github	42
2.6.3 Loading Packages	43
2.6.4 Upgrading Packages	45

2.7	Running Scripts from RStudio	46
2.7.1	RStudio shortcuts	48
2.8	Testing and Debugging Code	49
2.9	Creating Simple Objects	51
2.10	Creating Vectors	52
2.11	Knowing Your Environment and Objects	53
2.12	Displaying and Formatting Output	55
2.12.1	Customizing the Output	57
2.13	Finding the Size of Objects	59
2.14	Selecting Elements from an Atomic Vector	62
2.15	Removing Objects from the Memory	65
2.16	Displaying and Setting the Working Directory	66
2.17	Canceling Code Execution	68
2.18	Code Comments	69
2.19	Looking for Help	71
2.20	Using Code Completion with <i>tab</i>	72
2.21	Interacting with Files and the Operating System	76
2.21.1	Listing Files and Folders	76
2.21.2	Deleting Files and Directories	79
2.21.3	Downloading Files from the Internet	80
2.21.4	Using Temporary Files and Directories	81
2.22	Exercises	82
3	Writing Research Scripts	85
3.1	Stages of Research	85
3.2	Folder Structure	87
3.3	Important Aspects of a Research Script	90
3.4	Exercises	92
4	Importing Data from Local Files	93
4.1	<i>csv</i> files	94
4.1.1	Importing Data	97
4.1.2	Exporting Data	101
4.2	<i>Excel</i> Files (<i>xls</i> and <i>xlsx</i>)	103
4.2.1	Importing Data	103
4.2.2	Exporting Data	104
4.3	<i>RData</i> and <i>rds</i> Files	107
4.3.1	Importing Data	108
4.3.2	Exporting Data	109
4.4	<i>fst</i> files	110
4.4.1	Importing Data	110
4.4.2	Exporting Data	110
4.4.3	Timing the <i>fst</i> format	111

4.5	SQLite Files	113
4.5.1	Importing Data	113
4.5.2	Exporting Data	114
4.6	Unstructured Data and Other Formats	116
4.6.1	Importing Data	116
4.6.2	Exporting Data	118
4.7	How to Select a Format	118
4.8	Exercises	119
5	Importing Data from the Internet	121
5.1	Package <code>GetQuandlData</code>	121
5.2	Package <code>BatchGetSymbols</code>	127
5.3	Package <code>simfinR</code>	133
5.3.1	Example 01 - Apple Inc Annual Profit	134
5.3.2	Example 02 - Quarterly Net Profit of Many Companies	137
5.3.3	Example 03 - Fetching price data	139
5.4	Package <code>tidyquant</code>	140
5.5	Package <code>Rbitcoin</code>	142
5.6	Other Packages	144
5.7	Accessing Data from Web Pages (<i>webscraping</i>)	144
5.7.1	Scraping the Components of the SP500 Index from Wikipedia	145
5.7.2	Scraping the Website of the Reserve Bank of Australia	147
5.8	Exercises	149
6	Dataframes and other objects	153
6.1	<code>Dataframes</code>	154
6.1.1	Creating <code>dataframes</code>	154
6.1.2	Inspecting a Dataframe	156
6.1.3	The <i>pipeline</i> Operator (<code>%>%</code>)	158
6.1.4	Accessing Columns	159
6.1.5	Modifying a <code>dataframe</code>	162
6.1.6	Filtering rows of a <code>dataframe</code>	164
6.1.7	Sorting a <code>dataframe</code>	165
6.1.8	Combining and Aggregating <code>dataframes</code>	168
6.1.9	Extensions of the <code>dataframe</code> Class	172
6.1.10	Other Useful Functions for Handling <code>dataframes</code>	175
6.2	<code>Lists</code>	177
6.2.1	Creating <code>lists</code>	177
6.2.2	Accessing the Elements of a <code>list</code>	179
6.2.3	Adding and Removing Elements from a <code>list</code>	182
6.2.4	Processing the Elements of a <code>list</code>	184
6.2.5	Other Useful Functions	185

6.3	Matrices	186
6.3.1	Selecting Elements from a <code>matrix</code>	190
6.3.2	Other Useful Functions	191
6.4	Exercises	193
7	Basic Object Classes	195
7.1	Numeric Objects	195
7.1.1	Creating and Manipulating <code>numeric</code> Objects	196
7.1.2	Creating a <code>numeric</code> Sequence	198
7.1.3	Creating Vectors with Repeated Elements	199
7.1.4	Creating Vectors with Random Numbers	199
7.1.5	Accessing the Elements of a <code>numeric</code> Vector	204
7.1.6	Modifying and Removing Elements of a <code>numeric</code> Vector	206
7.1.7	Creating Groups	207
7.1.8	Other Useful Functions	209
7.2	Character Objects	211
7.2.1	Creating a Simple <code>character</code> Object	211
7.2.2	Creating Structured <code>character</code> Objects	212
7.2.3	<code>character</code> Constants	213
7.2.4	Selecting Pieces of a Text Object	214
7.2.5	Finding and Replacing Characters of a Text	215
7.2.6	Splitting Text	218
7.2.7	Finding the Number of Characters in a Text	219
7.2.8	Generating Combinations of Text	220
7.2.9	Encoding of <code>character</code> Objects	221
7.2.10	Other Useful Functions	222
7.3	Factor Objects	223
7.3.1	Creating <code>factors</code>	223
7.3.2	Modifying <code>factors</code>	224
7.3.3	Converting <code>factors</code> to Other Classes	226
7.3.4	Creating Contingency Tables	227
7.3.5	Other Useful Functions	228
7.4	Logical Objects	229
7.4.1	Creating <code>logical</code> Objects	229
7.5	Date and Time	232
7.5.1	Creating Simple Dates	232
7.5.2	Creating a Sequence of Dates	234
7.5.3	Operations with Dates	236
7.5.4	Dealing with Time	238
7.5.5	Customizing the Format of Dates and Times	239
7.5.6	Extracting Elements of a Date	241
7.5.7	Find the Current Date and Time	242

7.5.8	Other Useful Functions	243
7.6	Missing Data - NA (<i>Not available</i>)	246
7.6.1	Defining NA Values	246
7.6.2	Finding and Replacing NA	247
7.6.3	Other Useful Functions	249
7.7	Exercises	249
8	Programming and Data Analysis	253
8.1	R Functions	253
8.2	Using <code>for</code> Loops	254
8.3	Conditional Statements (<code>if</code> , <code>else</code> , <code>switch</code>)	254
8.4	Using <code>apply</code> Functions	254
8.4.1	Using <code>lapply</code>	255
8.4.2	Using <code>sapply</code>	255
8.4.3	Using <code>tapply</code>	255
8.4.4	Using <code>mapply</code>	256
8.4.5	Using <code>apply</code>	256
8.4.6	Using <code>by</code>	256
8.5	Using package <code>purrr</code>	257
8.5.1	Function <code>map_*</code>	257
8.5.2	Function <code>safely</code>	257
8.5.3	Function <code>pmap</code>	258
8.6	Data Manipulation with Package <code>dplyr</code>	258
8.6.1	Group Operations with <code>dplyr</code>	258
8.6.2	Complex Group Operations with <code>dplyr</code>	259
8.7	Exercises	259
8.8	Exercises	259
9	Cleaning and Structuring Data	261
9.1	The Format of a <code>dataframe</code>	261
9.1.1	Converting a <code>dataframe</code> Structure (long and wide) . .	262
9.2	Converting lists into <code>dataframes</code>	262
9.3	Removing Outliers	262
9.3.1	Treating Outliers in <code>dataframes</code>	263
9.4	Inflation and Price Data	263
9.5	Modifying Time Frequency and Aggregating Data	263
9.6	Exercises	264
9.7	Exercises	264
10	Creating and Saving Figures with <code>ggplot2</code>	267
10.1	The <code>ggplot2</code> Package	267
10.2	Using Graphics Windows	268
10.3	Creating Figures with Function <code>qplot</code>	268

10.4 Creating Figures with Function <code>ggplot</code>	268
10.4.1 The US Yield Curve	269
10.5 Using Themes	269
10.6 Creating Panels with <code>facet_wrap</code>	269
10.7 Using the Pipeline	270
10.8 Creating Statistical Graphics	270
10.8.1 Creating Histograms	270
10.8.2 Creating <i>boxplot</i> Figures	271
10.8.3 Creating <i>QQ</i> Plots	271
10.9 Saving Graphics to a File	271
10.10 Exercises	272
10.11 Exercises	272
11 Financial Econometrics with R	275
11.1 Linear Models (OLS)	275
11.1.1 Simulating a Linear Model	276
11.1.2 Estimating a Linear Model	276
11.1.3 Statistical Inference in Linear Models	276
11.2 Generalized Linear Models (GLM)	277
11.2.1 Simulating a GLM Model	277
11.2.2 Estimating a GLM Model	277
11.3 Panel Data Models	278
11.3.1 Simulating Panel Data Models	278
11.3.2 Estimating Panel Data Models	278
11.4 Arima Models	279
11.4.1 Simulating Arima Models	279
11.4.2 Estimating Arima Models	279
11.4.3 Forecasting Arima Models	280
11.5 GARCH Models	280
11.5.1 Simulating Garch Models	280
11.5.2 Estimating Garch Models	281
11.5.3 Forecasting Garch Models	281
11.6 Regime Switching Models	281
11.6.1 Simulating Regime Switching Models	282
11.6.2 Estimating Regime Switching Models	282
11.6.3 Forecasting Regime Switching Models	282
11.7 Dealing with Several Models	283
11.7.1 Using <code>tapply</code> and <code>sapply</code>	283
11.7.2 Using <code>by</code>	283
11.7.3 Using <code>dplyr::group_by</code>	284
11.8 Exercises	284
11.9 Exercises	284

12 Reporting Results	287
12.1 Reporting Tables	287
12.2 Reporting Models	288
12.3 Creating Reports with <i>RMarkdown</i>	288
12.4 Exercises	288
12.5 Exercises	288
13 Optimizing Code	291
13.1 Optimizing your Programming Time	291
13.2 Optimizing Code Speed	292
13.2.1 Profiling Code	292
13.2.2 Simple Strategies to Improve Code Speed	292
13.2.3 Using C++ code (package <i>Rcpp</i>)	293
13.2.4 Using cache (package <i>memoise</i>)	294
13.3 Exercises	294
13.4 Exercises	294
## Warning in install.packages :	
## package 'TFX' is not available for this version of R	
##	
## A version of this package for your version of R might be available elsewhere	
## see the ideas at	
## https://cran.r-project.org/doc/manuals/r-patched/R-admin.html#Installing	
## Warning in install.packages :	
## package 'splashr' is not available for this version of R	
##	
## A version of this package for your version of R might be available elsewhere	
## see the ideas at	
## https://cran.r-project.org/doc/manuals/r-patched/R-admin.html#Installing	

About New Edition

Since the first edition of this book in 2017, many things have changed. The second edition of the book complements and extends the previous material, considering new ways to use Rstudio, new R packages written by me and others, as well as the feedback from the first edition.

However, the format of the book has changed significantly. The order of chapters is now aligned with the steps of data-based research, from importing tables, cleaning it, testing hypotheses and presenting results. Thus, the reader can follow through the chapters according to his own work. Another important change is the alignment with the **tidyverse** tools – a set of packages that greatly facilitate the use of R –, especially for data importation and manipulation. Some native R functions are presented when necessary, but only when a **tidyverse** counterpart does not exist.

Three new chapters were added to this book, including a dedicated piece to the process of reporting research output and the use of the *RMarkdown* technology. The final chapter now discusses code optimization by presenting the best code practices and simple strategies to improve work efficiency.

For all R teachers in the world, this and all future editions of the book will include class material such as end-of-chapter exercises with solutions. All teaching material is publicly available in the internet¹ and released with a generous MIT license². So, if you are an R tutor in finance or economics, feel free to use and modify the teaching material to your liking. However, if you do so, please make sure that you cite the original source.

I hope you enjoy reading this book and, based on its content, you can find ways to improve your data analysis. Your feedback is always important and

¹https://www.msperlin.com/publication/2020_book-afedr-en/

²<https://opensource.org/licenses/MIT>

you can reach me at marceloperlin@gmail.com. This book is a special and lifelong project and I will keep improving it as much as I can over the years.

Revision 2021

The book was revised in march 2021, with the following changes:

- Improved and consistent css template for html and ebook.
- Over 100 end of chapter exercises. The exercises within the book are in the `exams` (Zeileis et al., 2022) format and can be compiled to pdf or html. Check out this blog post for details.
- To help the reader with topics that don't quite fit with the main text, new text boxes with important and cautionary messages were implemented in all formats.
- New hardcover format is available at Amazon.

All changes are fully reflected in Amazon.com. Those who previously purchase the second edition in ebook will have a new copy delivered to their kindle account.

Preface

Since you are reading this book, you are likely a data analyst looking for alternative and more efficient ways to add value to your organization, an undergraduate or graduate student in your first steps learning data science, or an experienced researcher, looking for new tools to use in your work. Be assured that you are in the right place. **This book will teach you to use R and RStudio for data analysis in finance and economics.**

The material in this book started out as class slides from my work as a university teacher and researcher in the south of Brazil. By observing students learning and using R in the classroom, I frequently see the positive impact that it has on their careers. They can autonomously do complex data tasks with their computers, providing a better and more comprehensive analysis to help the decision-making process in their organizations. They spend less time doing repetitive and soul-crushing spreadsheet data chores and more time thinking about their analysis and learning new tools. This book attempts to go beyond the classroom and reach an international and more diversified audience.

Another motivation for writing this book is my personal experience using code from other researchers. Usually, the code is not well-organized, lacks clarity, and, possibly, only works in the computer of its author! After being constantly frustrated, I realized the work required to figure out the code of other researchers would take more time than writing the procedure myself. These cases hurt the development of science, as one of its basic principles is the **reproducibility of experiments**. In the case of a computer-intensive field, such as empirical finance and economics, the underlying research code should run without effort in other people's computers. As researchers are expected to be good writers, it should also be expected that their code is in a proper format and readable by other people. With this book, I will tackle this

problem by presenting a code structure focused on scientific reproducibility, organization, and usability.

In this book, we will not work on the advanced uses of R. The content will be limited to simple and practical examples. One challenge of writing this book was defining the boundary between introductory and advanced material. Wherever possible, I gradually dosed the level of complexity. For readers interested in learning advanced features of the program and its inner workings, I suggest the work of Venables et al. (2004), Teator (2011) and Wickham (2019).

This is what you'll learn from this book:

Using R and RStudio In chapter 01 we will discuss the use of R as a programming platform designed to solve data-related problems in finance and economics. In chapter 02 we will explore basic commands and many functionalities of R and RStudio that will increase your productivity.

Importing financial and economic data In chapters 04 and 05 we will learn to import data from local files, such as an Excel spreadsheet, or the internet, using specialized packages that can download financial and economic data such as stock prices, economic indices, the US yield curve, corporate financial statements, and many others.

Cleaning, structuring and analyzing the data with R In chapters 06 and 07 we will concentrate our study on the ecosystem of basic and advanced classes of objects within R. We will learn to manipulate objects such as numeric vectors, dates and whole tables. In chapters 08 and 09 we'll study to use the programming tools to solve data-related problems such as cleaning and structuring messy data. In chapter 11 we will learn applications of the most common econometric models used in finance and economics including linear regression, generalized linear model, Arima model and others.

Creating a visual analysis of data In chapter 10 we'll learn to use functions from package `ggplot2` to create clever visualizations of our datasets, including the most popular applications in finance and economics, time series and statistical plots.

Reporting your results In chapter 12 we will see how to report our data analysis using specialized packages and the *RMarkdown* technology. It includes the topic of presenting and exporting tables, figures and models to a written report.

Writing better and faster code In the last chapter of the book we discuss best programming practices with R. We will look at how to profile code and search for bottlenecks and improving execution time with

caching strategies using package `memoize`, C++ code with `Rcpp` and parallel computing with `furrr`.

Supplement Material

All the material used in the book, including code examples separated by chapters, is publicly available on the Internet and distributed with an R package called `afedR`. It includes data files and several functions that can make it easier to run the examples of the book. If you plan to write some code as you read the book, this package will greatly help your journey.

In order to install the book package in your computer, you need to execute a couple of lines of code in R. For that, copy and paste the following commands into RStudio prompt (bottom left of screen, with a “>” sign) and press enter for each command. Be aware you’ll need R and RStudio installed in your computer (see section 1.4 for details).

```
# install devtools dependency
install.packages('devtools')

# install book package
devtools::install_github('msperlin/afedR')
```

What this code will do is to install package `devtools`, a required dependency for installing a package from Github, which is where the book bundle is hosted. After that, a call to `devtools::install_github('msperlin/afedR')` will install the package in your computer. You can safely ignore any warning messages during instalation.

After installing the book package `afedR`, you can, but its not necessary, to copy all book files to a local folder by simply executing the following command in R:

```
afedR::copy_book_files(path_to_copy = '~')
```

The previous code will unzip the book file into your “Documents/afedR-files” folder, as the tilda (~) is a shortcut to your “Documents” directory. The package also includes several other functions that will be used throughout the book. If you prefer the old-fashioned way of using an internet page, you can find and download the zip file in the book site³.

A suggestion, before you read the rest of the book: go to the book website and search for the related links page at the bottom. There you will find

³<https://www.msperlin.com/files/afedr-files/afedR-code-and-data.zip>

all internet addresses highlighted in the text, including the links for the installation of R and RStudio.

Content for Instructors

If you are an R instructor, you'll find plenty of material you can use with your classes. I made sure you get everything you need:

Over 100 exercises Every chapter in this book includes exercises that your students can practice, with solutions available in the web version of the book. Also, all exercises are available in the `exams` format (see this webpage for details), meaning that you can compile the same exercises in pdf or html. Moreover, you can export the exercises to *e-learning* platforms such as Moodle and Blackboard. See this blog post⁴ for instructions on how to use it with your students.

Web version The first seven chapters of the book is freely available at link <https://www.msperlin.com/afedR>, which is more than enough material for an introductory class on R and data analysis.

All of this content is released with the MIT license, so feel free to use and abuse it, as long as you give the credits to the original author. You can find the content within the book package `afedR` (see previous instructions on installation) or directly at the book site⁵.

I hope you enjoy this book and find it useful for your work.

Good reading!

Marcelo S. Perlin

⁴<https://www.msperlin.com/post/2021-02-28-dynamic-exercises-afedr/>

⁵https://www.msperlin.com/publication/2020_book-afedr-en/

Chapter 1

Introduction

In the digital era, information is abundant and accessible. From the ever-changing price of financial contracts to the unstructured data of social media websites, the high volume of information creates a strong need for data analysis in the workplace. A company or organization benefit immensely when it can create a bridge between raw information from its environment and making strategic decisions. Undoubtedly, this is a prolific time for professionals skilled in using the right tools for acquiring, storing, and analyzing data.

In particular, datasets related to Economics and Finance are widely available to the public. International and local institutions, such as central banks, government research agencies, financial exchanges, and many others, provide their data publicly, either by legal obligation or to foment research. Whether you are looking into statistics for a particular country or a company, most information is just two clicks away. By analyzing this information efficiently and effortlessly, you'll be able to offer valuable insights to your team.

Not surprisingly, fields with abundant access to data and practical applications, such as economics and finance, it is expected that a graduate student or a data analyst has learned at least one programming language that allows him/her to do his work efficiently. **Learning how to program is becoming a requisite for the job market.** In this setup, the role and contribution of R shine. In these sections, I will explain what R is and why you should use it.

1.1 What is R

R is a programming language specially designed to resolve statistical problems and display graphical representations of data. R is a modern version of S, a programming language originally created in Bell Laboratories (formerly AT&T, now Lucent Technologies). The base code of R was developed by two academics, **Ross Ihaka** and **Robert Gentleman**, resulting in the programming platform we have today. For anyone curious about the name, the letter R was chosen due to the common first letter of the name of their creators.

Today, R is almost synonymous with data analysis, with a large user base and consolidated modules. It is likely that researchers from various fields, from economics to biology, find in R significant preexisting code that facilitates their analysis.

On the business side, large and established companies, such as *Google* and *Microsoft*, already adopted R as the internal language for data analysis. R is maintained by the **R Foundation**¹ and the **R Consortium**², a collective effort to fund projects for extending the programming language.

1.2 Why Choose R

Learning a new programming language requires a lot of time and effort. Perhaps you're wondering why you should choose R and invest time in learning it. Here are the main arguments.

First, **R is a mature and stable platform, continuously supported and intensively used in the industry**. When choosing R, you will have the computational background not only for an academic career in scientific research but also to work as a data analyst in private organizations. Due to its open license, you can use R anywhere. Also, the strong support from the community means it is very unlikely the R platform will ever fade away or be substituted. Depending on your career choices, R might be the only programming language you ever need to learn.

Learning R is easy. My experience in teaching R allows me to confidently state that students, even those with no programming experience, have no problem learning the language and using it to create their own code. The language is intuitive and certain rules and functions can be extended to different cases. For example, the function `print` is used to show the contents

¹<https://www.r-project.org/foundation/>

²<https://www.r-consortium.org/>

of an object on the screen. You can use it for any kind of object. So, by learning the main concept, you'll be able to apply it in different scenarios. Once you understand how the software expects you to think, it is easy to discover new features starting from a previous logic. This generic notation facilitates the learning process.

The engine of R and the interface of RStudio creates a highly productive environment. The graphical interface provided by RStudio facilitates the use of R and increases productivity by introducing new features to the platform. By combining both, the user has at his disposal many tools that facilitate the development of research scripts and other projects.

CRAN Packages allow the user to do many different things with R. We will soon learn that we can import external code directly into R as individual modules (packages) and use it for different purposes. These packages extend the basic language of R and enable the most diverse functionalities. You can, for example, use R to write and publish a book, build and publish a blog, create exams with dynamic content, write random jokes and poems (seriously!), send emails, access and collect data from the internet, and many other features. It is truly impressive what you can do with just a couple of lines of code in R.

R is compatible with different operating systems and it can interface with different programming languages. If you need to execute code from another programming language, such as *C++*, *Python*, *Julia*, it is easy to integrate it with R. Therefore, the user is not restricted to a single programming language and can easily use features and functions from others. For example, the C++ code is well known for its superior speed in numerical tasks. From an R script, you can use package *Rcpp* (Eddelbuettel et al., 2022) to write a C++ function and effortlessly use it within your R code.

R is free! The main software and all its packages are free. A generous license motivates the adoption of the R language in a business environment, where obtaining individual and collective licenses of commercial software can be costly. This means you can take R anywhere you go.

1.3 What Can You Do With R and RStudio?

R is a fairly complete programming language and any computational problem can be solved based on it. Given the adoption of R for different areas of knowledge, the list is extensive. With finance and economics, I can highlight

the following possibilities:

- Substitute and improve data-intensive tasks from spreadsheet-like software;
- Develop routines for managing investment portfolios and executing financial orders;
- Creating tools for calculating and reporting economic indices such as inflation and unemployment;
- Performing empirical data research using statistical techniques, such as econometric models and hypothesis testing;
- Create dynamic *websites* with the **shiny** (Chang et al., 2021) package, allowing anyone in the world to use a computational tool created by you;
- Automate the process of writing technical reports with the **RMark-down** technology;

Moreover, public access to packages developed by users further expands these capabilities. The CRAN views website³ offers a *Task Views* panel for the topic of Finance⁴ and Econometrics⁵. There you can find the main packages to perform specific operations such as importing financial data from the internet, estimating econometric models, calculation of different risk estimates, among many other possibilities. Reading this page and the knowledge of these packages is essential for those who intend to work in Finance and Economics. It is worth noting, however, that the complete list of packages is much larger.

³<https://cran.r-project.org/web/views>

⁴<https://cran.r-project.org/web/views/Finance.html>

⁵<https://cran.r-project.org/web/views/Econometrics.html>



Be aware that R has a consistent release schedule. Every four months a new version of R is released, fixing *bugs* and implementing new solutions. There are two main types of releases, *major* and *minor*. For example, today, 2021-02-24, the latest version of R is 4.0.4. The first digit (“4”) indicates the *major* release while all others are of the *minor* type. Generally, the *minor* changes are very specific and, possibly, will have little impact on your work. However, unlike *minor* releases, **major releases are fully reflected in the R package ecosystem**. Every time you install a new *major* version of R, you will have to reinstall all packages. Particularly, the problem here is that it is not uncommon that a new major release comes with package incompatibility issues. My advice is: every time a new *major* release of R comes out, **wait a few months** before installing it on your machine. Thus, the authors of the packages will have more time to update their codes, minimizing the possibility of compatibility problems.

1.4 Installing R

Before going any further, let’s install the required software on your computer. The most direct and practical way to install R is to direct your browser to R website⁶ and click the *Download* link in the left side of the page, as shown in Figure 1.1.

The R Project for Statistical Computing

Getting Started

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred CRAN mirror.

If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

News

- **R version 4.0.4 (Lost Library Book) prerelease versions** will appear starting Friday 2021-02-05. Final release is scheduled for Monday 2021-02-15.
- **R version 4.0.3 (Bunny-Wunnies Freak Out)** has been released on 2020-10-10.
- Thanks to the organisers of useR! 2020 for a successful online conference. Recorded tutorials and talks from the conference are available on the [R Consortium YouTube channel](#).
- **R version 3.6.3 (Holding the Windsock)** was released on 2020-02-29.
- You can support the R Foundation with a renewable subscription as a [supporting member](#)

Figure 1.1: Initial page for downloading R

⁶<http://www.r-project.org/>

The next screen gives you a choice of the mirror to download the installation files. The CRAN repository (*R Comprehensive Archive network*) is mirrored in various parts of the world. You can choose one of the links from the nearest location to you. If undecided, just select the mirror *0-Cloud* (see Figure 1.2), which will automatically take you to the nearest location.

Location	Mirror URL	Description
0-Cloud	https://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
0-Cloud	http://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
Algeria	https://cran.usthb.dz/	University of Science and Technology Houari Boumediene
Algeria	http://cran.usthb.dz/	University of Science and Technology Houari Boumediene
Argentina	http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
Australia	https://cran.csiro.au/	CSIRO
Australia	http://cran.csiro.au/	CSIRO
Australia	https://mirror.aarnet.edu.au/pub/CRAN/	AARNET
Australia	https://cran.ms.unimelb.edu.au/	School of Mathematics and Statistics, University of Melbourne
Australia	https://cran.curtin.edu.au/	Curtin University of Technology

Figure 1.2: Choosing the CRAN mirror

The next step involves selecting your operating system, likely to be *Windows*. From now on, due to the greater popularity of this platform, we will focus on installing R in Windows. The instructions for installing R in other operating systems can be easily found online. Regardless of the underlying platform, using R is about the same. There are a few exceptions, especially when R interacts with the file system. In the content of the book, special care was taken to choose functions that work the same way in different operating systems. A few exceptions are highlighted throughout the book. So, even if you are using a Mac or a flavor of Linux, you can take full advantage of the material presented here.

After clicking the link *Download R for Windows*, as in Figure 1.3, the next screen will show the following download options: *base*, *contrib*, *old.contrib* and *RTools*. The first (*base*), should be selected. It contains the download link to the executable installation file of R in *Windows*.

If the user is interested in creating and distributing their own R packages, *RTools* should also be installed. For most users, however, this should not be the case. If you don't intend to write packages, you can safely ignore *Rtools* for now. The links to *contrib* and *old.contrib* relate to files for the current and old releases of R packages and can also be ignored. We will discuss the



[CRAN](#)
[Mirrors](#)
[What's new?](#)
[Task Views](#)
[Search](#)

[About R](#)
[R Homepage](#)
[The R Journal](#)

[Software](#)
[R Sources](#)
[R Binaries](#)
[Packages](#)
[Other](#)

[Documentation](#)
[Manuals](#)
[FAQs](#)
[Contributed](#)

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows** and **Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2020-10-10, Bunny-Wunnies Freak Out) [R-4.0.3.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).

Figure 1.3: Choosing the operating system

use of packages in the next chapter.



[CRAN](#)
[Mirrors](#)
[What's new?](#)
[Task Views](#)
[Search](#)

[About R](#)
[R Homepage](#)
[The R Journal](#)

[Software](#)
[R Sources](#)
[R Binaries](#)
[Packages](#)
[Other](#)

[Documentation](#)
[Manuals](#)
[FAQs](#)
[Contributed](#)

R for Windows

Subdirectories:

[base](#)

Binaries for base distribution. This is what you want to [install R for the first time](#).

[contrib](#)

Binaries of contributed CRAN packages (for R >= 2.13.x; managed by Uwe Ligges). There is also information on [third party software](#) available for CRAN Windows services and corresponding environment and make variables.

[old contrib](#)

Binaries of contributed CRAN packages for outdated versions of R (for R < 2.13.x; managed by Uwe Ligges).

[Tools](#)

Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

Figure 1.4: Installation options

After clicking the link *base*, the next screen will show the link to the *download*

of the R installation file (Figure 1.5). After downloading the file, open it and follow the steps in the installation screen. At this time, no special configuration is required. I suggest keeping all the default choices and simply hit *accept* in the displayed dialogue screens. After the installation of R, it is strongly recommended to install RStudio, which will be addressed next.

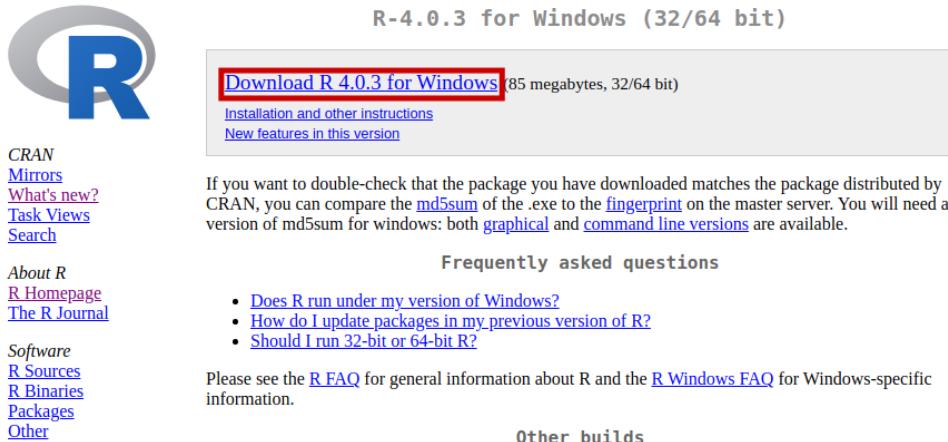


Figure 1.5: Downloading R

1.5 Installing RStudio

The base installation of R includes its own *GUI* (graphical user interface), where we can write and execute code. However, this native interface has several limitations. RStudio substitutes the original GUI and makes access to R more practical and efficient. One way to understand this relationship is with an analogy with cars. While R is the engine of the programming language, RStudio is the body and instrument panel, which significantly improves the user experience. Besides presenting a more attractive look, RStudio also adds several features that make the life of a programmer easier, allowing the creation of projects and packages, creation of dynamic documents, among others.

The installation of RStudio is simpler than that of R. The files are available in RStudio website⁷. After accessing the page, click *Download RStudio* and then *Download RStudio Desktop*. After that, just select the installation file relative to the operating system on which you will work. This option is probably *WINDOWS Vista 7/8/10*. Note that, as well as R, RStudio is also available for alternative platforms.

⁷<https://www.rstudio.com/>

I emphasize that using RStudio is not essential to develop programs in R. Other interfaces are available and can be used. However, in my experience, RStudio is the interface that offers a vast range of features for the language and is widely used, which justifies its choice.

1.6 Resources in the Web

The R community is vivid and engaging. There are many authors, such as myself⁸, that constantly release material about R in their blogs. It includes the announcement of new packages, posts about data analysis in real life, curiosities, rants, and tutorials. R-Bloggers⁹ is a website that aggregates these blogs, making it easier for anyone to access and participate. I strongly recommend to sign up for the R-Bloggers feed in RSS¹⁰, Facebook¹¹ or Twitter¹². Not only you'll be informed of what is happening in the R community, but also learn a lot by reading other people's code and articles.

Learning and using R can be a social experience. Several conferences and user-groups are available in many countries. You can find the complete list in this link¹³. I also suggest looking in social platforms for local R groups in your region.

1.7 Structure and Organization

This book presents a practical approach to using R in finance and economics. To get the most out of this book, I suggest you first seek to understand the code shown, and only then, try using it on your own computer. Whenever you find a piece of code that you do not understand, go on and study it. At first, it might seem like a daunting task but, with time, be confident that the learning process will get a lot easier as the code blocks start to make sense.

Learning to program in a new platform is like learning a foreign spoken language: the use in day-to-day problems is imperative to create fluency. All the code and data used in this book is available with the installation of package `afedR` (see the preface for instructions on how to install it). I

⁸<https://www.mspperl.com>

⁹<https://www.r-bloggers.com/>

¹⁰<https://feeds.feedburner.com/RBloggers>

¹¹<https://www.facebook.com/rbloggers/?fref=ts>

¹²<https://twitter.com/Rbloggers>

¹³<https://jumpingrivers.github.io/meetingsR/index.html>

suggest you test the code on your computer and *play* with it, modifying the examples and checking the effect of changes in the outputs. Whenever you have a computational problem, try using R to solve it. You'll stumble and make mistakes at first. But I guarantee that, soon enough, you'll be able to write complex data tasks effortlessly.

Throughout the book, every demonstration of code will have two parts: the R code and its output. The output is nothing more than the result of the commands on the screen. All inputs and outputs code will be marked in the text with a special format. See the following example:

```
# create a list
L <- list('abc', 1:5, 'dec')

# print list
print(L)
```

```
R> [[1]]
R> [1] "abc"
R>
R> [[2]]
R> [1] 1 2 3 4 5
R>
R> [[3]]
R> [1] "dec"
```

For the previous chunk of code, lines `L <- list('abc', 1:5, 'dec')` and `print(L)` are actual commands given to R. The output of this simple piece of code is the on-screen presentation of the contents of object L. The symbol `R>` is used for any code output. Notice also that inline comments are set with the symbol `#`. Anything on the right side of `#` is not evaluated by R. These comments serve as written notes about the code.

The code can also be spatially organized using newlines. This is a common strategy around arguments of functions. The next chunk of code is equivalent to the previous and will run the exact same way. Notice how we used a new line to vertically align the arguments of function `list`. You'll soon see that, throughout the book, this type of vertical alignment is constantly used.

```
# create a list
L <- list(
  'abc',
  1:5,
  'dec')

# print list
```

```
print(L)

R> [[1]]
R> [1] "abc"
R>
R> [[2]]
R> [1] 1 2 3 4 5
R>
R> [[3]]
R> [1] "dec"
```

The code also follows a well-defined structure. One decision in writing computer code is how to name objects and how to structure it. It is recommended to follow a clear pattern, so it is easy to maintain over time and be used and understood by others. For this book, a mixture of the author's personal choices with the coding style suggested by Google¹⁴ was used. The reader, however, may choose the structure he finds more efficient and aesthetically pleasing. Like many things in life, this is a choice. We will get back at discussing code structure in chapter 13.

1.8 Exercises

All solutions are available at <https://www.msperlin.com/afedR>.

01 - The R language was developed based on what other programming language?

- a) S
- b) C++
- c) Javascript
- d) Python
- e) Julia

02 - What are the names of the two authors of R?

- a) Linus Torvalds and Richard Stallman
- b) Guido van Rossum and Bjarne Stroustrup
- c) John Chambers and Robert Engle
- d) Roger Federer and Rafael Nadal
- e) Ross Ihaka and Robert Gentleman

¹⁴<https://google.github.io/styleguide/Rguide.xml>

03 - Why is R special when comparing to other programming languages, such as Python, C++, javascript and others?

- a) Makes it easy to write mobile apps
- b) It was designed for analyzing data and producing statistical output
- c) Works on any platform such as Windows, Unix, MacOS
- d) Easy to use
- e) Quick code execution

04 - What was the reason the programming language was named R?

- a) R = Reusable code.
- b) Letter R is cool!
- c) The mother of one of the authors is called Renata.
- d) Letter R is shared in the first names of its authors.
- e) It was the only available letter, not yet used as a programming language.

05 - Consider the following alternatives about R and RStudio:

I - R was developed in 2018 and is an innovative and unstable project;

II - RStudio is an alternative programming language to R;

III - R has compatibility with different programming languages;

Which alternatives are correct?

- a) FALSE, FALSE, TRUE
- b) TRUE, FALSE, FALSE
- c) FALSE, FALSE, FALSE
- d) TRUE, TRUE, FALSE
- e) TRUE, TRUE, TRUE

06 - Once you have R and RStudio installed, head over to the CRAN package website¹⁵ and look for technologies you use in your work. For example, if you use Google Sheets¹⁶ ostensibly in your work, you will soon discover that there is a package in CRAN that interacts with spreadsheets in the cloud.

07 - On the CRAN site you can also install the Rtools application. What is it for?

¹⁵https://cloud.r-project.org/web/packages/available_packages_by_date.html

¹⁶<https://www.google.com/sheets/about/>

- a) Make coffee (?).
- b) Compile graphics.
- c) Compile technical reports.
- d) Compile R packages locally
- e) Build web pages.

08 - Use Google to search for R groups in your region. Check if the meetings are frequent and, if you don't have a major impediment, go to one of these meetings and make new friends.

09 - Go to the RBloggers website¹⁷ and look for a topic of interest to you, such as football (*soccer*) or investments (*investments*). Read at least three of the found blog posts.

10 - If you work in an institution with data infrastructure, talk to the person in charge of the IT department and verify what technologies are used. Check if, through R, it is possible to access all tables in the databases. For now there is no need to write code, yet. Just check if this possibility exists.

¹⁷<https://www.r-bloggers.com/>

Basic Operations in R

Basic operations are the fundamental tasks that you will be repeating many times over your workday. As such, it is important to understand how to work with R and RStudio efficiently before you develop your data analysis. This includes understanding the RStudio interface, basic R commands, file extensions, available shortcuts and the autocomplete feature of RStudio.

In this section, we will go through the initial steps from the viewpoint of someone who has never worked with R and possibly never had contact with other programming languages. Those already familiar with the software may not find novel information here and, therefore, I suggest skipping to the next section. However, it is recommended, that you at least check the discussed topics so you can confirm your knowledge about the features of the software and how to use them for working smarter and not harder. This is especially true for RStudio, which offers several tools to increase your productivity.

2.1 Working With R

The greatest difficulty new users experience when developing routines in R is the format of work – the so-called development cycle. Our interaction with computers has been simplified over the years and we are currently comfortable with the *point&click* format. That is, if you want to perform an operation on the computer, just point the *mouse* to the specific location on the screen and click a button. Visual cues in a series of steps allow the execution of complex tasks. But, you need to be aware that this form of interaction is just one layer above what actually happens. Behind all these *clicks*, there is a command being executed on your computer. Any common

task such as opening a *pdf* file, a spreadsheet document, directing a *browser* to a web page has an underlying call to a code.

The “point&click” format of visual and motor interaction has its benefits in facilitating and popularizing the use of computers. However, it is not flexible and effective when working with computational procedures such as data analysis. By knowing the commands available to the user and how to execute them, it is possible to create a file containing several instructions in sequence and, in the future, simply request that the computer **execute** this file using the recorded procedures. There is no need to do a “scripted” point&click operation. You spend some time writing the program but, in the future, it will always execute the recorded procedure in the same way.

In the medium and long term, there is a significant gain in productivity between the use of a *script* (sequence of commands) and a *point&click* type of interface. Going further, the risk of human error in executing the procedure is almost nil because the commands and their required sequence of execution are recorded in the text file and will always be executed in the same way. This is one of the main reasons programming languages are popular in science. All steps of data-based research, including results, can be replicated.

In using R, the ideal format of work is to merge the mouse movement with commands. R and RStudio have some functionality with the *mouse*, but their capacity is optimized when we perform operations using code. When a group of commands is performed in a smart way, we have an R script that should preferably produce something important to us at the end of its execution. In finance and economics, this can be the current price of a stock, the value of an economic index such as inflation, the result of academic research, among many other possibilities.

Like other software, R allows us to import data and export files. We can use code to import a dataset stored in a local file – or the web–, analyze it and paste the results into a technical report. We can use RStudio and the *RMarkdown* technology to write a dynamic report, where code and content are integrated. For example, the book you’re reading was written using the *bookdown* package (Xie, 2022). The content of the book is compiled with the execution of the R codes and their outputs are recorded in the scope of the text. All figures and data tasks in the book can be updated with the execution of a simple command. Needless to say that by using the capabilities of R and RStudio, you will work smarter and faster.

The final product of working with R and RStudio will be an R script that produces elements for a data report. A good example of a simple and pol-

ished R script can be found at this link¹. Open it and you'll see the content of a file with extension *.R* that will download stock prices of two companies and create a plot and a table. By the end of the book, you can understand what is going on in the code and how it gets the job done. Even better, you'll be able to improve it. Soon, you'll learn to execute the code on your own computer. If impatient, simply copy the text content of the link to a new RStudio R script, save it, and press **control + shift + enter**.

2.2 Objects in R

In R, everything is an object, and each type of object has its properties. For example, the daily closing prices of the IBM stock can be represented as a numerical vector, where each element is a price recorded at the end of a trading day. Dates related to these prices can be represented as text (*string*) or as a unique **Date** class. Finally, we can represent the price data and the dates together by storing them in a single object of type **dataframe**, which is nothing more than a table with rows and columns. These objects are part of the R ecosystem, and through their manipulation, we take full advantage of the software.

While we represent data as objects in R, a special type is a **function**. It stores a pre-established manipulation of other objects available to the user. R has an extremely large number of functions, which enable the user to perform a wide range of operations. For example, the basic commands of R, available in the package **base**, adds up to a total of 1257 functions.

Each function has its own name and a programmer can write their own functions. For example, the **sort** function is a procedure that sorts elements within a vector. If we wanted to sort the elements of 2, 1, 4, 3, 1, simply insert the following command in the *prompt* (left bottom side of RStudio's screen) and press *enter*:

```
my_vec <- c(2, 1, 4, 3, 1)
sorted_vec <- sort(x = my_vec, decreasing = TRUE)
print(sorted_vec)
```

```
R> [1] 4 3 2 1 1
```

The **sort** function is used with start and end parentheses. These parentheses serve to highlight the entries (*inputs*), that is, the information sent to the function to produce something that will be saved in object **sorted_vec**. Note

¹https://github.com/msperlin/afedR/raw/master/inst/extdata/others/S_Example_Script.R

that each entry is separated by a comma, as in `my_fct(input1, input2, input3, ...)`. We also set option `decreasing = TRUE`. This is a specific directive for the `sort` function to order the value from highest to lowest. **Functions are at the heart of R** and we will dedicate a large part of this book to them. You can use the available functions or write your own. You can also publish functions as a package and let other people use your code.

2.3 International and Local Formats

Before explaining the use of R and RStudio, it is important to highlight some rules of formatting numbers, Latin characters and dates.

decimal: Following an international notation, the decimal point in R is defined by the period symbol (.), as in 2.5 and not a comma, as in 2,5. If this is not standard in your country, you'll have issues when importing local data from text files. Sometimes, such as with storing data in Microsoft Excel files, the conversion happens automatically in the importing process. This, however, is generally an exception. As a general rule of using R, only use commas to separate the inputs of a function. Under no circumstances should the comma symbol be used as the decimal point separator. Always give priority to the international format because it will be compatible with the vast majority of data.

Latin characters: Due to its international standard, R has problems understanding Latin characters, such as the cedilla and accents. If you can, avoid it, and do not use Latin characters in the names of your variables or files. In the content of character objects (text), you can use them without problems as long as the encoding of the script is correctly specified (e.g. UTF-8, Latin1). I strongly recommend the use of the English language for writing code and defining object names. This automatically eliminates the use of Latin characters and facilitates the usability of the code by people outside of your country.

date format: Dates in R are structured according to the ISO 8601² format. It follows the YYYY-MM-DD pattern, where YYYY is the year in four numbers, MM is the month and DD is the day. An example is 2022-11-23. This, however, may not be the case in your country. When importing local data sets, make sure the dates are in this format. If necessary, you can convert any date to the ISO format. Again, while you can work with your local format of dates in R, it is best advised to use the international notation. The conversion between one format and another is quite easy and will be presented in chapter 7.

²<https://www.iso.org/iso-8601-date-and-time-format.html>

If you want to learn more about your local format in R, use the following command by typing it in the prompt and pressing enter:

```
Sys.localeconv()
```

```
R>   decimal_point      thousands_sep      grouping
R>         "."           ""                 ""
R>   int_curr_symbol    currency_symbol  mon_decimal_point
R>         "BRL"          "R$"              ","
R>   mon_thousands_sep   mon_grouping    positive_sign
R>         "."           "\003\003"        ""
R>   negative_sign      int_frac_digits  frac_digits
R>         "-"           "2"                "2"
R>   p_cs_precedes      p_sep_by_space  n_cs_precedes
R>         "1"           "1"                "1"
R>   n_sep_by_space      p_sign_posn    n_sign_posn
R>         "1"           "1"                "1"
```

The output of `Sys.localeconv()` shows how R interprets decimal points and the thousands separator, among other things. As you can see from the previous output, this book was compiled using the Brazilian notation for the currency but uses the dot point for decimals.



Be careful when modifying the format that R interprets the different symbols and notations. As a rule of thumb, if you need to use a specific format, do it separately within the context of the code. Avoid permanent changes as you never know where such formats are being used. That way, you can avoid unpleasant surprises in the future.

2.4 Types of Files in R

Like any other programming platform, R has a file ecosystem and each type of file has a different purpose. In the vast majority of cases, however, the work will focus mostly on a couple of types. Next, I describe various file extensions. The items in the list are ordered by importance. Note that we omit graphic files such as `.png`, `.jpg`, `.gif` and data storage/spreadsheet files (`.csv`, `.xlsx`, ..) among others, as they are not exclusive to R.

Files with extension `.R`: text files containing R code. Besides, these are the files we will spend most of our time. They contain the sequence of commands that configures the main script and subroutines of the data research. Examples: `Script-stock-research.R`, `R-fcts.R`.

Files with extension *.RData* or *.rds*: files that store data in the native format. These files are used to save/write objects created in different sessions into your hard drive. For example, you can use a *.rds* file to save a table after processing and cleaning up the raw database. By *freezing* the data in a local file, we can later load it for subsequent analysis. Examples: *cleaned-inflation-data.rds*, *model-results.RData*.

Files with extension *.Rmd* and *.md*: files used for editing dynamic documents in the *RMarkdown* and *markdown* formats. Using these files allows the creation of documents where text and code output are integrated into the same document. In chapter 12 we have a dedicated section for RMarkdown, which will explore this functionality in detail. Example: *investment-report.Rmd*.

Files with extension *.Rproj*: contain files for editing projects in RStudio, such as a new R package, a *shiny* application or a book. While you can use the functionalities of RStudio projects to write R scripts, it is not a necessity. For those interested in learning more about this functionality, I suggest the RStudio manual³. Example: *project-retirement.Rproj*.

2.5 Explaining the RStudio Screen

After installing the two programs, R and RStudio, open RStudio by double-clicking its icon. Be aware that R also has its own interface and this often causes confusion. You should find the correct shortcut for RStudio by going through your software folders. In Windows, you can search for RStudio using the *Start* button and typing *Rstudio*.

After opening RStudio, the resulting window should look like Figure 2.1.

Note that RStudio automatically detected the installation of R and initialized your screen on the left side.

As a first exercise, click *file*, *New File*, and *R Script*. A text editor should appear on the left side of the screen. It is there that we will spend most of our time developing code. Commands are executed sequentially, from top to bottom. A side note, all *.R* files created in RStudio are just text files and can be edited anywhere. It is not uncommon for experienced programmers to use specific software to write code and another to run it.

³<https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects>

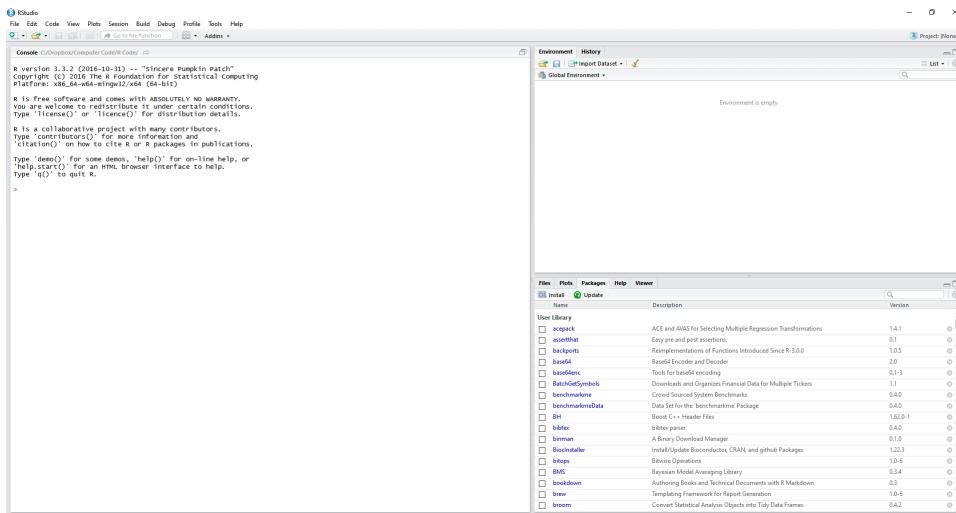


Figure 2.1: The RStudio screen

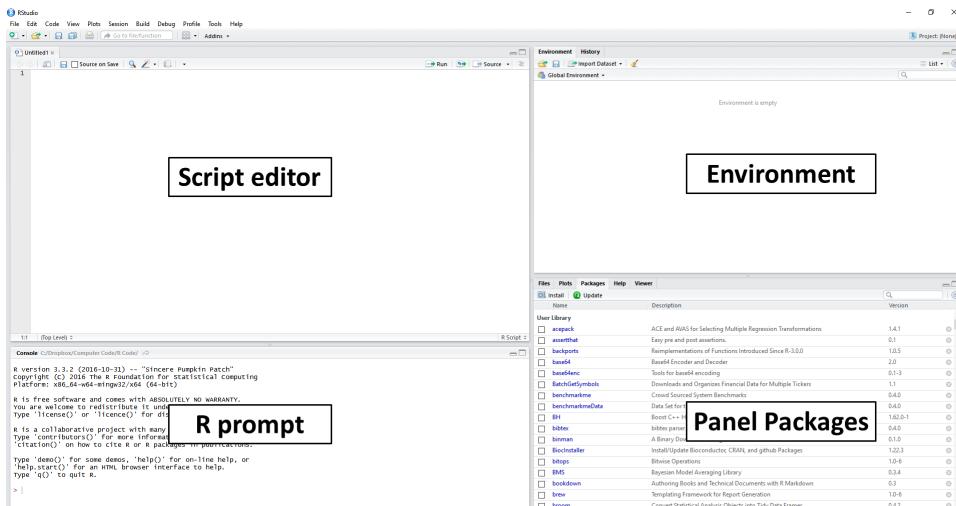


Figure 2.2: Explaining the RStudio screen



An important suggestion here is to change the color scheme of RStudio to a **dark mode** setting. It is not just an aesthetic issue, but also a strategy for preventing health problems. Since you will be spending a lot of time in front of the computer, it is smart to change the colors of the interface to relieve your eyes of the constant brightness of the screen. That way, you'll be able to work longer, without straining your vision. You can configure the color scheme of RStudio by going to the option *Tools*, *Global Options* and then *Appearance*. A dark color scheme that I personally like and suggest is *Ambience*.

After the previous steps in RStudio, the resulting screen should look like the image in Figure 2.2. The main items/panels of the RStudio screen in are:

Script Editor: located on the left side and above the screen. This panel is used to write scripts and functions, mostly on files with the *.R* extension;

R prompt: on the left side and below the script editor. It displays the *prompt*, which can also be used to give commands to R. The main purpose of the prompt is to test code and display the results of the commands entered in the script editor;

Environment: located on the top-right of the screen. Shows all objects, including variables and functions currently available to the user. Also note a *History* panel, which shows the history of commands previously executed by the user;

Panel Packages: shows the packages installed and loaded by R. Here you have four tabs: *Files*, to load and view system files; *Plots*, to view statistical figures created in R; *Help* to access the help system and *Viewer* to display dynamic and interactive results, such as a web page.

As an introductory exercise, let's initialize two objects in R. Inside the prompt (lower left side), insert these commands and press *enter* at the end of each. The `<-` symbol is nothing more than the result of joining `<` (less than) with the `-` (minus sign). The `'` symbol represents a single quotation mark and, in the computer keyboard, it is found under the escape (*esc*) key.

```
# set x
x <- 1

# set y
y <- 'My humble text'
```

If done correctly, notice that two objects appeared in the *environment* panel,

one called `x` with a value of 1, and another called `y` with the text content "My humble text". Also noticed how we used specific symbols to define objects `x` and `y`. The use of double quotes (" ") or single quotes (' ') defines objects of the class `character`. Numbers are defined by the value itself. As will be discussed later, understanding R object classes are important as each has a different behavior within the R code. After executing the previous commands, the *history tab* has been updated.

Now, let's show the values of `x` on the screen. To do this, type the following command:

```
# print contents of x  
print(x)
```

```
R> [1] 1
```

The `print` function is one of the main functions for displaying values in the *prompt* of R. The text displayed as [1] indicates the index of the first line number. To verify this, enter the following command, which will show a lengthy sequence of numbers on the screen:

```
# print a sequence  
print(50:100)
```

```
R> [1] 50 51 52 53 54 55 56 57 58 59 60 61 62 63  
R> [15] 64 65 66 67 68 69 70 71 72 73 74 75 76 77  
R> [29] 78 79 80 81 82 83 84 85 86 87 88 89 90 91  
R> [43] 92 93 94 95 96 97 98 99 100
```

Here, we use the colon symbol in `50:100` to create a sequence starting at 50 and ending at 100. Note that, on the left side of each line, we have the values [1], [13], and [25]. These represent the index of the first element presented in the line. For example, the fifteenth element of `50:100` is 64.

2.6 R Packages

One of the greatest benefits of using R is its package collection. A package is nothing more than a group of procedures aimed at solving a particular computational problem. R has at its core a collaborative philosophy. Users provide their codes for others to use. And, most importantly, **all packages are free**. For example, consider a case where the user is interested in accessing data about historical inflation in the USA. He can install and use an R module that is specifically designed for importing data from central banks and research agencies.

Every function in R belongs to a package. When R initializes, packages `stats`, `graphics`, `grDevices`, `utils`, `datasets`, `methods` and `base` are loaded by default. Almost every function we have used so far belongs to the package `base`. R packages can be accessed and installed from different sources. The main being **CRAN** (*The Comprehensive R Archive network*), and **Github**. It's worth knowing that the quantity and diversity of R packages increase every day.

CRAN is the official repository of R and it is built by the community. Anyone can send a package. However, there is an evaluation process to ensure that certain strict rules about code format and safety are respected. For those interested in creating and distributing packages, a clear roadmap on how to create and send packages to CRAN is presented on the site R packages⁴. Complete rules are available on the CRAN website⁵.

The suitability of the code to CRAN standards is the developer's responsibility. My personal experience, sending and publishing a package on CRAN demands a significant amount of work, especially in the first submission. After that, it becomes a lot easier. Don't be angry if your package is rejected. My own packages were rejected several times before entering CRAN. Listen to what the maintainers tell you and try fixing all problems before resubmitting. If you're having issues that you cannot solve or find a solution on the Internet, look for help in the R-packages mailing list⁶. You'll be surprised at how accessible and helpful the R community can be.

The complete list of packages available on CRAN, along with a brief description, can be accessed at the packages section of the R site⁷. A practical way to check if there is a package that does a specific procedure is to load the previous page and search in your *browser* for a keyword of interest (e.g. "SEC data"). If there is a package that does what you want, it is very likely that the keyword is used in the description field.

Another important source for finding packages is the CRAN Task Views⁸. There you can find the collection of noteworthy packages for a given area of expertise. See the *Task Views* screen in Figure 2.3.

A popular alternative to CRAN is Github⁹. Unlike the former, Github imposes no restrictions on the submitted code and, because of this and its version control system, it is a popular choice by developers. In practice, it is

⁴<http://r-pkgs.had.co.nz/intro.html>

⁵<https://cran.r-project.org/web/packages/policies.html>

⁶<https://www.r-project.org/mail.html>

⁷https://cran.r-project.org/web/packages/available_packages_by_date.html

⁸<https://cran.r-project.org/web/views/>

⁹<https://github.com/>

CRAN Task Views	
Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Envirometrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics & Survey Methodology
Optimization	Optimization and Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods
Psychometrics	Psychometric Models and Methods
ReproducibleResearch	Reproducible Research
Robust	Robust Statistical Methods

Figure 2.3: Task View screen

very common for developers to maintain a development version on Github and the official version in CRAN. When the development version reaches a certain stage of maturity, it is then sent to CRAN.

The most interesting part of this is that the Github packages can be accessed and installed directly from the prompt using the internet. To find out the current amount of packages on CRAN, type and execute the following commands in the prompt:

```
# get a matrix with available packages
df_cran_pkgs <- available.packages()

# find the number of packages
n_cran_packages <- nrow(df_cran_pkgs)

# print it
print(n_cran_packages)
```

R> [1] 18877

Currently, 2022-11-23 16:07:34, there are 18877 packages available on the CRAN servers, a very impressive mark for the community of developers as a whole.

You can also check the amount of **locally installed packages** in R with the `installed.packages` command:

```
# find number of packages currently installed
n_local_packages <- nrow(installed.packages())
```

```
# print it
print(n_local_packages)
```

R> [1] 539

In this case, the computer in which the book was written has 539 packages currently installed. Notice that, even as an experienced researcher and R programmer, I'm only using a small fraction of all packages available in CRAN! The number of installed packages is probably different from yours. Give it a try with the command `installed.packages()`!

2.6.1 Installing Packages from CRAN

To install a package, simply use the command `install.packages`. You only need to do it once for each new package. As an example, we will install a package called `readr` that will be used in future chapters. Note that we defined the package name in the installation as if it were text with the use of quotation marks (" ").

```
# install package readr
install.packages("readr")
```

That's it! After executing this simple command, package `readr` and all of its dependencies will be installed and the functions related to the package will be ready for use once the package is loaded in a script. If the installed package is dependent on another package, R detects this dependency and automatically installs the missing packages. Thus, all the requirements for using the installed package will already be satisfied and everything will work perfectly. It is possible, however, that a package has an external dependency. As an example, package `RnDTeXExams` depends on the existence of a LaTeX installation. These cases are usually announced in the description of the package and an error informs that a requirement is missing. External dependencies for R packages are not common, but they do happen.

2.6.2 Installing Packages from Github

To install a package hosted in Github, you must first install the `devtools` package, available on CRAN:

```
# install devtools
install.packages('devtools')
```

After that, use the function `devtools::install_github` to install a package

directly from Github. In the following example, we will install the development version of package `dplyr`:

```
# install ggplot2 from github  
devtools::install_github("hadley/dplyr")
```

Note that the username of the developer is included in the input string. In this case, the `hadley` name belongs to the developer of `dplyr`, Hadley Wickham. Throughout the book, you will notice that this name appears several times. Hadley is a prolific and competent developer of several popular R packages and currently works for RStudio.



Be aware that **github packages are not moderated**. Anyone can send code there and the content is not independently checked. Never install github packages without some confidence of the author's work. Although unlikely - it never happened to me for example - it is possible that they have malicious code.

2.6.3 Loading Packages

Within a script, use the function `library` to load a package, as in the following example.

```
# load package readr  
library(readr)
```

After running this command, all functions of the package will be available in the **current** R session. Whenever you close RStudio or start a new session, you'll lose all loaded packages. This is the reason why packages are usually loaded in the top of the script: starting from a clean memory, required packages are sequentially loaded before the actual R code is executed.

If the package you want to use is not available, R will throw an error message. See an example next, where we try to load a non-existing package called `unicorn`.

```
library(unicorn)
```

```
R> Error: There is no library called "unicorn"
```

Remember this error message. It will appear every time a package is not found. If you got the same message when running code from this book, you need to check what are the required packages of the example and install them using `install.packages`, as in `install.packages('unicorn')`.

Alternatively, if you use a specific package function and do not want to load all functions from the same package, you can do it with the use of double colons (::), as in the following example.

```
# example of using a function without loading package
fortunes::fortune(10)
```

```
R>
R> Overall, SAS is about 11 years behind R and S-Plus in
R> statistical capabilities (last year it was about 10 years
R> behind) in my estimation.
R>      -- Frank Harrell (SAS User, 1969-1991)
R>      R-help (September 2003)
```

Here, we use function **fortune** from the package **fortunes** (Zeileis and the R community, 2016), which shows on screen a potentially funny phrase chosen from the R mailing list. For our example, we selected message number 10. One interesting use of the package **fortune** is to display a random joke every time R starts and, perhaps, lighten up your day. As mentioned before, R is fully customizable. You can find many tutorials on how to achieve this effect by searching on the web for “customizing R startup”.

Another way of loading a package is by using the **require** function. A call to **require** has a different behavior than a call to **library**. Whenever you try to load an uninstalled package with the **library** function, it returns an error. This means that the script stops and no further code are evaluated. As for **require**, if a package is not found, it returns an object with value **FALSE** and the rest of the code is evaluated. So, in order to avoid code being executed without its explicit dependencies, it is best advised to always use **library** for loading packages in R scripts.

The use of **require** is left for loading up packages inside of functions. If you create a custom function that requires procedures from a particular package, you must load the package within the scope of the function. For example, see the following code, where we create a new function called **fct_example** that depends on the package **quantmod**:

```
fct_example <- function(x){

  require(quantmod)

  df <- getSymbols(x, auto.assign = F)
  return(df)
}
```

In this case, the first time that `fct_example` is called, it loads up the package `quantmod` and all of its functions. Using `require` inside a function is a good programming policy because the function becomes self-contained, making it easier to use it in the future. This was the first time where the complete definition of a user-created function in R is presented. Do not worry about it for now. We will explain it further in chapter 8.



Be aware that loading a package can cause a **conflict of functions**. For example, there is a function called `filter` in the `dplyr` package and also in the `stats` package. If we load both packages and call the `filter` function within the scope of the code, which one will R use? Well, the **preference is always for the last loaded package**. This is a type of problem that can be very confusing. Fortunately, note that R itself tests for conflicts when loading a package. Try it out: start a new R session and load the `dplyr` package. You will see that a message indicates that there are two conflicts with the `stats` package – functions `filter` and `lag` – and four with the `base` package.

A simple strategy to avoid bugs due to conflict of function is to call a function using the actual package name. For example, if I'm calling `lag` from `dplyr`, I can write the call as `dplyr::lag`. As you can see, the package name is explicit, avoiding any possible conflict.

2.6.4 Upgrading Packages

Over time, it is natural that packages available on CRAN are upgraded to accommodate new features, correct bugs and adapt to changes. Thus, it is recommended that users update their installed packages to a new version over the internet. In R, this procedure is quite easy. A direct way of upgrading packages is to click the button *Update* located in the package panel, lower right corner of RStudio, as shown in Figure 2.4.

The user can also update packages through the prompt. Simply type command `update.packages()` and hit *enter*, as shown below.

```
# update all installed packages  
update.packages()
```

The command `update.packages` compares the version of the installed packages with the versions available in CRAN. If it finds any difference, the new versions are downloaded and installed. After running the command, all packages will be synchronized with the versions available in CRAN.

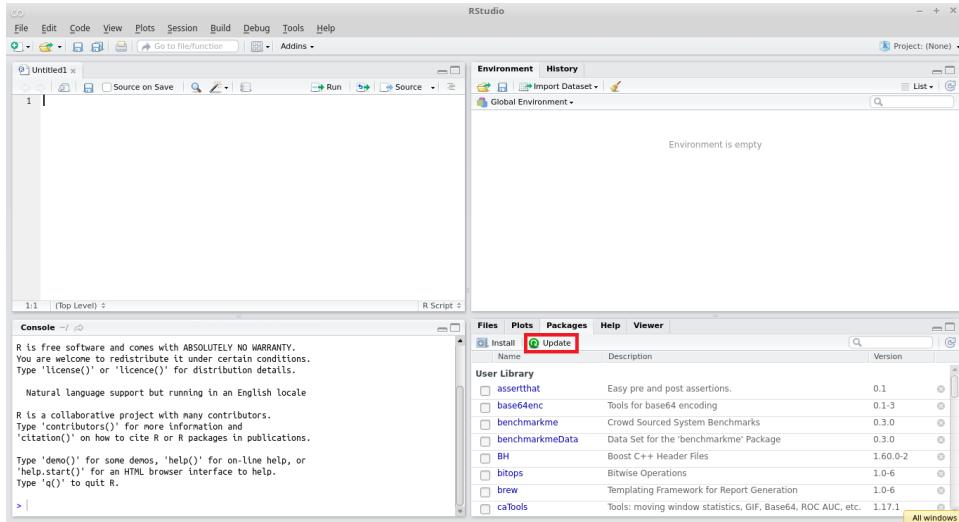


Figure 2.4: Updating R packages



Package versioning is an extremely important topic for keeping your code reproducible. Although it is uncommon to happen, a package update might modify, for the same data, results obtained previously. I have a particularly memorable experience when a scientific article returned from a journal review and, due to the update of one of the R packages, I was unable to reproduce the results presented in the article. In the end everything went well, but the trauma remains.

One solution to this problem is to freeze the package versions for each project using RStudio's `packrat` tool. In summary, `packrat` makes local copies of the packages used in the project, which have preference over system packages. Thus, if a package is updated in the system, but not in the project, the R code will continue to use the older version and the R code will always run under the same conditions.

2.7 Running Scripts from RStudio

Now, let's combine all the previously typed codes into a single file by copying and pasting all commands into the editor's screen (upper left side). The result looks like Figure 2.5.

After pasting all the commands in the editor, save the `.R` file to a personal folder where you have read and write permissions. In Windows,

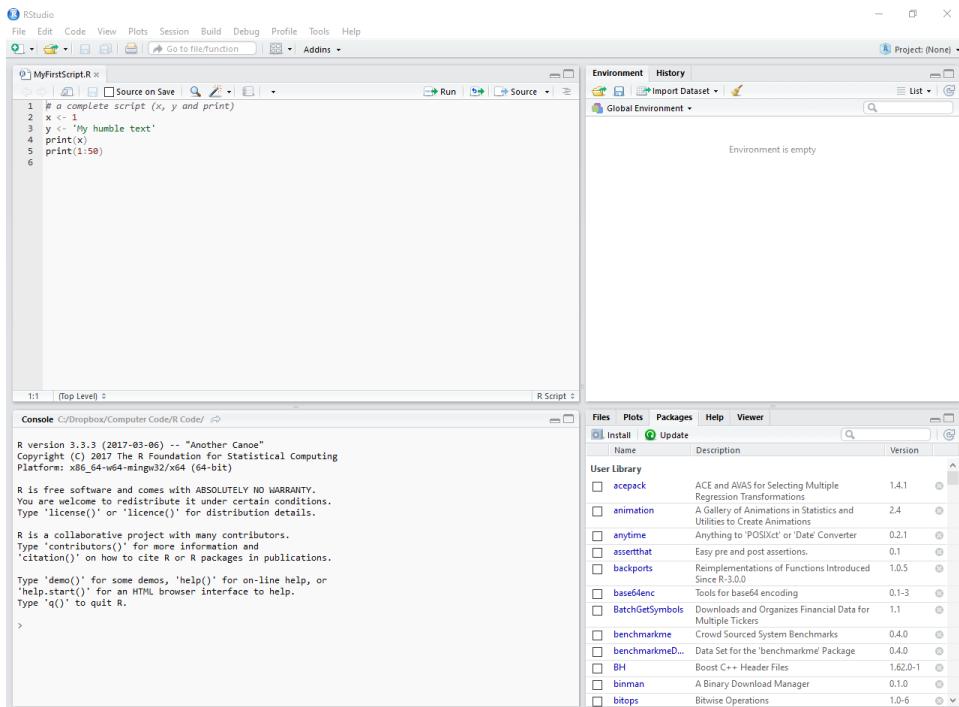


Figure 2.5: Example of an R script

one possibility is to save it in the `Documents` folder with a name like '`my_first_script.R`'. This saved file, which at the moment does nothing special, records the steps of a simple algorithm that creates several objects and shows their content.

2.7.1 RStudio shortcuts

In RStudio, there are some predefined and time-saving shortcuts for running code from the editor. To execute an entire script, simply press **control + shift + s**. This is the `source` command. With RStudio open, I suggest testing this key combination and checking how the code saved in a `.R` file is executed. The output of the script is shown in the prompt of R. The result in RStudio should look like Figure 2.6.

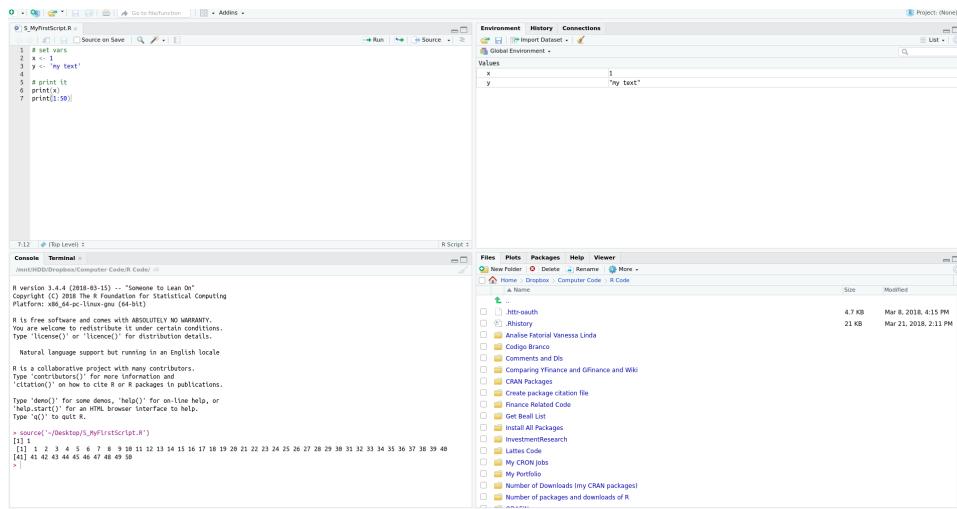


Figure 2.6: Example of a R script after execution

Another way of executing code is with the shortcut **control + enter**, which will only execute the line where the cursor is located. This shortcut is very useful in developing scripts because it allows each line of the code to be tested. As an example of usage, point the cursor to the `print(x)` line and press *control + enter*. As you will notice, only the line `print(x)` was executed and the cursor moves to the next line. Therefore, before running the whole script, you can test it line by line and check for possible errors.

Next, I highlight these and other RStudio shortcuts, which are also very useful.

control + shift + s executes (source) the current RStudio file;

control + shift + enter executes the current file with echo, showing the commands on the prompt;
control + enter executes the selected line, showing on-screen commands;
control + shift + b executes the codes from the beginning of the file to the cursor's location;
control + shift + e executes the codes of the lines where the cursor is until the end of the file.

I suggest using these shortcuts from day one, creating a healthy habit. Those who like to use the *mouse*, an alternate way to execute code is to click the *source* button in the upper-right corner of the code editor.

If you want to run code in a *.R* file within another *.R* file, you can use the `source` command. For example, imagine that you have the main script with your data analysis and another two scripts that performs some support operation such as importing data to R. These operations have been dismembered as a way of organizing the code.

To run the support *scripts*, just call it with function `source` in the main script, as in the following code:

```
# execute import script
source('01-import-data.R')

# execute analysis
source('02-build-tables')
```

Here, all code in `01-import-data.R` and `02-build-tables.R` will be executed sequentially. This equals manually opening each file and hitting *control + shift + s*.

2.8 Testing and Debugging Code

Developing code follows a cycle. At first, you will write a command line on a script, try it using *control + enter* and check the output on the prompt or the content of objects. A new line of code is written once the previous line worked as expected. A moving cycle is clear, writing code is followed by line execution, followed by result checking, modify and repeat if necessary. This is a normal and expected process. You need to make sure that every line of code is correctly specified before moving to the next one.

When trying to find an error in a preexisting script, R offers debugging tools for controlling and assessing its execution. This is especially useful when you have a long and complicated script. The simplest and easiest tool that

R and RStudio offer is code breakpoint. In RStudio, you can click on the left side of the script editor and a red circle will appear, as in Figure 2.7.

```

1 # set x
2 x <- 1
3
4 # set y
● 5 y <- 'My humble text'
6
7 # print contents of x
8 print(x)

```

Figure 2.7: Example of breakpoint in an R script

This red circle indicates a flag that will force the code to stop at that line. You can use it to test existing code and check its objects at a certain part of the execution. Pausing the code at a certain point might seem strange for a starting programmer but, for large scripts, with many functions and complex code, it is a necessity. When the execution hits the breakpoint, the prompt will change to `Browse[1]>` and you'll be able to try new code and verify the content of all current objects. From the Console, you have the option to continue the execution to the next breakpoint or stop it by pressing `shift+f8`. The same result can be achieved using a function called `browser`. Have a look:

```

# set x
x <- 1

# set y
y <- 'My humble text'

browser()

# print contents of x
print(x)

```

The practical result is the same as using RStudio's red circle, but it gives you more control for the case of several commands in the same line.

2.9 Creating Simple Objects

One of the most basic and most used commands in R is the creation of objects. As shown in previous sections, you can define an object using the `<-` command, which is verbally translated to *assign*. For example, consider the following code:

```
# set x
x <- 123

# set my_x, my_y and my_z in one line
my_x <- 1; my_y <- 2; my_z <- 3
```

We can read this code as *the value 123 is assigned to x*. The direction of the arrow defines where the value is stored. For example, using `123 -> x` also works, although this is not recommended as the code becomes less readable. Moreover, notice that you can create objects within the same line by separating the commands using a semi-colon.



Using an arrow symbol `<-` for object definition is specific to R. The reason for this choice was that, at the time of conception of the *S* language, keyboards had a specific key that directly defined the arrow symbol. This means that the programmer only had to hit one key in the keyboard to set the *assign* symbol. Modern keyboards, however, are different. If you find it troublesome to type this symbol, you can use a shortcut as well. In *Windows*, the shortcut for the symbol `<-` is *alt plus -*.

Most programming languages uses a equality symbol (`=`) to define objects and, often, this creates confusion. When using R, you can also define objects with `=`, as in `x = 123`, however, no one should ever recommend it. The equality symbol has a special use within an R code as it defines function arguments, as in `my_1 <- fct(arg1 = 1, arg2 = 3)`. For now, just remember to use `<-` for defining objects. We will learn more about functions and using the equality symbol in a future chapter.

The name of the object is important in R. With the exception of very specific cases, you can name objects as you please. This freedom, however, can work against you. It is desirable to set short object names that make sense to the content of the script and which are simple to understand. This facilitates the understanding of the code by other users and is part of the suggested set of rules for structuring code. Note that all objects created in this book have nomenclature in English and specific format, where the white space between

nouns are replaced by an underscore, as in `my_x <- 1` and `name_of_file <- 'my_data_file.csv'`. We will get back at code structure in chapter 13.

R executes the code looking for objects available in the environment, including functions. You also need to be aware that R is case sensitive. Likewise, object `m` differs from `M`. If we try to access an object that does not exist, R will return an error message and halt the execution of the rest of the code. Have a look:

```
print(z)
```

```
R> Error in print(z): object 'z' not found
```

The error occurred because the object `z` does not exist in the current environment. If we create a variable `z` as `z <- 321` and repeat the command `print(z)`, we will not have the same error message.

2.10 Creating Vectors

In the previous examples, we created simple objects such as `x <- 1` and `x <- 'ABC'`. While this is sufficient to demonstrate the basic commands in R, in practice, such commands are very limited. A real problem of data analysis will certainly have a greater volume of information.

When we gather many elements of the same class, such as `numeric`, into a single object, the result is an atomic vector. An example would be the representation of a series of daily stock prices as an atomic vector of the class `numeric`. Once you have a vector, you can manipulate it any way you want.

Atomic vectors are created in R using the `c` command, which comes from the verb *combine*. For example, if we want to combine the values 1, 2 and 3 in a single object, we can do it with the following command:

```
# create numeric atomic vector
x <- c(1, 2, 3)

# print it
print(x)
```

```
R> [1] 1 2 3
```

The `c` command works the same way for any other class of object, such as `character`:

```
# create character atomic vector
y <- c('text 1', 'text 2', 'text 3', 'text 4')
```

```
# print it
print(y)
```

```
R> [1] "text 1" "text 2" "text 3" "text 4"
```

The only restriction on the use of the `c` command is that all elements must have the same class. If we insert data from different classes in a call to `c()`, R will try to mutate all elements into the same class following its own logic. If the conversion of all elements to a single class is not possible, an error message is returned. Note the following example, where numeric values are set in the first and second element of `x` and a character in the last element.

```
# a mixed vector
x <- c(1, 2, '3')
```

```
# print result of forced conversion
print(x)
```

```
R> [1] "1" "2" "3"
```

The values of `x` are all of type `character`. The use of `class` command confirms this result:

```
# print class of x
class(x)
```

```
R> [1] "character"
```

2.11 Knowing Your Environment and Objects

After using various commands, further development of an R script requires you to understand what objects are available and if their content is as expected. You can find this information simply by looking at the *environment* tab in the upper right corner of RStudio. However, there is a command that shows the same information in the prompt. In order to know what objects are currently available in R's memory, you can use the command `ls`. Note the following example:

```
# set some objects
x <- 1
```

```
y <- 2
z <- 3

# print all objects in the environment
print(ls())
```

R> [1] "x" "y" "z"

Objects `x`, `y` and `z` were created and are available in the current working environment. If we had other objects, they would also appear in the output to `ls`.

To display the content of each object, just enter the names of objects and press `enter` in the *prompt*:

```
# print objects by their name
x
```

R> [1] 1

y

R> [1] 2

z

R> [1] 3

Typing the object name on the screen has the same effect as using the `print` command. In fact, when executing the sole name of a variable in the prompt or script, R internally passes the object to the `print` function.

In R, all objects belong to a class. As previously mentioned, to find the class of an object, simply use the `class` function. In the following example, `x` is an object of the class `numeric`, `y` is a text (`character`) object and `fct_example` is a function object.

```
# set objects
x <- 1
y <- 'a'
fct_example <- function(){}

# print their classes
print(class(x))
```

R> [1] "numeric"

```
print(class(y))  
  
R> [1] "character"  
print(class(fct_example))  
  
R> [1] "function"
```

Another way to learn more about an object is to check its textual representation. Every object in R has this property and we can find it with function **str**:

```
# set vec  
x <- 1:10  
# print the textual representation of a vector  
print(str(x))
```

```
R> int [1:10] 1 2 3 4 5 6 7 8 9 10  
R> NULL
```

We find that object **x** is a vector of class **int** (integer). Function **str** is particularly useful when trying to understand the details of a more complex object, such as a **dataframe** or a **list**.

2.12 Displaying and Formatting Output

You can show the value of an R object on the screen in two ways. You can either enter its name in the prompt or use the **print** function. Explaining it further, the **print** function focuses on the presentation of objects and can be customized for any type. For example, if we had an object of a class called **My_Table** to represent a specific type of table, we could create a function called **print.My_Table** that would show a table on the screen with a special format for the rows and column names. Function **print**, therefore, is oriented towards presenting objects and the user can customize it for different classes. The **base** package, which is automatically initialized with R, contains several **print** functions for various kinds of objects, such as **numeric** and **character**.

However, there are other specific functions to display text in the prompt. The main one is **message**. This function takes a text as input, processes it for specific symbols and displays the result on the screen. Function **message** is far more powerful and customizable than **print**.

For example, if we wanted to show the text, **The value of x is equal to 2** on-screen using a numerical object, we could do it as follows:

```
# set x
x <- 2

# print customized message
message('The value of x is', x)
```

R> The value of x is 2

You can also customize the screen output using specific commands. For example, if we wanted to break a line in the screen output, we could do it through the use of the reserved character \n:

```
# set text with break line
my_text <- 'First Line,\n Second line'

# print it
message(my_text)
```

R> First Line,
R> Second line

Note that the use of `print` would not result in the same effect as this command displays the text as it is, without processing it for specific symbols:

```
print(my_text)
```

R> [1] "First Line,\n Second line"

Another example in the use of specific commands for text is to add a *tab* space with the symbol \t. See an example next:

```
# set char with \t
my_text_1 <- 'A and B'
my_text_2 <- '\tA and B'
my_text_3 <- '\t\tA and B'

# print with message()
message(my_text_1)
```

R> A and B

```
message(my_text_2)
```

R> A and B

```
message(my_text_3)
```

```
R> A and B
```

We've only scratched the surface of the possible ways to manipulate text output. More details are found in the official R manual¹⁰.

2.12.1 Customizing the Output

Another way to customize text output is by using specific functions to manipulate objects of class `character`. For that, there are two very useful functions: `paste` and `format`.

Function `paste` *glues* a series of character objects. It is a very useful function and will be used intensively for the rest of the examples in this book. Consider the following example:

```
# set some text objects
my_text_1 <- 'I am a text'
my_text_2 <- 'very beautiful'
my_text_3 <- 'and informative.'

# paste all objects together and print
message(paste(my_text_1, my_text_2, my_text_3))
```

```
R> I am a text very beautiful and informative.
```

The previous result is not far from what we did in the example with the `print` function. Note, however, that the `paste` function adds a space between each text. If we did not want this space, we could use the function `paste0` as in:

```
# example of paste0
message(paste0(my_text_1, my_text_2, my_text_3))
```

```
R> I am a textvery beautifuland informative.
```



An alternative to the `message` function is `cat` (*concatenate and print*). It is not uncommon to find code where messages to the user are transmitted with `cat` and not `message`. As a rule of thumb, give preference to `message` which provides a output that is easier to control. For example, if the user wants to silence a function, omitting all outputs from the screen, he/she could just use the command `suppressMessages`.

Another very useful possibility with the `paste` function is to insert a text or

¹⁰<https://cran.r-project.org/doc/manuals/R-lang.html#Literal-constants>

symbol between the junction of texts. For example, if we wanted to add a comma (,) between each item to be pasted, we could do this by using the input option `sep` as follows:

```
# example using the argument sep
message(paste(my_text_1, my_text_2, my_text_3, sep = ', '))
```

R> I am a text, very beautiful, and informative.

If we had an atomic vector with all elements to be glued in a single object, we could achieve the same result using the `collapse` argument. See an example next.

```
# set character object
my_text <- c('I am a text', 'very beautiful', 'and informative.')
```



```
# example of using the collapse argument in paste
message(paste(my_text, collapse = ' '))
```

R> I am a text, very beautiful, and informative.

Another key feature of the `paste` command is that also works for vectors. For example, let's say I needed to create a series of text objects containing the phrase "My value is equal to X", where "X" goes from 1 to 10. We can do the following:

```
# set size and vector
my_size <- 10
my_vec <- 1:my_size

# define string vector
my_str <- paste0('My value is equal to ', my_vec)

# print it
print(my_str)
```

R> [1] "My value is equal to 1" "My value is equal to 2"
R> [3] "My value is equal to 3" "My value is equal to 4"
R> [5] "My value is equal to 5" "My value is equal to 6"
R> [7] "My value is equal to 7" "My value is equal to 8"
R> [9] "My value is equal to 9" "My value is equal to 10"

Going forward, command `format` is used to format numbers and dates. It is especially useful when we want to represent numbers in a more visually appealing way. By definition, R presents a set number of digits after the decimal point:

```
# example of decimal points in R
message(1/3)
```

R> 0.3333333333333333

If we wanted only two digits on the screen, we could use the following code:

```
# example of using the format on numerical objects
message(format(1/3, digits=2))
```

R> 0.33

Likewise, if we wanted to use a scientific format in the display, we could do the following:

```
# example of using a scientific format
message(format(1/3, scientific=TRUE))
```

R> 3.333333e-01

Function `format` has many more options. If you need your numbers to come out in a specific way, have a look at the help manual for this function. It is also a generic function and can be used for many types of objects.

2.13 Finding the Size of Objects

In R, an object size can mean different things but most likely it is defined as the number of individual elements that constitute the object. This information serves not only to assist the programmer in checking possible code errors but also to know the length of iteration procedures such as *loops*, which will be treated in a later chapter of this book.

In R, the size of an object can be checked with the use of four main functions: `length`, `nrow`, `ncol` and `dim`.

Function `length` is intended for objects with a single dimension, such as atomic vectors:

```
# create atomic vector
x <- c(2, 3, 3, 4, 2,1)

# get length of x
n <- length(x)

# display message
message('The length of x is ', n)
```

```
R> The length of x is 6
```

For objects with more than one dimension, such as a matrix, use functions `nrow`, `ncol` and `dim` (dimension) to find the number of rows (first dimension) and the number of columns (second dimension). See the difference in usage below.

```
# create a matrix
M <- matrix(1:20, nrow = 4, ncol = 5)
```

```
# print matrix
print(M)
```

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

```
# calculate size in different ways
my_nrow <- nrow(M)
my_ncol <- ncol(M)
my_n_elements <- length(M)
```

```
# display messages
message('The number of lines in M is ', my_nrow)
```

```
R> The number of lines in M is 4
```

```
message('The number of columns in M is ', my_ncol)
```

```
R> The number of columns in M is 5
```

```
message('The number of elements in M is ', my_n_elements)
```

```
R> The number of elements in M is 20
```

The `dim` function shows the dimension of the object, resulting in a numeric vector as output. This function should be used when the object has more than two dimensions. In practice, however, such cases are rare as most data-related problems can be solved with a bi-dimensional representation. An example is given next:

```
# get dimension of M
my_dim <- dim(M)
```

```
# print it
```

```
print(my_dim)
```

```
R> [1] 4 5
```

In the case of objects with more than two dimensions, we can use the `array` function to create the object and `dim` to find its size. Have a look at the next example:

```
# create an array with three dimensions
my_array <- array(1:9, dim = c(3, 3, 3))

# print it
print(my_array)
```

```
R> , , 1
```

```
R>
```

```
R>      [,1] [,2] [,3]
R> [1,]    1    4    7
R> [2,]    2    5    8
R> [3,]    3    6    9
```

```
R>
```

```
R> , , 2
```

```
R>
```

```
R>      [,1] [,2] [,3]
R> [1,]    1    4    7
R> [2,]    2    5    8
R> [3,]    3    6    9
```

```
R>
```

```
R> , , 3
```

```
R>
```

```
R>      [,1] [,2] [,3]
R> [1,]    1    4    7
R> [2,]    2    5    8
R> [3,]    3    6    9
```

```
# display its dimensions
print(dim(my_array))
```

```
R> [1] 3 3 3
```

An important note here is that **the use of functions `length`, `nrow`, `dim` and `ncol` are not intended to discover the number of letters in a text**. This is a common mistake and you should be aware of it. For example, if we had a `character` type of object and we use the `length` function, the

result would be the following:

```
# set text object
my_char <- 'abcde'

# print result of length
print(length(my_char))
```

```
R> [1] 1
```

This occurred because the `length` function returns the number of elements in an object. In this case, `my_char` has only one element. To find out the number of characters in the object, we use the `nchar` function as follows:

```
# find the number of characters in an character object
print(nchar(my_char))
```

```
R> [1] 5
```

2.14 Selecting Elements from an Atomic Vector

After creating an atomic vector of a class, it is possible that the user is interested in only one or more elements of it. For example, if we were updating the value of an investment portfolio, our interest is only for the latest price of the stocks. All values from other dates are not relevant to our analysis and therefore could be safely ignored.

The selection of *pieces* of an atomic vector is called indexing and it is accomplished with the use of square brackets (`[]`). Consider the following example:

```
# set x
my_x <- c(1, 5, 4, 3, 2, 7, 3.5, 4.3)
```

If we wanted only the third element of `my_x`, we use the bracket operator as follows:

```
# get the third element of x
elem_x <- my_x[3]

# print it
print(elem_x)
```

```
R> [1] 4
```

Indexing also works using vectors containing the desired locations. If we are only interested in the last and penultimate values of `my_x`, we use the following code:

```
# set vector with indices
my_idx <- (length(my_x)-1):length(my_x)

# get last and penultimate value of my_x
piece_x_1 <- my_x[my_idx]

# print it
print(piece_x_1)
```

R> [1] 3.5 4.3

A cautionary note: **a unique property of the R language is that if a non-existing element of an object is accessed, the program returns the value NA (*not available*)**. See the next example code, where we attempt to obtain the fourth value of a vector with only three components.

```
# set object
my_vec <- c(1, 2, 3)

# print non-existing fourth element
print(my_vec[4])
```

R> [1] NA

This is important because NA elements are contagious. That is, anything that interacts with NA will also become NA. The lack of treatment of these errors can lead to problems that are difficult to identify. In other programming languages, attempting to access non-existing elements generally returns an error and cancels the execution of the rest of the code.



Generally, the occurrence of NA (*Not Available*) values suggests a code problem. Always remember that NA indicates lack of data and are contagious: anything that interacts with an NA value will turn into another NA. **You should become suspicious about your code every time that NA values are found unexpectedly.** A manual inspection in the length and indexation of vectors may be required.

The use of indices is very useful when you are looking for items of a vector that satisfy some condition. For example, if we wanted to find out all values

in `my_x` that are greater than 3, we could use the following command:

```
# find all values in my_x that is greater than 3
piece_x_2 <- my_x[my_x>3]

# print it
print(piece_x_2)
```

```
R> [1] 5.0 4.0 7.0 3.5 4.3
```

It is also possible to index elements by more than one condition using the logical operators `&` and `|` (*or*). For example, if we wanted the values of `my_x` greater than 2 **and** lower than 4, we could use the following command:

```
# find all values of my_x that are greater than 2 and lower then 4
piece_x_3 <- my_x[ (my_x > 2) & (my_x < 4) ]
print(piece_x_3)
```

```
R> [1] 3.0 3.5
```

Likewise, if we wanted all items that are lower than 3 **or** greater than 6, we use:

```
# find all values of my_x that are lower than 3 or higher than 6
piece_x_4 <- my_x[ (my_x < 3) | (my_x > 6) ]

# print it
print(piece_x_4)
```

```
R> [1] 1 2 7
```

Moreover, logic indexing also works with the interaction of different objects. That is, we can use a logical condition in one object to select items from another:

```
# set my_x and my.y
my_x <- c(1, 4, 6, 8, 12)
my_y <- c(-2, -3, 4, 10, 14)

# find all elements of my_x where my.y is higher than 0
my_piece_x <- my_x[my_y > 0]

# print it
print(my_piece_x)
```

```
R> [1] 6 8 12
```

Looking more closely at the indexing process, it is worth noting that, when we use a data indexing condition, we are in fact creating a variable of the `logical` type. This object takes only two values: `TRUE` and `FALSE`. Have a look in the code presented next, where we create a `logical` object, print it and present its class.

```
# create a logical object
my_logical <- my_y > 0

# print it
print(my_logical)
```

```
R> [1] FALSE FALSE TRUE TRUE TRUE
```

```
# find its class
class(my_logical)
```

```
R> [1] "logical"
```

Logical objects are very useful whenever we are testing a particular condition on a data set. We will learn more about this and other basic classes in chapter 7.

2.15 Removing Objects from the Memory

After creating several variables, the R environment can become full of used and disposable content. In this case, it is desirable to clear the memory to erase objects that are no longer needed. Generally, this is accomplished at the beginning of a script, so that every time the script runs, the memory will be cleared before any calculation. In addition to cleaning the computer's memory, it also helps to avoid possible errors in the code. In most cases, cleaning the working environment should be performed only once at the beginning of the script.

For example, given an object `x`, we can delete it from memory with the command `rm`, as shown next:

```
# set x
x <- 1

# remove x
rm('x')
```

After executing the command `rm('x')`, the value of `x` is no longer available in the R session. In practical situations, however, it is desirable to clean up

all the memory used by all objects created in R. We can achieve this goal with the following code:

```
rm(list = ls())
```

The term `list` in `rm(list = ls())` is a function argument of `rm` that defines which objects will be deleted. The `ls()` command shows all the currently available objects. Therefore, by chaining together both commands, we erase all current objects available in the environment. As mentioned before, it is a good programming policy to clear the memory before running the script. However, you should only wipe out all of R's memory if you have already saved the results of interest or if you can replicate them.



Clearing memory in *scripts* is a controversial topic. Some authors argue that it is better not to clear the memory as this can erase important results. In my opinion, I think it is important to clear the memory at the top of the script, as long as all results are reproducible. When you start a code in a clean state – no variables or functions – it becomes easier to understand and solve possible *bugs*.

2.16 Displaying and Setting the Working Directory

Like other programming platforms, **R always works in a directory**. If no directory is set, a default value is used when R starts up. It is based on the working directory that R searches for files to load data or other R scripts. It is in this directory that R saves any output if we do not explicitly define a path on the computer. This output can be a graphic file, text or a spreadsheet. **It is a good programming policy is to change the working directory to the same place where the *script* is located.**

The simplest way of checking the current working directory is looking at RStudio's prompt panel. At the top, in a small font and just below the word "Console", you'll see the working path. Using code, we can check the current working directory with function `getwd`:

```
# get current dir
my_dir <- getwd()

# display it
print(my_dir)
```

```
R> C:/Dropbox/06-My Books/afedR-ed2/Book Content
```

The result of the previous code shows the folder in which this book was written and compiled. As you can see, the book files are saved in a sub-folder of my Dropbox directory.

The change of the working directory is performed with the `setwd` command. For example, if we wanted to change our working directory to *C:/My Research/*, simply type in the *prompt*:

```
# set where to change directory
my_d <- 'C:/My Research/'

# change it
setwd(my_d)
```

After changing the directory, importing and saving files in the *C:/My Research/* folder will be a lot easier.

As for simple cases such as the above, remembering the directory name is not difficult. In practical cases, however, the working directory can be in a deeper directory of the file system. A simple and very efficient way of setting the working directory is using the *autocomplete* feature of RStudio. Explained briefly, we can use the *tab* key in the keyboard to navigate the computer's folders within RStudio. Give it a try with the following steps:

- 1) Write command `setwd('')` in a script as it is, with an empty content for the inner character;
- 2) Place your cursor between the ' symbols;
- 3) Press the *tab* key.

Now you'll be able to see your folders in a small window and use the arrows and *enter* key to navigate. The autocomplete feature of RStudio goes much deeper than that. We will learn more about it in a future section of this chapter.

Another, more modern, way of setting the directory is to use the RStudio API package, which contains a set of functions that only work inside RStudio and provides information about current file, project and many more. To find out the path of the current R script being edited in RStudio and set the working directory to there, you can write:

```
my_path <- dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(my_path)
```

This way, the script will change the directory to its own location, no matter where you copy it. Be aware, however, that this trick only works in RStudio's script editor and within a saved file. It will not work from the prompt.

Once you are working on the same path as the script, using relative paths is preferable. For example, if you are working in a folder that contains a subdirectory called `data`, you can enter this sub-folder with the code:

```
# change to subfolder
setwd('data')
```

Another possibility is to go to a previous level of the directory using `..`, as in:

```
# change to the previous level
setwd('..')
```

So, if you are working in directory `C:/My Research/` and execute the command `setwd('..')`, the current folder becomes `C:/`, which is one level below `C:/My Research/`.

2.17 Canceling Code Execution

Whenever R is running some code, a visual cue in the shape of a small red circle in the right corner of the *prompt* will appear. If you read it, the text shows the *stop* word. This button is not only an indicator of running code but also a shortcut for canceling its execution. Another way to cancel an execution is to point the mouse to the *prompt* and press the *escape* (*esc*) button from the keyboard.

To try it out, run the next chunk of code in RStudio and cancel its execution using *esc*.

```
for (i in 1:100) {
  message('\nRunning code (please make it stop by hitting esc!)')
  Sys.sleep(1)
}
```

In the previous code, we used a `for` loop and function `Sys.sleep` to display the message '`\nRunning code (please make it stop by hitting ESC!)`' one hundred times, every second. For now, do not worry about the code and functions used in the example. We will discuss the use of loops in chapter 8.



Another very useful trick for defining working directories in R is to use the ~ symbol. The tilda defines the “Documents” folder in *Windows*, which is unique for each user. Therefore, by running `setwd('~')`, you will direct R to a folder that is easily accessible.

2.18 Code Comments

In R, comments are set using the hashtag symbol `#`. Anything after this symbol will not be processed by R. This gives you the freedom to write whatever you want within the script. An example:

```
# this is a comment (R will not parse it)
# this is another comment (R will again not parse it)

x <- 'abc' # this is an inline comment
```

Comments are an effective way to communicate any important information that cannot be directly inferred from the code. In general, you should avoid using comments that are too obvious or too generic:

```
# read CSV file
df <- read.csv('data/data_file.csv')
```

As you can see, it is quite obvious from the line `df <- read.csv('..')` that the code is reading a .csv file. The name of the function already states that. So, the comment was not a good one as it did not add any new information to the user. A better approach at commenting would be to set the author, description of the script and better explain the origin and last update of the data file. Have a look:

```
# Script for reproducing the results of JOHN (2019)
# Author: Mr data analyst (dontspamme@emailprovider.com)
# Last script update: 2020-01-10
#
# File downloaded from www.site.com/data-files/data_file.csv
# The description of the data goes here
# Last file update: 2020-01-10

df <- read.csv('data/data_file.csv')
```

So, by reading the comments, the user will know the purpose of the script, who wrote it and the date of the last edit. It also includes the origin of the data file and the date of the latest update. If the user wants to update the data, all he has to do is go to the referred website and download the new

file.

Another productive use of comments is to set sections in the code, such as in:

```
# Script for reproducing the results of JOHN (2019)
# Author: Mr data analyst (dontspamme@emailprovider.com)
# Last script update: 2020-01-10
#
# File downloaded from www.site.com/data-files/data_file.csv
# The description of the data goes here
# Last file update: 2020-01-10

# Clean data -----
# - remove outliers
# - remove unnecessary columns

# Create descriptive tables -----

# Estimate models -----

# Report results -----
```

The use of a long line of dashes (-) at each section of the code is intentional. It causes RStudio to identify and bookmark the sections, with a link to them at the bottom of the script editor. Test it yourself, copy and paste the above code into a new R script, save it, and you'll see that the sections appear on a button between the editor and the prompt. Such a shortcut can save plenty of time in lengthy scripts.



When you start to share code with other people, you'll soon realize that comments are essential and expected. They help transmit information that is not available from the code. This is one way of a discerning novice from experienced programmers. The latter is always very communicative in its comments (sometimes too much!). A note here, throughout the book you'll see that the code comments are, most of the time, a bit obvious. This was intentional as clear and direct messages are important for new users, which is part of the audience of this book.

2.19 Looking for Help

A common task in the use of R is to seek help. Even advanced users often seek instructions on specific tasks, whether it is to better understand the details of some functions or simply to study a new procedure. The use of the R help system is part of the everyday routine with the software and you should master it as soon as possible.

You can get help by using the *help* panel in RStudio or directly from the *prompt*. Simply enter the question mark next to the object on which you want help, as in `?mean`. In this case, object `means` is a function and the use of the `help` command will open a panel on the right side of RStudio.

In R, the help screen of a function is the same as shown in Figure 2.8. It presents a general description of the function, explains its input arguments and the format of the output. The help screen follows with references and suggestions for other related functions. More importantly, examples of usage are given last and can be copied to the prompt or script in order to accelerate the learning process.

```
mean {base}                                         R Documentation

Arithmetic Mean

Description
Generic function for the (trimmed) arithmetic mean.

Usage
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
x      An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects.
Complex vectors are allowed for trim = 0, only.
trim   the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim
outside that range are taken as the nearest endpoint.
na.rm  a logical value indicating whether NA values should be stripped before the computation proceeds.
```

Figure 2.8: Help screen for function `mean`

If we are looking for help for a given text and not a function name, we can use double question marks as in `?"standard deviation"`. This operation will search for the occurrence of the term in all packages of R and it is very useful to learn how to perform a particular task. In this case, we looked for the available functions to calculate the standard deviation of a vector.

As a suggestion, the easiest and most direct way to learn a new function is by trying out the examples in the manual. This way, you can see which type of input objects the function expects and what type of output it gives. Once you have it working, read the help screen to understand if it does exactly what you expected and what are the options for its use. If the function performs the desired procedure, you can copy and paste the example for your own *script*, adjusting where necessary.

Another very important source of help is the Internet itself. Sites like stackoverflow and specific *mailing lists* and blogs, whose content is also on the Internet, are a valuable source of information. If you find a problem that could not be solved by reading the standard help files, the next logical step is to seek a solution using your error message or the description of the problem in search engines. In many cases, your problem, **no matter how specific it is, has already occurred and has been solved by other users**. In fact, it is more surprising **not** to find the solution for a programming problem on the internet, than the other way around.



Whenever you ask for help on the internet, always try to 1) describe your problem clearly and 2) add a reproducible code of your problem. Thus, the reader can easily verify what is happening by running the example on his computer. I have no doubt that if you respect both rules, a charitable person will soon help you with your problem.

2.20 Using Code Completion with *tab*

A very useful feature of RStudio is *code completion*, an editing tool that facilitates the search of object names, packages, function arguments, and files. Its usage is very simple. After you type any first letter in the keyboard, just press *tab* (left side of the keyboard, above *capslock*) and a number of options will appear. See Figure 2.9 where, after entering the *f* letter and pressing *tab*, a small window appears with a list of object names that begin with that letter.

The autocomplete feature is self-aware and will work differently depending on where it is called. As such, it works perfectly for searching for packages. For that, type `library()` in the prompt or editor, place the cursor in between the parentheses and press *tab*. The result should look something like Figure 2.10, shown next.

Note that a description of the package or object is also offered by the code

```
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.
```

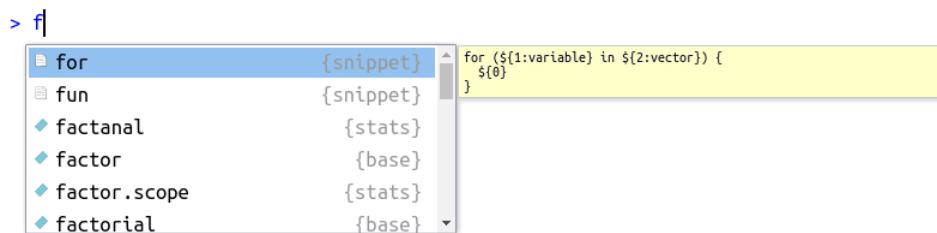


Figure 2.9: Usage of autocomplete for object name

```
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.
```

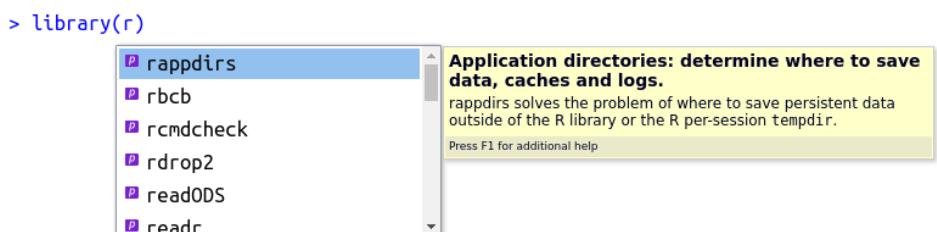


Figure 2.10: Usage of autocomplete for packages

completion tool. This greatly facilitates the day to day work as the memorization of package names and R objects is not an easy task. The use of the *tab* decreases the time to look up names, also avoiding possible coding errors.

The use of this tool becomes even more beneficial when objects and functions are named with some sort of pattern. In the rest of the book, you will notice that objects tend to be named with the prefix *my*, as *my_x*, *my_num*, *my_df*. Using this naming rule (or any other) facilitates the lookup for the names of objects created by the user. You can just type *my_*, press *tab*, and a list of objects will appear.

As mentioned in the previous section, you can also find files and folders on your computer using *tab*. To try it, write the command `my_file <- ""` in the prompt or a script, point the cursor to the middle of the quotes and press the *tab* key. A screen with the files and folders from the current working directory should appear, as shown in Figure 2.11.

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

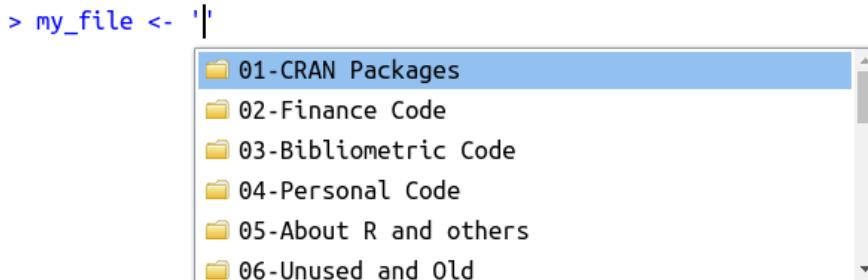


Figure 2.11: Usage of autocomplete for files and folders

The use of autocomplete is also possible for finding the name and description of function arguments. To try it out, write `message()` and place the mouse cursor inside the parentheses. After that, press *tab*. The result should be similar to Figure 2.12. By using *tab* inside of a function, we have the names of all arguments and their description – a mirror of the information found in the help files.

Likewise, you can also search for a function within a package with *tab*. For that, simply type the name of the package followed by two commas, as in `readr::`, and press *tab*. The result should be similar to Figure 2.13

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> sort()
  x =      for sort an R object with a class or a numeric, complex,
decreasing =    character or logical vector. For sort.int, a numeric, complex,
... =          character or logical vector, or a factor.
Press F1 for additional help
```

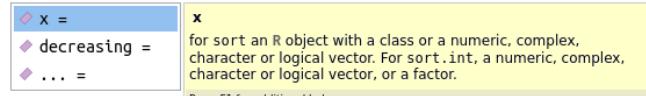


Figure 2.12: Usage of autocomplete for function arguments

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> readr:::
  AccumulateCallback      for AccumulateCallback
  as.col_spec             {readr}
  ChunkCallback           {readr}
  clipboard               {readr}
  col_character           {readr}
  col_date                {readr}
```

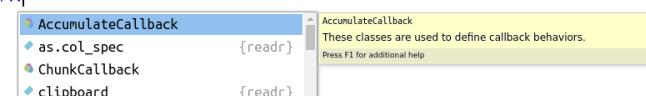


Figure 2.13: Usage of autocomplete for finding functions within a package

Summing up, using code completion will make you more productive. You'll find names of files, objects, arguments, and packages much faster. Use it as much as you can and form a habit out of it.



Autocomplete is one of the most important tools of RStudio, helping users to find object names, locations on the hard disk, packages and functions. Get used to using the *tab* key and, soon enough, you'll see how much the *autocomplete* tool can help you write code quickly, and without typos.

2.21 Interacting with Files and the Operating System

As you are learning R, soon enough you'll find a data-related problem with a demand for interacting with files on the computer, either by creating new folders, decompressing and compressing files, listing and removing files from the hard drive of the computer or any other type of operation. R has full support for such type of operations. You can automate any type of computer task within an R script, if so needed.

2.21.1 Listing Files and Folders

To list files from your computer, use function `list.files`, where the `path` argument sets the directory to list the files from. For the compilation of the book, I've created a directory called `data`. This folder contains all the data needed to recreate the book's examples. You can check the files in the sub-folder `data` with the following code:

```
# list files in data folder
my_files <- list.files(path = "data", full.names = TRUE)
print(my_files)
```

```
R> [1] "data/batchgetsymbols_parallel_example.rds"
R> [2] "data/FileWithLatinChar_ISO-8859-9.txt"
R> [3] "data/FileWithLatinChar_UTF-8.txt"
R> [4] "data/Financial Sample.xlsx"
R> [5] "data/grunfeld.csv"
R> [6] "data/MySQLiteDatabase.SQLITE"
R> [7] "data/pride_and_prejudice.txt"
R> [8] "data/SP500_Excel.xlsx"
R> [9] "data/SP500_long_yearly_2010-01-01_2019-11-04.rds"
```

```
R> [10] "data/SP500-Stocks_long.csv"
R> [11] "data/SP500-Stocks_wide.csv"
R> [12] "data/SP500-Stocks-WithRet.rds"
R> [13] "data/SP500.csv"
R> [14] "data/SQLite_db.SQLITE"
R> [15] "data/temp_file.xlsx"
R> [16] "data/temp_fst.fst"
R> [17] "data/temp_rds.rds"
R> [18] "data/temp_writexl.xlsx"
R> [19] "data/temp_xlsx.xlsx"
R> [20] "data/temp.csv"
R> [21] "data/temp.fst"
R> [22] "data/temp.RData"
R> [23] "data/temp.rds"
R> [24] "data/temp.txt"
R> [25] "data/temp.xlsx"
R> [26] "data/top25babynames-by-sex-2005-2017.csv"
R> [27] "data/UCI_Credit_Card.csv"
```

There are several files with different extensions in this directory. These files contain data that will be used in future chapters. When using `list.files`, it is recommended to set input `full.names` as `TRUE`. This option makes sure that the names returned by the function contain the full path. It facilitates further manipulation, such as reading and importing information from data files. It is worth noting that you can also list the files recursively, that is, list all files from all sub-folders contained in the original address. To check it, try using the following code in your computer:

```
# list all files for all subfolders (IT MAY TAKE SOME TIME...)
list.files(path = getwd(), recursive = T, full.names = TRUE)
```

The previous command will list all files in the current folder and sub-folders. Depending on the current working directory, it may take some time to run it all. If you executed it, be patient or just cancel it pressing `esc` in your keyboard.

To list folders (directories) on your computer, use the command `list.dirs`. See below.

```
# store names of directories
my_dirs <- list.dirs(recursive = F)

# print it
print(my_dirs)
```

```
R> [1] "./_book"                  "./_bookdown_files"
R> [3] "./Rproj.user"           "./vscode"
R> [5] "./afedR_ed_02_cache"   "./afedR_ed_02_files"
R> [7] "./bib-files"            "./blocks"
R> [9] "./css"                  "./data"
R> [11] "./ebook_files"         "./eqs"
R> [13] "./fig_ggplot"          "./figs"
R> [15] "./ftp_files"           "./gdfpd2_cache"
R> [17] "./html_code"           "./images"
R> [19] "./js"                  "./latex"
R> [21] "./many_datafiles"     "./many_datafiles_2"
R> [23] "./mem_cache"           "./Other_chapters"
R> [25] "./quandl_cache"        "./Scripts"
R> [27] "./simfin_cache"        "./tabs"
```

The command `list.dirs(recursive = F)` listed all directories of the current path without recursion. The output shows the directories used to write the book. It includes the output directory of the book (`./_book`), the directory with the data (`./data`), among others. In this same directory, you can find the chapters of the book, organized by files and based on the *RMarkdown* language (`.Rmd` file extension). To list only files with the extension `.Rmd`, we can use the `pattern` input in function `list.files` as follows:

```
# list all files with the extension .Rmd
list.files(pattern = "*.Rmd")
```

```
R> [1] "_Welcome.Rmd"
R> [2] "00a-About-new-edition.Rmd"
R> [3] "00b-Preface.Rmd"
R> [4] "01-Introduction.Rmd"
R> [5] "02-Basic-operations.Rmd"
R> [6] "03-Research-scripts.Rmd"
R> [7] "04-Importing-exporting-local.Rmd"
R> [8] "05-Importing-internet.Rmd"
R> [9] "06-Data-structure-objects.Rmd"
R> [10] "07-Basic-objects.Rmd"
R> [11] "08-Programming--ONLINE.Rmd"
R> [12] "09-Cleaning-data--ONLINE.Rmd"
R> [13] "10-Figures--ONLINE.Rmd"
R> [14] "11-Models--ONLINE.Rmd"
R> [15] "12-Reporting-results--ONLINE.Rmd"
R> [16] "13-Optimizing-code--ONLINE.Rmd"
R> [17] "14-References.Rmd"
```

```
R> [18] "afedR_ed02-ONLINE.Rmd"
R> [19] "index.Rmd"
```

The files presented above contain all the contents of this book, including this specific paragraph, located in file 02-Basic-operations.Rmd!

2.21.2 Deleting Files and Directories

You can also use an R session to delete files and directories from your computer. This might come in handy when dealing with disposable data files. Use these commands with responsibility. If not careful, you can easily break the operating system of your computer.

You can delete files with command `file.remove`:

```
# create temporary file
my_file <- 'data/tempfile.csv'
write.csv(x = data.frame(x=1:10),
          file = my_file)

# delete it
file.remove(my_file)
```

```
R> [1] TRUE
```

Remember that you must have permission from your operating system to manipulate the hard drive and delete a file. In the previous chunk of code, the returned value `TRUE` tells us that the operation was successful.

To delete directories and all their elements, we use `unlink`:

```
# create temp dir
dir.create('temp')

# create a file inside of temp
my_file <- 'temp/tempfile.csv'
write.csv(x = data.frame(x=1:10),
          file = my_file)

unlink(x = 'temp', recursive = TRUE)
```

Notice that, unlike `file.remove`, function `unlink` returns nothing. If needed, we can check if the deletion of a directory was successful with command `dir.exists`:

```
dir.exists('temp')
```

R> [1] FALSE

As expected, the directory was not found.

2.21.3 Downloading Files from the Internet

We can also use R to download files from the Internet with function `download.file`. See the following example, where we download an Excel spreadsheet from Microsoft's website:

```
# set link
link_dl <- 'go.microsoft.com/fwlink/?LinkID=521962'
local_file <- 'data/temp_file.xlsx' # name of local file

download.file(url = link_dl,
              destfile = local_file)
```

Using `download.file` is quite handy when you are working with Internet data that is constantly updated. Just re-download the file with data at the beginning of the *script*. After that, we could continue the code by reading the downloaded file and performing our analysis.

One trick worth knowing is that you can also download files from cloud services such as Dropbox¹¹ and Google Drive¹². So, if you need to send a data file to a large group of people and update it frequently, just pass the file link from the cloud service. This way, any local change in the data file in your computer will be reflected for all users with the file link.



Needless to say, **be very careful** with commands `file.remove` and `unlink`, especially when using recursion (`recursive = TRUE`). One simple mistake and important parts of your hard drive can be erased, leaving your computer inoperable. Be aware that R **permanently deletes** files and do move them to the trash folder. Therefore, when deleting directories with `unlink`, you will be unable to recover the files easily.

¹¹<https://www.dropbox.com/>

¹²<https://drive.google.com/>

2.21.4 Using Temporary Files and Directories

An interesting aspect of R is that every new session is linked to a temporary folder within the computer. This folder is used to store any disposable files and folders generated by R. The location of this directory is available with `tempdir`:

```
windows_tempdir <- tempdir()
print(windows_tempdir)
```

```
R> C:\Users\NAME\AppData\Local\Temp\Rtmp8E
```

The name of the temporary directory, in this case ‘Rtmp8E’, is randomly defined at the start of every new R session. When the computer is rebooted, all temporary directories are deleted.

The same dynamic is found for file names. If you want to use a temporary random name for some reason, use `tempfile`:

```
windows_tempfile <- tempfile(pattern = 'temp_',
                               fileext = '.xlsx')
message(windows_tempfile)
```

```
R> C:\Users\NAME\AppData\Local\Temp\Rtmp8E\temp_4365730565.xlsx
```

You can also set its extension and name:

```
windows_tempfile <- tempfile(pattern = 'temp_',
                               fileext = '.csv')
message(windows_tempfile)
```

```
R> C:\Users\NAME\AppData\Local\Temp\Rtmp8E\temp_43664e87729.csv
```

As a practical case of using temporary files and folders, let’s *download* the Excel worksheet from Microsoft into a temporary folder and read its content for the first five rows:

```
# set link
link_dl <- 'go.microsoft.com/fwlink/?LinkId=521962'
local_file <- tempfile(fileext = '.xlsx', tmpdir = tempdir())

download.file(url = link_dl,
              destfile = local_file)

df_msft <- readxl::read_excel(local_file)

print(head(df_msft))
```

```
R> # A tibble: 6 x 16
R>   Segment      Country Product Disco~1 Units~2 Manuf~3 Sale ~4
R>   <chr>        <chr>    <chr>    <chr>     <dbl>     <dbl>     <dbl>
R> 1 Government  Canada   Carret~ None      1618.      3       20
R> 2 Government  Germany  Carret~ None      1321       3       20
R> 3 Midmarket   France   Carret~ None      2178       3       15
R> 4 Midmarket   Germany  Carret~ None      888        3       15
R> 5 Midmarket   Mexico   Carret~ None      2470       3       15
R> 6 Government  Germany  Carret~ None      1513       3      350
R> # ... with 9 more variables: `Gross Sales` <dbl>,
R> #   Discounts <dbl>, Sales <dbl>, COGS <dbl>, Profit <dbl>,
R> #   Date <dttm>, `Month Number` <dbl>, `Month Name` <chr>,
R> #   Year <chr>, and abbreviated variable names
R> #   1: `Discount Band`, 2: `Units Sold`,
R> #   3: `Manufacturing Price`, 4: `Sale Price`
```

The example Excel file contains the sales report of a company. Do notice that the imported file becomes a `dataframe` in our R session, a table like an object with rows and columns.

By using `tempfile`, we do not need to delete (or worry) about the downloaded file because it will be removed from the computer's hard disk when the system is rebooted.

2.22 Exercises

All solutions are available at <https://www.mspperl.com/afedR>.

01 - In RStudio, create a new *script* and save it in a personal folder. Now, write R commands in the script that define two objects: one holding a sequence between 1 and 100 and the other with the text of your name (ex. 'Richard'). Execute the whole script with the keyboard shortcuts.

02 - In the previously created *script*, use function `message` to display the following phrase in R's *prompt*: "My name is".

03 - Within the same script, show the current working directory (see function `getwd`, as in `print(getwd())`). Now, change your working directory to *Desktop* (*Desktop*) and show the following message on the *prompt* screen: 'My desktop address is'. Tip: use and abuse of RStudio's *autocomplete* tool to quickly find the *desktop* folder.

04 - Use R to download the compressed zip file with the book material,

available at this link¹³. Save it as a file in the temporary session folder (see function `tempfile`).

05 - Use the `unzip` function to unzip the downloaded file from previous question to a directory called '`afedR-files`' inside the "Desktop" folder. How many files are available in the resulting folder? Tip: use the `recursive = TRUE` argument with `list.files` to also search for all available subdirectories.

06 - Every time the user installs an R package, all package files are stored locally in a specific directory of the hard disk. Using command `Sys.getenv('R_LIBS_USER')` and `list.dirs`, list all the directories in this folder. How many packages are available in this folder on your computer?

07 - In the same topic as previous exercise, list all files in all subfolders in the directory containing the files for the different packages (see command `Sys.getenv('R_LIBS_USER')`). On average, how many files are needed for each package?

08 - Use the `install.packages` function to install the `BatchGetSymbols` package on your computer. After installation, use function `BatchGetSymbols` to download price data for the IBM stock in the last 15 days. Tips: 1) use function `Sys.Date()` to find out the current date and `Sys.Date() - 15` to calculate the date located 15 days in the past; 2) note that the output of `BatchGetSymbols` is a list – a special type of object – and that the price data is located in the second element of that list.

09 - The `cranlogs` package allows access to downloads statistics of CRAN packages. After installing `cranlogs` on your computer, use the `cranlogs::cran_top_downloads` function to check which are the 10 most installed packages by the global community in the last month. Which package comes first? Tip: Set the `cran_top_downloads` function input to `when = 'last-month'`. Also, be aware that the answer here may not be the same as you got because it depends on the day the R code was executed.

- a) textshaping
- b) ggplot2
- c) cli
- d) rlang
- e) dplyr

10 - Using the `devtools` package, install the development version of the `ggplot2` package, available in the Hadley Hickman repository. Load the

¹³<https://www.msperlin.com/files/afedR-files/afedR-code-and-data.zip>

package using `library` and create a simple figure with the code `qplot(y = rnorm(10), x = 1:10)`.

11 - Using your programming ability check on your computer which folder, from the “Documents” directory (shortcut = ~), has the largest number of files. Display the five folders with the largest number of files on R’s prompt.

Writing Research Scripts

So far we learned how to use R for basic tasks such as interacting with the computer, creating simple vectors and downloading files from the internet. Although, before we import large volume tables into R and analyze them, we need to discuss the structure of a research script and, more specifically, how to organize our work efficiently.

An organized code facilitates sharing and future use. As a research code becomes larger and complex, organization is a necessity. In this chapter, I will suggest a way to organize files and folders. So, I recommend that you follow these guidelines – or at least your own version of them – in every project you work on.

3.1 Stages of Research

Unlike other software designs, every research script has clear consecutive steps to achieve its goal.

1. **Importation of data:** Raw (original) data is imported from a local file or the internet. At this stage, no manual data manipulation, such as renaming columns names at a .csv file, should happen. The raw data should be imported “as it is”. Save any required and reproducible manipulation for the next stage.
2. **Cleaning and structuring the data:** The raw data imported in the previous step is further cleaned and structured according to the need of the research. Abnormal records and errors in observations

can be removed or treated. The structure of the data can also be manipulated, binding different datasets and adding other variables of interest. Preferably, at the end of this stage, there should be a couple of final datasets that will be used in the next stage.

3. **Visual analysis and hypothesis testing:** After cleansing and structuring the data, the work continues with implementing the visual analysis and hypothesis testing. Here, you can create graphical representations of the data for your audience and use statistical tools, such as econometric models, to test a particular hypothesis. This is the *heart* of the research and the stage most likely to take more development time.
4. **Reporting the results:** The final stage of a research script is reporting the results. Likely, we will be exporting selected tables and figures from R to a text processing software such as Latex, Writer (LibreOffice) or Word (Microsoft).

Each of the mentioned steps can be structured in a single *.R* script or in several separate files. Using multiple files is preferable when the first steps of the research demand significant processing time. For example, in importing and organizing a large volume database, it is worth the trouble to separate the code in different files. It will be easier to find bugs and maintain the code. Each script will do one job, and do it well.

A practical example would be the analysis of a large dataset of financial transactions. Importing and cleansing the data takes plenty of computer time. A smart organization is to insert these primary data procedures in a *.R* file and save the final objects of this stage in an external file. This local archive serves as a bridge to the next step, hypothesis testing, where the previously created file with clean data is imported. Every time a change is made to the hypothesis testing script, it is unnecessary to rebuild the whole dataset. This simple organization of files saves a lot of time. The underlying logic is simple, isolate the parts of the script that demand more computational time – and less development –, and connect them to the rest of the code using external data files.

If you are working with multiple files, one suggestion is to create a naming structure that informs the steps of the research in an intuitive way. An example would be to name the data importing code as `01-Import-and-clean-data.R`, the modeling code as `02-build-report-models.R` and so on. The practical effect is that using a number in the first letter of the filenames clarifies the order of execution. We can also create a *master* script called `0-run-it-all.R` or `0-main.R`

that runs (`source`) all other scripts. So, every time we make an update to the original data, we can simply run `0-run-it-all.R` and will have the new results, without the need to run each script individually.

3.2 Folder Structure

A proper folder structure also benefits the reproducibility and organization of research. In simple scripts, with a small database and a low number of procedures, it is unnecessary to spend much time thinking about the organization of files, just place all files in the same directory. This is certainly the case for most of the code in this book. More complex programs, with several stages of data cleansing, hypothesis testing, and several sources of data, organizing the file structure is essential.

A suggestion for an effective folder structure is to create a single directory and, within it, create subdirectories for each input and output element. For example, you can create a subdirectory called `data`, where all the original data will be stored, a directory `fig` and `tables`, where figures and tables with final results will be exported. If you are using many custom-written functions in the scripts, you can also create a directory called `R-Fcts` and save all files with function definitions at this location. As for the root of the directory, you should only find the main research scripts there. An example of a file structure that summarizes this idea is:

```
/Capital Markets and Inflation/
  /data/
    stock_indices.csv
    inflation_data.csv
  /figs/
    SP500_and_inflation.png
  /tables/
    Table1_descriptive_table.tex
    Table2_model_results.tex
  /R-Fcts/
    fct_models.R
    fct_clean_data.R
  0-run-it-all.R
  1-import-and-clean-data.R
  2-run-research.R
```

The research code should also be self-contained, with all files available within a sub-folder of the root directory. If you are using many different R packages, it is advisable to add a comment in the first lines of `0-run-it-all.R`

that indicates which packages are necessary to run the code. The most friendly way to inform it is by adding a commented line that installs all required packages, as in `#install.packages('pkg1', 'pkg2', ...)`. So, when someone receives the code for the first time, all he (or she) needs to do is uncomment the line and execute it. External dependencies and steps for their installation should also be informed.

The benefits of this directory format are as follows. If you need to share the code with other researchers, simply compress the directory to a single file and send it to the recipient. After decompressing the file, the structure of the folder immediately informs the user where to change the original data, the order of execution of the scripts in the root folder, and where the outputs are saved. The same benefit goes when you reuse your code in the future, say three years from now. By working smarter, you will be more productive, spending less time with repetitive and unnecessary steps.

An example for the content of file `0-run-it-all.R` would be:

```
# clean up workspace
rm(list=ls())

# close all figure windows created with x11()
graphics.off()

# load packages
library(pkg1)
library(pkg2)
library(pkg3)

# change directory
my_dir <- dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(my_dir)

# list functions in 'R-Fcts'
my_R_files <- list.files(path='R-Fcts',
                          pattern = '*.R',
                          full.names=TRUE)

# Load all functions in R
sapply(my_R_files, source)

# Import data script
source('01-import-and-clean-data.R')
```

```
# run models and report results
source('02-run-research.R')
```

This is the first time we use functions `graphics.off` and `sapply`. The first one simply closes all windows used to display a figure. In a research script, sometimes you may have many graphical windows opened and it is wise to close them all. Command `sapply` will apply a function, in this case `source`, to a series of elements. Here, the practical effect is that all files located at folder `R-Fcts` and with extension `.R` will be executed. In chapter 8 we will learn more about `sapply` and its variants.

Notice that, assuming all packages are installed, no extra step is needed to run the above code successfully on another computer. We could also automate the copy of the figure and table files used in the report with `file.copy`. From there, you can create a link in the text for each figure file. As an example, in LaTeX, you can include a figure file with the command `\includegraphics{file_name_here}`. You can also create a direct link for the figure file in the research folder, although this method is not recommended since it creates an external dependency on the written report. Either way, whenever the main code is executed, all research figures will be automatically updated in the text. If needed, you can also produce table files in different formats using packages `xtable` (Dahl et al., 2019) and `texreg` (Leifeld, 2022). We will go deeper into this subject in chapter 12.

Another way of setting up directories in a research script is by using an RStudio project. For that, open RStudio and go to *File, New Project, New Directory*, and choose a folder and project name. RStudio will create a file with the `.RProj` extension in the chosen directory. Every time you want to work on the project, just open the project file in RStudio.

The benefit of this approach is that it is unnecessary to change the directory in the code. The project automatically changes it to the location of the `.RProj` file. Various information is saved, including the history of files being edited, a history of project commands, among other things. Besides that, custom options for the project can also be made. It's worth noting that `.RProj` file is pure text and editable. You can open it in any text editor of your operating system and see how it stores many startup options for R.

3.3 Important Aspects of a Research Script

In this section I'll be making some suggestions for how you can conduct research with R. Making it clear, these are personal positions from my experience as a researcher and teacher. Many points raised here are specific to the academic environment but can be easily extended to the practice of research outside universities. In short, these are suggestions I wish I knew when I first started my career.

Firstly, **know your data!**. I can't stress enough how this is important! The first instinct of every passionate data analyst when encountering a new set of tables is to immediately import it into R and perform an analysis. However, a certain level of caution is needed. Every time you come across a new set of data, ask yourself how much you **really** know about this data:

- How was the data collected? To what purpose?
- How do the available data compare with data used in other studies?
- Is there any possibility of bias within the data collection?

Furthermore, you need to remember that the ultimate goal of any research is communication. Thus, it is very likely that you will report your results to people who will have an informed opinion about the subject, including the sources and individualities of different datasets. The worst case scenario is when a research effort of three to six months in between coding and writing is nullified by a simple lapse in data checking. Unfortunately, this is not uncommon.

As an example, consider the case of analyzing the long term performance of companies in the retail business. For that, you gather a recent list of available companies and download financial records about their revenue, profit and adjusted stock price for the past twenty years. Well, the problem here is in the selection of the companies. By taking those that are available today, you missed all companies that went bankrupt during the 20 year period. By looking only at companies that stayed active during the whole period, you indirectly selected those that are profitable and presented good performance. This is a well-known effect called **survival bias**. The right way of doing this research is gathering a list of companies in the retail business twenty years ago and keep track of those that went bankrupt and those that stayed alive.

The message is clear. **Be very cautious about the data you are using.** Your raw tables stand at the base of the research. A small detail that goes unnoticed can invalidate your whole work. If you are lucky and the database

is accompanied by a written manual, break it down to the last detail. If the information is not clear, do not be shy about sending questions to the responsible party. Likewise, if there is an inevitable operational bias in your dataset, be open and transparent about it.

The second point here is the code. After you finish reading this book, you will have the knowledge to conduct research with R. The computer will be a powerful ally in making your research ideas come true, no matter how gigantic they may be. However, **a great power comes with great responsibility**. Likewise, you need to be aware that a single misplaced line in a code can easily bias and invalidate your research.

Remember that analyzing data is your profession and **your reputation is your most valuable asset**. If you have no confidence in the produced code, do not publish or communicate your results. The code is entirely your responsibility and no one else's. Check it as many times as necessary. Always be skeptical about your own work:

- Do the descriptive statistics of the variables faithfully report the database?
- Is there any relationship between the variables that can be verified in the descriptive table?
- Do the main findings of the research make sense to the current literature of the subject? If not, how to explain them?
- Is it possible that a *bug* in the code has produced the results?

I'm constantly surprised by how many studies submitted to respected periodicals can be denied publication based on a simple analysis of the descriptive table. Basic errors in variable calculations can be easily spotted with a trained eye. The process of continuous evaluation of your research will not only make you stronger as a researcher but will also serve as practice for peer evaluation, much used in academic research. If you do not have enough confidence to report results, test your code extensively. If you have already done so and are still not confident, identify the lines of code you have doubts and seek help with a colleague or your advisor, if there is one. The latter is a strong ally who can help you in dealing with problems he/she already had.

All of the research work is, to some extent, based on existing work. Today it is extremely difficult to carry out ground-breaking research. Knowledge is built in the form of blocks, one over the other. There is always a collection of literature that needs to be consulted. Particularly in the case of data research. Therefore, you should always compare your results with the results already presented in the subject, especially when it is replicated. If the main

results are not similar to those found in the literature, one should ask himself: could a code error have created this result?

I clarify that it is possible that the results of research differ from those of the literature, but the opposite is more likely. Knowledge of this demands care with your code. *Bugs* and code errors are quite common, especially in early versions of scripts. As a data analyst, it is important to recognize this risk and manage it.

3.4 Exercises

All solutions are available at <https://www.msperlin.com/afedR>.

01 - Imagine a survey involving analyzing your household budget over time. Data are available in electronic spreadsheets separated by month, for 10 years. The objective of the research is to understand if it is possible to purchase a real state property in the next five years. Within this setup, detail in text the elements in each stage of the study, from importing the data to the construction of the report.

02 - Based on the study proposed earlier, create a directory structure on your computer to accommodate the study. Create mock files for each subdirectory (see directory structure at section 3.2). Be aware you can create mock files and direction all in R (see functions `cat` and `dir.create`).

Importing Data from Local Files

Surely, the very first step of script is getting your data into R. In this chapter, we will learn to import and export data available as local files in the computer. Although the task is not particularly difficult, a good data analyst should understand the different characteristics of file formats and how to take the best advantage of them in every situation. While some are best suited for sharing and collaboration, others can offer a significant boost in reading and writing speeds.

Here we will draw a comprehensive list of file formats for importing and exporting data, including:

- Text data with comma-separated values (*csv*);
- Microsoft Excel (*xls*, *xlsx*);
- R native files (*RData*, *rds*);
- **fst** format;
- SQLite;
- Unstructured text data.

The first lesson in importing data from local files is that the location of the file must be explicitly stated in the code. The path of the file is then passed to a function that will read the file. The best way to work with paths is to use the *autocomplete* feature of RStudio (see section 2.20). An example of full path is:

```
my_file <- 'C:/My Research/data/SP500_Data.csv'
```

Note the use of forwarding slashes (/) to designate the file directory. Relative references also work, as in:

```
my_file <- 'data/SP500_Data.csv'
```

Here, it is assumed that in the current working folder there is a directory called `data` and, inside of it, exists a file called `SP500_Data.csv`. If the file path is simply its name, such as in `my_file <- 'SP500_Data.csv'`, it is implicitly assumed that the file is located in the root of the working directory. From the previous chapter, recall you can use `setwd` to change the working folder to where the work is being done and simply use the relative path of the data file. An example:

```
setwd('C:/My Research')
my_file <- 'data/SP500_Data.csv'
```



Here, again, I reinforce the use of `tab` and the *autocomplete* tool of RStudio. It is much more **easier and practical** to find files on your computer's hard disk using `tab` navigation than to copy and paste the address from your file explorer. To use it, open double or quotes in RStudio, place the *mouse* cursor in between the quotes and press `tab`.

Another very important point here is that **the data will be imported and exported in R as an object of type `dataframe`**. That is, a table contained in an Excel or `.csv` file will become a `dataframe` object in R. When we export data, the most common format is this same type of object. Conveniently, `dataframes` are nothing more than tables, with rows and columns.

Each column in the `dataframe` will have its own class, the most common being numeric (`numeric`), text (`character`), factor (`factor`) and date (`Date`). When importing the data, **it is imperative that each column is represented in the correct class**. A vast amount of errors can be avoided by simply checking the column classes in the `dataframe` resulting from the import process. For now, we only need to understand this basic property of `dataframes`. We will study the details of this object in chapter 6.

4.1 `csv` files

Consider a data file called `SP500.csv`, available in the `book` package. It contains daily closing prices of the SP500 index from 2010-01-04 until 2020-12-30. We will now use package `afedR` for finding the file and copying it to your local folder. If you followed the instructions in the book preface

chapter, you should have package `afedR` already installed. If not, execute the following code:

```
# install devtools dependency
install.packages('devtools')

# install book package
devtools::install_github('msperlin/afedR')
```

Once you installed package `afedR`, file `SP500.csv` and all other data files used in the book were downloaded from Github. The package also includes functions for facilitating the reproduction of code examples. Command `afedR::get_data_file` will return the local path of a book data file by its name.

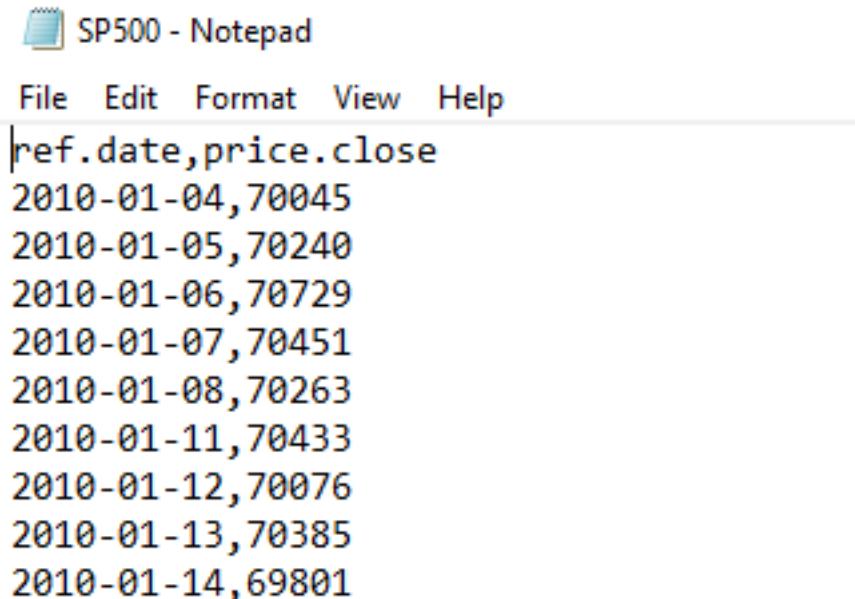
Let's copy `SP500.csv` to your "My Documents" folder with the following code using the tilde (~) shortcut:

```
my_f <- afedR::get_data_file('SP500.csv')
file.copy(from = my_f, to = '~')
```

Now, if it is your first time working with `.csv` files, use a file browser (Explorer in Windows) and open `SP500.csv` in the "My Documents" folder with any text editor software such as Notepad (see figure 4.1). The first lines of `SP500.csv`, also called header lines, show the column names. Following international standards, rows are set using line breaks, and all columns are separated by commas (,).

The content of `SP500.csv` is standard and we should have no problem importing the data in R. However, you should be aware this is not always the case. So, if you want to avoid the common issues, I suggest that you use a set of steps that can avoid most problems in importing data from `.csv` files:

- 1) Check the existence of text before the actual data. A standard `.csv` file will only have the contents of a table but, sometimes, you will find a header text with some details about the data. In R, you can control how many lines you skip in the `csv` reading function;
- 2) Verify the existence of names for all columns and if those names are readable;
- 3) Check the symbol for column separation. Normally it is a comma, but you never know for sure;
- 4) For the numerical data, verify the decimal symbol. R will expect it to be a dot. If necessary, you can adjust this information in the CSV



SP500 - Notepad

File Edit Format View Help

ref.date,price.close

2010-01-04,70045
2010-01-05,70240
2010-01-06,70729
2010-01-07,70451
2010-01-08,70263
2010-01-11,70433
2010-01-12,70076
2010-01-13,70385
2010-01-14,69801

Figure 4.1: Ibov.csv in Notepad

reading function.

- 5) Check the encoding of the text file. Normally it is one of UTF-8, Latin1 (ISO-8859) or Windows 1252. These are broad encoding formats and should suffice for most languages. Whenever you find strange symbols in the text columns of the resulting `dataframe`, the problem is due to a difference in encoding. While the file is encoded in a specific format, R is reading it with a different encoding structure. Windows users can check the encoding of a text file by opening it in Notepad++¹. The information about the encoding will be available in the bottom right corner of the Notepad++ editor. Linux and Mac users can find the same information in any advanced text editor software such as Kate².

¹<https://notepad-plus-plus.org/>

²<https://kate-editor.org/>



Whenever you find an unexpected text structure in a `.csv` file, use the arguments of the `csv` reading function to import the information correctly. As a rule of thumb, **never modify raw .csv data manually**. Its far more efficient to use the R code to deal with different structures of `.csv` files. It takes a bit of work, but such a policy will save you a lot of time in the future as, in a couple of months, you are unlikely to remember how you manually cleaned that `.csv` file for your R script.

4.1.1 Importing Data

The `base` package of R includes a native function called `read.csv` for importing data from `.csv` files. However, we will prefer the `tidyverse` alternative, `readr::read_csv` (Wickham et al., 2022c), as it is more efficient and easier to work with. In short, the benefit is that it reads the data very quickly, and it uses clever rules for defining the classes of imported columns.

This is the first package from the `tidyverse` that we will use. Before doing so, it is necessary to install it in your R session. A simple way of installing all `tidyverse` packages as a bundle is as follows:

```
install.packages('tidyverse')
```

After running the previous code, all `tidyverse` packages will be installed on your computer. Once it finishes, let's load the `tidyverse` set of packages.

```
# load library  
library(tidyverse)
```

Back to importing data from `.csv` files, to load the contents of file `SP500.csv` in R, use the `readr::read_csv` function.

```
# set file to read  
my_f <- afedR::get_data_file('SP500.csv')  
  
# read file  
my_df_sp500 <- read_csv(my_f)
```

```
R> Rows: 2718 Columns: 2  
R> -- Column specification -----  
R> Delimiter: ","  
R> dbl  (1): price.close  
R> date (1): ref.date  
R>  
R> i Use `spec()` to retrieve the full column specification for this data.
```

```
R> i Specify the column types or set `show_col_types = FALSE` to quiet this
# print it
print(head(my_df_sp500))

R> # A tibble: 6 x 2
R>   ref.date    price.close
R>   <date>        <dbl>
R> 1 2010-01-04     70045
R> 2 2010-01-05     70240
R> 3 2010-01-06     70729
R> 4 2010-01-07     70451
R> 5 2010-01-08     70263
R> 6 2010-01-11     70433
```

The contents of the imported file becomes a `dataframe` object in R. As mentioned in the previous chapter, each column of a `dataframe` has a class. We can check the classes of `my_df_sp500` using function `glimpse` from package `dplyr`, which is also part of the `tidyverse`:

```
# Check the content of dataframe
glimpse(my_df_sp500)
```

```
R> Rows: 2,718
R> Columns: 2
R> $ ref.date    <date> 2010-01-04, 2010-01-05, 2010-01-06, 2~
R> $ price.close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Note that the column of dates – `ref.date` – was imported as a `Date` vector and the closing prices – `price.close` – as numeric (`dbl`, double accuracy). This is exactly what we expected. Internally, function `read_csv` identifies columns classes according to their content.

Notice how the previous code presented a message entitled `Parsed with column specification:` This message shows how the function sets the attributes of the columns by reading the first 1000 lines of the file. Column `ref.date` was imported as `date` and column `price.close` was imported as `double` (numeric). We can use this information in our own code by copying the text and assigning it to a variable. Have a look:

```
# set cols from import message
my_cols <- cols(ref.date = col_date(),
                  price.close = col_character() )

# read file with readr::read_csv
my_df_sp500 <- read_csv(my_f, col_types = my_cols)
```

As an exercise, Let's import the same data, but use a `character` class for column `date`:

```
# set cols from import message
my_cols <- cols(ref.date = col_character(),
                  price.close = col_character() )

# read file with readr::read_csv
my_df_sp500 <- read_csv(my_f, col_types = my_cols)

# glimpse the dataframe
glimpse(my_df_sp500)
```

```
R> Rows: 2,718
R> Columns: 2
R> $ ref.date    <chr> "2010-01-04", "2010-01-05", "2010-01-0~
R> $ price.close <chr> "70045", "70240", "70729", "70451", "7~
```

As expected, both columns are of class `character`. So, a possible set of steps using `readr::read_csv` is, first, to read the file without arguments in `read_csv`, copy the default column classes from the output message, add it as argument `col_types`, and re-execute the script. This is handy when the imported file has several columns and manually defining each column class requires lots of typing.

There is also a simpler way of using the classes defined by `read_csv`, just set `col_types = cols()`. This way you don't need to manually copy and paste the message from `read_csv`.

```
# read file with readr::read_csv
my_df_sp500 <- read_csv(my_f,
                        col_types = cols())

# glimpse the dataframe
glimpse(my_df_sp500)
```

```
R> Rows: 2,718
R> Columns: 2
R> $ ref.date    <date> 2010-01-04, 2010-01-05, 2010-01-06, 2~
R> $ price.close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Going further, `read_csv` has several other input options such as:

- change the format of the import data, including symbols for decimal

- places and encoding (`locale` option);
- change column names (argument `col_names`);
- skip n lines before importation (`skip` option);
- custom definition for NA values (`na` option)

Now, let's study a more abnormal case of `.csv` file. In the book package we have a file called `funky_csv_file.csv` where:

- the header has textual information;
- the file will use the comma as a decimal;
- the file text will contain Latin characters.

The first 5 lines of the files contain the following content:

```
R> Example of csv file with alternative format:  
R> - columns separated by ";"  
R> - decimal as ","  
R>  
R> Data compiled in 2021-02-26
```

We ave a header text up to line number 7 and the columns being separated by a semicolon (";"). When importing the data with standard (and wrong) options, we will have the following result:

```
my_f <- afedR::get_data_file('funky_csv_file.csv')  
  
df_funky <- read_csv(my_f,  
                      col_types = cols())  
  
glimpse(df_funky)
```

```
R> Rows: 2  
R> Columns: 1  
R> $ `Example of csv file with alternative format:` <chr> "- ~
```

Clearly something went wrong, with the issue of several *warning* messages. To solve it, we use the following code, making sure the particularities of the file are handled:

```
df_not_funky <- read_delim(  
  file = my_f,  
  skip = 7, # how many lines do skip  
  delim = ';', # column separator  
  col_types = cols(), # column types
```

```

    locale = locale(decimal_mark = ',')# locale
)

glimpse(df_not_funky)

R> Rows: 100
R> Columns: 6
R> $ COD.UF      <dbl> 35, 21, 35, 35, 41, 31, 31, 21, 29, 26,~
R> $ COD          <dbl> 3546306, 2103109, 3514700, 3538105, 411~
R> $ NOME         <chr> "Santa Cruz das Palmeiras", "Cedral", "~"
R> $ state        <chr> "São Paulo", "Maranhão", "São Paulo", "~"
R> $ SIGLA        <chr> " SP", " MA", " SP", " SP", " PR", " MG~
R> $ number_col   <dbl> 1.902087, 69.808750, 81.509312, 56.8400~
```

Note that the data has now been correctly imported, with the correct column classes. For that, we use the alternative function `readr::read_delim` with custom inputs. Package `readr` also provides several other functions for specific import situations. If the `read_csv` function does not solve your problem in reading some structured text file, surely another function of this package will.

4.1.2 Exporting Data

To write a `.csv` file, use the `readr::write_csv` function. First, we create a new dataframe with some random data:

```

# set the number of rows
N <- 100

# set dataframe
my_df <- data.frame(y = runif(N),
                      z = rep('a',N))

# print it
print(head(my_df))
```

```

R>           y z
R> 1 0.0866294 a
R> 2 0.7850703 a
R> 3 0.4275280 a
R> 4 0.3631070 a
R> 5 0.6539552 a
R> 6 0.2050078 a
```

And now we use `write_csv` to save it in a new `.csv` file:

```
# set file out
f_out <- 'data/temp.csv'

# write to files
write_csv(x = my_df,
          file = f_out)
```

In the previous example, we save the object `my_df` into a file with path `data/temp.csv`. We can read it back and check its contents using `read_csv` once again:

```
# read it
my_df_imported <- read_csv(f_out)
```

```
R> Rows: 100 Columns: 2
R> -- Column specification -----
R> Delimiter: ","
R> chr (1): z
R> dbl (1): y
R>
R> i Use `spec()` to retrieve the full column specification for this data.
R> i Specify the column types or set `show_col_types = FALSE` to quiet this
# print first five rows
print(head(my_df_imported))
```

```
R> # A tibble: 6 x 2
R>       y   z
R>   <dbl> <chr>
R> 1 0.0866 a
R> 2 0.785  a
R> 3 0.428  a
R> 4 0.363  a
R> 5 0.654  a
R> 6 0.205  a
```

As we can see, the data imported from the file is identical to the one created in the previous code chunk.

4.2 *Excel* Files (*xls* and *xlsx*)

Although it is not an efficient or portable data storage format, Excel is a very popular software due to its spreadsheet-like capacities. It is not uncommon for data to be stored and distributed in this format, especially in the finance industry.

The downside of using Excel files for storing data is its low portability and the longer time required to read and write it. This may not be a problem for small tables, but when handling a large volume of data, using Excel files can be very frustrating. If you can, avoid the use of Excel files in your work cycle.

4.2.1 Importing Data

R does not have a native function for importing Excel files; therefore, we must install and use packages to perform this operation. There are several good options including **XLConnect** (Mirai Solutions GmbH, 2022), **xlsx** (Dragulescu and Arendt, 2020), **readxl** (Wickham and Bryan, 2022) and **tidyxl** (Garmonsway, 2022).

Despite their similar goals, each package has its peculiarities. If reading Excel files is important to your work, I strongly advise the study of each package. For example, package **tidyxl** was specially designed to read unstructured Excel files, where the desired information is not contained in a tabular format. Package **XLConnect** allows the user to open a live connection and, from R, control an Excel file, making it possible to export and send data, format cells, and so on.

In this section, we will give priority to package **readxl**, one of the most straightforward packages to use. It also does not require the installation of external software such as *Java*. Let's start with an example. Consider a file called **SP500-Excel.xlsx** that contains the same SP500 data. We can import the information from the file using function **read_excel** from **readxl**:

```
library(readxl)

# set excel file
my_f <- afedR::get_data_file('SP500_Excel.xlsx')

# read excel file
my_df <- read_excel(my_f, sheet = 'Sheet1')
```

```
# print classes
print(sapply(my_df, class))

R> $ref.date
R> [1] "POSIXct" "POSIXt"
R>
R> $price.close
R> [1] "numeric"

# print with head (first five rows)
print(head(my_df))

R> # A tibble: 6 x 2
R>   ref.date      price.close
R>   <dttm>          <dbl>
R> 1 2010-01-04 00:00:00     1133.
R> 2 2010-01-05 00:00:00     1137.
R> 3 2010-01-06 00:00:00     1137.
R> 4 2010-01-07 00:00:00     1142.
R> 5 2010-01-08 00:00:00     1145.
R> 6 2010-01-11 00:00:00     1147.
```

As we can see, one benefit of using Excel files is that the column's classes are directly inherited. If the classes are correct in the Excel file, then they will automatically be correct in R. In our case, the date column of file SP500_Excel.xlsx was correctly set as a `dttm` object, a special type of `DateTime` class. Likewise, even if the Excel file used commas for decimals, the import process would still succeed as the conversion is handled internally.

4.2.2 Exporting Data

Exporting a `dataframe` to an Excel file is also easy. Again, no native function in R performs this procedure. We can, however, use packages `xlsx` and `writexl`. A requisite for using `xlsx` is the installation of Java JDK in the operating system. For Windows users, visit the Java site³ and install it with option *Windows Off-line (64 bits)*. After that, install `xlsx` with command `install.packages('xlsx')` and try loading it with `library(xlsx)`. If you got an error message about `Java`, try rebooting your system.

An example of `xlsx` usage is given next:

³https://www.java.com/pt_BR/

```
library(xlsx)

# create dataframe
N <- 50
my_df <- data.frame(y = seq(1,N), z = rep('a',N))

# set excel file
f_out <- 'data/temp.xlsx'

# write to excel
write.xlsx(x = my_df, file = f_out, sheetName = "my df")
```

If you want to save several `dataframes` into several worksheets of the same Excel file, you must use the input option `append=TRUE` in the call to `write.xlsx`. Otherwise, the function will create a new file on each call and erase all previous content. See the following example, where we export two `dataframes` for two different sheets in the same Excel file:

```
# create two dataframes
N <- 25
my_df_A <- data.frame(y = seq(1, N),
                       z = rep('a', N))

my_df_B <- data.frame(z = rep('b', N))

# set file out
f_out <- 'data/temp.xlsx'

# write in different sheets
write.xlsx(x = my_df_A,
           file = f_out,
           sheetName = "my df A")

write.xlsx(x = my_df_B,
           file = f_out,
           sheetName = "my df B",
           append = TRUE )
```

After executing the code, we can open the excel files and check their contents to see that they mirror the artificial data.

As for package `writexl`, its innovation is that a Java installation is not needed. Writing speed is also significantly increased. See an example next.

```

library(writexl)
# set number of rows
N <- 25

# create random dfs
my_df_A <- data.frame(y = seq(1, N),
                        z = rep('a', N))

write_xlsx(x = my_df_A,
            path = f_out)

```

In order to compare writing performance, let's calculate the difference of time from `xlsx` to `writexl`:

```

library(writexl)
library(readxl)
library(xlsx)

# set number of rows
N <- 2500

# create random dfs
my_df_A <- data.frame(y = seq(1,N),
                        z = rep('a',N))

# set files
my_file_1 <- 'data/temp_writexl.xlsx'
my_file_2 <- 'data/temp_xlsx.xlsx'

# test export
time_write_writexl <- system.time(write_xlsx(x = my_df_A,
                                                path = my_file_1))

time_write_xlsx <- system.time(write.xlsx(x = my_df_A,
                                            file = my_file_2))

# test read
time_read_readxl <- system.time(read_xlsx(path = my_file_1 ))
time_read_xlsx <- system.time(read.xlsx(file = my_file_2,
                                         sheetIndex = 1 ))

```

And now we show the results:

```
# results
my_formats <- c('xlsx', 'readxl')
results_read <- c(time_read_xlsx[3], time_read_readxl[3])
results_write<- c(time_write_xlsx[3], time_write_writexl[3])

# print text
my_text <- paste0('\nTime to WRITE dataframe with ',
                  my_formats, ': ',
                  format(results_write, digits = 4),
                  ' seconds', collapse = '')

message(my_text)
```

```
R>
R> Time to WRITE dataframe with xlsx: 1.660 seconds
R> Time to WRITE dataframe with readxl: 0.013 seconds

my_text <- paste0('\nTime to READ dataframe with ',
                  my_formats, ': ',
                  format(results_read, digits = 4),
                  ' seconds', collapse = '')

message(my_text)
```

```
R>
R> Time to READ dataframe with xlsx: 2.418 seconds
R> Time to READ dataframe with readxl: 0.008 seconds
```

As we can see, even for low-volume data, a dataframe with 2500 rows and 2 columns, the run-time difference is significant. If you are working with large spreadsheets, the use of packages `readxl` and `writexl` for reading and writing Excel files is strongly recommended.

4.3 *RData* and *rds* Files

R offers native formats to write objects to a local file. The great benefit of using both native formats, *RData*, and *rds*, is that the saved file is compact and its access is very fast. The downside is the low portability, i.e., it's difficult to use the files in other software.

The difference between *RData* and *rds* is that the first can save many R objects in a single file, while the latter only one. This, however, is not a hard restriction for the *rds* format as we can incorporate several objects into a single one using a `list`, a special type of object. In practice, a *rds* file can store as many objects as needed.

4.3.1 Importing Data

To create a new *RData* file, use the `save` function. See the following example, where we create a *RData* file with some content, clear R's memory, and then load the previously created file:

```
# set a object
my_x <- 1:100

# set name of RData file
my_file <- 'data/temp.RData'

# save it
save(list = c('my_x'), file = my_file)
```

We can verify the existence of the file with the `file.exists` function:

```
# check if file exists
file.exists(my_file)
```

R> [1] TRUE

As expected, file temp.RData is available.

Importing data from *rds* files is very similar. For that we use function `readr::read_rds`:

```
# set file path
my_file <- 'data/temp.rds'

# load content into workspace
my_y <- read_rds(file = my_file)
```

Comparing the code between using *RData* and *rds* files, note that the *rds* format allows the explicit definition of the output object. The contents of `my_file` in `read_rds` is saved in `my_y`. When we use the `load` function for *RData* files, we cannot name the output directly. This is particularly inconvenient when you need to modify the name of the imported object.



As a suggestion, give preference to the *rds* format, which is more practical, resulting in cleaner code. The difference in speed between one and the other is minimal. The benefit of importing multiple objects into the same *RData* format file becomes irrelevant when using `list` objects, which can incorporate other objects into its content.

4.3.2 Exporting Data

We can create a new *RData* file with command `save`:

```
# set vars
my_x <- 1:100
my_y <- 1:100

# write to RData
my_file <- 'data/temp.RData'
save(list = c('my_x', 'my_y'),
     file = my_file)
```

We can check if the file exists with function `file.exists`:

```
file.exists(my_file)
```

```
R> [1] TRUE
```

The result is `TRUE` as expected.

As for *.rds* files, we save it with function `readr::write_rds`:

```
# set data and file
my_x <- 1:100
my_file <- 'data/temp.rds'

# save as .rds
write_rds(x = my_x,
           file = my_file)

# read it
my_x2 <- read_rds(file = my_file)

# test equality
print(identical(my_x, my_x2))
```

```
R> [1] TRUE
```

Command `identical` tests if both objects are equal. Again, as expected, we find the result to be `TRUE`.

4.4 *fst* files

The *fst* format⁴ is specially designed to enable quick writing and reading time from tabular data, with minimal disk space. Using this format is particularly beneficial when working with large databases in powerful computers. The trick here is the use of all computer cores to import and export data, while all other formats only use one. If you have a computer with several cores, the gain in speed is impressive, as we will soon learn.

4.4.1 Importing Data

Using *fst* file format is similar to the previous cases. We use function `read_fst` from package `fst` (Klik, 2022) to read files:

```
library(fst)

R> fst package v0.9.8
# set file location
my_file <- afedR::get_data_file('temp.fst')

# read fst file
my_df <- read_fst(my_file)
```

```
R> fstcore package v0.9.12
R> (OpenMP detected, using 16 threads)
# check contents
glimpse(my_df)
```

```
R> Rows: 1,000
R> Columns: 1
R> $ x <dbl> 0.70968891, 0.83903044, 0.70026554, 0.78120026, ~
```

As with the other cases, the data from file `temp.fst` is available in the workspace.

4.4.2 Exporting Data

We use function `fst::write_fst` to save dataframes in the *fst* format:

```
library(fst)
```

⁴<http://www.fstpackage.org/>

```
# create dataframe
N <- 1000
my_file <- 'data/temp.fst'
my_df <- data.frame(x = runif(N))

# write to fst
write_fst(x = my_df, path = my_file)
```

4.4.3 Timing the *fst* format

As a test of the potential of the **fst** format, we will now time the read and write time between **fst** and **rds** for a large table: 5.000.000 rows and 2 columns. We will also report the size of the resulting file.

```
library(fst)

# set number of rows
N <- 5000000

# create random dfs
my_df <- data.frame(y = seq(1,N),
                      z = rep('a',N))

# set files
my_file_1 <- 'data/temp_rds.rds'
my_file_2 <- 'data/temp_fst.fst'

# test write
time_write_rds <- system.time(write_rds(my_df, my_file_1 ))
time_write_fst <- system.time(write_fst(my_df, my_file_2 ))

# test read
time_read_rds <- system.time(readRDS(my_file_1))
time_read_fst <- system.time(read_fst(my_file_2))

# test file size (MB)
file_size_rds <- file.size(my_file_1)/1000000
file_size_fst <- file.size(my_file_2)/1000000
```

And now we check the results:

```
# results
my_formats <- c('.rds', '.fst')
```

```

results_read <- c(time_read_rds[3], time_read_fst[3])
results_write<- c(time_write_rds[3], time_write_fst[3])
results_file_size <- c(file_size_rds , file_size_fst)

# print text
my_text <- paste0('\nTime to WRITE dataframe with ',
                  my_formats, ': ',
                  results_write, ' seconds', collapse = '')
message(my_text)

```

```

R>
R> Time to WRITE dataframe with .rds: 1.08700000000005 seconds
R> Time to WRITE dataframe with .fst: 0.1030000000000009 seconds

my_text <- paste0('\nTime to READ dataframe with ',
                  my_formats, ': ',
                  results_read, ' seconds', collapse = '')
message(my_text)

```

```

R>
R> Time to READ dataframe with .rds: 1.0810000000002 seconds
R> Time to READ dataframe with .fst: 0.093999999999941 seconds

my_text <- paste0('\nResulting FILE SIZE for ',
                  my_formats, ': ',
                  results_file_size, ' MBs', collapse = '')
message(my_text)

```

```

R>
R> Resulting FILE SIZE for .rds: 65.01011 MBs
R> Resulting FILE SIZE for .fst: 14.791938 MBs

```

The difference is very impressive! The `fst` not only reads and writes faster but also results in smaller file sizes. Be aware, however, this result is found in a 16 core computer in which the book was compiled. You may not be able to replicate the same result in a more modest machine.



Due to the use of all the computer's cores, the `fst` format is highly recommended when working with large data on a powerful computer. Not only will the resulting files be smaller, but the writing and reading process will be considerably faster.

4.5 SQLite Files

The use of `.csv` or `.rds` files for storing objects has its limits as the size of the files increases. If you are waiting a long time to read a `dataframe` from a file or if you are only interested in a small portion of a large table, you should look for alternatives. Likewise, if you are working in a network of computers from your institution and many people are using the same tables, it makes sense to keep and distribute the information from a central server. This way, every user can access the same information, concurrently.

This brings us to the topic of **database software**. These specific programs usually work with a query language, called *SQL* (*Structured Query Language*). It allows the user to read portions of the data and even manipulate it efficiently. Many options of database software integrate nicely with R. The list includes **MySQL**, **SQLite** and **MariaDB**. Here, we will provide a quick tutorial on this topic using SQLite, which is the easiest one to work.

Before moving to the examples, we need to understand how to use database software. First, R will connect to the database and return a connection object. Based on this connection, we will send queries for importing data using the *SQL* language. The main advantage is we can have a large database of, let's say, 10 GB and only load a small portion of it in R. This operation is also very quick, allowing efficient access to the available tables.

4.5.1 Importing Data

Assuming the existence of an SQLite file in the computer, we can import its tables with package **RSQLite** (Müller et al., 2022):

```
library(RSQLite)

# set name of SQLITE file
f_sqlite <- afedR::get_data_file('SQLite_db.SQLITE')

# open connection
my_con <- dbConnect(drv = SQLite(), f_sqlite)

# read table
my_df <- dbReadTable(conn = my_con,
                      name = 'MyTable1') # name of table in sqlite

# print with str
glimpse(my_df)
```

```
R> Rows: 1,000,000
R> Columns: 2
R> $ x <dbl> 0.007504194, 0.439465174, 0.178387480, 0.9857759~
R> $ G <chr> "B", "B", "B", "B", "A", "B", "A", "B", "B", "B"~
```

It worked. The `dataframe` from table `MyTable1` is exactly as expected.

Another example of using SQLite is with the actual SQL statements. Notice, in the previous code, we used function `dbReadTable` to get the contents of all rows in table `MyTable1`. Now, let's use an SQL command to get from `MyTable2` only the rows where the `G` column is equal to `A`.

```
# set sql statement
my_SQL_statement <- "select * from myTable2 where G='A'""

# get query
my_df_A <- dbGetQuery(conn = my_con,
                       statement = my_SQL_statement)

# disconnect from db
dbDisconnect(my_con)

# print with str
print(str(my_df_A))
```

```
R> 'data.frame':    499522 obs. of  2 variables:
R>   $ x: num  0.0637 0.1982 0.2894 0.7389 0.0669 ...
R>   $ G: chr  "A" "A" "A" "A" ...
R> NULL
```

It also worked, as expected.

In this simple example, we can see how easy it is to create a connection to a database, retrieve tables, and disconnect. If you have to work with large tables, which, in my opinion, is any database that occupies more than 4 GB of your computer memory, it is worth moving it to proper database software. You'll be able to retrieve data faster, without the need of loading the whole database in the computer's memory. If you have a server available in your workplace, I strongly advise learning how to connect to it and use the SQL language to your advantage.

4.5.2 Exporting Data

As an example of exporting data to an SQLite file, let's first create an SQLite database. For that, we will set two large `dataframes` with random data and

save both in an SQLite file using the package `RSQLite`.

```
library(RSQLite)

# set number of rows in df
N = 10^6

# create simulated dataframe
my_large_df_1 <- data.frame(x=runif(N),
                             G= sample(c('A','B'),
                                       size = N,
                                       replace = TRUE))

my_large_df_2 <- data.frame(x=runif(N),
                             G = sample(c('A','B'),
                                       size = N,
                                       replace = TRUE))

# set name of SQLITE file
f_sqlite <- 'data/SQLite_db.SQLITE'

# open connection
my_con <- dbConnect(drv = SQLite(), f_sqlite)

# write df to sqlite
dbWriteTable(conn = my_con, name = 'MyTable1',
             value = my_large_df_1)
dbWriteTable(conn = my_con, name = 'MyTable2',
             value = my_large_df_2)

# disconnect
dbDisconnect(my_con)
```

The TRUE output of `dbWriteTable` indicates everything went well. A connection was opened using function `dbConnect`, and the `dataframes` were written to an SQLite file, called `SQLite_db.SQLITE`. Unlike other database software, SQLite stores data and configurations from a single file, without the need of a formal server. Also, notice how we disconnected from the database using the function `dbDisconnect`.

4.6 Unstructured Data and Other Formats

The previous packages and functions are sufficient for getting most of the work done. Nevertheless, it is worth mentioning that R can also import data from other formats such as SPSS, Stata, Matlab, among many others. If that is your case, I suggest a thorough study of the `foreign` package (R Core Team, 2022).

Another example is the case of importing data from unstructured text files. Reading raw text files is the last resort for data importation. If none of the previous packages can read the data, then it must be parsed line by line. Let's explore this problem.

4.6.1 Importing Data

You can read the contents of a text file with function `readr::read_lines`:

```
# set file to read
my_f <- afedR::get_data_file('pride_and_prejudice.txt')

# read file line by line
my_txt <- read_lines(my_f)

# print 50 characters of first fifteen lines
print(str_sub(string = my_txt[1:15],
              start = 1,
              end = 50))
```

```
R> [1] "The Project Gutenberg EBook of Pride and Prejudice"
R> [2] ""
R> [3] "This eBook is for the use of anyone anywhere at no"
R> [4] "almost no restrictions whatsoever. You may copy i"
R> [5] "re-use it under the terms of the Project Gutenberg"
R> [6] "with this eBook or online at www.gutenberg.org"
R> [7] ""
R> [8] ""
R> [9] "Title: Pride and Prejudice"
R> [10] ""
R> [11] "Author: Jane Austen"
R> [12] ""
R> [13] "Posting Date: August 26, 2008 [EBook #1342]"
R> [14] "Release Date: June, 1998"
```

```
R> [15] "Last Updated: March 10, 2018"
```

In this example, file `pride_and_prejudice.txt` contains the whole content of the book *Pride and Prejudice* by Jane Austen, freely available in the Gutenberg⁵ project. We imported the entire content of the file as a `character` vector named `my_txt`. Each element of `my_txt` is a line from the raw text file. Based on it, we can calculate many things such as the number of lines in the book and the number of times that the name 'Bennet', one of the protagonists, appears in the text:

```
# count number of lines
n_lines <- length(my_txt)

# set target text
name_to_search <- 'Bennet'

# set function for counting words
fct_count_bennet <- function(str_in, target_text) {

  require(stringr)

  n_words <- length(str_locate_all(string = str_in,
                                    pattern = target_text)[[1]])

  return(n_words)
}

# use fct for all lines of Pride and Prejudice
n_times <- sum(sapply(X = my_txt,
                      FUN = fct_count_bennet,
                      target_text = name_to_search))

# print results
my_msg <- paste0('The number of lines found in the file is ',
                 n_lines, '\n',
                 'The word "', name_to_search, '" appears ',
                 n_times, ' in the book.')
message(my_msg)
```

```
R> The number of lines found in the file is 13427.
```

```
R> The word "Bennet" appears 664 in the book.
```

⁵<http://www.gutenberg.org/>

In the example, we once again used `sapply`. In this case, it allowed us to use a function for each element of `my_txt`. We searched and counted the number of times the word “Bennet” was found. Notice we could simply change `name_to_search` for any other name if we wanted to.

4.6.2 Exporting Data

A typical case of exporting unstructured text is saving the log record of a procedure. This is quite simple. Using function `readr::write_lines`, use the input `file` to set the name of the local file and `x` for the actual textual content.

```
# set file
my_f <- 'data/temp.txt'

# set some string
my_text <- paste0('Today is ', Sys.Date(), '\n',
                  'Tomorrow is ', Sys.Date()+1)

# save string to file
write_lines(x = my_text, file = my_f, append = FALSE)
```

In the previous example, we created a simple text object and saved it in `data/temp.txt`. We can check the result with the `readr::read_lines` function:

```
print(read_lines(my_f))
```

```
R> [1] "Today is 2022-11-23"      "Tomorrow is 2022-11-24"
```

As we can see, it worked as expected.

4.7 How to Select a Format

The choice of file format is an important topic and might actually be a time-saver at your work. In that decision, we must consider three points:

- speed of reading and writing operations;
- size of the resulting file;
- compatibility with other software and operating systems.

Usually, the use of `csv` files easily satisfies these requirements. A `csv` file is nothing more than a text file that can be opened, viewed, and imported

into any other statistical software. This makes it easy to share it with other people. Also, the size of *csv* files is usually not restrictive and, if needed, it can be compressed using the *zip* function. For these reasons, the use of *csv* files for importing and exporting data is preferable in the vast majority of situations.

However, there are cases where the execution speed of import and export operations matter. If you don't mind giving up portability, the *rds* format is a great choice for most projects. If you have good hardware and execution speed with *rds* is still not great, then the best alternative is the *fst* format, which uses all cores to import and export data. Finally, if you can, my most important suggestion is to **avoid the Excel format at all costs**.

4.8 Exercises

All solutions are available at <https://www.mspperl.com/afedR>.

01 - Create a **dataframe** with the following code:

```
library(dplyr)

my_N <- 10000
my_df <- tibble(x = 1:my_N,
                 y = runif(my_N))
```

Export the resulting **dataframe** to each of the five formats: *csv*, *rds*, *xlsx*, *fst*. Which of the formats took up the most space in the computer's memory? Tip: **file.size** calculates the size of files within R.

02 - Improve the previous code by measuring the execution time for saving the data in different formats. Which file format resulted in the fastest execution for exporting data? Tip: use the **system.time** function or the **tic toc** package to calculate the execution times.

03 - For the previous code, reset the value of **my_N** to 1000000. Does it change the answers to the last two questions?

04 - Use **afedR::get_data_file** function to access the **SP500.csv** file in the book's data repository. Import the contents of the file into R with the function **readr::read_csv**. How many lines are there in the resulting **dataframe**?

05 - Within link <https://eeecon.uibk.ac.at/~zeileis/grunfeld/Grunfeld.csv> you will find a *.csv* file for the *Grunfeld* data. This is a particularly famous table due to its use as reference data in econometric models. Us-

ing `readr::read_csv` function, read this file using the direct link as input `read_csv`. How many columns do you find in the resulting `dataframe`?

06 - Use function `afedR::get_data_file` function to access the `example_tsv.csv` file in the book's data repository. Note that the columns of the data are separated by the tab symbol ('\t'), and not the usual comma. After reading the `readr::read_delim` manual, import the information from this file to your R session. How many rows does the resulting `dataframe` contain?

07 - In the book package there is a data file called '`funky_csv2.csv`'. This has a particularly bizarre format for the data. Open it in a text editor and try to understand how the columns are separated and what is symbol for decimals. After that, study the inputs of function `read.table` and import the table into your R session. If we add the number of rows to the number of columns in the imported table, what is the result?

Importing Data from the Internet

One of the great advantages of R is a large amount of data that can be imported using the internet and specialized packages. This means that you can avoid all the tedious and soul-crushing work of manual data collection. It also becomes easy to share reproducible code, as anyone can feasibly download the same tables with a single line of code.

In this chapter, I will describe and give examples of the most important and reliable packages for data importation in the fields of finance and economics. It is a small, but comprehensive list of packages that cover a large range of research topics. The list includes:

- GetQuandlData (Perlin, 2019)** Imports economical and financial data from the Quandl platform.
- BatchGetSymbols (Perlin, 2022)** Imports adjusted and unadjusted stock price data from Yahoo Finance.
- simfinR (Perlin, 2021)** Imports financial statements and adjusted stock prices from the SimFin project¹.
- tidyquant (Dancho and Vaughan, 2022)** Imports several financial information about stock prices and fundamental data.
- Rbitcoin (Gorecki, 2016)** Imports data for cryptocurrencies.

5.1 Package GetQuandlData

Quandl is an established and comprehensive platform that provides access to a series of free and paid data. Several central banks and research institutions

¹<https://simfin.com/>

provide free economic and financial information on this platform. I strongly recommend browsing the available tables from the Quandl website². It is likely that you'll find datasets that you're familiar with.

In R, package `Quandl` (Raymond McTaggart et al., 2021) is the official extension offered by the company and available in CRAN. However, the package has some issues (see blog post here³) and it is uncomfortable to work with the `tidyverse` collection of packages. The alternative package `GetQuandlData` (Perlin, 2019) is, in my humble and biased opinion as an author, a better alternative.

The **first and mandatory** step in using `GetQuandlData` is to register a user at the Quandl website. Soon after, go to *account settings* and click *API KEY*. This page should show a code, such as `Asv8Ac7zuZzJSCGxynfG`. Copy it to the clipboard (*control + c*) and, in R, define a character object containing the copied content as follows:

```
# set FAKE api key to quandl
my_api_key <- 'Asv8Ac7zuZzJSCGxynfG'
```

This API key is unique to each user, and the one presented here will not work on your computer. You'll need to get your own API key to run the examples of the book. After finding and setting your key, go to Quandl's website and use the search box to look for the symbol of the time series of interest. As an example, we will use data for gold prices in the London Market, with a Quandl code equivalent to '`LBMA/GOLD`'. Do notice that the structure of a Quandl code is always the same, with the name of the main database at first, and the name of table second, separated by a forward slash (/).

Now, with the API key and the Quandl symbol, we use function `get_Quandl_series` to download the data from 1980-01-01 to 2019-01-01:

```
library(GetQuandlData)
library(tidyverse)

# set symbol and dates
my_symbol <- c('GOLD' = 'LBMA/GOLD')
first_date <- '1980-01-01'
last_date <- '2021-01-01'

# get data!
df_quandl <- get_Quandl_series(id_in = my_symbol,
```

²<https://www.quandl.com/>

³<https://www.msperlin.com/post/2019-10-01-new-package-getquandldata/>

```

        api_key = my_api_key,
        first_date = first_date,
        last_date = last_date,
        do_cache = FALSE)

# check it
glimpse(df_quandl)

```

```

R> Rows: 10,840
R> Columns: 9
R> $ `USD (AM)` <chr> "1747", "1739.65", "1764.75", "1764.55~"
R> $ `USD (PM)` <chr> "1742.95", "1740.4", "1751.6", "1758.6~
R> $ `GBP (AM)` <chr> "1473.59", "1474.01", "1481.15", "1484~"
R> $ `GBP (PM)` <chr> "1467.74", "1471.86", "1473.27", "1491~"
R> $ `EURO (AM)` <chr> "1698.78", "1699.64", "1702.63", "1701~"
R> $ `EURO (PM)` <chr> "1696.76", "1696.02", "1694.85", "1701~"
R> $ series_name <chr> "GOLD", "GOLD", "GOLD", "GOLD", "GOLD"~
R> $ ref_date <date> 2022-11-22, 2022-11-21, 2022-11-18, 2~
R> $ id_quandl <chr> "LBMA/GOLD", "LBMA/GOLD", "LBMA/GOLD", ~

```

Notice how we set the name of the time series inline `id_in = c('GOLD' = 'LBMA/GOLD')`. The name of the element becomes the value of column `series_name` in `df_quandl`. If we had more time series, they would be stacked in the same table, but with different `series_name` value.

Worth knowing that other Quandl API options are available with inputs `order`, `collapse` and `transform`. If using Quandl is important to your work, I strongly recommend reading the available parameters for querying data⁴. Several choices for data transformations can be passed to function `get_Quandl_series`.

As an inspection check, let's plot the prices of Gold in USD over time.

⁴<https://docs.quandl.com/docs/parameters-2>



Overall, gold prices were fairly stable between 1980 and 2000, reaching a spike after 2010. One possible explanation is the higher demand for safer assets, such as gold, after the 2009 financial crisis. However, gold was never an efficient long term investment. To show that, let's calculate its compound annual return from 1980-01-02 to 2022-11-22:

```
# sort the rows
df_quandl <- df_quandl %>%
  mutate(USD = as.numeric(`USD (AM)`)) %>%
  arrange(ref_date)

total_ret <- last(df_quandl$USD)/first(df_quandl$USD) - 1
total_years <- as.numeric(max(df_quandl$ref_date) -
                           min(df_quandl$ref_date) )/365

comp_ret_per_year <- (1 + total_ret)^(1/total_years) - 1

print(comp_ret_per_year)
```

R> [1] 0.0269065

We find the result that Gold prices in USD compounded in a rate equal to 2.69% per year. This is not an impressive investment result by any means. As a comparison, the annual inflation for the US in the same period is 3.12%. This means that by buying gold in 1980, the investor received less than the

inflation as a nominal return.

5.1.0.1 Fetching many time series

When asking for multiple time series from Quandl, package `GetQuandlData` stacks all the data in a single `dataframe`, making it easier to work with the `tidyverse` tools. As an example, let's look at Quandl database `RATEINF`, which contains a time series of inflation rates around the world. First, we need to see what are the available datasets:

```
library(GetQuandlData)
library(tidyverse)

# database to get info
db_id <- 'RATEINF'

# get info
df_db <- get_database_info(db_id, my_api_key)

glimpse(df_db)
```

```
R> Rows: 26
R> Columns: 8
R> $ code      <chr> "CPI_ARG", "CPI_AUS", "CPI_CAN", "CPI~
R> $ name       <chr> "Consumer Price Index - Argentina", "~"
R> $ description <chr> "Please visit <a href=http://www.rate~
R> $ refreshed_at <dttm> 2020-10-10 02:03:32, 2022-11-19 02:0~
R> $ from_date   <date> 1988-01-31, 1948-09-30, 1989-01-31, ~
R> $ to_date     <date> 2013-12-31, 2022-09-30, 2022-10-31, ~
R> $ quandl_code <chr> "RATEINF/CPI_ARG", "RATEINF/CPI_AUS", ~
R> $ quandl_db    <chr> "RATEINF", "RATEINF", "RATEINF", "RAT~
```

Column `name` contains the description of tables. If we dig deeper, we'll find the following names:

```
print(unique(df_db$name))

R> [1] "Consumer Price Index - Argentina"
R> [2] "Consumer Price Index - Australia"
R> [3] "Consumer Price Index - Canada"
R> [4] "Consumer Price Index - Switzerland"
R> [5] "Consumer Price Index - Germany"
R> [6] "Consumer Price Index - Euro Area"
R> [7] "Consumer Price Index - France"
```

```
R> [8] "Consumer Price Index - UK"
R> [9] "Consumer Price Index - Italy"
R> [10] "Consumer Price Index - Japan"
R> [11] "Consumer Price Index - New Zealand"
R> [12] "Consumer Price Index - Russia"
R> [13] "Consumer Price Index - USA"
R> [14] "Inflation YOY - Argentina"
R> [15] "Inflation YOY - Australia"
R> [16] "Inflation YOY - Canada"
R> [17] "Inflation YOY - Switzerland"
R> [18] "Inflation YOY - Germany"
R> [19] "Inflation YOY - Euro Area"
R> [20] "Inflation YOY - France"
R> [21] "Inflation YOY - UK"
R> [22] "Inflation YOY - Italy"
R> [23] "Inflation YOY - Japan"
R> [24] "Inflation YOY - New Zealand"
R> [25] "Inflation YOY - Russia"
R> [26] "Inflation YOY - USA"
```

What we want is the '**Inflation YOY - ***' datasets, which contain the year-on-year inflation rates for different countries. Let's filter the `dataframe` to keep the series with the yearly inflation, and select four countries:

```
selected_series <- c('Inflation YOY - USA',
                     'Inflation YOY - Canada',
                     'Inflation YOY - Euro Area',
                     'Inflation YOY - Australia')

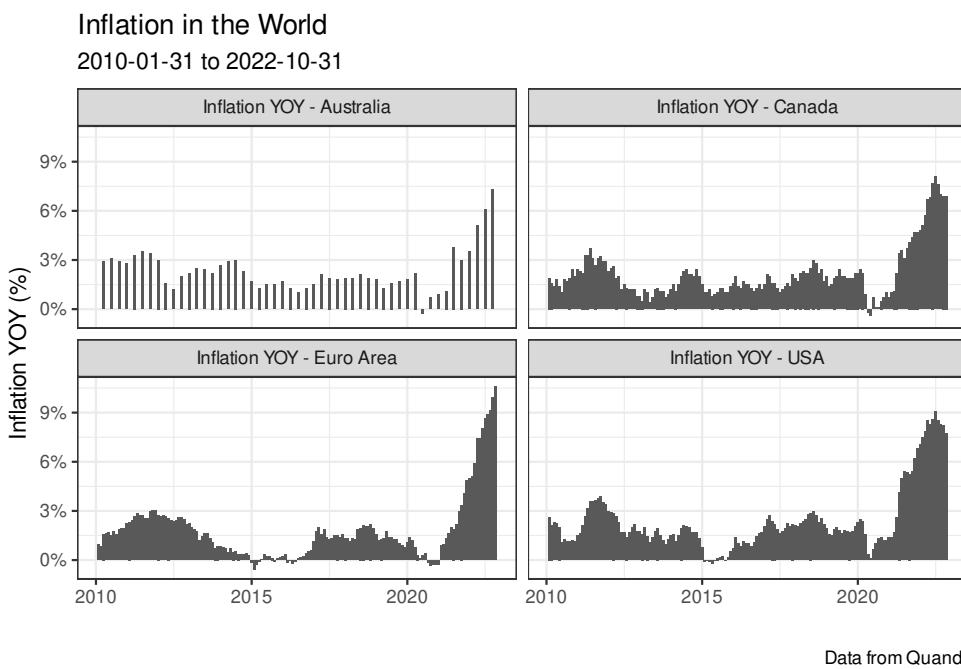
# filter selected countries
idx <- df_db$name %in% selected_series
df_db <- df_db[idx, ]
```

Now we grab the data using get Quandl series:

```
glimpse(df_inflation)
```

```
R> Rows: 513
R> Columns: 4
R> $ series_name <chr> "Inflation YOY - Australia", "Inflatio~
R> $ ref_date    <date> 2022-09-30, 2022-06-30, 2022-03-31, 2~
R> $ value       <dbl> 7.3, 6.1, 5.1, 3.5, 3.0, 3.8, 1.1, 0.9~
R> $ id_quandl   <chr> "RATEINF/INFLATION_AUS", "RATEINF/INFL~
```

And, finally, we create an elegant plot to see the behavior of the inflation rates in the selected countries:



As you can see, the `GetQuandlData` output is formatted to work well with the `tidyverse` tools. The resulting data shows some correlation between the inflation rates, specially between the Euro Area and USA.

5.2 Package BatchGetSymbols

Package `BatchGetSymbols` (Perlin, 2022) is all about downloading stock price data from Yahoo Finance. Unlike other packages, `BatchGetSymbols` focuses on large batch downloads of structured and clean/tidy data. Its main features are:

Cleanliness and organization: All financial data from different *tickers* is kept in the same `dataframe` in a format that facilitates further analysis using packages from `tidyverse`.

Control of import errors: All download errors are registered in the output. If a particular stock does not have available data in Yahoo finance, it will be skipped and all other stock data are still downloaded and returned to the user;

Comparison of dates to a benchmark: Individual asset data is compared to data available for a benchmark, usually a market index. If the number of missing dates exceeds a certain limit set by the user, the stock is removed from the final `dataframe`. Moreover, you can set the minimal accepted volume of data for each stock, removing all stock cases with a low number of rows.

Caching system: By default, all imported data is locally saved using a clever caching system. Whenever the user repeats a data query, the cache system is used. If the desired data is not available in the cache, the function will only download the missing data. This significantly increases data access speed while minimizing the use of an Internet connection;



Since version 2.6 (2020-11-22) of `BatchGetSymbols`, the default cache folder is located in the temporary directory of the R session. Thus, the cache is persistent only for the current user session. This change was motivated by structural breaks in the *Yahoo Finance* data, where the past data recorded in cache was no longer correct due to corporate events such as splits and dividends. The user, however, can change the cache folder using the `cache.folder` entry.

Access to *tickers* in market indices: The package includes functions to download composition of stock indexes. This makes it easy to import data for a large number of stocks. You can, for example, download prices for all stocks constituents of the SP500 index.

Use of multiple cores: If the user is downloading a large batch of stock data, an option for parallel execution is available. The practical effect is, depending on the number of cores in the computer, a significant decrease in total download time.

Flexible output format: The package also offers functions to modify the format of the data. If you need the wide format `dataframe`, such as a price

matrix where tickers are columns and prices are rows, all you need to do is call function `BatchGetSymbols::reshape.wide`. Likewise, changing the frequency of the data is also possible. You can, for example, download weekly or monthly prices.

As an example of usage, let's download the prices of four stocks in the previous five years using function `BatchGetSymbols`. We choose these companies: Microsoft (MSFT), Google (GOOGL), JP Morgan (JPM) and General Electric (GE).

In the call to function `BatchGetSymbols`, we set arguments `thresh.bad.data = 0.95` and `bench_ticker = '^GSPC'`. These choices make sure that all returned data have at least 95% of valid prices when compared to data from the SP500 index (ticker '`^GSPC`').

```
library(BatchGetSymbols)

# set tickers
tickers <- c('MSFT', 'GOOGL', 'JPM', 'GE')

# set dates
first_date <- Sys.Date() - 5 * 365
last_date <- Sys.Date()
thresh_bad_data <- 0.95    # sets percent threshold for bad data
bench_ticker <- '^GSPC'    # set benchmark as ibovespa

l_out <- BatchGetSymbols(tickers = tickers,
                         first.date = first_date,
                         last.date = last_date,
                         bench.ticker = bench_ticker,
                         thresh.bad.data = thresh_bad_data)
```

The output of `BatchGetSymbols` is an object of type `list`, which was not yet presented in this book. For now, all you need to know is that a `list` is a container for other objects. We will further study this class in chapter 6.



Note that the entries of the `BatchGetSymbols::BatchGetSymbols` function use `"."` in their names, such as `thresh.bad.data`, and `bench.ticker`, while the book is written using the underscore (`_`), such as `asthresh_bad_data`, and `bench_ticker`. This difference can result in problems if, in the absence of attention, the user substitutes one for the other. As a rule of thumb, prioritize the use of underscores in object names. Unfortunately some functions written in the past still have an old code structure and, in order not to harm old users, the names of the inputs were kept.

Back to our example, object `l_out` has two elements, a table called `df.control` and another table called `df.tickers`. We can access each element using operator `$`, such as in `l_out$df.control`. The first table, `df.control`, contains the result of the download process. As previously mentioned, the package not only downloads the data but also keeps track of possible errors and missing values. Let's check the content of this table.

```
# print result of download process
print(l_out$df.control)
```

```
R> # A tibble: 4 x 6
R>   ticker src  download.status total.obs perc.benc~1 thres~2
R>   <chr>  <chr> <chr>          <int>      <dbl> <chr>
R> 1 MSFT    yahoo OK             1258       1 KEEP
R> 2 GOOGL   yahoo OK             1258       1 KEEP
R> 3 JPM     yahoo OK             1258       1 KEEP
R> 4 GE      yahoo OK             1258       1 KEEP
R> # ... with abbreviated variable names
R> #   1: perc.benchmark.dates, 2: threshold.decision
```

Column `threshold.decision` from `df.control` shows that all tickers were valid, and we got 1258 valid observations (rows) for each company.

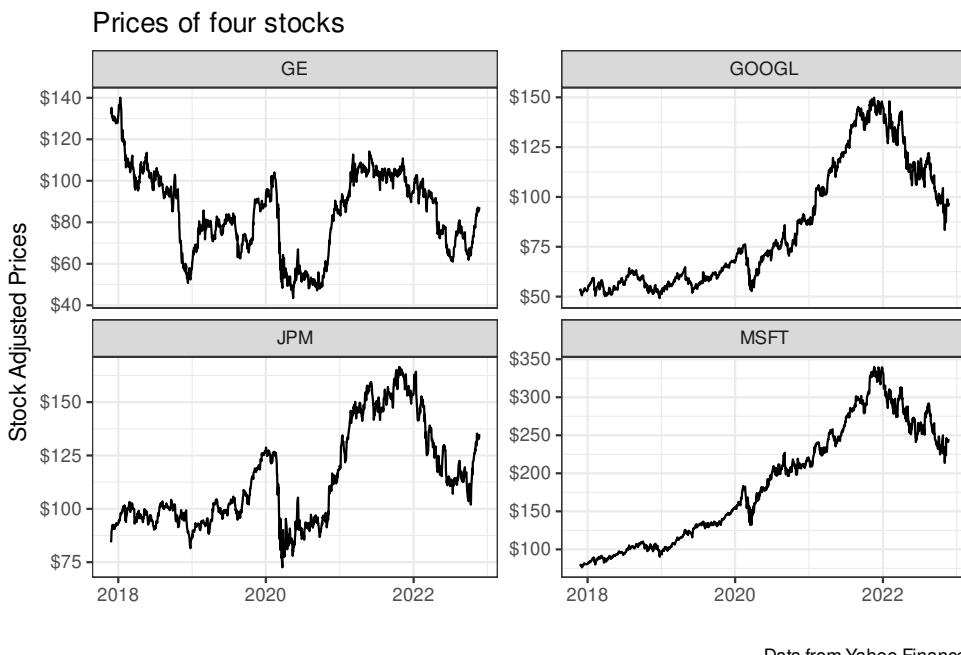
As for the actual financial data, it is contained in element `df.tickers` of `l_out`. Let's have a look:

```
# print df.tickers
glimpse(l_out$df.tickers)
```

```
R> Rows: 5,032
R> Columns: 10
R> $ price.open      <dbl> 83.01, 83.31, 84.07, 84.71, 83~
```

```
R> $ price.high          <dbl> 83.43, 83.98, 85.06, 84.92, 84~
R> $ price.low           <dbl> 82.78, 83.30, 84.02, 83.18, 83~
R> $ price.close         <dbl> 83.26, 83.87, 84.88, 83.34, 84~
R> $ volume              <dbl> 7425600, 18265200, 21926000, 2~
R> $ price.adjusted      <dbl> 78.40719, 78.98163, 79.93274, ~
R> $ ref.date             <date> 2017-11-24, 2017-11-27, 2017~~
R> $ ticker               <chr> "MSFT", "MSFT", "MSFT", "MSFT"~
R> $ ret.adjusted.prices <dbl> NA, 0.0073263691, 0.0120421802~
R> $ ret.closing.prices   <dbl> NA, 0.0073264591, 0.0120423747~
```

As expected, we find information about stock prices and traded volume. Notice it also includes column `ticker`, which contains the symbols of the stocks. In the tidy format, each stock has a chunk of data that is pilled in top of each other. Later, in chapter 8, we will use this column to split the data and build summary tables. To inspect the data, let's look at its prices with `ggplot2`.



Data from Yahoo Finance

We see that General Electric (GE) stock was not kind to its investors. Someone that bought the stock at its peak in mid-2016 has found its current value at less than half. Now, when it comes to the GOOGL, JPM and MSFT, we see an upward increase in stock prices. These are profitable and competitive companies in their sectors and not surprisingly, the stock prices surged over time.

Now, let's look at an example of a large batch download of stock prices. For that, we had to find the group of tickers of stocks that currently belong to the SP500 index with function `BatchGetSymbols::GetSP500Stocks`. After that, we simply pass the vector of tickers to `BatchGetSymbols`. We will also use parallel computing to speed up the importation:

```
library(BatchGetSymbols)

# set tickers
df_SP500 <- GetSP500Stocks()
sp500_tickers <- df_SP500$Tickers

# set dates
first_date <- '2010-01-01'
last_date <- '2021-01-01'
thresh_bad_data <- 0.95    # sets percent threshold for bad data
bench_ticker <- '^GSPC'    # set benchmark as ibovespa

# set number of cores (half of available cores)
future::plan(future::multisession,
             workers = floor(parallel::detectCores()/2))

l_out <- BatchGetSymbols(tickers = sp500_tickers,
                         first.date = first_date,
                         last.date = last_date,
                         bench.ticker = bench_ticker,
                         thresh.bad.data = thresh_bad_data,
                         do.parallel = TRUE)
```

And now we check the resulting data:

```
glimpse(l_out$df.tickers)
```

```
R> Rows: 1,027,664
R> Columns: 10
R> $ price.open      <dbl> 83.09, 82.80, 83.88, 83.32, 83~
R> $ price.high       <dbl> 83.45, 83.23, 84.60, 83.76, 84~
R> $ price.low        <dbl> 82.67, 81.70, 83.51, 82.12, 83~
R> $ price.close      <dbl> 83.02, 82.50, 83.67, 83.73, 84~
R> $ volume           <dbl> 3043700, 2847000, 5268500, 447~
R> $ price.adjusted   <dbl> 64.09430, 63.69284, 64.59611, ~
R> $ ref.date          <date> 2010-01-04, 2010-01-05, 2010~~
R> $ ticker            <chr> "MMM", "MMM", "MMM", "MMM", "M~
R> $ ret.adjusted.prices <dbl> NA, -0.0062636148, 0.014181783~
```

```
R> $ ret.closing.prices <dbl> NA, -0.0062635150, 0.014181793~
```

We get a table with 1027664 rows in `l_out$df.tickers`. Looking deeper into `l_out$df.control` we find that, out of 503 tickers, only 454 passed the consistency check. This is a fairly sized table. Notice how easy it was to get that large volume of data from Yahoo Finance into an R session.



Please be aware that Yahoo Finance (YF) data for **adjusted prices of single stocks** is not trustworthy and, if you compare to other data vendors, you'll easily find large differences. The issue is that Yahoo Finance does not adjust for dividends, only for stock splits. This means that, when looking at price series over a long period of time, there is a downward bias in overall return. As a rule of thumb, in a formal research, **never use individual stock data from Yahoo Finance**, specially if the stock return is important to the research. The exception is for financial indexes, such as the SP500, where Yahoo Finance data is quite reliable since indexes do not undergo the same adjustments as individual stocks.

5.3 Package simfinR

SimFin⁵ is a special project for making financial data as accessible as possible. It works by gathering data from different stock exchanges and financial reports, cleaning and verifying it against official records, and distributing the tables from an API (*access point interface*). From its own website:

Our core goal is to make financial data as freely available as possible because we believe that having the right tools for investing/research shouldn't be the privilege of those that can afford to spend thousands of dollars per year on data.

The platform is free, with a daily limit of 2000 api calls. This is not bad and should suffice for most users. If you need more calls, the premium version⁶ is just 10 euros a month, a fraction of what other data vendors usually request.

Package `simfinR` facilitates importing data from the SimFin API. First, it makes sure the requested data exists and only then calls the api. As usual, all api queries are saved locally using package `memoise`. This means that

⁵<https://simfin.com/>

⁶<https://simfin.com/simfin-plus>

the second time you ask for a particular data about a company/year, the function will load a local copy, and will not call the web api, helping you stay below the API limits.

5.3.1 Example 01 - Apple Inc Annual Profit

The first step in using `simfinR` is registering at the SimFin website. Once done, click on Data Access⁷. It should now show an API key such as '`'rluwS1N304NpyJeBjlxZPspfBBhfJR4o'`'. Save it in an R object for later use.

```
my_api_key <- 'rluwS1N304NpyJeBjlxZPspfBBhfJR4o'
```

You need to be aware that the **API key in `my_api_key` is fake and will not work for you**. You need to get your own to execute the examples.

With the API key in hand, the second step is to find the numerical id of the company of interest. For that, we can find all available companies and their respective ids and ticker with `simfinR_get_available_companies`.

```
library(simfinR)
library(tidyverse)

# get info
df_info_companies <- simfinR_get_available_companies(my_api_key)

# check it
glimpse(df_info_companies)
```

```
R> Rows: 3,410
R> Columns: 3
R> $ simId <int> 171401, 901704, 1243193, 901866, 994625, 45~
R> $ ticker <chr> "ZYXI", "ZYNE", "ZY", "ZVO", "ZUO", "ZUMZ", ~
R> $ name   <chr> "ZYNEX INC", "Zynerba Pharmaceuticals, Inc.~
```

Digging deeper into the `dataframe`, we find that the numerical id of Apple is 111052. Let's use it to download the annual financial information since 2009.

```
id_companies <- 111052 # id of APPLE INC
type_statements <- 'pl' # profit/loss
periods = 'FY' # final year
years = 2009:2018
```

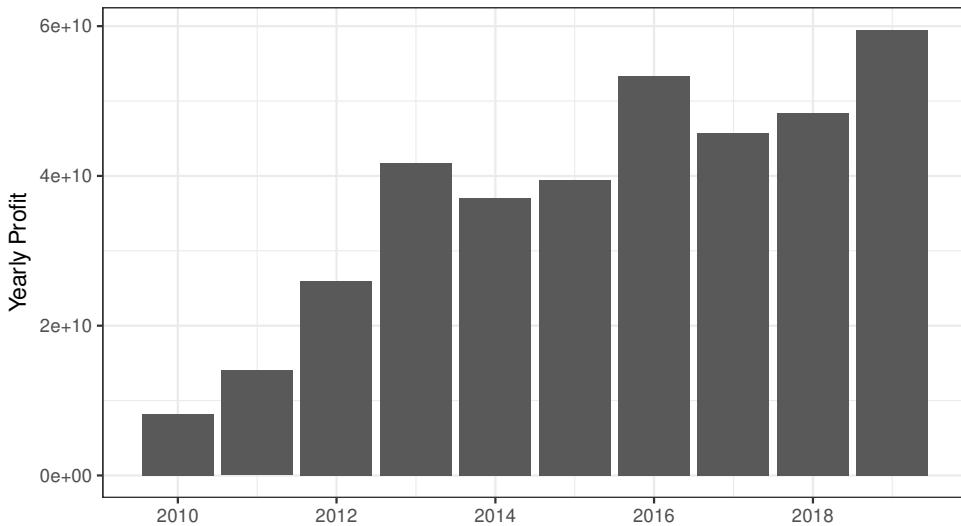
⁷<https://simfin.com/data/access/api>

```
df_fin_FY <- simfinR_get_fin_statements(  
  id_companies,  
  type_statements = type_statements,  
  periods = periods,  
  years = years,  
  api_key = my_api_key)  
  
glimpse(df_fin_FY)
```

```
R> Rows: 580  
R> Columns: 13  
R> $ company_name    <chr> "APPLE INC", "APPLE INC", "APPLE IN~  
R> $ company_sector <chr> "Computer Hardware", "Computer Hard~  
R> $ type_statement  <fct> pl, pl, pl, pl, pl, pl, pl, pl, pl,~  
R> $ period          <fct> FY, FY, FY, FY, FY, FY, FY, FY,~  
R> $ year            <int> 2009, 2009, 2009, 2009, 2009, 2009,~  
R> $ ref_date        <date> 2009-12-31, 2009-12-31, 2009-12-31~  
R> $ acc_name         <chr> "Revenue", "Sales & Services Revenu~  
R> $ acc_value        <dbl> 4.2905e+10, NA, NA, NA, -2.5683e+10~  
R> $ tid              <chr> "1", "3", "5", "6", "2", "7", "8", ~  
R> $ uid              <chr> "1", "0", "0", "0", "2", "0", "0", ~  
R> $ parent_tid       <chr> "4", "1", "1", "1", "4", "2", "2", ~  
R> $ display_level    <chr> "0", "1", "1", "1", "0", "1", "1", ~  
R> $ check_possible   <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, ~
```

And now we plot the results of the Net Income (profit/loss) for all years:

Yearly Profit of APPLE INC



Data from simfin <<https://simfin.com/>>

Not bad! Apple has been doing very well over the years. We can also grab data for all quarters and get more detailed information:

```
type_statements <- 'pl' # profit/loss
periods = c('Q1', 'Q2', 'Q3', 'Q4') # final year
years = 2009:2018

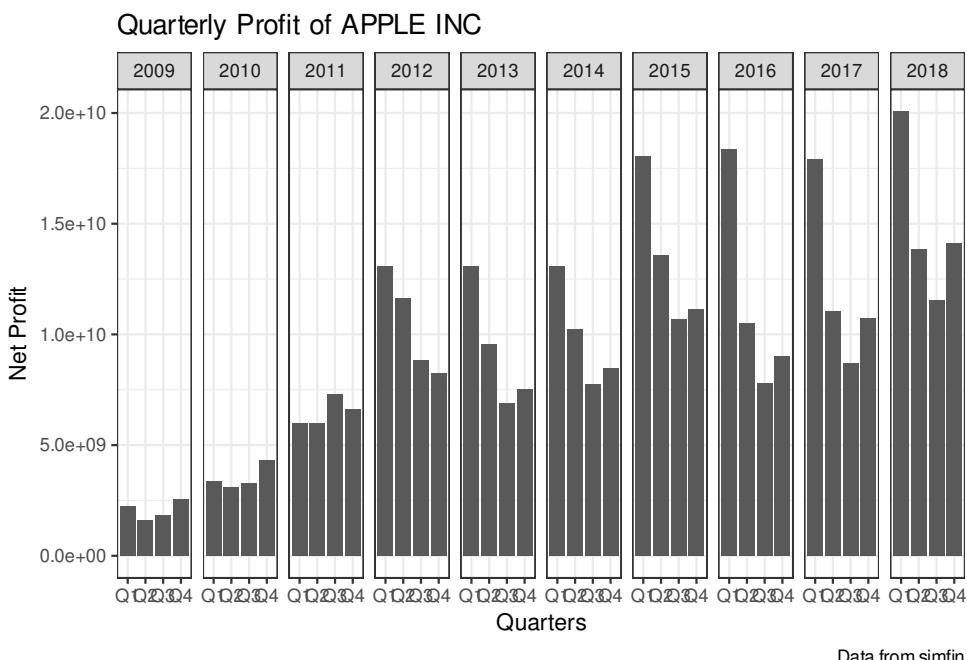
df_fin_quarters <- simfinR_get_fin_statements(
  id_companies,
  type_statements = type_statements,
  periods = periods,
  years = years,
  api_key = my_api_key)

glimpse(df_fin_quarters)
```

```
R> Rows: 2,320
R> Columns: 13
R> $ company_name    <chr> "APPLE INC", "APPLE INC", "APPLE IN~
R> $ company_sector <chr> "Computer Hardware", "Computer Hard~
R> $ type_statement  <fct> pl, pl, pl, pl, pl, pl, pl, pl, ~
R> $ period          <fct> Q1, Q1, Q1, Q1, Q1, Q1, Q1, Q1, ~
R> $ year            <int> 2009, 2009, 2009, 2009, 2009, 2009, ~
R> $ ref_date        <date> 2009-03-31, 2009-03-31, 2009-03-31~
```

```
R> $ acc_name      <chr> "Revenue", "Sales & Services Revenue"
R> $ acc_value     <dbl> 1.188e+10, NA, NA, NA, -7.373e+09, ~
R> $ tid           <chr> "1", "3", "5", "6", "2", "7", "8", ~
R> $ uid           <chr> "1", "0", "0", "0", "2", "0", "0", ~
R> $ parent_tid    <chr> "4", "1", "1", "1", "4", "2", "2", ~
R> $ display_level <chr> "0", "1", "1", "1", "0", "1", "1", ~
R> $ check_possible <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, ~
```

And plot the results:



Nice and impressive profit record. The first quarter (Q1) seems to present the best performance, probably due to the effect of Christmas in the retail business. Apple sells quality products at a premium price. Not surprisingly, many people want the new iphone for Christmas and that translates into more sells in the months of december, which are registered at the first quarter of the year.

5.3.2 Example 02 - Quarterly Net Profit of Many Companies

Package `simfinR` can also fetch information for many companies in a single call. Let's run another example by selecting four random companies and creating the same previous graph:

```

set.seed(100)
my_ids <- sample(df_info_companies$simId, 4)
type_statements <- 'pl' # profit/loss
periods = 'FY' # final year
years = 2010:2018

df_fin <- simfinR_get_fin_statements(
  id_companies = my_ids,
  type_statements = type_statements,
  periods = periods,
  years = years,
  api_key = my_api_key)

net_income <- df_fin %>%
  dplyr::filter(acc_name == 'Net Income')

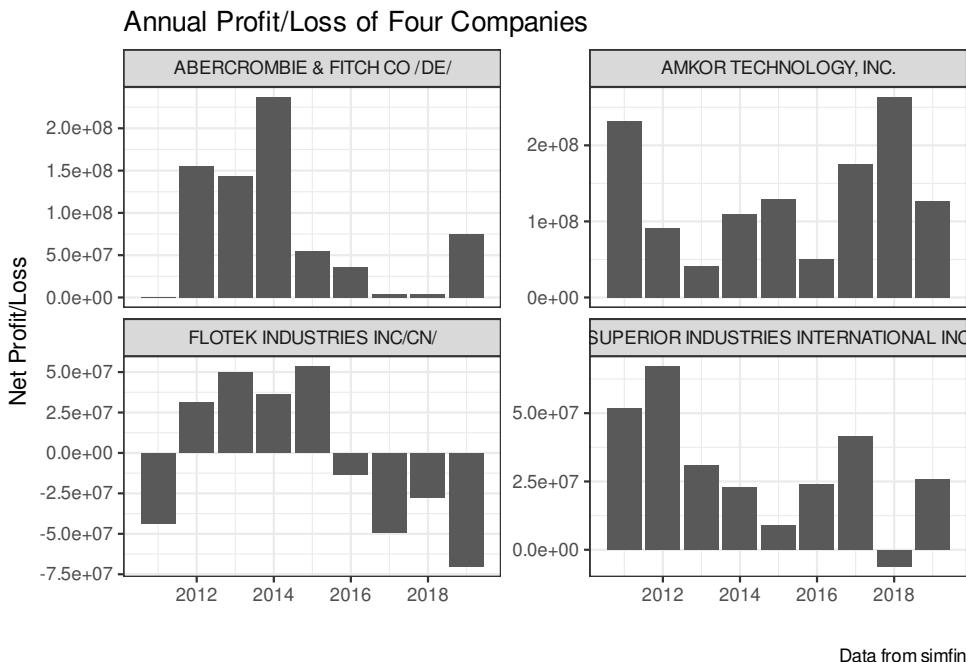
glimpse(net_income)

```

```

R> Rows: 36
R> Columns: 13
R> $ company_name    <chr> "ABERCROMBIE & FITCH CO /DE/", "ABE~"
R> $ company_sector <chr> "Retail - Apparel & Specialty", "Re~
R> $ type_statement  <fct> pl, pl, pl, pl, pl, pl, pl, pl, pl, ~
R> $ period          <fct> FY, FY, FY, FY, FY, FY, FY, FY, ~
R> $ year            <int> 2010, 2011, 2012, 2013, 2014, 2015, ~
R> $ ref_date        <date> 2010-12-31, 2011-12-31, 2012-12-31~
R> $ acc_name         <chr> "Net Income", "Net Income", "Net In~
R> $ acc_value        <dbl> 254000, 155709000, 143934000, 23701~
R> $ tid              <chr> "55", "55", "55", "55", "55", "55", ~
R> $ uid              <chr> "55", "55", "55", "55", "55", "55", "55", ~
R> $ parent_tid       <chr> "58", "58", "58", "58", "58", "58", "58", ~
R> $ display_level    <chr> "0", "0", "0", "0", "0", "0", "0", "0", ~
R> $ check_possible   <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, ~

```



As you can see, it is fairly straightforward to download financial data for multiple companies using `simfinR`.

5.3.3 Example 03 - Fetching price data

The simfin project also provides prices of stocks, adjusted for dividends, splits and other corporate events. Have a look:

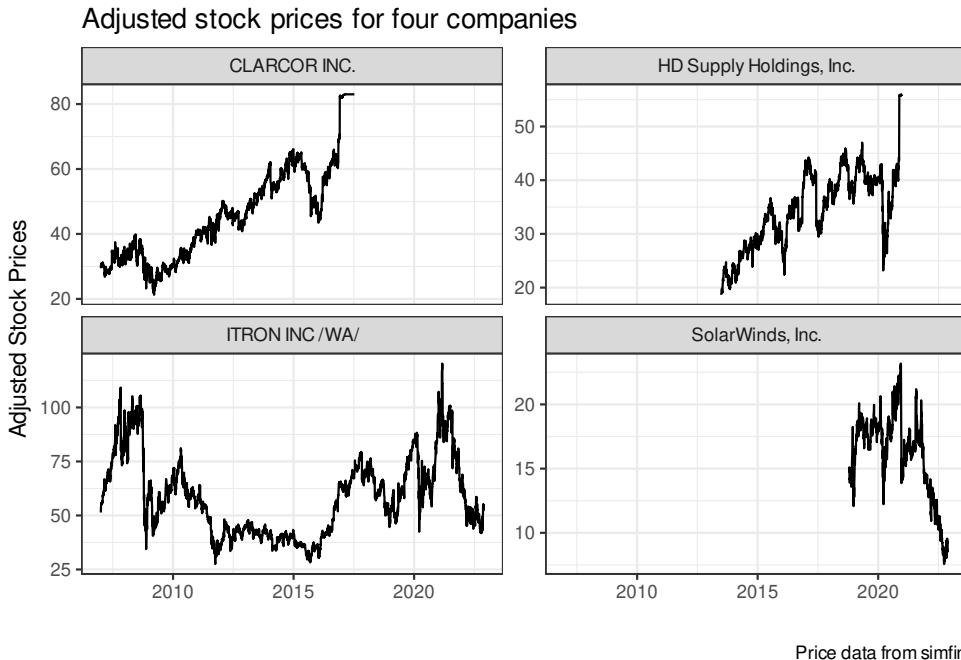
```
set.seed(15)
my_ids <- sample(df_info_companies$simId, 4)

df_price <- simfinR_get_price_data(id_companies = my_ids,
                                      api_key = my_api_key)

glimpse(df_price)

R> Rows: 9,503
R> Columns: 8
R> $ ref_date      <date> 2017-07-10, 2017-07-07, 2017-07-~
R> $ close_adj     <dbl> 83, 83, 83, 83, 83, 83, 83, 83, 8~
R> $ split_coef    <chr> NA, NA, NA, NA, NA, NA, NA, NA, N~
R> $ share_class_id <int> 553306, 553306, 553306, 553306, 5~
R> $ share_class_name <chr> "Common shares", "Common shares", ~
```

```
R> $ share_class_type <chr> "common", "common", "common", "co~  
R> $ currency           <chr> "USD", "USD", "USD", "USD", "USD"~  
R> $ company_name       <chr> "CLARCOR INC.", "CLARCOR INC.", "~
```



Price data from simfin

As you can see, the data is comprehensive and should suffice for many different corporate finance research topics.

5.4 Package tidyquant

Package `tidyquant` (Dancho and Vaughan, 2022) provides functions related to financial data acquisition and analysis. It is an ambitious project that offers many solutions in the field of finance. As you might suspect, `tidyquant` is designed to interact well with the *tidyverse* format, also known as the *long* format, discussed in chapter 6. The package includes functions for obtaining financial data from the web, manipulation of such data, and the calculation of performance measures of portfolios.

In its current version, `tidyquant` has 159 functions. Let's look at its main functionalities. First, we will obtain price data for Apple stocks (AAPL) using function `tq_get`.

```
library(tidyquant)
```

```
# set stock and dates
ticker <- 'AAPL'
first_date <- '2019-01-01'
last_date <- Sys.Date()

# get data with tq_get
df_prices <- tq_get(ticker,
                      get = "stock.prices",
                      from = first_date,
                      to = last_date)

glimpse(df_prices)

R> Rows: 982
R> Columns: 8
R> $ symbol <chr> "AAPL", "AAPL", "AAPL", "AAPL", "AAPL", "~"
R> $ date <date> 2019-01-02, 2019-01-03, 2019-01-04, 2019-
R> $ open <dbl> 38.7225, 35.9950, 36.1325, 37.1750, 37.39~
R> $ high <dbl> 39.7125, 36.4300, 37.1375, 37.2075, 37.95~
R> $ low <dbl> 38.5575, 35.5000, 35.9500, 36.4750, 37.13~
R> $ close <dbl> 39.4800, 35.5475, 37.0650, 36.9825, 37.68~
R> $ volume <dbl> 148158800, 365248800, 234428400, 21911120~
R> $ adjusted <dbl> 38.10514, 34.30958, 35.77423, 35.69461, 3~
```

As we can see, except for column names, the price data has a similar format to the one we got with `BatchGetSymbols`. This is not surprising as both share the same origin, Yahoo Finance.

We can also get information about components of an index using function `tq_index`. The available market indices are:

```
# print available indices
print(tq_index_options())
```

```
R> [1] "DOW"           "DOWGLOBAL"    "SP400"        "SP500"
R> [5] "SP600"
```

Let's get information for "DOWGLOBAL".

```
# get components of "DOWJONES"
print(tq_index("DOWGLOBAL"))
```

```
R> Getting holdings for DOWGLOBAL
```

```
R> # A tibble: 156 x 8
```

```
R>   symbol company      ident~1 sedol  weight sector share~2
R>   <chr>  <chr>       <chr>  <chr>  <dbl> <chr>  <dbl>
R> 1 SLB    Schlumberger~ 806857~ 2779~ 0.00860 Energy  16427
R> 2 UCG-IT UniCredit S.~ BYMXPS  BYMX~ 0.00843 Finan~  62007
R> 3 GILD   Gilead Scien~ 375558~ 2369~ 0.00843 Healt~  9983
R> 4 SIE-DE Siemens AG  572797  5727~ 0.00815 Indus~  6158
R> 5 SAP-DE SAP SE     484628  4846~ 0.00813 Infor~  7467
R> 6 NFLX   Netflix Inc. 64110L~ 2857~ 0.00801 Commu~  2805
R> 7 ALV-DE Allianz SE  523148  5231~ 0.00800 Finan~  3835
R> 8 CAT    Caterpillar ~ 149123~ 2180~ 0.00789 Indus~  3392
R> 9 MRK    Merck & Co. ~ 58933Y~ 2778~ 0.00783 Healt~  7408
R> 10 DD    DuPont de Ne~ 26614N~ BK0V~ 0.00779 Mater~  11157
R> # ... with 146 more rows, 1 more variable:
R> #   local_currency <chr>, and abbreviated variable names
R> #   1: identifier, 2: shares_held
```

We only looked into a few functions from the package `tidyquant`. It also offers solutions for the usual manipulations, such as calculating returns and functions for portfolio analytics. You can find more details about this package in its website⁸.

5.5 Package Rbitcoin

Given the popularity of crypto-currencies, another package worth mentioning is `RBitcoin`. It allows access to trade data from several Bitcoin exchanges. Here, let's show a simple example of importing trade data from the 'Kraken' exchange, using Euro as the currency.

```
library(Rbitcoin)

# set mkt, currency pair and type of action
mkt <- "kraken"
currency <- c("BTC", "EUR")
action <- 'trades'

# import data
my_l <- market.api.process(market = mkt,
                           currency_pair = currency,
                           action = action)

# print it
```

⁸<https://business-science.github.io/tidyquant/>

```
print(my_l)

R> $market
R> [1] "kraken"
R>
R> $base
R> [1] "BTC"
R>
R> $quote
R> [1] "EUR"
R>
R> $timestamp
R> [1] "2022-11-23 16:08:49 -03"
R>
R> $market_timestamp
R> [1] NA
R>
R> $trades
R>           date   price     amount
R> 1: 2022-11-23 18:36:07 15864.2 0.03527983
R> 2: 2022-11-23 18:36:11 15865.2 0.00966255
R> 3: 2022-11-23 18:36:18 15869.7 0.00406972
R> 4: 2022-11-23 18:36:19 15869.4 0.00176628
R> 5: 2022-11-23 18:36:24 15865.6 0.00602280
R> ---
R> 996: 2022-11-23 19:08:40 15977.3 0.03305618
R> 997: 2022-11-23 19:08:42 15976.8 0.00321661
R> 998: 2022-11-23 19:08:43 15976.9 0.01600000
R> 999: 2022-11-23 19:08:43 15976.9 0.00025863
R> 1000: 2022-11-23 19:08:46 15983.2 0.00531965
R>           tid type
R> 1:             <NA> ask
R> 2:             <NA> ask
R> 3:             <NA> bid
R> 4:             <NA> bid
R> 5:             <NA> bid
R> ---
R> 996:             <NA> ask
R> 997:             <NA> ask
R> 998:             <NA> bid
R> 999:             <NA> bid
R> 1000: 1669230526572052588 bid
```

The output of `market.api.process` is a `list` object with information about Bitcoin in the 'Kraken' market. The actual trades are available in the `trade` slot of `my_1`. Let's have a look at its content:

```
glimpse(my_1$trades)
```

```
R> Rows: 1,000
R> Columns: 5
R> $ date    <dttm> 2022-11-23 18:36:07, 2022-11-23 18:36:11, ~
R> $ price   <dbl> 15864.2, 15865.2, 15869.7, 15869.4, 15865.6~
R> $ amount   <dbl> 0.03527983, 0.00966255, 0.00406972, 0.00176~
R> $ tid     <chr> NA, ~
R> $ type    <chr> "ask", "ask", "bid", "bid", "bid", "bid", "~~
```

It includes price and time information for the past 1000 trades. The package also includes functions for looking into the order book of each market and managing Bitcoin wallets. One can find more details about the functionalities of the package in its website⁹.

5.6 Other Packages

In CRAN, you'll find many more packages for importing financial datasets in R. In this section, we focused on packages, which are free and easy to use. Interface with commercial data sources is also possible. Several companies provide APIs for serving data to their clients. Packages such as `Rblpapi` (Bloomberg), `IBrokers` (Interactive Brokers), `TFX` (TrueFX), `rdatastream` (Thomson Dataworks) can make R communicate with these commercial platforms. If the company you use is not presented here, check the list of packages in CRAN¹⁰. It is very likely you'll find what you need.

5.7 Accessing Data from Web Pages (*web-scraping*)

Packages from the previous section make it easy to import data directly from the web with a single line of code. However, in many cases, the information of interest is not available through a package, but on a web page. Fortunately, we can use R to read the HTML data and import the desired information into an R session. The main advantage is that every time we execute the R code, we get the same content as the target website.

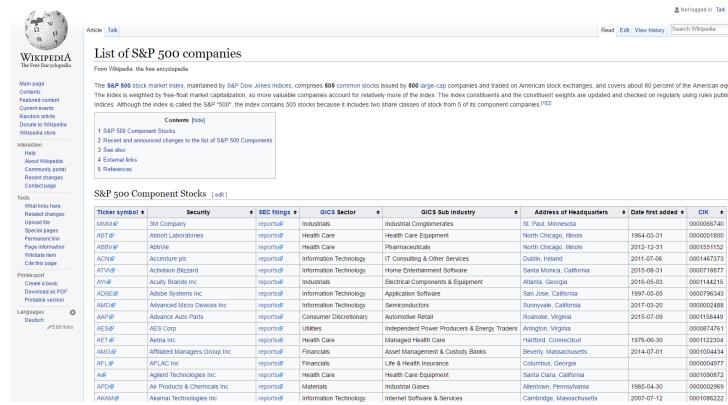
⁹<https://github.com/jangorecki/Rbitcoin>

¹⁰<https://cran.r-project.org/>

The process of extracting information from web pages is called *webscraping*. Depending on the structure and technology used on the internet page, importing its content can be as trivial as a single line in R or a complex process, taking hundreds of lines of code. Let's look at two simple examples; first, we will retrieve tabular information about the SP500 index from Wikipedia and, second, we will extract current inflation and interest rate from the Reserve Bank of Australia (RBA) website.

5.7.1 Scraping the Components of the SP500 Index from Wikipedia

In its website, Wikipedia offers a section¹¹ about the components of the SP500 index. This information is presented in a tabular format, Figure 5.1.



The screenshot shows a Wikipedia article titled "List of S&P 500 companies". The page includes a header with links for "Article", "Talk", "Read", "Edit", "View history", and "Search Wikipedia". Below the header is a summary box stating: "The S&P 500 stock market index, maintained by S&P Dow Jones Indices, comprises 500 common stocks issued by 500 large-cap companies listed on American stock exchanges, and covers about 60 percent of the American equity market. The index is weighted by free-float market capitalization, so more valuable companies account for relatively more of the index. The index constituents and the constituent weights are updated and checked on regularly using rules listed below. Although the index is called the S&P '500', the index contains 505 stocks because it includes two share classes of stock from 5 of its component companies." A note at the bottom of the summary box says: "This page is a mirror of the Wikipedia page 'List of S&P 500 companies'. The original page is located at https://en.wikipedia.org/w/index.php?title=List_of_S%26P_500_companies&oldid=96222".

Table: S&P 500 Component Stocks

Ticker symbol	Security	SEC filings	GICS Sector	GICS Sub Industry	Address of Headquarters	Date first added	GIC
AET	Aetna Inc.	reporting	Health Care	Health Care Equipment	St. Paul, Minnesota	1964-01-31	0000097460
AETLP	Aetna Laboratories	reporting	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	000151152
ABIV	AbbVie	reporting	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	000151152
ADM	Academy Sports + Outdoors	reporting	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	0001467373
ATVI	Activision Blizzard	reporting	Information Technology	Home Entertainment Software	Santa Monica, California	2010-01-31	0000718877
AYH	Acuity Brands Inc.	reporting	Industrials	Electrical Components & Equipment	Atlanta, Georgia	2016-06-03	0001144215
ADBE	Adobe Systems Inc.	reporting	Information Technology	Application Software	Sunnyvale, California	1997-01-31	0000095245
ADBEI	Advanced Micro Devices Inc.	reporting	Information Technology	Semiconductors	Sunnyvale, California	2017-10-20	0000094048
AAPL	Advance Auto Parts	reporting	Consumer Discretionary	Auto Parts	Roanoke, Virginia	2015-07-09	000158449
AES	AES Corp	reporting	Utilities	Independent Power Producers & Energy Traders	Arlington, Virginia	2008-07-01	0000874761
AT&T	At&T Inc.	reporting	Health Care	Managed Health Care	Hartford, Connecticut	1976-06-30	0001123004
AMGN	Affiliated Managers Group Inc.	reporting	Financials	Asset Management & Custody Banks	Beverly, Massachusetts	2014-07-01	0000040434
AT&T	AT&T Inc.	reporting	Financials	Life & Health Insurance	Atlanta, Georgia	2009-07-01	000009777
AAC	AAC Technologies Inc.	reporting	Health Care	Health Care Equipment	Santa Clara, California	2001-08-02	000150873
APD	Air Products & Chemicals Inc.	reporting	Materials	Industrial Goods	Altoona, Pennsylvania	1995-04-30	000002969
AKAM	Akamai Technologies Inc.	reporting	Information Technology	Internet Software & Services	Cambridge, Massachusetts	2007-07-12	0001096222

Figure 5.1: Mirror of Wikipedia page on SP500 components

The information on this web page is constantly updated, and we can use it to import information about the stocks belonging to the SP500 index. Before delving into the R code, we need to understand how a webpage works. Briefly, a webpage is nothing more than a lengthy HTML code interpreted by your browser. A numerical value or text presented on the website can usually be found within the code. This code has a particular tree-like structure with branches and classes. Moreover, every element of a webpage has an address, called *xpath*. In chrome and firefox browsers, you can see the actual code of a webpage by using the mouse to right-click any part of the webpage and selecting *View page source*.

The first step in webscraping is finding out the location of the information you need. In Chrome, you can do that by right-clicking in the specific

¹¹https://en.wikipedia.org/wiki/List_of_S%26P_500_companies

location of the number/text on the website and selecting *inspect*. This will open an extra window in the browser. Once you do that, right-click in the selection and chose *copy* and *copy xpath*. In Figure 5.2, we see a mirror of what you should be seeing in your browser.

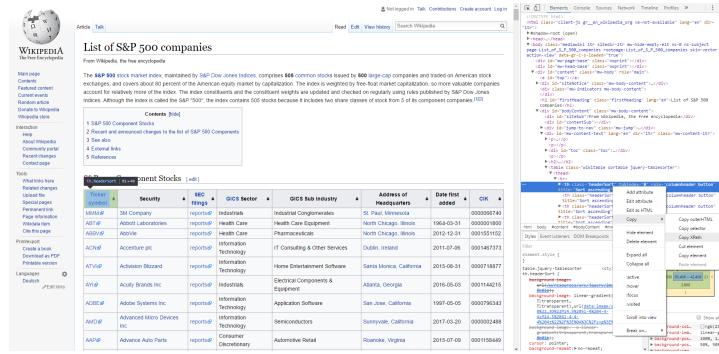


Figure 5.2: Finding xpath from website

Here, the copied *xpath* is:

```
'//*[@id="mw-content-text"]/table[1]/thead/tr/th[2]'
```

This is the address of the header of the table. For the whole content of the table, including header, rows, and columns, we need to set an upper level of the HTML tree. This is equivalent to address `'//*[@id="MW-content-text"]/table[1]`.

Now that we have the location of what we want, let's load package `rvest` (Wickham, 2022a) and use functions `read_html`, `html_nodes` and `html_table` to import the desired table into R:

```
library(rvest)

# set url and xpath
my_url <- paste0('https://en.wikipedia.org/wiki/',
                  'List_of_S%26P_500_companies')
my_xpath <- '//*[@id="mw-content-text"]/div/table[1]'

# get nodes from html
out_nodes <- html_nodes(read_html(my_url),
                        xpath = my_xpath)

# get table from nodes (each element in
# list is a table)
df_SP500_comp <- html_table(out_nodes)
```

```
# isolate it
df_SP500_comp <- df_SP500_comp[[1]]

# change column names (remove space)
names(df_SP500_comp) <- make.names(names(df_SP500_comp))

# print it
glimpse(df_SP500_comp)
```

```
R> Rows: 503
R> Columns: 9
R> $ Symbol <chr> "MMM", "AOS", "ABT", "ABBV", ~
R> $ Security <chr> "3M", "A. O. Smith", "Abbott~
R> $ SEC.filings <chr> "reports", "reports", "repor~
R> $ GICS.Sector <chr> "Industrials", "Industrials"~
R> $ GICS.Sub.Industry <chr> "Industrial Conglomerates", ~
R> $ Headquarters.Location <chr> "Saint Paul, Minnesota", "Mi~
R> $ Date.first.added <chr> "1976-08-09", "2017-07-26", ~
R> $ CIK <int> 66740, 91142, 1800, 1551152, ~
R> $ Founded <chr> "1902", "1916", "1888", "201~
```

Object `df_SP500_comp` contains a mirror of the data from the Wikipedia website. The names of the columns require some work, but the raw data is intact and could be further used in a script. Notice how the output is the exact same data as the previous use of function `BatchGetSymbols::GetSP500Stocks`. By executing command `edit(BatchGetSymbols::GetSP500Stocks)` you'll see that the source of data is the same and the code is similar to the one we just executed.

5.7.2 Scraping the Website of the Reserve Bank of Australia

As another example of webscraping with R, let's import information from the Reserve Bank of Australia. When accessed in 2021-02-26, its home page¹² mirrors Figure 5.3.

The website presents several information such as current news and interest rates. Let's assume we are interested in the information about the current cash/bank rate and inflation, right upper corner of the webpage.

The first step of *webscrapping* is finding out the *xpath* of the information we

¹²<http://www.rba.gov.au/>

Figure 5.3: Website for the Reserve Bank of Australia

want. Using the procedure described in the previous example, we find out the address of both values, market rate and current inflation:

```
xpath_inflation <- '//*[@id="content"]/section[1]/div/div[3]/p'  
xpath_int_rate <- '//*[@id="content"]/section[1]/div/div[1]/p'
```

A difference from the previous example is we are not importing a table, but a simple text from the website. For that, we use function `html_text` and not `html_table`. The full code and its output are presented next.

```
# get interest rate from html
my_int_rate <- html_text(html_nodes(x = html_code,
                                     xpath = xpath_int_rate ))
```

And now we print the result:

```
# print result
cat("\nCurrent inflation in AUS:", my_inflation)
```

```
R>
R> Current inflation in AUS: 0.9%
```

```
cat("\nCurrent interest rate AUS:", my_int_rate)
```

```
R>
R> Current interest rate AUS: 0.1%
```

Using *webscraping* techniques can become a strong ally of the researcher. It can give you access to an immense amount of information available on the web. However, each scenario of *webscraping* is particular. It is not always the case you can import data directly and easily as in previous examples.

Another problem is that the webscrapping code depends on the structure of the website. Any simple change in the *html* structure and your code will fail. You should be aware that maintaining a *webscraping* code can demand significant time and effort from the developer. If possible, you should always check for alternative sources of the same information.

Readers interested in learning more about this topic should study the functionalities of packages **XML** (Temple Lang, 2022), **RSelenium** (Harrison, 2022) and **splashr** (?). Each one of these is best suited to solve a particular web-scraping problem.

5.8 Exercises

All solutions are available at <https://www.msperlin.com/afedR>.

01 - Using the **BatchGetSymbols** package, download daily data of the Facebook stock (META) from *Yahoo Finance* for the period between 2019 and 2020. What is the lowest **unadjusted closing price (column price.close)** in the analyzed period?

02 - If you have not already done so, create a profile on the Quandl website¹³ and download the arabica coffee price data in the CEPEA database (Center

¹³<https://www.quandl.com/>

for Advanced Studies in Applied Economics)) between 2010-01-01 and 2020-12-31. What is the value of the most recent price?

03 - Use function `simfinR::simfinR_get_available_companies` to import data about all available companies in Simfin. How many companies do you find? Be aware that the number of rows is not necessarily equal to the number of companies. You need to check the number of unique values in column `name` (see function `dplyr::n_distinct`).

- a) 3409
- b) 5380
- c) 6078
- d) 4682
- e) 3984

04 - With package `simfinR`, download the PL (profit/loss) statement for FY (final year) data of company TESLA INC (id = 56317) for years 2018, 2019, 2020. What is the latest Net Income of the company?

- a) \$130,048,745
- b) \$417,949,743
- c) \$561,900,243
- d) \$273,999,244
- e) \$721,000,000

05 - Using function `tidyquant::tq_index`, download current composition of index SP400. What is the company with the highest percentage in the composition of the index?

Be aware that the answer is time-dependent and the reported result might be different from what you actually got in your R session.

- a) Kinsale Capital Group Inc.
- b) Steel Dynamics Inc.
- c) Deckers Outdoor Corporation
- d) Grand Canyon Education Inc.
- e) Vontier Corp

06 - Using again the `BatchGetSymbols` package, download data between 2019-01-01 and 2020-01-01 for the following tickers:

- AAPL: Apple Inc

- BAC: Bank of America Corporation
- GE: General Electric Company
- TSLA: Tesla, Inc.
- SNAP: Snap Inc.

Using the **adjusted closing price** column, what company provided higher return to the stock holder during the analyzed period?

Tip: this is an advanced exercise that will require some coding. To solve it, check out function `split` to split the dataframe of price data and `lapply` to map a function to each dataframe.

- a) TSLA
- b) SNAP
- c) AAPL
- d) BAC
- e) GE

Chapter

6

Dataframes and other objects

In R, everything is an object with its own properties. A numeric vector can interact with other numeric objects in operations such as multiplication, division, and addition. The same is not for objects of the character class, where mathematical properties are not valid or intuitive – it does not make sense to add a numeric value to a text or to divide a text for other text. But, the character class has other properties, such as allowing the user to look for a specific chunk of characters, splitting parts of a text, and replacing specific characters, among many other possibilities. **One of the most important aspects of working with R is learning the functionalities of the object classes.**

The basic object classes in R include numeric values, characters (text), factors, dates, among many other cases. However, base classes are stored in more complex data structures, such as **dataframes** (tables) and lists. Such an organization makes the job of manipulating data a lot easier. Imagine, for example, conducting a study using price and volume data for five hundred stocks. If we used one numeric object – a vector – for each stock's volume and price, we would have one thousand objects to handle in our *environment*. Although it is possible to work this way, the resulting code would be disorganized, difficult to understand, and prone to errors – the so-called spaghetti code¹. So twisted and tangled that it becomes difficult to use and maintain.

To avoid that, a simpler way to organize our data is to create an object named **my_data** and allocate prices and volumes there. All the information needed to perform the study would be in this object, facilitating the import and export of data. Our code would also be simpler and more structured

¹https://en.wikipedia.org/wiki/Spaghetti_code

as would only need to manipulate one object – a table, expanding it with new variables or relating it to other tables. Said that, let’s have a closer look at objects for structuring datasets, including `dataframes`, `lists` and `matrices`.

6.1 Dataframes

Without a doubt, the `dataframe` class is the most used and most important object to understand while learning R. You will spend most of your time manipulating one or many `dataframes` to get the result you need. A `dataframe` is simply a table with rows and columns. It allows for each column to have a different class. We can organize text data into a `dataframe` along with numbers, for example. This flexibility makes the `dataframe` an efficient object to represent heterogeneous datasets. Internally, a `dataframe` is a special type of a `list`, where each column is an atomic vector.

A `dataframe` can organize our work significantly. The tabular structure of a `dataframe` forces the data to be *paired*, where each row is a different data point with several pieces of information (columns). This simple data structure can accommodate an infinite variety of information. As new data points arrive, the number of rows increases. When a new variable is inserted in the analysis, we simply add a new column to the existing table.

Another positive aspect of using the `dataframe` class in R is that several functions expect a `dataframe` as input. For example, the data manipulation package `dplyr` (Wickham et al., 2022b) and the graphical package `ggplot2` (Wickham et al., 2022a) work from a `dataframe` only. Operations of importing and exporting information are mostly `dataframe` oriented. Without a doubt, `dataframes` are at the centre of functionalities in R, and you **must** master the manipulation of this object.

6.1.1 Creating `dataframes`

The `dataframe` object is one of R’s native classes and is implemented in the `base` package. We can, for example, create a `dataframe` with function `base::data.frame`. However, the `tidyverse` universe offers its own version of a `dataframe`, called `tibble`. Converting a `dataframe` to `tibble` is internal and automatic. A `tibble` object is more flexible than native `dataframes`, making it significantly easier to use. Following our preference for the `tidyverse` functions, from now on we will use `tibbles` as our version of `dataframes`.

We call function `tibble::tibble` to create a `dataframe`. Notice that func-

tion `tibble:data_frame` does the same job, but has been deprecated. This means that developers of the `tibble` package will give preference to the `tibble` function in the future and, as users, we should follow the recommendation. We will now use this function for creating a `dataframe` with financial data of different stocks.

```
library(tidyverse)

# set tickers
tickers <- c(rep('AAP',5),
             rep('COG', 5),
             rep('BLK', 5),
             rep('CAM',5) )

# set a date vector
dates <- as.Date(rep(c("2010-01-04", "2010-01-05", "2010-01-06",
                      "2010-01-07", "2010-01-08"), 4) )

# set prices
prices <- c(40.38,  40.14,  40.49,  40.48,  40.64,
           46.23,  46.17,  45.97,  45.56,  45.46,
           238.58, 239.61, 234.67, 237.25, 238.92,
           43.43,  43.96,  44.26,  44.5,   44.86)

# create tibble/dataframe
my_df <- tibble(tickers, dates, prices)

# print its first 5 rows
print(head(my_df))
```

```
R> # A tibble: 6 x 3
R>   tickers     dates    prices
R>   <chr>     <date>    <dbl>
R> 1 AAP      2010-01-04  40.4
R> 2 AAP      2010-01-05  40.1
R> 3 AAP      2010-01-06  40.5
R> 4 AAP      2010-01-07  40.5
R> 5 AAP      2010-01-08  40.6
R> 6 COG      2010-01-04  46.2
```

We used the function `rep` to replicate and facilitate the creation of the raw data for the `dataframe` object. Notice how all our data is now stored in a single object, facilitating access and organization of the resulting code. The

content of `my_df` can also be viewed in the RStudio interface. To do so, click on the object name in the *environment* tab, top right of the screen. After that, a viewer will appear on the main screen of the program, as in 6.1.

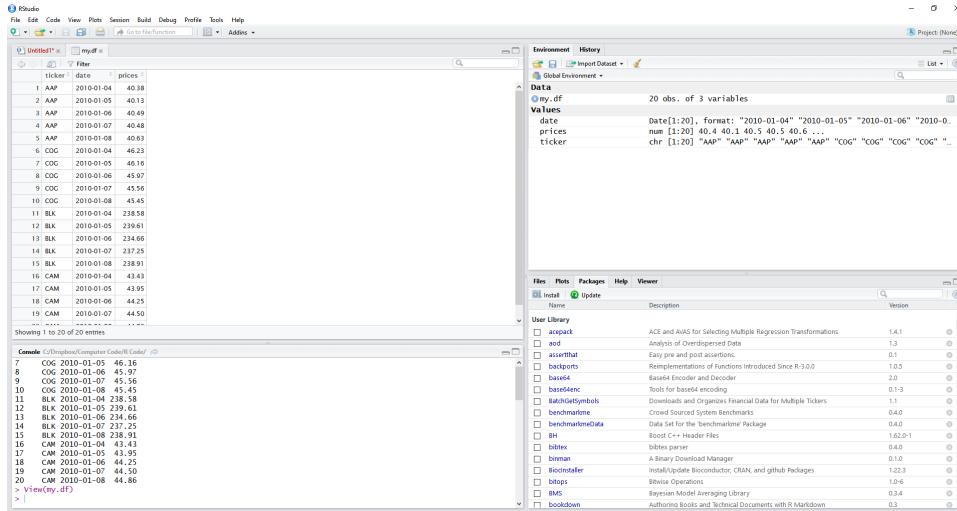


Figure 6.1: Example of viewing a dataframe in RStudio

The advantage of using the viewer is that you can explore the data and easily sort the columns by clicking on their names. For those who like to use the prompt, you can open the viewer with function `View`, as in `View(my_df)`.

6.1.2 Inspecting a Dataframe

Once you have a dataframe in your R session, a **mandatory** step is to check its contents. With time, this should become a healthy habit in your work. You need to be aware of:

- Properly defined column's names and classes;
- Correct number of rows and columns;
- The existence (or not) of missing data (`NA`).

We often have no control over how we get our data and errors in column names are very common. With that in mind, having meaningful column names that are intuitive, easy to access and without special characters is essential. Therefore, the first step in checking a `dataframe` is to analyze the names of columns and, if necessary, adjust the code. As a rule, **never touch the content of raw tables**, always use code to further manipulate the data.

It is also very important to make sure that the classes of columns from the imported `dataframe` are correctly specified. Otherwise, future column operations may cause an error. For example, if a column of numeric values is imported with the text class (`character`), any mathematical operation with that column will cause an error. **Checking column's classes right after importing the data is very important** and will save you a lot of time in the future.

You should also check for the number of NA values in the different columns. Whenever you find a large proportion of NA values in the imported table, you should find out what is going on and if the information is being correctly imported. As mentioned in the previous chapter, NA values are contagious and will turn anything it touches into another NA.

Back to the code, one of the most recommended functions for familiarizing yourself with a `dataframe` is `dplyr::glimpse`. It shows the name and class of the columns and the number of rows/columns. We used and abused this function in previous chapters. Here's a simple example:

```
# check content of my_df
glimpse(my_df)
```

```
R> Rows: 20
R> Columns: 3
R> $ tickers <chr> "AAP", "AAP", "AAP", "AAP", "AAP", "COG", ~
R> $ dates   <date> 2010-01-04, 2010-01-05, 2010-01-06, 2010--~
R> $ prices  <dbl> 40.38, 40.14, 40.49, 40.48, 40.64, 46.23, ~
```

Usually, the use of `glimpse` is sufficient to check if the data import process has succeeded. However, a deeper analysis can also help to understand how each column varies in the imported data and if that makes sense to our problem. Here comes the role of the `base::summary` function:

```
# check variation my_df
summary(my_df)
```

	tickers	dates	prices
R> Length:	20	Min. :2010-01-04	Min. : 40.14
R> Class :	character	1st Qu.:2010-01-05	1st Qu.: 42.73
R> Mode :	character	Median :2010-01-06	Median : 45.16
R>		Mean :2010-01-06	Mean : 92.08
R>		3rd Qu.:2010-01-07	3rd Qu.: 93.34
R>		Max. :2010-01-08	Max. :239.61

The objective of `summary` is to provide a grasp of the content for each column of the `dataframe`. Notice that `summary` interprets each column differently.

For the first case, column `ticker`, a *character* vector, it shows only its length. But, for column `prices`, a *numeric* vector, it presents its maximum, minimum, median and quartiles. We can use a simple call to `base::summary` for inspecting the contents of all columns. For example, an extreme observation (*outlier*) could easily be identified by analyzing the output of `summary`.



Whenever you start to work on a new `dataframe`, check its contents with functions `dplyr::glimpse` and `base::summary` and verify possible problems with the importing process or the content of the file itself. With experience you will notice that many future errors in code can be remedied by a simple inspection of the imported tables.

6.1.3 The *pipeline* Operator (%>%)

An important feature of the `tidyverse` universe is the extensive use of the *pipeline* operator, first proposed in package `magrittr` (Bache and Wickham, 2022) and defined by symbol `%>%`. The *pipeline* operator allows data operations to be performed sequentially and in a modular fashion, increasing readability and maintainability of the resulting code.

Imagine a situation where we have three functions to be applied to a `dataframe`. Each function depends on the output of another function. This requires chaining your calls, so one output feeds an object to the next call. Using the *pipeline* operator, we can write the `dataframe` manipulation procedure with the following code:

```
my_tab <- my_df %>%
  fct1(arg1) %>%
  fct2(arg2) %>%
  fct3(arg3)
```

We use symbol `%>%` at the end of each line to “glue” the operations. The `fct*` functions are operations performed at each step. The `arg*` objects are just arguments (options) for each function call. The result of each code line is passed to the next function sequentially. Thus, there is no need to create intermediate objects. To compare, here are two ways to perform the same operation **without** the *pipeline* operator:

```
# version 1
my_tab <- fct3(fct2(fct1(my_df,
                         arg1),
                         arg2),
```

```

    arg1)

# version 2
temp1 <- fct1(my_df, arg1)
temp2 <- fct2(temp1, arg2)

my_tab <- fct3(temp1, arg3)

```

Notice how the alternatives result in a messy code structure. Version one is the ugliest. You need to pay good attention to understand the code. You probably haven't noticed, but both codes have typos and would cause a bug. For the first, the last `arg1` argument should be `arg3` and, in the second, the `fct3` function is using the `temp1` dataframe and not `temp2`. This example shows how using *pipelines* makes the code more elegant and readable. From now on we will use the pipeline operator extensively.

6.1.4 Accessing Columns

To discover the names of the columns of a `dataframe`, we have two functions, `names` and `colnames`, with the exact same behavior:

```

# get names of columns with names
names(my_df)

R> [1] "tickers" "dates"    "prices"
colnames(my_df)

R> [1] "tickers" "dates"    "prices"

```

Both can also modify column names:

```

# set temp df
temp_df <- my_df

# change names
names(temp_df) <- paste0('Col', 1:ncol(temp_df))

# check names
names(temp_df)

R> [1] "Col1" "Col2" "Col3"

```

In this example, the way we use `names` differs greatly from other R functions. Here, we use the function on the left side of the `assign` symbol (`<-`).

Internally, we are defining an attribute of the `temp_df` object, the name of its columns.

To access a particular column of a `dataframe` as a vector, we can use operator `$` or the name/position of the column with double brackets:

```
# isolate columns of df
my_tickers <- my_df$tickers
my_prices <- my_df$prices

# print the results
print(head(my_tickers))
```

```
R> [1] "AAP" "AAP" "AAP" "AAP" "AAP" "COG"
print(head(my_prices))
```

```
R> [1] 40.38 40.14 40.49 40.48 40.64 46.23
```

It's worth knowing that, internally, dataframes are stored as `lists`, where each element is a column. This is important because some properties of `lists` also work for `dataframes`. One example is using a double bracket (`[[` `]]`) for selecting columns:

```
# select column in dataframe with list notation
print(my_df[[2]])
```

```
R> [1] "2010-01-04" "2010-01-05" "2010-01-06" "2010-01-07"
R> [5] "2010-01-08" "2010-01-04" "2010-01-05" "2010-01-06"
R> [9] "2010-01-07" "2010-01-08" "2010-01-04" "2010-01-05"
R> [13] "2010-01-06" "2010-01-07" "2010-01-08" "2010-01-04"
R> [17] "2010-01-05" "2010-01-06" "2010-01-07" "2010-01-08"
print(my_df[['tickers']])
```

```
R> [1] "AAP" "AAP" "AAP" "AAP" "AAP" "COG" "COG" "COG" "COG"
R> [10] "COG" "BLK" "BLK" "BLK" "BLK" "BLK" "CAM" "CAM" "CAM"
R> [19] "CAM" "CAM"
```

To access specific rows and columns of a `dataframe`, use single brackets with atomic vectors that indicate positions:

```
# accessing rows 1:5, column 2
print(my_df[1:5, 2])
```

```
R> # A tibble: 5 x 1
R>   dates
```

```
R> <date>
R> 1 2010-01-04
R> 2 2010-01-05
R> 3 2010-01-06
R> 4 2010-01-07
R> 5 2010-01-08

# accessing rows 1:5, columns 1 and 2
print(my_df[1:5, c(1,2)])
```

```
R> # A tibble: 5 x 2
R>   tickers dates
R>   <chr>    <date>
R> 1 AAP      2010-01-04
R> 2 AAP      2010-01-05
R> 3 AAP      2010-01-06
R> 4 AAP      2010-01-07
R> 5 AAP      2010-01-08

# accessing rows 1:5, all columns
print(my_df[1:5, ])
```

```
R> # A tibble: 5 x 3
R>   tickers dates     prices
R>   <chr>    <date>     <dbl>
R> 1 AAP      2010-01-04  40.4
R> 2 AAP      2010-01-05  40.1
R> 3 AAP      2010-01-06  40.5
R> 4 AAP      2010-01-07  40.5
R> 5 AAP      2010-01-08  40.6
```

Column selection can also be performed using names, as in the following example:

```
# selecting rows 1 to 3, columns 'ticker' and 'prices'
print(my_df[1:3, c('tickers', 'prices')])
```

```
R> # A tibble: 3 x 2
R>   tickers     prices
R>   <chr>       <dbl>
R> 1 AAP         40.4
R> 2 AAP         40.1
R> 3 AAP         40.5
```

Or, using the pipeline operator and function `dplyr::slice` and

```
dplyr::select:

my.temp <- my_df %>%
  select(tickers, prices) %>%
  slice(1:3) %>%
  glimpse()
```

```
R> Rows: 3
R> Columns: 2
R> $ tickers <chr> "AAP", "AAP", "AAP"
R> $ prices  <dbl> 40.38, 40.14, 40.49
```

6.1.5 Modifying a dataframe

To create new columns in a dataframe, simply use function `dplyr::mutate` and the *pipeline* operator.

```
# add columns with mutate
my_df <- my_df %>%
  mutate(ret = prices/lag(prices) - 1,
        seq_1 = 1:nrow(my_df),
        seq_2 = seq_1 +9) %>%
  glimpse()
```

```
R> Rows: 20
R> Columns: 6
R> $ tickers <chr> "AAP", "AAP", "AAP", "AAP", "AAP", "COG", ~
R> $ dates   <date> 2010-01-04, 2010-01-05, 2010-01-06, 2010-~
R> $ prices  <dbl> 40.38, 40.14, 40.49, 40.48, 40.64, 46.23, ~
R> $ ret      <dbl> NA, -0.0059435364, 0.0087194818, -0.000246~
R> $ seq_1    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ~
R> $ seq_2    <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20~
```

All new columns are defined as arguments in `dplyr::mutate`. Also note we use the `price` column to construct `ret`, the daily return of prices. A special case here is the creation of column `seq_2` based on `seq_1`, that is, even before it is explicitly calculated, it is possible to use the new column to create another one.

Another, more traditional way of creating new columns is using operator `$:`

```
# add new column with base R
my_df$seq_3 <- 1:nrow(my_df)

# check it
```

```
glimpse(my_df)
```

```
R> Rows: 20
R> Columns: 7
R> $ tickers <chr> "AAP", "AAP", "AAP", "AAP", "AAP", "COG", ~
R> $ dates   <date> 2010-01-04, 2010-01-05, 2010-01-06, 2010-~-
R> $ prices  <dbl> 40.38, 40.14, 40.49, 40.48, 40.64, 46.23, ~
R> $ ret      <dbl> NA, -0.0059435364, 0.0087194818, -0.000246~-
R> $ seq_1    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ~
R> $ seq_2    <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20~-
R> $ seq_3    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ~
```

Therefore, you can use `$` to either access or modify a dataframe.

Going further, if we try to create a column with the number of elements different than the number of rows of the target `dataframe`, an error will appear.

```
my_df <- my_df %>%
  mutate(seq_3 = 1:100) %>%
  glimpse()
```

```
R> Error: Column `seq_3` must be length 20 (the number of rows) ...
```

However, due to the simplified recycling rule, we can use single values to fill up a whole column:

```
my_df <- my_df %>%
  mutate(seq_3 = 1) %>%
  glimpse()
```

```
R> Rows: 20
R> Columns: 7
R> $ tickers <chr> "AAP", "AAP", "AAP", "AAP", "AAP", "COG", ~
R> $ dates   <date> 2010-01-04, 2010-01-05, 2010-01-06, 2010-~-
R> $ prices  <dbl> 40.38, 40.14, 40.49, 40.48, 40.64, 46.23, ~
R> $ ret      <dbl> NA, -0.0059435364, 0.0087194818, -0.000246~-
R> $ seq_1    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ~
R> $ seq_2    <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20~-
R> $ seq_3    <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
```

To remove columns from a `dataframe`, use function `dplyr::select` with the minus symbol for the undesired columns:

```
# removing columns
my_df.temp <- my_df %>%
```

```
select(-seq_1, -seq_2, -seq_3) %>%
glimpse()
```

```
R> Rows: 20
R> Columns: 4
R> $ tickers <chr> "AAP", "AAP", "AAP", "AAP", "AAP", "COG", ~
R> $ dates   <date> 2010-01-04, 2010-01-05, 2010-01-06, 2010-~ 
R> $ prices  <dbl> 40.38, 40.14, 40.49, 40.48, 40.64, 46.23, ~
R> $ ret      <dbl> NA, -0.0059435364, 0.0087194818, -0.000246~
```

Using base R, the traditional way of removing columns is to allocate a single value `NULL` to its contents:

```
# set temp df
temp_df <- my_df

# remove cols
temp_df$prices <- NULL
temp_df$dates  <- NULL
temp_df$ret    <- NULL
temp_df$tickers <- NULL

# check it
glimpse(temp_df)
```

```
R> Rows: 20
R> Columns: 3
R> $ seq_1 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1~ 
R> $ seq_2 <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ~
R> $ seq_3 <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
```

6.1.6 Filtering rows of a dataframe

A fairly common `dataframe` operation in R is to filter rows according to one or more conditions. For example, if we only wanted data from the '`COG`' stock, we could use the `filter` function to filter the table:

```
# filter df for single stock
my_df.temp <- my_df %>%
  filter(tickers == 'COG') %>%
  glimpse()
```

```
R> Rows: 5
R> Columns: 7
```

```
R> $ tickers <chr> "COG", "COG", "COG", "COG", "COG"
R> $ dates   <date> 2010-01-04, 2010-01-05, 2010-01-06, 2010-0~
R> $ prices  <dbl> 46.23, 46.17, 45.97, 45.56, 45.46
R> $ ret      <dbl> 0.137549213, -0.001297859, -0.004331817, ~
R> $ seq_1    <int> 6, 7, 8, 9, 10
R> $ seq_2    <dbl> 15, 16, 17, 18, 19
R> $ seq_3    <dbl> 1, 1, 1, 1, 1
```

We can go further and also filter data for 'COG' and dates after '2010-01-05':

```
# filter df for single stock and date
my_df.temp <- my_df %>%
  filter(tickers == 'COG',
         dates > as.Date('2010-01-05')) %>%
  glimpse()
```

```
R> Rows: 3
R> Columns: 7
R> $ tickers <chr> "COG", "COG", "COG"
R> $ dates   <date> 2010-01-06, 2010-01-07, 2010-01-08
R> $ prices  <dbl> 45.97, 45.56, 45.46
R> $ ret      <dbl> -0.004331817, -0.008918860, -0.002194908
R> $ seq_1    <int> 8, 9, 10
R> $ seq_2    <dbl> 17, 18, 19
R> $ seq_3    <dbl> 1, 1, 1
```

Here we used symbol `==` to test for equality in column `ticker` and *greater than* (`>`) for selecting the rows where the dates are after 2010-01-05. There are plenty more operators for all kinds of logical conditions. We will study these in chapter 7.

6.1.7 Sorting a dataframe

After creating or importing a `dataframe`, we can sort its rows according to the values of any column. A common case where a sort operation is needed is when financial or economic data is imported, but the dates are not ascending. Depending on the situation, it may be easier – or expected – to deal with data where the dates are always increasing along the rows, from top to bottom. The sorting operation in `dataframes` is performed using function `dplyr::arrange` or `base::order`.

As an example, consider creating a `dataframe` with these values:

```
# set new df
my_df <- tibble(col1 = c(4, 1, 2),
                 col2 = c(1, 1, 3),
                 col3 = c('a','b','c'))

# print it
print(my_df)
```

```
R> # A tibble: 3 x 3
R>   col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     4     1 a
R> 2     1     1 b
R> 3     2     3 c
```

We use function `dplyr::arrange` and the *pipeline* operator to order the whole `dataframe` by the ascending values of column `col1`:

```
# sort ascending, by col1
my_df <- my_df %>%
  arrange(col1) %>%
  print()
```

```
R> # A tibble: 3 x 3
R>   col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     2     3 c
R> 3     4     1 a
```

We can also sort by descending values using `desc`:

```
# sort descending, col1 and col2
my_df <- my_df %>%
  arrange(desc(col1)) %>%
  print()
```

```
R> # A tibble: 3 x 3
R>   col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     4     1 a
R> 2     2     3 c
R> 3     1     1 b
```

And, for multiple columns, using extra arguments in `arrange`:

```
# sort ascending, by col2 and col1
my_df <- my_df %>%
  arrange(col2, col1) %>%
  print()
```

```
R> # A tibble: 3 x 3
R>   col1   col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     4     1 a
R> 3     2     3 c
```

As for base R, function `order` returns the position of the elements for the sorted vector. With the first column of `my_df`, the positions of the elements in ascending order are:

```
# set index with positions of ascending order in col1
idx <- order(my_df$col1)

# print it
print(idx)
```

```
R> [1] 1 3 2
```

Therefore, when using the output of function `order` as an index of an existing `dataframe`, you get a new version of the `dataframe`, where all rows are set according to the ascending values of a particular column. See an example next:

```
# order my_df by col1
my_df.2 <- my_df[order(my_df$col1), ]

# print result
print(my_df.2)
```

```
R> # A tibble: 3 x 3
R>   col1   col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     2     3 c
R> 3     4     1 a
```

This operation may also be performed considering more than one column. See the following example, where we sort the rows of `my_df` using columns `col2` and `col1`.

```
# sort df with col2 and col1
my_df.3 <- my_df[order(my_df$col2, my_df$col1), ]

# print result
print(my_df.3)
```

```
R> # A tibble: 3 x 3
R>   col1   col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     4     1 a
R> 3     2     3 c
```

6.1.8 Combining and Aggregating dataframes

In the practice of manipulating data, often you must aggregate multiple **dataframes** into a single one. This usually happens when the heterogeneous data is imported from different sources and we must bind them into a single table. In the simplest case of combining **dataframes**, we join them according to the rows (vertically) or columns (horizontally). For that, we have functions `dplyr::bind_rows` (alternative to `base::rbind`) and `dplyr::bind_cols` (alternative to `base::cbind`). Examples of usage are given next.

```
# set two dfs with same colnames
my_df_1 <- tibble(col1 = 1:5,
                   col2 = rep('a', 5))
my_df_2 <- tibble(col1 = 6:10,
                   col2 = rep('b', 5))

# bind them by rows
my_df <- bind_rows(my_df_1, my_df_2)

# print result
print(my_df)
```

```
R> # A tibble: 10 x 2
R>   col1   col2
R>   <int> <chr>
R> 1     1 a
R> 2     2 a
R> 3     3 a
R> 4     4 a
```

```
R> 5      5 a
R> 6      6 b
R> 7      7 b
R> 8      8 b
R> 9      9 b
R> 10     10 b
```

Notice that, in the previous example, the names of columns are the same between `my_df_1` and `my_df_2`. Function `dplyr::bind_rows` is very clever and will search for shared names and correctly bind the data, even if the column positions are different. If we swapped the positions of the columns, there would be no change in the result of the bind operation.

Another interesting aspect of `dplyr::bind_rows` is that, if the names of the columns don't match, the unmatched columns will return a `NA` result (not available). This means we can bind tables with different column names:

```
# set two df with different colnames
my_df_1 <- tibble(col1 = 1:5,
                   col2 = rep('a', 5))
my_df_2 <- tibble(col1 = 6:10,
                   col3 = rep('b', 5))

# bind them by rows (NA values for missing cols)
my_df <- bind_rows(my_df_1,
                    my_df_2)

# print result
print(my_df)
```

```
R> # A tibble: 10 x 3
R>   col1 col2 col3
R>   <int> <chr> <chr>
R> 1     1 a    <NA>
R> 2     2 a    <NA>
R> 3     3 a    <NA>
R> 4     4 a    <NA>
R> 5     5 a    <NA>
R> 6     6 <NA> b
R> 7     7 <NA> b
R> 8     8 <NA> b
R> 9     9 <NA> b
R> 10    10 <NA> b
```

For the case of column bind with function `dplyr::bind_cols`, the names of the columns must be different, but the number of rows must be the same:

```
# set two dfs
my_df_1 <- tibble(col1 = 1:5,
                    col2 = rep('a', 5))
my_df_2 <- tibble(col3 = 6:10,
                    col4 = rep('b', 5))

# column bind dfs
my_df <- cbind(my_df_1, my_df_2)

# print result
print(my_df)
```

```
R>   col1 col2 col3 col4
R> 1     1     a     6     b
R> 2     2     a     7     b
R> 3     3     a     8     b
R> 4     4     a     9     b
R> 5     5     a    10     b
```

Sometimes, aggregating different tables won't be as easy as simply row or column binding. For example, imagine you have data for annual unemployment rates in one `dataframe` and data for monthly inflation in another. There, you can't just bind their columns as their frequency and number of rows are different. The solution is to use an index (or table key) – a vector of dates – that relates both tables.

For that, you can use functions `dplyr::join*` to merge two dataframes using one or more indexes. This includes `dplyr::inner_join`, `dplyr::left_join`, `dplyr::full_join`, and many others. The main difference between them is how you treat the row cases without a match. For example, `dplyr::inner_join` keeps only the data where it finds a matching index, while `dplyr::full_join` keep everything from both tables and fills the missing information with `NA` values. Let's have a closer look by using a practical example of both.

```
# set df
my_df_1 <- tibble(date = as.Date('2016-01-01')+0:10,
                   x = 1:11)

my_df_2 <- tibble(date = as.Date('2016-01-05')+0:10,
                   y = seq(20,30, length.out = 11))
```

Please do notice that both dataframes share a column called `date`, which will be automatically used as a matching index.

```
# aggregate tables
my_df <- inner_join(my_df_1,
                     my_df_2)

R> Joining, by = "date"
glimpse(my_df)

R> Rows: 7
R> Columns: 3
R> $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-01-08, 2016-01-09, 2016-01-10, 2016-01-11
R> $ x     <int> 5, 6, 7, 8, 9, 10, 11
R> $ y     <dbl> 20, 21, 22, 23, 24, 25, 26
```

Now with `dplyr::full_join`:

```
# aggregate tables
my_df <- full_join(my_df_1,
                     my_df_2)

R> Joining, by = "date"
glimpse(my_df)

R> Rows: 15
R> Columns: 3
R> $ date <date> 2016-01-01, 2016-01-02, 2016-01-03, 2016-01-04, 2016-01-05, 2016-01-06, 2016-01-07, 2016-01-08, 2016-01-09, 2016-01-10, 2016-01-11, NA, NA, NA, NA, NA
R> $ x     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, NA, NA, NA, NA, NA
R> $ y     <dbl> NA, NA, NA, NA, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31
```

Notice the difference in the number of rows from one to the other. When using `dplyr::full_join`, all unmatched cases are set to `NA`.

If we had `dataframes` with different column names, we can also set the index explicitly with argument `by` of `dplyr::*_join`. See the next example, where we set different column names and still match the data using a vector of dates:

```
# set df
my_df_3 <- tibble(ref_date = as.Date('2016-01-01')+0:10,
                    x = 1:11)

my_df_4 <- tibble(my_date = as.Date('2016-01-05')+0:10,
                    y = seq(20,30, length.out = 11))
```

```
# join by my_df.3$ref.date and my_df.4$my.date
my_df <- inner_join(my_df_3, my_df_4,
                     by = c('ref_date' = 'my_date'))

glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ ref_date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-
R> $ x          <int> 5, 6, 7, 8, 9, 10, 11
R> $ y          <dbl> 20, 21, 22, 23, 24, 25, 26
```

Whenever you need to combine tables that share information, use one of the `dplyr::*_join` functions. The decision of how to treat the unmatched cases will depend on the problem you’re analyzing.

6.1.9 Extensions of the `dataframe` Class

As mentioned in the previous chapter, one benefit of using R is the existence of a range of packages designed to deal with specific problems. This is also true for extensions of the basic data structure. While the `tibble` class is a good solution for most cases, sometimes, it can make more sense to store the data in a specific type of custom object. Over time, several solutions have been developed.

For example, it is common in economic and financial research to work with numeric data indexed by time. We can store this data in a matrix format, so each line represents dates and each column represents a variable. With this structure, time operations, such as period aggregations, are easier to perform. This is the main idea of package `xts` (Ryan and Ulrich, 2022). The great benefit of this alternative `dataframe` is that several functions for time aggregation and manipulation are available. We can turn a whole set of daily data for several variables to the weekly frequency in one line of code. In addition, various other functions automatically recognize the time index and adapt accordingly. One example is the creation of a figure with the values of a variable over time. The horizontal axes of the figure are automatically arranged as dates.

See the following example, where we represent the previous stock data as an `xts` object:

```
# load pkg
library(xts)
```

```

# set ticker symbols as a vector
tickers <- c('AAP', 'COG', 'BLK', 'CAM')

# set a date vector
dates <- as.Date(c("2010-01-04", "2010-01-05", "2010-01-06",
                  "2010-01-07", "2010-01-08"))

# set prices as matrix
price_matrix <- matrix(c(40.38, 40.13, 40.49, 40.48, 40.63,
                         46.23, 46.16, 45.97, 45.56, 45.45,
                         238.58, 239.61, 234.66, 237.25, 238.91,
                         43.43, 43.95, 44.25, 44.5, 44.86),
                         nrow = length(dates))

# set xts object
my_xts <- xts(price_matrix, order.by = dates)

# set colnames
colnames(my_xts) <- tickers

# print it
print(my_xts)

```

```

R>          AAP   COG    BLK    CAM
R> 2010-01-04 40.38 46.23 238.58 43.43
R> 2010-01-05 40.13 46.16 239.61 43.95
R> 2010-01-06 40.49 45.97 234.66 44.25
R> 2010-01-07 40.48 45.56 237.25 44.50
R> 2010-01-08 40.63 45.45 238.91 44.86

```

```

# show its class
class(my_xts)

```

```
R> [1] "xts" "zoo"
```

In creating the `xts` object, notice how the time index is explicitly defined using argument `order.by`. This is a necessary step in creating every `xts` object.

The previous code can give the impression that the object `my_xts` is similar to a native `dataframe`. However, make no mistake. By having an explicit time index, object `my_xts` can be used for several privileged procedures. See the following example, where we create a new `xts` object with two columns

and calculate the average of each column on a weekly basis.

```
# set number of time periods
N <- 500

# create matrix with data
my_mat <- matrix(c(seq(1, N), seq(N, 1)), nrow=N)

# set xts object
my_xts <- xts(my_mat, order.by = as.Date('2018-01-01')+1:N)

# apply mean function for each week
my_xts_weekly_mean <- apply.weekly(my_xts, mean)

# print result
print(head(my_xts_weekly_mean))
```

```
R>           X.1   X.2
R> 2018-01-07 3.5 497.5
R> 2018-01-14 10.0 491.0
R> 2018-01-21 17.0 484.0
R> 2018-01-28 24.0 477.0
R> 2018-02-04 31.0 470.0
R> 2018-02-11 38.0 463.0
```

In finance and economics, these time aggregations with `xts` objects are useful when working with data at different time frequencies. It is common to aggregate transaction data in the financial market for high-frequency intervals of 5 by 5 minutes. Such a procedure is easily accomplished in R through the correct representation of the data as `xts` objects. There are several other features in this package. Users that work frequently with time-indexed data are encouraged to read the manual² and learn more about it.

Package `xts` is not alone as an alternative to `dataframes`. For example, the data structure proposed by package `data.table` (Dowle and Srinivasan, 2022) prioritizes processing time and uses a compact notation. If you like short notations for writing code and need a quick execution time, `data.table` is an impressive and powerful package. As another example, package `tibbletime` (Vaughan and Dancho, 2022) is, well, a time-oriented version of `tibble`, bringing together the benefits of `tidyverse` and `xts`.

²<https://cran.r-project.org/web/packages/xts/xts.pdf>

6.1.10 Other Useful Functions for Handling dataframes

head Returns the first **n** rows of a **dataframe**. This function is mostly used for showing only a small part of a **dataframe** in the prompt.

```
# set df
my_df <- tibble(col1 = 1:5000,
                  col2 = rep('a', 5000))

# print its first 5 rows
print(head(my_df, 5))
```

```
R> # A tibble: 5 x 2
R>   col1 col2
R>   <int> <chr>
R> 1     1 a
R> 2     2 a
R> 3     3 a
R> 4     4 a
R> 5     5 a
```

tail - Returns the last **n** rows of a **dataframe**. Also used to glimpse the last rows of a **dataframe**.

```
# print its last 5 rows
print(tail(my_df, 5))
```

```
R> # A tibble: 5 x 2
R>   col1 col2
R>   <int> <chr>
R> 1 4996 a
R> 2 4997 a
R> 3 4998 a
R> 4 4999 a
R> 5 5000 a
```

complete.cases - Returns a logical vector with the same length as the number of rows of the **dataframe**, containing TRUE when all columns have non NA values and FALSE otherwise.

```
# create df
my_df <- tibble(x = c(1:5, NA, 10),
                  y = c(5:10, NA))
```

```
# show df
print(my_df)

R> # A tibble: 7 x 2
R>   x     y
R>   <dbl> <int>
R> 1 1     5
R> 2 2     6
R> 3 3     7
R> 4 4     8
R> 5 5     9
R> 6 NA    10
R> 7 10    NA

# print logical test of complete.cases
print(complete.cases(my_df))

R> [1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE

# print all rows where there is at least one NA
print(which(!complete.cases(my_df)))

R> [1] 6 7

na.omit - Returns a dataframe without the rows where a NA in any column is found.

print(na.omit(my_df))

R> # A tibble: 5 x 2
R>   x     y
R>   <dbl> <int>
R> 1 1     5
R> 2 2     6
R> 3 3     7
R> 4 4     8
R> 5 5     9

unique - Returns a dataframe where all duplicated rows are removed and only the unique cases, row-wise, are kept.

# set df with repeating rows
my_df <- data.frame(col1 = c(1, 1, 2, 3, 3, 4, 5),
                     col2 = c('A', 'A', 'A', 'C', 'C', 'B', 'D'))

# print it
```

```

print(my_df)

R>   col1 col2
R> 1    1    A
R> 2    1    A
R> 3    2    A
R> 4    3    C
R> 5    3    C
R> 6    4    B
R> 7    5    D

# print unique df
print(unique(my_df))

```

```

R>   col1 col2
R> 1    1    A
R> 3    2    A
R> 4    3    C
R> 6    4    B
R> 7    5    D

```

6.2 Lists

A **list** is a flexible container that can hold many elements. Unlike atomic vectors, a **list** has no restriction on the classes or types of elements – we can group **numeric** objects with **character** objects, **factor** with **Dates** and even **lists** within **lists**. Likewise, each element of a list need not have the same length as the others. We can also name each element within a **list**. These properties make the **list** class the most flexible object in R. It is not by accident that several functions in R return an object of type **list**.

6.2.1 Creating lists

A **list** can be created with the **base:::list** command, followed by their comma-separated elements:

```

# create list
my_l <- list(c(1, 2, 3),
              c('a', 'b'),
              factor('A', 'B', 'C'),
              data.frame(col1 = 1:5))

```

```
# use base::print
print(my_1)
```

```
R> [[1]]
R> [1] 1 2 3
R>
R> [[2]]
R> [1] "a" "b"
R>
R> [[3]]
R> [1] <NA>
R> Levels: C
R>
R> [[4]]
R>   col1
R> 1     1
R> 2     2
R> 3     3
R> 4     4
R> 5     5
```

```
# use dplyr::glimpse
glimpse(my_1)
```

```
R> List of 4
R> $ : num [1:3] 1 2 3
R> $ : chr [1:2] "a" "b"
R> $ : Factor w/ 1 level "C": NA
R> $ :'data.frame':    5 obs. of  1 variable:
R>   ..$ col1: int [1:5] 1 2 3 4 5
```

Notice how we gather four objects: a numeric vector, a character vector, a factor and a dataframe/tibble into a single **list**. Also, notice that a **list** type object is printed differently than atomic vectors. The elements of the **list** are separated vertically and the content appears within double brackets ([[]]). We will soon learn that we can access its elements in the same way, by using double brackets.

Following other objects, the elements of a **list** can also be named, making it easier to work with heterogenous data. For example, let's consider a set of data with information about a particular company traded in the NYSE market:

```
# set named list
my_named_1 <- list(tickers = 'CMPY',
                    markets = 'NYSE',
                    df_prices = data.frame(P = c(1,1.5,2,2.3),
                                           ref_date = Sys.Date()+0:3))

# check content
glimpse(my_named_1)
```

```
R> List of 3
R> $ tickers : chr "CMPY"
R> $ markets : chr "NYSE"
R> $ df_prices:'data.frame': 4 obs. of 2 variables:
R>   ..$ P      : num [1:4] 1 1.5 2 2.3
R>   ..$ ref_date: Date[1:4], format: "2022-11-23" ...
```

The data is organized in a single object, facilitating the import/export of the data and the analysis.



Every time you work with lists, make your life easier by naming all elements intuitively. Avoid the use of the position of the element in a list, which can change when lists expand or contract. Using names facilitates access and avoids possible errors in the code.

6.2.2 Accessing the Elements of a list

As mentioned, the individual elements of a `list` can be accessed with double brackets (`[[]]`), as in:

```
# accessing elements from list
print(my_named_1[[2]])
```

```
R> [1] "NYSE"
print(my_named_1[[3]])

R>      P    ref_date
R> 1 1.0 2022-11-23
R> 2 1.5 2022-11-24
R> 3 2.0 2022-11-25
R> 4 2.3 2022-11-26
```

You can also access the elements of a `list` with simple brackets (`[]`), but be careful with this operation as the result will not be the element itself, but

another `list`. This is a common mistake. See it below:

```
# set list
my_l <- list('a',
              c(1, 2, 3),
              factor('a', 'b'))
```



```
# check classes
class(my_l[[2]])
```

```
R> [1] "numeric"
class(my_l[2])
```

```
R> [1] "list"
```

If we try to add an element to `my_l[2]`, we will receive an error message.

```
# adding an element to a list (WRONG)
my_l[2] + 1
```

```
R> Error in my_l[2] + 1: non-numeric argument to binary operator
```

An error is returned because a `list` object cannot be summed with a `numeric` object. To fix it, simply use double brackets, as in `my_l[[2]] + 1`. Accessing elements of a list with simple brackets is only useful when looking for a sub-list within a larger list. As an example, if we wanted to obtain the first and second elements of `my_l`, we would write:

```
# set new list with the first and second element of my_l
my_new_l <- my_l[c(1,2)]
```

```
# print result
print(my_new_l)
```

```
R> [[1]]
R> [1] "a"
R>
R> [[2]]
R> [1] 1 2 3
```

With the named lists, we can access its elements with operator `$` as in `my_named_l$df_prices` or using the element's name. In general, this is a more efficient and advised way of working with `lists`. Avoid using positional access of a `list`. The problem is, if you are working interactively with a `list`, the position of the elements may change as new data arrives. Using

names prevents this problem because, by modifying the `list` and adding elements, you can change the order of elements, but not the names.



Be aware that RStudio's *autocomplete* tool also works for lists. To use it, enter the list name followed by `$` and press *tab*. A dialog box with all the elements available in the list will appear. From there, just select the desired element by pressing *enter*.

Next, we provide several examples of how to access the elements of a `list` using operator `$` and double brackets.

```
# different ways to access a list
my_named_1$tickers
my_named_1$markets
my_named_1[['tickers']]
my_named_1[['markets']]
```

Another useful trick for working with lists is you can access all inner elements directly – in one line of code – by simply using consecutive brackets or names. See below:

```
my_l <- list(slot1 = c(num1 = 1,
                      num2 = 2,
                      num3 = 3),
              slot2 = c('a', 'b'))

# access the second value of the first element of my_l
print(my_l[[1]][2])
```

```
R> num2
R>     2
# access the first value of the second element of my_l
print(my_l[[2]][1])
```

```
R> [1] "a"
# access the value 'num3' in 'slot1'
print(my_l[['slot1']]['num3'])
```

```
R> num3
R>     3
```

This operation is very useful when interested in a few elements within a larger object. It avoids the need for creating intermediate objects.

6.2.3 Adding and Removing Elements from a list

To add or replace elements in a `list`, just set the new object in the desired position:

```
# set list
my_l <- list('a', 1, 3)
glimpse(my_l)

R> List of 3
R> $ : chr "a"
R> $ : num 1
R> $ : num 3

# add new elements to list
my_l[[4]] <- c(1:5)
my_l[[2]] <- c('b')

# print result
glimpse(my_l)
```

```
R> List of 4
R> $ : chr "a"
R> $ : chr "b"
R> $ : num 3
R> $ : int [1:5] 1 2 3 4 5
```

This operation is also possible with the use of names and `$`:

```
# set list
my_l <- list(elem1 = 'a',
             name1=5)

# set new element
my_l$name2 <- 10

# check it
glimpse(my_l)
```

```
R> List of 3
R> $ elem1: chr "a"
R> $ name1: num 5
R> $ name2: num 10
```

To remove elements from a `list`, set the element to the reserved symbol `NULL`, as in:

```
# set list
my_l <- list(text = 'b', num1 = 2, num2 = 4)
glimpse(my_l)
```

```
R> List of 3
R> $ text: chr "b"
R> $ num1: num 2
R> $ num2: num 4
```

```
# remove elements
my_l[[3]] <- NULL
glimpse(my_l)
```

```
R> List of 2
R> $ text: chr "b"
R> $ num1: num 2
```

```
# remove elements
my_l$num1 <- NULL
glimpse(my_l)
```

```
R> List of 1
R> $ text: chr "b"
```

Another way of removing elements from a `list` is to use a negative index, which will exclude it from the returned object. See the next example, where we remove the second element of a `list` using a negative index.

```
# set list
my_l <- list(a = 1,
             b = 'text')

# remove second element
glimpse(my_l[[-2]])
```

```
R> num 1
```

As with atomic vectors, removing elements of a `list` can also be accomplished with logical conditions. See next:

```
# set list
my_l <- list(1, 2, 3, 4)

# remove elements by condition
my_l[my_l > 2] <- NULL
glimpse(my_l)
```

```
R> List of 2
R> $ : num 1
R> $ : num 2
```

However, note this operation only works because all elements of `my_1` are numeric, and a logical test can be applied to all cases. If that is impossible for a particular element, R will return an `NA` value.

6.2.4 Processing the Elements of a list

A very important point about working with `lists` is that its elements can be iterated in a very simple and direct way. For example, if you have a `list` with twenty dataframes, you can easily apply function `base::summary` to all of them in a single line of code. As an example, consider a list of numeric vectors of different sizes:

```
# set list with different numerical vectors.
my_1_num <- list(c(1,2,3),
                  seq(1:50),
                  seq(-5,5, by=0.5))
```

Let's assume we need to calculate the average of each vector in `my_1_num` and store the result in an atomic vector. We could do this operation by calling the `mean` function to each element of the `list`, as in:

```
# calculate means
mean_1 <- mean(my_1_num[[1]])
mean_2 <- mean(my_1_num[[2]])
mean_3 <- mean(my_1_num[[3]])

# print result
print(c(mean_1, mean_2, mean_3))
```

```
R> [1] 2.0 25.5 0.0
```

However, the code looks bad and it took three lines of code, not one! An easier, more elegant, and smarter way of doing that would be to use the `sapply` function, which will apply a function to each element of a list. All you need is the name of the list object and the name of the function used to process each element:

```
# using sapply
my_mean <- sapply(my_1_num, mean)

# print result
```

```
print(my_mean)

R> [1] 2.0 25.5 0.0
```

As expected, the result is identical to the previous example. Using function `sapply` is preferable, because it is more compact and efficient than the alternative - creating `mean_1`, and `mean_2` and `mean_3`. Notice the first example code only works for a `list` with three elements. If we had a fourth element and we wanted to keep this code structure, we would have to add a new line `mean_4 <- mean(my_1_num[[4]])` and modify the output command to `print <-c(mean_1, mean_2, mean_3, mean_4))`.

Intelligently, function `sapply` works the same way in `lists` of any size. If we had more elements in `my_1_num`, no modification is necessary for `my_mean <- sapply(my_1_num, mean)`, making it easier to extend the code for more information. By combining a flexible object, such as a `list`, with the programming capacity of R, performing extensive operations in many complex objects becomes easy.

Using generic procedures is one premise of good and efficient programming practices. For the case of R, the rule is simple: **always write code flexible to the size of your objects**. The arrival of new data should never require modifications in the code. This is called the *DRY* rule (**don't repeat yourself**). If you are repeating lines of code, as in the previous example, certainly a more elegant and flexible solution could be used. This section only gives a taste of the programming capacity of R. This will be explained in greater detail in chapter 8.

6.2.5 Other Useful Functions

`unlist` - Returns the elements of a `list` in a single atomic vector.

```
my_named_1 <- list(ticker = 'XXXX4',
                     price = c(1,1.5,2,3),
                     market = 'Be')
my_unlisted <- unlist(my_named_1)
print(my_unlisted)
```

```
R> ticker  price1  price2  price3  price4  market
R> "XXXX4"    "1"    "1.5"    "2"    "3"    "Be"
class(my_unlisted)
```

```
R> [1] "character"
```

as.list - Converts an object to the `list` type.

```
my_x <- 10:13
my_x_as_list <- as.list(my_x)
print(my_x_as_list)
```

```
R> [[1]]
R> [1] 10
R>
R> [[2]]
R> [1] 11
R>
R> [[3]]
R> [1] 12
R>
R> [[4]]
R> [1] 13
```

names - Returns or defines the names of the elements of a `list`.

```
my_l <- list(value1 = 1, value2 = 2, value3 = 3)
print(names(my_l))
```

```
R> [1] "value1" "value2" "value3"
my_l <- list(1,2,3)
names(my_l) <- c('num1', 'num2', 'num3')
print(my_l)
```

```
R> $num1
R> [1] 1
R>
R> $num2
R> [1] 2
R>
R> $num3
R> [1] 3
```

6.3 Matrices

A matrix is a two-dimensional representation of numbers, arranged in rows and columns. Using matrices is a powerful way of representing numerical data in two dimensions and, in certain situations, matrix functions can simplify complex mathematical operations.

In R, matrices are objects with two dimensions, where all elements must have the same class. You can think of matrices as atomic vectors with one extra dimension. In matrices, lines and columns can be named. When used correctly, `matrix` objects can facilitate the storage and context of the data.

A simple example of using matrices in finance is the representation of stock prices over time. The rows of the matrix represents the different dates, and the columns set each stock apart:

```
R> New names:
R> * `` -> `...`
```

	AAP	COG	BLK	CAM
1	40.38	46.23	238.58	43.43
2	40.14	46.17	239.61	43.96
3	40.49	45.97	234.67	44.26
4	40.48	45.56	237.25	44.50
5	40.64	45.46	238.92	44.86

The above matrix could be created in R with the following code:

```
# set raw data with prices
raw_data <- c(40.38, 40.14, 40.49, 40.48, 40.64,
             46.23, 46.17, 45.97, 45.56, 45.46,
             238.58, 239.61, 234.67, 237.25, 238.92,
             43.43, 43.96, 44.26, 44.5, 44.86)

# create matrix
my_mat <- matrix(raw_data, nrow = 5, ncol = 4)
colnames(my_mat) <- c('AAP', 'COG', 'BLK', 'CAM')
rownames(my_mat) <- c("2010-01-04", "2010-01-05", "2010-01-06",
                      "2010-01-07", "2010-01-08")

# print result
print(my_mat)
```

```
R>           AAP    COG    BLK    CAM
R> 2010-01-04 40.38 46.23 238.58 43.43
R> 2010-01-05 40.14 46.17 239.61 43.96
R> 2010-01-06 40.49 45.97 234.67 44.26
R> 2010-01-07 40.48 45.56 237.25 44.50
R> 2010-01-08 40.64 45.46 238.92 44.86
```

We set the number of rows and columns explicitly with arguments `nrow = 4` and `ncol = 3` in `base::matrix`. The names of rows and columns are defined with functions `colnames` and `rownames`, using a left side notation as in `rownames(my_mat) <- c(...)`. Going further, we can also retrieve the names of rows and columns with the same functions:

```
# print the names of columns
print(colnames(my_mat))
```

```
R> [1] "AAP" "COG" "BLK" "CAM"
```

```
# print the names of rows
print(rownames(my_mat))
```

```
R> [1] "2010-01-04" "2010-01-05" "2010-01-06" "2010-01-07"
```

```
R> [5] "2010-01-08"
```

After matrix `my_mat` is created, we have at our disposal all its numerical properties. A simple example of using matrix operations in finance is the calculation of the value of a portfolio. If an investor has 200 shares of AAP, 300 share of COG, 100 of BLK and 50 of CAM, the value of his portfolio over time can be calculated as follows:

$$V_t = \sum_{i=1}^4 N_i P_{i,t}$$

In this formula, N_i is the number of shares purchased for each asset, and $P_{i,t}$ is the price of stock i at date t . This is a simple operation to be performed with a matrix multiplication. Translating the procedure to R code, we have:

```
# set vector with shares purchased
my_stocks <- as.matrix(c(200, 300, 100, 50), nrow = 4)

# get value of portfolio with matrix multiplication
my_port <- my_mat %*% my_stocks

# print result
print(my_port)
```

```
R> [,1]
R> 2010-01-04 47974.5
R> 2010-01-05 48038.0
R> 2010-01-06 47569.0
R> 2010-01-07 47714.0
```

```
R> 2010-01-08 47901.0
```

In this last example, we use symbol `%*%`, which does a matrix multiplication between two objects of the class `matrix`. The output shows the value of the portfolio over time, resulting in a small loss for the investor on the last date.

A `matrix` object is also flexible with its content, as long as it is a single class. For example, you can create matrices with `character` elements:

```
# create matrix with character
my_mat_char <- matrix(rep(c('a','b','c'), 3),
                       nrow = 3,
                       ncol = 3)

# print it
print(my_mat_char)
```

```
R>      [,1] [,2] [,3]
R> [1,] "a"   "a"   "a"
R> [2,] "b"   "b"   "b"
R> [3,] "c"   "c"   "c"
```

Now with a `logic` type:

```
# create matrix with logical
my_mat_logical <- matrix(sample(c(TRUE,FALSE),
                               size = 3*3,
                               replace = TRUE),
                           nrow = 3,
                           ncol = 3)

# print it
print(my_mat_logical)
```

```
R>      [,1] [,2] [,3]
R> [1,] FALSE TRUE TRUE
R> [2,] FALSE TRUE TRUE
R> [3,] FALSE TRUE TRUE
```

This flexibility allows the user to expand the representation of two-dimensional data beyond numerical values.

6.3.1 Selecting Elements from a matrix

Following the same notation as the atomic vector, you can select *pieces* of a `matrix` using indexes. A difference here is that matrices are two-dimensional objects, while atomic vectors are one-dimensional.³ The extra dimension of matrices requires selecting elements not only by lines, but also by columns. The elements of an array can be accessed with the notation `[i, j]` where i represents the row and j the column. See the following example:

```
# create matrix
my_mat <- matrix(1:9, nrow = 3)

# display it
print(my_mat)
```

```
R>      [,1] [,2] [,3]
R> [1,]     1     4     7
R> [2,]     2     5     8
R> [3,]     3     6     9
```

```
# display element in [1,2]
print(my_mat[1,2])
```

```
R> [1] 4
```

To select an entire row or column, simply leave a blank index, as in the following example:

```
# select all rows from column 2
print(my_mat[, 2])
```

```
R> [1] 4 5 6
```

```
# select all columns from row 1
print(my_mat[1, ])
```

```
R> [1] 1 4 7
```

Notice the result of indexing is an atomic vector, not a `matrix`. If we wanted the extracted piece to maintain its `matrix` class, with vertical or horizontal orientation, we could force this conversion using functions `as.matrix` and `matrix`:

³To avoid confusion, atomic vectors in R have no dimension attribute in the strict sense of the function. When using the `dim` function in an atomic vector, such as `dim(c(1,2,4))`, the result is `NULL`. Away from the computing environment, however, atomic vectors can be considered one-dimensional objects as it can only increase its size in one direction.

```
# force matrix conversion and print result
print(as.matrix(my_mat[, 2]))
```

```
R>      [,1]
R> [1,]    4
R> [2,]    5
R> [3,]    6
```

```
# force matrix conversion for one row and print result
print(matrix(my_mat[1, ], nrow=1))
```

```
R>      [,1] [,2] [,3]
R> [1,]    1    4    7
```

Pieces of the `matrix` can also be selected using vectors. If we wanted a new `matrix` with all elements from the second row and first column to the third row and second column, we could use the following code:

```
# select some elements and print them
print(my_mat[2:3, 1:2])
```

```
R>      [,1] [,2]
R> [1,]    2    5
R> [2,]    3    6
```

Finally, using logical tests to select elements of matrices is also possible:

```
# set matrix
my_mat <- matrix(1:9, nrow = 3)
```

```
# print logical matrix where value is higher than 5
print(my_mat >5)
```

```
R>      [,1] [,2] [,3]
R> [1,] FALSE FALSE TRUE
R> [2,] FALSE FALSE TRUE
R> [3,] FALSE  TRUE TRUE
```

```
# print the result
print(my_mat[my_mat >5])
```

```
R> [1] 6 7 8 9
```

6.3.2 Other Useful Functions

`as.matrix` - Transforms raw data to a `matrix` object.

```
my_mat <- as.matrix(1:5)
print(my_mat)
```

```
R>      [,1]
R> [1,]    1
R> [2,]    2
R> [3,]    3
R> [4,]    4
R> [5,]    5
```

t - Returns a transposed matrix.

```
my_mat <- matrix(seq(10,20,
                     length.out = 6),
                  nrow = 3)
print(my_mat)
```

```
R>      [,1] [,2]
R> [1,]    10   16
R> [2,]    12   18
R> [3,]    14   20
print(t(my_mat))
```

```
R>      [,1] [,2] [,3]
R> [1,]    10   12   14
R> [2,]    16   18   20
```

rbind - Returns the merger (bind) of matrices, with row orientation.

```
my_mat_1 <- matrix(1:5, nrow = 1)
print(my_mat_1)
```

```
R>      [,1] [,2] [,3] [,4] [,5]
R> [1,]    1    2    3    4    5
```

```
my_mat_2 <- matrix(10:14, nrow = 1)
print(my_mat_2)
```

```
R>      [,1] [,2] [,3] [,4] [,5]
R> [1,]    10   11   12   13   14
```

```
my_rbind_mat <- rbind(my_mat_1, my_mat_2)
print(my_rbind_mat)
```

```
R>      [,1] [,2] [,3] [,4] [,5]
```

```
R> [1,]    1    2    3    4    5
R> [2,]   10   11   12   13   14
```

cbind - Returns the merger (bind) of matrices, with column orientation.

```
my_mat_1 <- matrix(1:4, nrow = 2)
print(my_mat_1)
```

```
R>      [,1] [,2]
R> [1,]    1    3
R> [2,]    2    4
```

```
my_mat_2 <- matrix(10:13, nrow = 2)
print(my_mat_2)
```

```
R>      [,1] [,2]
R> [1,]   10   12
R> [2,]   11   13
```

```
my_cbind_mat <- cbind(my_mat_1, my_mat_2)
print(my_cbind_mat)
```

```
R>      [,1] [,2] [,3] [,4]
R> [1,]    1    3   10   12
R> [2,]    2    4   11   13
```

rowMeans - Returns the mean of a matrix, row wise.

```
my_mat <- matrix(1:9, nrow=3)
print(rowMeans(my_mat))
```

```
R> [1] 4 5 6
```

colMeans - Returns the mean of a matrix, column wise.

```
my_mat <- matrix(1:9, nrow=3)
print(colMeans(my_mat))
```

```
R> [1] 2 5 8
```

6.4 Exercises

All solutions are available at <https://www.msperlin.com/afedR>.

01 - Using function `dplyr::tibble`, create a `dataframe` called `my_df` with a column called `x` containing a sequence from -100 to 100 and another column

called `y` with the value of column `x` added by 5. How many values in column `x` are greater than 10 and lower than 25?

02 - Create a new column in object `my_df` called `cumsum_x`, containing the cumulative sum of `x` (`cumsum` function). In this new column, how many values are greater than -3500?

03 - Use function `dplyr::filter` function and the pipeline operator to filter `my_df`, keeping only the rows where the value of the `y` column is greater than 0. What is the number of rows in the resulting table?

04 - If you have not already done so, repeat exercises 1, 2 and 3 using the functions of the `tidyverse` universe and the *pipeline* operator.

05 - Use the `BatchGetSymbols` package to download Google (GOOG) stock data, fromr `first_date` to 2020-12-31. If the investor had bought 1000 USD in Google shares on the first day of the data and kept the investment until today, what would be the value of his portfolio?

- a) \$7,485.01
- b) \$9,972.50
- c) \$5,571.73
- d) \$8,728.75
- e) \$6,241.27

06 - Use functions `afedR::get_data_file` and `readr::read_csv` to import data from the `grunfeld.csv` file. Now, use function `dplyr::glimpse` functions to find out the number of lines in the imported data. What is the number of rows in the imported table?

07 - Create a list type object with three `dataframes` in its contents, `df1`, `df2` and `df3`. The content and size of the dataframes is your personal choice. Now, use the `sapply` function and a custom function to find out the number of rows and columns in each `dataframe`.

08 - Within an R session, create an identity matrix (value 1 in the diagonal, zero in any other) of size 3X3. Tip: use the `diag` function to define the diagonal of the matrix.

Basic Object Classes

The basic classes are the most primary elements of data representation in R. Previously, we used the basic classes to translate raw information from files or the internet. The numeric data became a `numeric` column in a `dataframe`, while text data became a `character` object.

In this chapter, we will study R's basic object classes with depth, including the manipulation of their content. This chapter will show you what operations are possible with each object class and how you can use functions and packages to manipulate the information efficiently. It includes the following types of objects:

- Numeric (`numeric`)
- Text (`character`)
- Factors (`factor`)
- Logical Values (`logical`)
- Dates and Time (`Date` and `timedate`)
- Missing Data (`NA`)

7.1 Numeric Objects

The objects of type `numeric` represent quantities and, unsurprisingly, are one of the most used objects in data research. For example, the price of a stock at a given date, the value of inflation in a given period and country, the net profit of a company at the end of the fiscal year, among many other possibilities.

7.1.1 Creating and Manipulating numeric Objects

It is easy to create and manipulate the `numeric` objects. As expected, we can use the common symbols of mathematical operations, such as sum (+), difference (-), division (/) and multiplication (*). When working with `numeric` vectors, all mathematical operations are carried out using an **element by element** orientation and using vector notation. For example, this means that we can add the elements from two vectors in a single line of code.

As you can see in the next example, where we have created two vectors and perform various operations.

```
# create numeric vectors
x <- 1:5
y <- 2:6

# print sum
print(x+y)
```

```
R> [1] 3 5 7 9 11
```

```
# print multiplication
print(x*y)
```

```
R> [1] 2 6 12 20 30
```

```
# print division
print(x/y)
```

```
R> [1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
# print exponentiation
print(x^y)
```

```
R> [1] 1 8 81 1024 15625
```

The difference between R and other programming languages is that operations between vectors of different sizes are accepted. That is, we can also add a `numeric` vector with four elements to another with only two. Whenever that happens, R calls for the **recycling rule**. It states that, if two different sized vectors are interacting, the smaller vector is repeated as often as necessary to obtain the same number of elements as the larger vector. See the following example:

```
# set x with 4 elements and y with 2
x <- 1:4
y <- 2:1
```

```
# print sum
print(x + y)
```

```
R> [1] 3 3 5 5
```

The result of `x + y` is equivalent to `1:4 + c(2, 1, 2, 1)`. If you try to operate with vectors in which the length of the largest vector is not a multiple of the length of the smaller, R performs the same recycling procedure. However, it also sends a `warning` message to inform the user that the recycling procedure did not result in a perfect match.

```
# set x = 4 elements and y with 3
x <- c(1, 2, 3, 4)
y <- c(1, 2, 3)

# print sum (recycling rule)
print(x + y)
```

```
R> Warning in x + y: longer object length is not a multiple of
R> shorter object length
```

```
R> [1] 2 4 6 5
```

The first three elements of `x` were summed to the first three elements of `y`, as expected. However, the fourth element of `x` was summed to the first element of `y`. In order to complete the operation, R cycled through the values of the vector, restarting with the first element of `y`.

Elements of a `numeric` vector can also be named. See an example next, where we create a vector with several named items.

```
# create named vector
x <- c(item1 = 10,
       item2 = 14,
       item3 = 9,
       item4 = 2)

# print it
print(x)
```

```
R> item1 item2 item3 item4
R>     10     14      9      2
```

Empty `numeric` vectors can also be created. Sometimes, you need to preallocate an empty vector to be filled with values later. In some situations this

simple procedure can make the code faster; and for that, you need to use the `numeric` function:

```
# create empty numeric vector of length 10
my_x <- numeric(length = 10)

# print it
print(my_x)
```

```
R> [1] 0 0 0 0 0 0 0 0 0 0
```

As you can see, when using `numeric(length = 10)`, all values are set to zero.

7.1.2 Creating a numeric Sequence

In R, you have two ways to create a sequence of numerical values. The first with operator `:` as in `my_seq <- 1:10`. This method is practical because the notation is clear and direct.

However, using the operator `:` can be restrictive. Did you notice that it only creates sequences where the difference between adjacent elements is $+1$ or -1 ? A more powerful version for the creation of sequences is the use of function `seq`. With it, you can set the intervals between each value with the argument `by`. See an example next:

```
# create sequence with seq
my_seq <- seq(from = -10,
               to = 10,
               by = 2)

# print it
print(my_seq)
```

```
R> [1] -10 -8 -6 -4 -2 0 2 4 6 8 10
```

Another interesting feature of function `seq` is the possibility of creating equally spaced vectors with an initial value, a final value, and the desired number of elements. This is accomplished using option `length.out`. In the following code, we create a array from 0 to 10 with 20 elements:

```
# create sequence with defined number of elements
desired_len <- 20
my_seq <- seq(from = 0,
               to = 10,
               length.out = desired_len)
```

```
# print it
print(my_seq)

R> [1] 0.0000000 0.5263158 1.0526316 1.5789474 2.1052632
R> [6] 2.6315789 3.1578947 3.6842105 4.2105263 4.7368421
R> [11] 5.2631579 5.7894737 6.3157895 6.8421053 7.3684211
R> [16] 7.8947368 8.4210526 8.9473684 9.4736842 10.0000000
```

The final number of elements in object `my_seq` is exactly 20. Function `seq` automatically calculates and sets the difference of 0.5263 between the adjacent elements. That is, if we calculate the difference from one element to the other, we will always find the same result of 0.5263.

7.1.3 Creating Vectors with Repeated Elements

Another way to create `numeric` vectors is by using repetition. For example, imagine that we are interested in a vector with the value 1 repeated ten times. For that, we use function `rep`:

```
# repeat vector three times
my_x <- rep(x = 1, times = 10)

# print it
print(my_x)
```

```
R> [1] 1 1 1 1 1 1 1 1 1 1
```

It also works with vectors. For example, let's say you need to create a vector with the repeated values of `c(1, 2)`. The result should be equal to `c(1, 2, 1, 2, 1, 2)`. For that, we use `rep` the same way:

```
# created a vector with repeated elements
my_x <- rep(x = c(1, 2),
              times = 3)

# print it
print(my_x)
```

```
R> [1] 1 2 1 2 1 2
```

7.1.4 Creating Vectors with Random Numbers

Some applications in finance and economics require the use of random numbers to simulate mathematical models. For example, the simulation method

of Monte Carlo can generate asset prices based on random numbers from the Normal distribution.

In R, several functions create random numbers for different statistical distributions. The most commonly used, however, are functions `stats::rnorm` and `stats::runif`. Remember that package `stats` is automatically loaded when R starts. So, unless you changed the default settings of R's startup, there is no need to make a call to `library(stats)`.

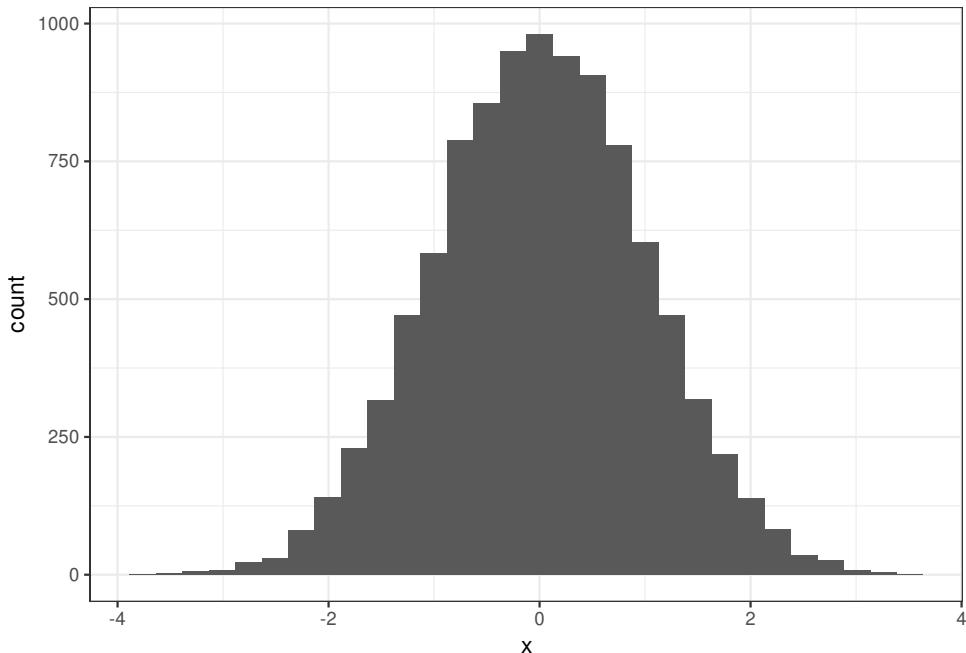
Function `rnorm` generates random numbers from the Normal distribution, with options for the mean and standard deviation. The `mean` will set the point with the highest frequency and `sd` (standard deviation) will change the dispersion of the histogram.

```
# generate 10 random numbers from a Normal distribution
my_rnd_vec <- rnorm(n = 10000,
                      mean = 0,
                      sd = 1)

# print it
glimpse(my_rnd_vec)
```

```
R> num [1:10000] 0.544 1.041 0.198 -1.63 0.121 ...
```

We generated ten thousand random numbers from a Normal distribution, with mean zero and standard deviation equal to one. Let's see if its distribution of numbers looks close to the Normal, a bell shaped distribution:



Yes, the shape of the frequency distribution is visually close to the bell curve. You can change the parameters `mean` and `sd` for different shapes of a Normal distribution.

Function `runif` generates random values uniformly distributed between a maximum and a minimum. It is commonly used to simulate probabilities with values between zero and one. The `runif` function has three input parameters: the desired number of random values, the minimum value, and maximum value. See the following example:

```
# create a random vector with minimum and maximum
my_rnd_vec <- runif(n = 10,
                      min = -5,
                      max = 5)

# print it
print(my_rnd_vec)
```

```
R> [1] -0.0858712  4.3401557  2.9388118 -2.2792339  3.1741687
R> [6] -0.3947496  3.4054654  3.1049157  3.5276227 -1.5256521
```

Note that both functions, `rnorm` and `runif`, are limited to their respective distribution. An alternative and flexible way to generate random values is to use the `sample` function. It accepts any vector as input and returns a scrambled version of its elements. Its flexibility lies in the fact that the

input vector can be anything. For example, if we wanted to create a random vector with elements taken from vector `c(0, 5, 15, 20, 25)`, we could do it like this:

```
# create sequence
my_vec <- seq(from = 0, to = 25, by=5)

# sample sequence
my_rnd_vec <- sample(my_vec)

# print it
print(my_rnd_vec)
```

R> [1] 5 10 25 0 20 15

Function `sample` also allows the random selection of a fixed number of elements. If we wanted to select only one element of `my_vec` randomly, we could write the code as:

```
# sample one element of my_vec
my_rnd_vec <- sample(my_vec, size = 1)

# print it
print(my_rnd_vec)
```

R> [1] 20

If we wanted two random elements from `my_rnd_vec`:

```
# sample two elements of my_vec
my_rnd_vec <- sample(my_vec, size = 2)

# print it
print(my_rnd_vec)
```

R> [1] 25 20

Besides, you can select values from a smaller vector to create a larger vector. Consider the case where you have a vector with numbers `c(5, 10, 15)` and want to create a random vector with ten elements. For that, we use the option `replace = TRUE`.

```
# create vector
my_vec <- c(5, 10, 15)

# sample
```

```
my_rnd_vec <- sample(x = my_vec, size = 10, replace = TRUE)
print(my_rnd_vec)
```

```
R> [1] 15 10 5 5 10 5 5 15 10 10
```

Another important feature of `sample` is it works for any type of vector, not only for those of the `numeric` class:

```
# example of sample with characters
print(sample(c('elem 1','elem 2','elem 3'),
             size = 1))
```

```
R> [1] "elem 2"
# example of sample with list
print(sample(list(x = c(1,1,1),
                 y = c('a', 'b')),
            size = 1))
```

```
R> $y
R> [1] "a" "b"
```

At this point, it is important to acknowledge that **the generation of random values in R is not entirely random!** Internally, the computer makes selections based on a deterministic queue. Every time that chance-related functions, such as `rnorm`, `runif`, and `sample`, are called in the code, the computer chooses a different place in this queue according to various parameters. The practical effect is that the actual chosen values are unpredictable from the user's viewpoint. However, for the computer, this selection is absolutely predictable and deterministic.

One neat trick is that we can select the starting place in the queue of random values using function `base::set.seed`. In practical terms, the result is that, after a call to `set.seed`, all subsequent numbers and random selections will be the same in every code execution. Using `set.seed` is strongly recommended for the reproducibility of code involving randomness. Anyone can replicate the exact same results, even if it involves the selection of random numbers. See the following example.

```
# set seed with integer 10
set.seed(seed = 10)

# create and print "random" vectors
my_rnd_vec_1 <- runif(5)
print(my_rnd_vec_1)
```

```
R> [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
my_rnd_vec_2 <- runif(5)
print(my_rnd_vec_2)
```

```
R> [1] 0.2254366 0.2745305 0.2723051 0.6158293 0.4296715
```

In the previous code, the value 10 in `set.seed(10)` is an integer chosen by the user. After the call to `set.seed(10)`, all selections and random numbers will start from the same point in the queue. Therefore, the random vectors are the same. By running that previous chunk of code in your computer, you'll see that the values of `my_rnd_vec_1` and `my_rnd_vec_2` will be exactly the same as the ones printed in this book.

Function `set.seed` also works for `sample`:

```
# fix seed
set.seed(seed = 15)

# print vectors
print(sample(1:10))
```

```
R> [1] 5 2 1 6 8 10 3 7 9 4
print(sample(10:20))
```

```
R> [1] 13 15 10 17 20 14 19 12 11 18 16
```

Likewise, if you execute the previous code in your R session, you'll see the exact same selections.

7.1.5 Accessing the Elements of a `numeric` Vector

All elements of a numerical vector can be accessed with brackets (`[]`). For example, if we wanted only the first element of `x`, we can use `x[1]`:

```
# set vector
x <- c(-1, 4, -9, 2)

# get first element
first_elem_x <- x[1]

# print it
print(first_elem_x)
```

```
R> [1] -1
```

The same notation is used to extract parts of a vector. If we wanted to create a sub-vector with the first and second element of `x`, we can achieve this goal with the next chunk of code:

```
# sub-vector of x
sub_x <- x[1:2]

# print it
print(sub_x)
```

```
R> [1] -1  4
```

To access named elements of a numeric array, simply use its name as a `character` value or vector inside the brackets.

```
# set named vector
x <- c(item1 = 10, item2 = 14, item3 = -9, item4 = -2)

# access elements by name
print(x['item2'])
```

```
R> item2
R>     14
print(x[c('item2','item4')])
```

```
R> item2 item4
R>     14      -2
```

We can also access the elements of a numerical vector using logical tests. For example, if we were interested in knowing which values of `x` are larger than 0, we could use the following code:

```
# find all values of x higher than zero
print(x[x > 0])
```

```
R> item1 item2
R>     10     14
```

The selection of elements from a vector, according to some criteria, is called logical indexing. Objects of type `logical` will be treated later in this same chapter.

7.1.6 Modifying and Removing Elements of a numeric Vector

The modification of a vector is very simple. Just indicate the changes with the *assign* symbol (<-):

```
# set vector
my_x <- 1:4

# modify first element to 5
my_x[1] <- 5

# print result
print(my_x)
```

R> [1] 5 2 3 4

This modification can also be performed block-wise:

```
# set vector
my_x <- 0:5

# set the first three elements to 5
my_x[1:3] <- 5

# print result
print(my_x)
```

R> [1] 5 5 5 3 4 5

Using conditions to change values in a vector is also possible:

```
# set vector
my_x <- -5:5

# set any value lower than 2 to 0
my_x[my_x<2] <- 0

# print result
print(my_x)
```

R> [1] 0 0 0 0 0 0 2 3 4 5

The removal of elements of a vector is carried out using a negative index:

```
# create vector
my_x <- -5:5

# remove first and second element of my_x
my_x <- my_x[-(1:2)]

# show result
print(my_x)
```

R> [1] -3 -2 -1 0 1 2 3 4 5

Notice how using negative index simply returns the original vector, without the elements in the brackets.

7.1.7 Creating Groups

A common data task is to create groups based on numerical data. For example, we can calculate how many cases in the sample are located within a certain numerical range. Now, let's imagine that we have a vector of daily returns of a stock, the percentage change in prices from one day to another. A simple way to measure the risk of the investment is to divide the return interval into five parts and verify the percentage of occurrences of returns at each range. This simple calculation will show, for example, how many days the return was negative, with a drop of the stock price.

In R, the function used to create intervals from a numerical vector is `base::cut`. See the following example, where we create a random vector from the Normal distribution and five groups from intervals defined by the data.

```
# set random vector
my_x <- rnorm(10000)

# create groups with 5 breaks
my_cut <- cut(x = my_x, breaks = 5)

# print it!
print(head(my_cut))
```

R> [1] (-2.18,-0.71] (-0.71,0.759] (-2.18,-0.71] (-0.71,0.759]
R> [5] (-0.71,0.759] (-2.18,-0.71]
R> 5 Levels: (-3.66,-2.18] (-2.18,-0.71] ... (2.23,3.7]

Be aware that ranges define the names in `my_cut`, and the result is a `factor`

object. We will cover this object in a future section. For now, it is worthwhile to say **factors** are simply groups within our data. Going forward, we can also check how often we find cases in each group using **base::table**:

```
print(table(my_cut))
```

```
R> my_cut
R> (-3.66,-2.18] (-2.18,-0.71] (-0.71,0.759] (0.759,2.23]
R>          154         2132        5459        2149
R> (2.23,3.7]
R>          106
```

As expected, the distribution of values is balanced, with a higher occurrence of values around zero – the mean.

With the **cut** function, you can also define custom breaks in data and group names. See next, where we use a **tibble** to organize our data:

```
# create random vector in tibble
my_df <- tibble(x = rnorm(10000))

# define breaks and labels manually
my_breaks <- c(min(my_x)-1, -1, 1, max(my_x)+1)
my_labels <- c('Low', 'Normal', 'High')

# create group from numerical vector
my_df <- my_df %>%
  mutate(cut_x = cut(x = x,
                     breaks = my_breaks,
                     labels = my_labels))

# glimpse it!
glimpse(my_df)
```

```
R> Rows: 10,000
R> Columns: 2
R> $ x      <dbl> -0.42217453, 1.21510691, -0.48018365, 0.0904~
R> $ cut_x <fct> Normal, High, Normal, Normal, Low, Normal, N~
```

Notice that, in this example of creating a group from a numerical vector, the breaks were defined in **my_breaks** and the names in **my_labels**. We can once again check the distribution of groups with function **table**:

```
print(table(my_df$cut_x))
```

```
R>
```

```
R>      Low Normal   High
R> 1593    6770   1637
```

7.1.8 Other Useful Functions

as.numeric - Converts an object to the **numeric** class.

```
my_text <- c('1', '2', '3')
class(my_text)
```

```
R> [1] "character"
my_x <- as.numeric(my_text)
print(my_x)
```

```
R> [1] 1 2 3
```

```
class(my_x)
```

```
R> [1] "numeric"
```

unique - Returns all unique values of a numeric vector.

```
my_x <- c(1, 1, 2, 3, 3, 5)
print(unique(my_x))
```

```
R> [1] 1 2 3 5
```

sum - Sums all elements of a **numeric** vector.

```
my_x <- 1:50
my_sum <- sum(my_x)
print(my_sum)
```

```
R> [1] 1275
```

max - Returns the maximum value of a **numeric** vector.

```
x <- c(10, 14, 9, 2)
max_x <- max(x)
print(max_x)
```

```
R> [1] 14
```

min - Returns the minimum value of a **numeric** vector.

```
x <- c(12, 15, 9, 2)
min_x <- min(x)
print(min_x)
```

```
R> [1] 2
```

which.max - Returns the position of the maximum value of a **numeric** object.

```
x <- c(100, 141, 9, 2)
which_max_x <- which.max(x)
cat(paste('The position of the max value of x is ', which_max_x))
```

```
R> The position of the max value of x is 2
cat(' and its value is ', x[which_max_x])
```

```
R> and its value is 141
```

which.min - Returns the position of the minimum value of a **numeric** object.

```
x <- c(10, 14, 9, 2)
which_min_x <- which.min(x)
cat(paste('The position of the min value of x is ', which_min_x, ' and its value is ', x[which_min_x]))
```

```
R> The position of the min value of x is 4 and its value is 2
```

sort - Returns a sorted (ascending or descending) version of a **numeric** vector.

```
x <- runif(5)
print(sort(x, decreasing = FALSE))
```

```
R> [1] 0.1032680 0.3004002 0.4337767 0.5133841 0.7782837
print(sort(x, decreasing = TRUE))
```

```
R> [1] 0.7782837 0.5133841 0.4337767 0.3004002 0.1032680
```

cumsum - Returns the cumulative sum of the elements of a **numerical** vector.

```
my_x <- 1:25
my_cumsum <- cumsum(my_x)
print(my_cumsum)
```

```
R> [1] 1 3 6 10 15 21 28 36 45 55 66 78 91 105
R> [15] 120 136 153 171 190 210 231 253 276 300 325
```

prod - Returns the product (multiplication) of all the elements of a numerical vector.

```
my_x <- 1:10
my_prod <- prod(my_x)
print(my_prod)
```

```
R> [1] 3628800
```

cumprod - Returns the cumulative product of the elements of a **numeric** vector.

```
my_x <- 1:10
my_prod <- cumprod(my_x)
print(my_prod)
```

```
R> [1]      1      2      6     24    120    720   5040
R> [8] 40320 362880 3628800
```

7.2 Character Objects

The **character** class, or simply **text** class, is used to store textual information. A practical example in finance would be to extract sentiments from Facebook and Twitter posts and use that to predict future market prices. It is worth knowing that analyzing textual information is an upward trend in research (Gentzkow et al., 2017), with many R packages developed over the last few years.

R has several features that facilitate the creation and manipulation of text type objects. The base functions shipped with the installation of R are comprehensive and suited for most cases. However, package **stringr** (Wickham, 2022b) provides many functions that greatly expand the basic functionality.

A positive aspect of **stringr** is that all functions start with the name **str_** and are informative. So, using the auto-completion feature (*tab* key) described in the previous chapter, it is easy to find the names of functions. Following our preference for the packages in the **tidyverse**, we will focus only on the functions of **stringr**. The base functions for string manipulation will be presented but in a limited way.

7.2.1 Creating a Simple **character** Object

In R, every **character** object is created by encapsulating a text with double quotation marks (" ") or single (' '). To create an array of characters with stock *tickers*, we can do it with the following code:

```
tickers <- c('MMM', 'FB', 'ICE')
print(tickers)
```

```
R> [1] "MMM" "FB"  "ICE"
```

We can confirm the class of the created object with function `class`:

```
class(tickers)
```

```
R> [1] "character"
```

7.2.2 Creating Structured character Objects

We can also use R to create a text vector with some sort of structure. For example, vector `c('ticker 1', 'ticker 2', ..., 'ticker 19', 'ticker 20')` has a clear logic. It combines a text `ticker` with values from a vector that starts in 1 and ends in 20.

To create a text vector with the junction of text and numbers, use the `stringr::str_c` or `paste` function. See the following example, where we create the previous structured text in two ways, with space between the characters and numbers, and without it.

```
library(stringr)
```

```
# create sequence and text
my_seq <- 1:20
my_text <- 'text'

# paste objects together (without space)
my_char <- str_c(my_text, my_seq)
print(my_char)
```

```
R> [1] "text1"   "text2"   "text3"   "text4"   "text5"   "text6"
R> [7] "text7"   "text8"   "text9"   "text10"  "text11"  "text12"
R> [13] "text13"  "text14"  "text15"  "text16"  "text17"  "text18"
R> [19] "text19"  "text20"
```

```
# paste objects together (with space)
my_char <- str_c(my_text,
                  my_seq,
                  sep = ' ')
print(my_char)
```

```
R> [1] "text 1"  "text 2"  "text 3"  "text 4"  "text 5"
```

```
R> [6] "text 6"  "text 7"  "text 8"  "text 9"  "text 10"
R> [11] "text 11" "text 12" "text 13" "text 14" "text 15"
R> [16] "text 16" "text 17" "text 18" "text 19" "text 20"
```

We can do the same procedure with text vectors:

```
# set character value
my_x <- 'My name is'

# set character vector
my_names <- c('Marcelo', 'Ricardo', 'Tarcizio')

# paste and print
print(str_c(my_x, my_names, sep = ' '))
```

```
R> [1] "My name is Marcelo"  "My name is Ricardo"
R> [3] "My name is Tarcizio"
```

Another possibility of building structured text is the repetition of the content of another object. With `character` objects, use function `stringr!str_dup` or `base::strrep` for this purpose. Consider the following example:

```
# repeat 'abc' five times
my_char <- str_dup(string = 'abc', times = 5)

# print it
print(my_char)
```

```
R> [1] "abcabcabcabcabc"
```

7.2.3 character Constants

R also allows direct access to all letters of the Roman alphabet. They are stored in the reserved (constant) objects, called `letters` and `LETTERS`. See an example next.

```
# print all letters in alphabet (no cap)
print(letters)
```

```
R> [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
R> [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
# print all letters in alphabet (WITH CAP)
print(LETTERS)
```

```
R> [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
```

```
R> [15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Note that, in both cases, `letters` and `LETTERS` are not functions. They are `character` objects automatically embedded as constants in R. Even though they do not appear in the environment, and they are always available for use. You may overwrite their object names, such as in `letters <- 'other char'`, but this is not advised. You never know where it is being used.

Other constant `character` objects in R are `month.abb`, which shows an abbreviation of months and `month.name`. Their content is presented next.

```
# print abbreviation and full names of months
print(month.abb)
```

```
R> [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
R> [10] "Oct" "Nov" "Dec"
```

```
print(month.name)
```

```
R> [1] "January"    "February"   "March"       "April"
R> [5] "May"        "June"        "July"        "August"
R> [9] "September"  "October"    "November"   "December"
```

7.2.4 Selecting Pieces of a Text Object

A common beginner's mistake is to select characters of a text using brackets, as it is done for selecting elements of a vector. Consider the following code:

```
# set char object
my_char <- 'ABCDE'

# print its second element (WRONG - RESULT is NA)
print(my_char[2])
```

```
R> [1] NA
```

The `NA` value indicates the second element of `my_char` does not exist. This happens because using square brackets is reserved for accessing the elements of an atomic vector, not characters within a larger text. Watch what happens when we use `my_char[1]`:

```
print(my_char[1])
```

```
R> [1] "ABCDE"
```

The result is simply the `ABCDE` text, on the first item of `my_char`. To select pieces of text, we need to use function `stringr::str_sub` or `base::substr`:

```
# print third and fourth characters
my_substr <- str_sub(string = my_char,
                      start = 2,
                      end = 2)
print(my_substr)
```

```
R> [1] "B"
```

These functions also work for atomic vectors. Let's assume you imported text data, and the raw dataset contains a 3-letter identifier of a company, always in the same location of the string. Let's simulate the situation in R:

```
# build char vec
my_char_vec <- paste0(c('ABC', 'VBC', 'ZMN'),
                       ' - other ignorable text')
print(my_char_vec)
```

```
R> [1] "ABC - other ignorable text"
R> [2] "VBC - other ignorable text"
R> [3] "ZMN - other ignorable text"
```

Here, we want the information in the first three characters of each element only in `my_char_vec`. To select them, we can use the same functions as before.

```
# get ids with stringr::str_sub
ids_vec <- str_sub(my_char_vec, 1, 3)
print(ids_vec)
```

```
R> [1] "ABC" "VBC" "ZMN"
```



Vector operations in character objects are very common in R. Almost anything you can do to a single element can be expanded to vectors. This facilitates the development of research scripts as you can easily perform complicated tasks to a series of elements in a single line of code.

7.2.5 Finding and Replacing Characters of a Text

A useful operation in handling texts is to locate specific patterns of text within a `character` object with functions `stringr::str_locate/base::regexpr` and `stringr::str_locate_all/base::gregexpr`. However, before we move on to the examples, it is important to point out that, by default,

these functions use expressions of the type `regex` – regular expressions (Thompson, 1968). This is a specific format for the identification of patterns in text. When `regex` is used correctly, it can make your life a lot easier. You'll be able to find complex string patterns effortlessly.

The most common case of string search is to verify the position or the existence of a smaller text within a larger text. For these cases, however, using language `regex` is unnecessary. Therefore, the location and replacement of characters in the next example is of the fixed type, i.e., without using `regex`. Such information is passed to the `stringr` function by setting the `pattern` input with `stringr::fixed`, as in `str_locate(str_in, pattern = fixed('pattern to match'))`.

The following example shows how to find the *D* character from a range of characters.

```
# set character object
my_char <- 'ABCDEF-ABCDEF-ABC'

# find position of 'D' using str_locate
pos <- str_locate(my_char, fixed('D'))
print(pos)

R>      start end
R> [1,]     4   4
```

Note the `str_locate` function returns only the **first occurrence** of *D*. To locate all instances, we use function `str_locate_all`.

```
# set object
my_char <- 'ABCDEF-ABCDEF-ABC'

# find position of ALL 'D' using str_locate_all
pos <- str_locate_all(my_char, fixed('D'))
print(pos)

R> [[1]]
R>      start end
R> [1,]     4   4
R> [2,]    11  11
```

To replace characters in a text, use functions `str_replace` and `str_replace_all` from `stringr` or `sub` and `gsub` from the base package. As with previous example, `stringr::str_replace` replaces the first occurrence of the character, while `stringr::str_replace_all` performs a global substitution; applies to all matches. Here are the differences:

```
# set char object
my_char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with str_replace
my_char <- str_replace(string = my_char,
                        pattern = 'ABC',
                        replacement = 'XXX')
print(my_char)
```

R> [1] "XXXDEF-ABCDEF-ABC"

And now, we globally substitute characters.

```
# set char object
my_char <- 'ABCDEF-ABCDEF-ABC'

# substitute ALL 'ABC' for 'XXX' with str_replace_all
my_char <- str_replace_all(string = my_char,
                            pattern = 'ABC',
                            replacement = 'XXX')

# print result
print(my_char)
```

R> [1] "XXXDEF-XXXDEF-XXX"

Again, it is worth pointing out that the operations of replacements of strings also work for vectors. Have a look at the next example.

```
# set char object
my_char <- c('ABCDEF', 'DBCFE', 'ABC')

# create an example of vector
my_char_vec <- paste(sample(my_char, 5, replace = T),
                      sample(my_char, 5, replace = T),
                      sep = ' - ')

# show it
print(my_char_vec)
```

R> [1] "ABC - DBCFE" "ABCDEF - ABCDEF" "DBCFE - DBCFE"
R> [4] "ABCDEF - ABC" "ABCDEF - ABCDEF"

```
# substitute all occurrences of 'ABC'
my_char_vec <- str_replace_all(string = my_char_vec,
```

```

        pattern = 'ABC',
        replacement = 'XXX')

# print result
print(my_char_vec)

R> [1] "XXX - DBCFE"      "XXXDEF - XXXDEF"  "DBCFE - DBCFE"
R> [4] "XXXDEF - XXX"     "XXXDEF - XXXDEF"

```

7.2.6 Splitting Text

Eventually, you will need to break a text into different parts. Most of the time, you want to isolate a piece of particular information in the full string by using a delimiter in the text. For example, the text 'ABC;DEF;GHI' has three sub-characters divided by the symbol ;. To separate a text into several parts, use `stringr::str_split` or `base::strsplit`. Both functions break the original text into several fractions, according to a chosen delimiter character:

```

# set char
my_char <- 'ABC;ABC;BCD'

# split it based on ';' and using stringr::str_split
splitted_char <- str_split(my_char, ';')

# print result
print(splitted_char)

R> [[1]]
R> [1] "ABC" "ABC" "BCD"

```

The output of this function is an object of type `list`. To access the text BCD in object `splitted_char`, we can use the following code:

```

print(splitted_char[[1]][3])

R> [1] "BCD"

```

For an example of a split in character vectors, see the next code.

```

# set char
my_char_vec <- c('ABCDEF', 'DBCFE', 'ABFC', 'ACD')

# split it based on 'B' and using stringr::strsplit
splitted_char <- str_split(my_char_vec, 'B')

```

```
# print result
print(split_text)
```

```
R> [[1]]
R> [1] "A"      "CDEF"
R>
R> [[2]]
R> [1] "D"      "CFE"
R>
R> [[3]]
R> [1] "A"      "FC"
R>
R> [[4]]
R> [1] "ACD"
```

Notice how, again, an object of type `list` is returned, where each element is the split text from the input vector.

7.2.7 Finding the Number of Characters in a Text

If we want to discover the number of characters in a `character` object, you can use functions `stringr::str_length` and `base::nchar`. Both functions also work for atomic vectors. See the examples below:

```
# set char
my_char <- 'abcdef'

# print number of characters using stringr::str_length
print(str_length(my_char))
```

```
R> [1] 6
```

And now an example with vectors.

```
#set char
my_char <- c('a', 'ab', 'abc')

# print number of characters using stringr::str_length
print(str_length(my_char))
```

```
R> [1] 1 2 3
```

7.2.8 Generating Combinations of Text

One useful trick in R is to use functions `outer` and `expand.grid` to create all possible combinations of elements in different objects. This is useful when you want to create a `character` vector by combining all possible elements from different vectors. For example, if we wanted to create a vector with all combinations between `c('a', 'b')` and `'c('A','A')` as `c('a-A', 'a-B', ...)`, we could write:

```
# set char vecs
my_vec_1 <- c('a','b')
my_vec_2 <- c('A','B')

# combine in matrix
comb_mat <- outer(my_vec_1, my_vec_2, paste, sep='-' )

# print it!
print(comb_mat)
```

```
R>      [,1]  [,2]
R> [1,] "a-A" "a-B"
R> [2,] "b-A" "b-B"
```

The output of `outer` is a `matrix` type of object. If we wanted to change `comb_mat` to an atomic vector, we can use function `as.character`:

```
print(as.character(comb_mat))
```

```
R> [1] "a-A" "b-A" "a-B" "b-B"
```

Another way to reach the same objective is by using function `tidyverse::expand_grid` or `base::expand.grid`. Look at the next example, where we create different phrases based on all combinations of character vectors.

```
library(tidyverse)

# set vectors
my_vec_1 <- c('John ', 'Claire ', 'Adam ')
my_vec_2 <- c('is fishing.', 'is working.')

# create df with all combinations
my_df <- expand_grid(name = my_vec_1,
                      verb = my_vec_2)

# print df
```

```
print(my_df)

R> # A tibble: 6 x 2
R>   name      verb
R>   <chr>     <chr>
R> 1 "John "    is fishing.
R> 2 "John "    is working.
R> 3 "Claire "  is fishing.
R> 4 "Claire "  is working.
R> 5 "Adam "    is fishing.
R> 6 "Adam "    is working.

# paste columns together in tibble
my_df <- my_df %>%
  mutate(phrase = paste0(name, verb) )

# print result
print(my_df)
```

```
R> # A tibble: 6 x 3
R>   name      verb      phrase
R>   <chr>     <chr>     <chr>
R> 1 "John "    is fishing. John is fishing.
R> 2 "John "    is working. John is working.
R> 3 "Claire "  is fishing. Claire is fishing.
R> 4 "Claire "  is working. Claire is working.
R> 5 "Adam "    is fishing. Adam is fishing.
R> 6 "Adam "    is working. Adam is working.
```

Here, we used the function `expand.grid` to create a `dataframe` containing all possible combinations of `my_vec_1` and `my_vec_2`. We pasted the contents of these columns using `paste0`.

7.2.9 Encoding of character Objects

For R, a text string is just a sequence of *bytes*. The translation of *bytes* to actual characters is achieved using a particular encoding structure. Most of the time, and especially in English speaking countries, the character encoding is not an issue. However, when dealing with text data from different spoken languages, such as Portuguese, the character encoding is something that you must learn and understand.

Let's explore an example. Here, we will import data from a text file with the '`ISO-8859-9`' encoding and check the result.

```
library(readr)

# read text file
my_f <- afedR::get_data_file('FileWithLatinChar_Latin1.txt')

my_char <- read_lines(my_f)

# print it
print(my_char)
```

```
R> [1] "A casa \xe9 bonita e tem muito espa\xe7o"
```

The original content of the file is a text in Portuguese. As you can see, the output of `readLines` shows all Latin characters as ugly, unreadable symbols. It happens that the encoding of the file has been manually changed to '`ISO-8859-9`', while the `readLines` function uses '`UTF-8`' as *default*.

The easiest solution is to modify input `locale` of `readr::read_lines` so that the function is aware of the different encoding format:

```
my_char <- read_lines(my_f,
                      locale = locale(encoding='Latin1'))

# print it
print(my_char)
```

```
R> [1] "A casa é bonita e tem muito espaço"
```

The Latin characters are now correct because the encoding in `read_lines` is the same as the file, '`Latin1`'. A good policy in this topic is always to check the encoding of imported text files and combine it into R. Most import functions have an option to do so. If you can, give preference to the '`UTF-8`' encoding in exporting and importing data, which is the most popular in text files.

7.2.10 Other Useful Functions

`stringr::str_to_lower/base::tolower` - Converts a string to small caps.

```
print(stringr::str_to_lower('ABC'))
```

```
R> [1] "abc"
```

`stringr::str_to_upper/base::toupper` - Converts a string to upper caps.

```
print(stringr::str_to_upper('abc'))
```

```
R> [1] "ABC"
```

7.3 Factor Objects

Object class **factor** is used to represent groups (categories) in a database. Imagine a dataset containing the financial expenses of different people over a whole year. In this database, you find a column that defines the gender of the individual – male or female. This information can be imported in R as a **character** object; however, the best way to represent it is by mutating it to class **factor**.

The **factor** class integrates nicely with statistical procedures and packages, so the work of dealing with groups becomes easier. For example, if we wanted to create a chart for each group within our database, we could do it by simply telling the **graph** function that we have a grouping variable of type **factor**. If we wanted to check whether the medians of different groups are statistically different from each other, all we need to do is to pass the numerical values and the grouping factor to the function that performs the statistical test. When the categories of data are appropriately represented in R, working with them becomes easier and more efficient.

7.3.1 Creating factors

The creation of factors is accomplished with function **factor**:

```
# create factor
my_factor <- factor(c('M', 'F', 'M',
                       'M', 'F', 'F'))

# print it
print(my_factor)
```

```
R> [1] M F M M F F
```

```
R> Levels: F M
```

Notice that, in the previous example, the presentation of factors with function **print** shows its content and an extra item called **Levels**, which identifies the possible groups in the object, in this case, only M and F. If we had a larger number of groups, the item **Levels** increases. See next:

```
# create factor with 3 levels
my_factor <- factor(c('M', 'F', 'M',
                      'M', 'F', 'F',
                      'ND'))

# print factor
print(my_factor)
```

```
R> [1] M F M M F F ND
R> Levels: F M ND
```

Here, we also have the `ND` (not defined) group.

An important point about creating factors is that the `Levels` are inferred from the data, and that may not correspond to reality. Consider the following example:

```
# set factors with 1 level
my_status <- factor(c('Single', 'Single', 'Single'))

# print it
print(my_status)
```

```
R> [1] Single Single Single
R> Levels: Single
```

Accidentally, the data in `my_status` only shows one category: `Single`. However, it is well-known that another category, `Married`, is expected. If we used `my_status` as it is, we might omit important information, and that may cause problems in future parts of the code. The correct procedure is to define the `Levels` manually, as follows:

```
my_status <- factor(c('Single', 'Single', 'Single'),
                     levels = c('Single', 'Married'))

print(my_status)

R> [1] Single Single Single
R> Levels: Single Married
```

7.3.2 Modifying factors

An important point about the `factor` type of objects is their `Levels` are immutable and will not update with the input of new data. You cannot modify the `Levels` after the creation of a `factor`. All new groups not in the

current **Levels** will be transformed into **NA** (*not available*) and a **warning** message will appear on the screen. This behavior may seem strange, at first, but it avoids possible errors in the code. See the following example:

```
# set factor
my_factor <- factor(c('a', 'b', 'a', 'b'))

# change first element of a factor to 'c'
my_factor[1] <- 'c'

R> Warning in `<-factor`(`*tmp*`, 1, value = "c"): invalid
R> factor level, NA generated

# print result
print(my_factor)

R> [1] <NA> b     a     b
R> Levels: a b
```

As we expected, the first element of `my_factor` becomes an **NA**. Here, the proper way to add a new factor is to first transform the **factor** object to a **character** object, change the content and, finally, change the class back from **character** to **factor**.

```
# set factor
my_factor <- factor(c('a', 'b', 'a', 'b'))

# change factor to character
my_char <- as.character(my_factor)

# change first element
my_char[1] <- 'c'

# mutate it back to class factor
my_factor <- factor(my_char)

# show result
print(my_factor)

R> [1] c b a b
R> Levels: a b c
```

Using these steps, we have the desired result in vector `my_factor`, with three **Levels**: **a**, **b** and **c**.

The **tidyverse** universe also has its own package for handling factors:

`forcats`. For the current factor modification problem, we can use `forcats::fct_recode` function. Here's an example where we change the values of factors and levels:

```
library(forcats)

# set factor
my_factor <- factor(c('A', 'B', 'C',
                       'A', 'C', 'M',
                       'N'))

# modify factors
my_factor <- fct_recode(my_factor,
                        'D' = 'A',
                        'E' = 'B',
                        'F' = 'C')

# print result
print(my_factor)
```

```
R> [1] D E F D F M N
R> Levels: D E F M N
```

Using `forcats::fct_recode` is intuitive. All we need to do is to set the new names with the equality symbol.

7.3.3 Converting factors to Other Classes

Attention is required when converting a `factor` to another class. When converting a `factor` to the `character` class, the result is as expected:

```
# create factor
my_char <- factor(c('a', 'b', 'c'))

# convert and print
print(as.character(my_char))
```

```
R> [1] "a" "b" "c"
```

However, when the same procedure is performed for conversion from `factor` to the `numeric` class, the result is far from expected:

```
# set factor
my_values <- factor(5:10)
```

```
# convert to numeric (WRONG)
print(as.numeric(my_values))
```

```
R> [1] 1 2 3 4 5 6
```

As you can see, all elements in `my_values` were converted to `c(1, 2, 3, 4, 5)`, which are obviously wrong. It happens that, internally, `factors` are stored as numerical counters, ranging from 1 to the total number of `Levels`. This simple transformation minimizes the use of computer memory. When we asked R to transform the `factor` object into numbers, it returned the values of the counters, not the actual numbers stored as `factors`. Solving this problem and getting the result we want is easy; just turn the `factor` object into a `character` and then to `numeric`, as shown next:

```
# converting factors to character and then to numeric
print(as.numeric(as.character(my_values)))
```

```
R> [1] 5 6 7 8 9 10
```



Always be careful when transforming factors into numbers. This bug is hard to catch and may go unnoticed until it breaks your code or jeopardizes your analysis. Remember that, internally, R converts a numerical factor to its index level number, and not the actual number.

7.3.4 Creating Contingency Tables

After creating a factor, we can find the number of times that each group, or combination of groups, is found with function `table`. This is also called a contingency table. In a simple case, with only one factor, function `table` counts the number of occurrences of each category:

```
# create factor
my_factor <- factor(sample(c('Pref', 'Ord'),
                           size = 20,
                           replace = TRUE))

# print contingency table
print(table(my_factor))
```

```
R> my_factor
R> Ord Pref
R>    10    10
```

A more advanced usage of function `table` is to consider more than one factor:

```
# set factors
my_factor_1 <- factor(sample(c('Pref', 'Ord'),
                             size = 20,
                             replace = TRUE))

my_factor_2 <- factor(sample(paste('Group', 1:3),
                             size = 20,
                             replace = TRUE))

# print contingency table with two factors
print(table(my_factor_1,
            my_factor_2))
```

```
R>          my_factor_2
R> my_factor_1 Group 1 Group 2 Group 3
R>      Ord      4      2      8
R>      Pref     1      1      4
```

The table that we created previously demonstrates the number of occurrences for each combination of groups. Therefore, it is worth knowing you can also use it with more than two factors.

7.3.5 Other Useful Functions

levels - Returns the Levels an object of class `factor`.

```
my_factor <- factor(c('A', 'A', 'B', 'C', 'B'))
print(levels(my_factor))
```

```
R> [1] "A" "B" "C"
```

as.factor - Transforms an object to the class `factor`.

```
my_y <- c('a','b', 'c', 'c', 'a')
my_factor <- as.factor(my_y)

print(my_factor)
```

```
R> [1] a b c c a
R> Levels: a b c
```

split - Based on a grouping variable and another vector, creates a list with subsets of groups of the target object. This function is best used to separate

different samples according to groups.

```
my_factor <- factor(c('A','B','C','C','C','B'))
my_x <- 1:length(my_factor)

my_l <- split(x = my_x, f = my_factor)

print(my_l)
```

```
R> $A
R> [1] 1
R>
R> $B
R> [1] 2 6
R>
R> $C
R> [1] 3 4 5
```

7.4 Logical Objects

Logical tests are at the heart of R. In one line of code, and we can test a condition for a large vector of data. This procedure is commonly used to find outliers in a dataset or to split the sample according to some condition, such as a particular period.

7.4.1 Creating logical Objects

Objects of class `logical` are created based on the use of condition tests on other objects. For example, in a sequence from 1 to 10, we can check what elements are higher than five with the following code:

```
# set numerical
my_x <- 1:10

# print a logical test
print(my_x > 5)
```

```
R> [1] FALSE FALSE FALSE FALSE FALSE TRUE  TRUE  TRUE  TRUE
R> [10] TRUE

# print position of elements from logical test
print(which(my_x > 5))

R> [1] 6 7 8 9 10
```

In the previous example, function `which` returned the index (position) where the condition is true (TRUE).

To perform equality tests, simply use the equality symbol twice (==):

```
# create char
my_char <- rep(c('abc', 'bcd'),
                 times = 5)

# print its contents
print(my_char)

R> [1] "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc"
R> [10] "bcd"

# print logical test
print(my_char == 'abc')

R> [1] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
R> [10] FALSE
```

For an inequality test, use symbol !=, as shown in the next code:

```
# print inequality test
print(my_char != 'abc')

R> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
R> [10] TRUE
```

It is also possible to test multiple logical conditions. For simultaneous occurrences of events, use operator &. For example, if we wanted to check the values from a sequence between 1 and 10 that are larger than 4 **and** smaller than 7, we write:

```
my_x <- 1:10

# print logical for values higher than 4 and lower than 7
print((my_x > 4)&(my_x < 7) )

R> [1] FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE
R> [10] FALSE

# print the actual values
idx <- which( (my_x > 4)&(my_x < 7) )
print(my_x[idx])

R> [1] 5 6
```

For non-simultaneous conditions, i.e., the occurrence of one event or other, use the operator `|`. For instance, considering the previous sequence, we can find the values greater than 7 **or** lower than four by writing:

```
# location of elements higher than 7 or lower than 4
idx <- which( (my_x > 7) | (my_x < 4) )

# print elements from previous condition
print(my_x[idx])
```

```
R> [1] 1 2 3 8 9 10
```

Be aware that we used parentheses to encapsulate the logical conditions for both cases. Although it is not strictly necessary, it is a good coding practice. We could have used `idx <- which(my_x > 7 | my_x < 4)` for the same result but, using parentheses makes the code cleaner by isolating the logical tests.

Another interesting use of logical objects is to test whether an item or more is found or not in another vector. For this, we use the operator `%in%`. For example, suppose you have the names of two countries, `c('Country 1', 'Country 2')`, and you want to know if you can find the same countries in another vector. This is an operation similar to using the equality test but in vector notation. Here's an example:

```
library(dplyr)
# location of elements higher than 7 or lower than 4
my_countries <- c('Country 1', 'Country 2')

# set df
n_obs <- 100
df_temp <- tibble(country = str_c('Country ',
                                    sample(1:10,
                                           size = n_obs,
                                           replace = TRUE)),
                    inflation.rate = rnorm(n_obs, sd = 0.05) ) %>%
glimpse()
```

```
R> Rows: 100
R> Columns: 2
R> $ country      <chr> "Country 3", "Country 1", "Country ~
R> $ inflation.rate <dbl> -0.037301808, -0.060783522, -0.0062~
# filter rows of df with selected tickers
df_temp <- df_temp %>%
```

```
filter(country %in% my_countries) %>%
glimpse()
```

```
R> Rows: 18
R> Columns: 2
R> $ country      <chr> "Country 1", "Country 2", "Country ~
R> $ inflation.rate <dbl> -0.060783522, -0.002706447, -0.0193~
```

The resulting dataframe only has rows for 'Country 1' and 'Country 2'. With operator `%in%`, we filtered our table so it only keeps the desired rows.

7.5 Date and Time

The representation and manipulation of dates is an important aspect of research. When you have dates in your dataset, you must be certain they are correctly represented in R with the correct timezone. In this section, we will give priority to package `lubridate` (Spinu et al., 2022), which offers efficient and diverse functions for time manipulation. There are, however, many packages that can also help the user. If the reader must perform a date operation not covered here, I suggest looking into packages `chron` (James and Hornik, 2022), `timeDate` (Wuertz et al., 2022), and `bizdays` (Freitas, 2022).

7.5.1 Creating Simple Dates

In R, several classes can represent dates. The choice between one to another depends on the required precision of time representation. Some situations only require the knowledge of the day (and not time), while in others, the clock time can be very important as the data is collected over a day and knowing the time of day of each data point can affect the research.

The most basic class, indicating the day, month, and year, is `Date`. Using package `lubridate`, we create a date class object with functions `ymd` (year-month-date), `dmy` (day-month-year) e `mdy` (month-day-year). The order of components, and choice of function, is set according to the input character. Have a look:

```
library(lubridate)

# set Date object (YMD)
print(ymd('2020-06-24'))
```

```
R> [1] "2020-06-24"
```

```
# set Date object (DMY)
print(dmy('24-06-2020'))
```

```
R> [1] "2020-06-24"
```

```
# set Date object (MDY)
print(mdy('06-24-2020'))
```

```
R> [1] "2020-06-24"
```

Note that the functions return the exact same object. The difference in usage is only by the way the input string is structured with the position of the day, month, and year.

One benefit of using the **lubridate** package is that its functions are smart when dealing with different formats. You should note that we defined the data elements using the dash (-) separator and numeric values, as in '2020-06-24' in the previous case. Other formats are also automatically recognized:

```
# set Date object
print(ymd('2020/06/24'))
```

```
R> [1] "2020-06-24"
```

```
# set Date object
print(ymd('2020&06&24'))
```

```
R> [1] "2020-06-24"
```

```
# set Date object
print(ymd('2020 june 24'))
```

```
R> [1] "2020-06-24"
```

```
# set Date object
print(dmy('24 of june 2020'))
```

```
R> [1] "2020-06-24"
```

This is a very useful property of **lubridate**, making it easy to import date information in different formats.

Now, using the **base** package, we can create a date with function **as.Date**:

```
# set Date from dd/mm/yyyy with the definition of format
my_date <- as.Date('24/06/2021', format = '%d/%m/%Y')
```

```
# print result
print(my_date)
```

R> [1] "2021-06-24"

The symbols used in *input format*, such as %d, %m, and %Y, indicate how the character object should be converted and where the day, month and year are in the text. Likewise, many other symbols may be used for processing dates in specific formats. An overview of the main symbols is given next.

Symbol	Description	Example
%d	day of month (decimal)	0
%m	month (decimal)	12
%b	month (abbreviation)	Apr
%B	month (complete name)	April
%y	year (2 digits)	16
%Y	month (4 digits)	2020

By using the previous table, you'll be able to create and represent dates in a vast number of ways. Notice how the **lubridate** functions, regarding **base**, are simpler and easier to use.

7.5.2 Creating a Sequence of Dates

An interesting aspect of objects **Date** is they interact with **numeric** objects and can be used for logical tests. If we wanted to add a day after a particular date, all we need to do is to add value 1 to the object, as shown next:

```
# create date
my_date <- ymd('2021-06-01')

# find next day
my_date_2 <- my_date + 1

# print result
print(my_date_2)
```

R> [1] "2021-06-02"

This property also works with vectors, facilitating the creation of **Date** sequences. See an example next.

```
# create a sequence of Dates
my_date_vec <- my_date + 0:15

# print it
print(my_date_vec)

R> [1] "2021-06-01" "2021-06-02" "2021-06-03" "2021-06-04"
R> [5] "2021-06-05" "2021-06-06" "2021-06-07" "2021-06-08"
R> [9] "2021-06-09" "2021-06-10" "2021-06-11" "2021-06-12"
R> [13] "2021-06-13" "2021-06-14" "2021-06-15" "2021-06-16"
```

A more customizable way for creating `Date` sequences is using function `seq`. It allows the creation of date sequences with custom time intervals or with a fixed vector size. For example, if we wanted a `Date` sequence every two days, we could use the following code:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-03-20')

# set sequence
my_vec_date <- seq(from = my_date_1,
                     to = my_date_2,
                     by = '2 days')

# print result
print(my_vec_date)
```

```
R> [1] "2021-03-07" "2021-03-09" "2021-03-11" "2021-03-13"
R> [5] "2021-03-15" "2021-03-17" "2021-03-19"
```

Likewise, if we wanted a sequence of dates for every two weeks, we can simply change input `by` to '`2 weeks`':

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-04-20')

# set sequence
my_vec_date <- seq(from = my_date_1,
                     to = my_date_2,
                     by = '2 weeks')

# print result
```

```
print(my_vec_date)
```

```
R> [1] "2021-03-07" "2021-03-21" "2021-04-04" "2021-04-18"
```

Another way to use function `seq` is by setting the desired length of the sequence of dates. For example, if we wanted an array of dates with 10 elements, we would write:

```
# set dates
my_date_1 <- as.Date('2021-06-27')
my_date_2 <- as.Date('2021-07-27')

# set sequence with 10 elements
my_vec_date <- seq(from = my_date_1,
                     to = my_date_2,
                     length.out = 10)

# print result
print(my_vec_date)
```

```
R> [1] "2021-06-27" "2021-06-30" "2021-07-03" "2021-07-07"
R> [5] "2021-07-10" "2021-07-13" "2021-07-17" "2021-07-20"
R> [9] "2021-07-23" "2021-07-27"
```

Once again, the interval between the dates is automatically defined by the function.

7.5.3 Operations with Dates

We can calculate difference of days between two dates by simply subtracting one from the other. Have a look:

```
# set dates
my_date_1 <- ymd('2015-06-24')
my_date_2 <- ymd('2020-06-24')

# calculate difference
diff_date <- my_date_2 - my_date_1

# print result
print(diff_date)
```

```
R> Time difference of 1827 days
```

The output of the subtraction operation is an object of class `difftime`. Just

like in a `list`, we can access its elements using double brackets. The numerical value of the difference of days is contained in the first element of `diff_date`:

```
# print difference of days as numerical value
print(diff_date[[1]])
```

```
R> [1] 1827
```

Going further, we can also use mathematical operators to test whether a date is more recent or not than another:

```
# set date and vector
my_date_1 <- ymd('2021-06-20')
my_date_vec <- ymd('2021-06-20') + seq(-5,5)

# test which elements of my_date_vec are older than my_date_1
my.test <- (my_date_vec > my_date_1)

# print result
print(my.test)
```

```
R> [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
R> [10] TRUE TRUE
```

The previous operation is useful when selecting a certain period of time in your dataset. This is a common practice in research. We set the first and last dates of the period we are interested in and use a logical test to find all dates in between:

```
library(dplyr)
library(lubridate)

# set first and last dates
first_date <- ymd('2021-06-01')
last_date <- ymd('2021-06-15')

# create data frame and glimpse it
my_temp_df <- tibble(date.vec = ymd('2020-05-25') + seq(0,30),
                      prices=seq(1,10,
                                 length.out = length(date.vec)))

# find dates that are between the first and last date
my_idx <- (my_temp_df$date.vec >= first_date) &
  (my_temp_df$date.vec <= last_date)
```

```
# use index to filter dataframe
my_temp_df_filtered <- my_temp_df %>%
  filter(my_idx) %>%
  glimpse()
```

```
R> Rows: 0
R> Columns: 2
R> $ date.vec <date>
R> $ prices <dbl>
```

In the previous code, the object `my_temp_df_filtered` will only contain rows for the time period between 2021-06-01 and 2021-06-15.

7.5.4 Dealing with Time

Using the `Date` class is sufficient when dealing only with days, and the hours are irrelevant. When it is necessary to consider time, we have to use an object of type `datetime`.

In the `base` package, one class used for this purpose is `POSIXlt`, which stores the contents of a date as a list. Another class is `POSIXct`, which stores dates as seconds counted from 1970-01-01. Due to its storage format, the `POSIXct` class takes up less computer memory and is the one used in package `lubridate`. Given that, we will prioritize it in these sections. Worth knowing that all examples presented here can also be replicated with the `POSIXlt` class.

In R, the time/date format also follows the ISO 8601 standard¹ and is represented as “year-month-day hours:minutes:seconds timezone” (YYYY-MM-DD HH:mm:SS TZM). See the following example:

```
# creating a POSIXct object
my_timedate <- as.POSIXct('2021-01-01 16:00:00')

# print result
print(my_timedate)
```

```
R> [1] "2021-01-01 16:00:00 -03"
```

The `lubridate` package also offers intelligent functions for creating date-time objects. These follow the same structural intuition as the date creation functions.

¹<https://www.iso.org/iso-8601-date-and-time-format.html>

```
library(lubridate)

# creating a POSIXlt object
my_timedate <- ymd_hms('2020-01-01 16:00:00')

# print it
print(my_timedate)
```

R> [1] "2020-01-01 16:00:00 UTC"

You should note that this class automatically adds the time zone from the operating system. If you need to represent a different time zone, you can do so with the `tz` argument:

```
# creating a POSIXlt object with custom timezone
my_timedate_tz <- ymd_hms('2020-01-01 16:00:00',
                           tz = 'GMT')

# print it
print(my_timedate_tz)
```

R> [1] "2020-01-01 16:00:00 GMT"

An important note in the case of `POSIXlt` and `POSIXct` objects, **the operations of sum and subtraction refer to seconds**, not days, as with objects from the `Date` class:

```
# Adding values (seconds) to a POSIXlt object and printing it
print(my_timedate_tz + 30)
```

R> [1] "2020-01-01 16:00:30 GMT"

7.5.5 Customizing the Format of Dates and Times

The ISO format for representing dates and `datetime` object in R may not be what we need. When writing reports, using a date-time format different than the local one can unnecessarily generate confusion in your audience.

In the same way as objects of class `Date`, there are specific symbols for dealing with components of a `POSIXlt` object. Next, we have a table with the main symbols and their meanings.

Symbol	Description	Example
%H	Hour (decimal, 24 hours)	23

Symbol	Description	Example
%I	Hour (decimal, 12 hours)	11
%M	Minutes (decimal, 0-59)	12
%p	AM/PM indicator	AM
%S	Seconds (decimal, 0-59)	50

To format a date, use the `format` function. Using the symbols presented in the previous table, the user can create any desired customization. See the following example, where we change a date vector to the American format (MM/DD/YYYY):

```
# create vector of dates
my_dates <- seq(from = as.Date('2020-01-01'),
                 to = as.Date('2020-01-15'),
                 by = '1 day')

# change format
my_dates_US_format <- format(my_dates, '%m/%d/%Y')

# print result
print(my_dates_US_format)
```

```
R> [1] "01/01/2020" "01/02/2020" "01/03/2020" "01/04/2020"
R> [5] "01/05/2020" "01/06/2020" "01/07/2020" "01/08/2020"
R> [9] "01/09/2020" "01/10/2020" "01/11/2020" "01/12/2020"
R> [13] "01/13/2020" "01/14/2020" "01/15/2020"
```

The same procedure can be used for `POSIXlt` objects:

```
# create vector of date-time
my_datetime <- as.POSIXlt('2020-02-01 12:00:00') + seq(0,560,60)

# change to US format
my_dates_US_format <- format(my_datetime, '%m/%d/%Y %H:%M:%S')

# print result
print(my_dates_US_format)

R> [1] "02/01/2020 12:00:00" "02/01/2020 12:01:00"
R> [3] "02/01/2020 12:02:00" "02/01/2020 12:03:00"
R> [5] "02/01/2020 12:04:00" "02/01/2020 12:05:00"
R> [7] "02/01/2020 12:06:00" "02/01/2020 12:07:00"
R> [9] "02/01/2020 12:08:00" "02/01/2020 12:09:00"
```

Likewise, we can customize our dates for very specific formats:

```
# set custom format
my_dates_my_format <- format(my_dates,
                           'Year=%Y | Month=%m | Day=%d')

# print result
print(my_dates_my_format)

R> [1] "Year=2020 | Month=01 | Day=01"
R> [2] "Year=2020 | Month=01 | Day=02"
R> [3] "Year=2020 | Month=01 | Day=03"
R> [4] "Year=2020 | Month=01 | Day=04"
R> [5] "Year=2020 | Month=01 | Day=05"
R> [6] "Year=2020 | Month=01 | Day=06"
R> [7] "Year=2020 | Month=01 | Day=07"
R> [8] "Year=2020 | Month=01 | Day=08"
R> [9] "Year=2020 | Month=01 | Day=09"
R> [10] "Year=2020 | Month=01 | Day=10"
R> [11] "Year=2020 | Month=01 | Day=11"
R> [12] "Year=2020 | Month=01 | Day=12"
R> [13] "Year=2020 | Month=01 | Day=13"
R> [14] "Year=2020 | Month=01 | Day=14"
R> [15] "Year=2020 | Month=01 | Day=15"
```

7.5.6 Extracting Elements of a Date

We can use function `format` to extract data elements such as the year, month, day, hour, minute and second. Look at the next example, where we retrieve only the hours of a `POSIXct` object:

```
library(lubridate)

# create vector of date-time
my_datetime <- seq(from = ymd_hms('2020-01-01 12:00:00'),
                   to = ymd_hms('2020-01-01 18:00:00'),
                   by = '1 hour')

# get hours from POSIXlt
my_hours <- format(my_datetime, '%H')

# print result
print(my_hours)
```

```
R> [1] "12" "13" "14" "15" "16" "17" "18"
```

Likewise, we can use symbols %M and %S to extract the hours, minutes and seconds of a `POSIXct` object:

```
# create vector of date-time
n_dates <- 10
my_datetime <- seq(from = ymd_hms('2020-01-01 12:00:00'),
                    to = ymd_hms('2020-01-01 18:00:00'),
                    length.out = n_dates) + sample(1:59,
                                                   size = n_dates)

# get minutes from POSIXlt
my_minutes <- format(my_datetime, '%M')

# print result
print(my_minutes)
```

```
R> [1] "00" "40" "20" "00" "40" "20" "00" "40" "20" "00"
```

Alternatively, we can use `lubridate` functions such as `hour` and `minute`:

```
# get hours with lubridate
print(hour(my_datetime))
```

```
R> [1] 12 12 13 14 14 15 16 16 17 18
```

```
# get minutes with lubridate
print(minute(my_datetime))
```

```
R> [1] 0 40 20 0 40 20 0 40 20 0
```

Functions for extracting other components of a date, such as `lubridate::year` and `lubridate::second`, are also available.

7.5.7 Find the Current Date and Time

R also allows the user to find the current date and time from the operating system. This is particularly useful when it is important for the user to know the date and time when the code was executed.

If you want to find the present day, use function `base::Sys.Date` or `lubridate::today`

```
# get today from base
print(Sys.Date())
```

```
R> [1] "2022-11-23"
# get today from lubridate
print(lubridate::today())
```

R> [1] "2022-11-23"

If you want to find the current date and time, we use function `base::Sys.time` or `lubridate::now`:

```
# get time from base
print(Sys.time())
```

R> [1] "2022-11-23 16:09:20 -03"

```
# get time from lubridate
print(lubridate::now())
```

R> [1] "2022-11-23 16:09:20 -03"

Going further, based on these functions, we can write:

```
# example of log message
my_sys_info <- Sys.info()
my_str <- str_c('Log of execution\n',
                 'Time of execution: ', now(), '\n',
                 'User: ', my_sys_info['user'], '\n',
                 'Computer: ', my_sys_info['nodename'])

# print it
cat(my_str)
```

```
R> Log of execution
R> Time of execution: 2022-11-23 16:09:20
R> User: msperlin
R> Computer: msperlin-Inspiron-5675
```

This is the exact time when this book was compiled in its final version. Notice we also got some details regarding username and computer with function `Sys.info`.

7.5.8 Other Useful Functions

`weekdays` - Returns the day of the week from one or more dates.

```
# set date vector
my_dates <- seq(from = ymd('2020-01-01'),
```

```

    to = ymd('2020-01-5'),
    by = '1 day')

# find corresponding weekdays
my_weekdays <- weekdays(my_dates)

# print it
print(my_weekdays)

```

```
R> [1] "Wednesday" "Thursday"  "Friday"      "Saturday"
R> [5] "Sunday"
```

months - Returns the month of one or more dates.

```

# create date vector
my_dates <- seq(from = ymd('2020-01-01'),
                 to = ymd('2020-12-31'),
                 by = '1 month')

# find months
my_months <- months(my_dates)

# print result
print(my_months)

```

```
R> [1] "January"    "February"   "March"       "April"
R> [5] "May"        "June"       "July"        "August"
R> [9] "September"  "October"    "November"   "December"
```

quarters - Returns the location of one or more dates within the year quartiles.

```

# get quartiles of the year
my_quarters <- quarters(my_dates)

# print it
print(my_quarters)

```

```
R> [1] "Q1" "Q1" "Q1" "Q2" "Q2" "Q2" "Q3" "Q3" "Q3" "Q4" "Q4"
R> [12] "Q4"
```

OlsonNames - Returns an array with the time zones available in R. In total, there are over 500 items. Here, we present only the first five elements.

```
# get possible timezones
possible_tz <- OlsonNames()

# print it
print(possible_tz[1:5])

R> [1] "Africa/Abidjan"      "Africa/Accra"
R> [3] "Africa/Addis_Ababa" "Africa/Algiers"
R> [5] "Africa/Asmara"
```

Sys.timezone - Returns the current timezone of the operating system.

```
print(Sys.timezone())
```

```
R> [1] "America/Sao_Paulo"
```

cut - Returns a factor by grouping dates and time.

```
# set example date vector
my_dates <- seq(from = as.Date('2020-01-01'),
                 to = as.Date('2020-03-01'),
                 by = '5 days')

# group vector based on monthly breaks
my_month_cut <- cut(x = my_dates,
                      breaks = 'month',
                      labels = c('Jan', 'Fev', 'Mar'))

# print result
print(my_month_cut)
```

```
R> [1] Jan Jan Jan Jan Jan Jan Jan Fev Fev Fev Fev Fev Mar
R> Levels: Jan Fev Mar
```

```
# set example datetime vector
my_datetime <- as.POSIXlt('2020-01-01 12:00:00') + seq(0,250,15)

# set groups for each 30 seconds
my_cut <- cut(x = my_datetime, breaks = '30 secs')

# print result
print(my_cut)
```

```
R> [1] 2020-01-01 12:00:00 2020-01-01 12:00:00
R> [3] 2020-01-01 12:00:30 2020-01-01 12:00:30
```

```
R> [5] 2020-01-01 12:01:00 2020-01-01 12:01:00
R> [7] 2020-01-01 12:01:30 2020-01-01 12:01:30
R> [9] 2020-01-01 12:02:00 2020-01-01 12:02:00
R> [11] 2020-01-01 12:02:30 2020-01-01 12:02:30
R> [13] 2020-01-01 12:03:00 2020-01-01 12:03:00
R> [15] 2020-01-01 12:03:30 2020-01-01 12:03:30
R> [17] 2020-01-01 12:04:00
R> 9 Levels: 2020-01-01 12:00:00 ... 2020-01-01 12:04:00
```

7.6 Missing Data - NA (*Not available*)

One of the main innovations of R is the representation of missing data with objects of class `NA` (*Not Available*). The lack of data can have many reasons, such as failure to collect information or simply the absence of it. These cases are generally treated by removing or replacing the missing information before analyzing the data. The identification of these cases, therefore, is imperative.

7.6.1 Defining NA Values

To define omissions in the dataset, use symbol `NA` without quotes:

```
# a vector with NA
my_x <- c(1, 2, NA, 4, 5)

# print it
print(my_x)
```

```
R> [1] 1 2 NA 4 5
```

An important information that you must remember from previous chapters is that an `NA` object is contagious. Any object that interacts with an `NA` will turn into the same class of missing data:

```
# a vector
my_y <- c(2, 3, 5, 4, 1)

# example of NA interacting with other objects
print(my_x + my_y)
```

```
R> [1] 3 5 NA 8 6
```

This property demands special attention if you are calculating a value recursively, such as when using functions `cumsum` and `cumprod`. In these cases,

any value after `NA` will turn into `NA`:

```
# set vector with NA
my_x <- c(1:5, NA, 5:10)

# print cumsum (NA after sixth element)
print(cumsum(my_x))
```

```
R> [1] 1 3 6 10 15 NA NA NA NA NA NA NA
```

```
# print cumprod (NA after sixth element)
print(cumprod(my_x))
```

```
R> [1] 1 2 6 24 120 NA NA NA NA NA NA NA
```

Therefore, when using functions `cumsum` and `cumprod`, make sure no `NA` value is found in the input vector.



Every time you use the `cumsum` and `cumprod` functions, make sure that there is no `NA` value in the input vector. Remember that every `NA` is contagious and recursive calculations will result in a vector full of missing data.

7.6.2 Finding and Replacing NA

To find `NA` values, use function `is.na`:

```
# set vector with NA
my_x <- c(1:2, NA, 4:10)

# Test and find location of NA
print(is.na(my_x))
```

```
R> [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
R> [10] FALSE
```

```
print(which(is.na(my_x)))
```

```
R> [1] 3
```

To replace it, use indexing with the output of `is.na`:

```
# set vector
my_x <- c(1, NA, 3:4, NA)

# replace NA for 2
```

```
my_x[is.na(my_x)] <- 2

# print result
print(my_x)
```

```
R> [1] 1 2 3 4 2
```

Another way to remove NA values is to use the function `na.omit`, which returns the same object, but without the NA values. Note, however, that the vector size will change and the output will be an object of class `omit`. Have a look:

```
# set vector
my_char <- c(letters[1:3], NA, letters[5:8])
```

```
# print it
print(my_char)
```

```
R> [1] "a" "b" "c" NA "e" "f" "g" "h"
```

```
# use na.omit to remove NA
my_char <- na.omit(my_char)
```

```
# print result
print(my_char)
```

```
R> [1] "a" "b" "c" "e" "f" "g" "h"
```

```
R> attr(,"na.action")
```

```
R> [1] 4
```

```
R> attr(,"class")
```

```
R> [1] "omit"
```

Although the class of the object has changed due to the use of `na.omit`, the basic properties of the initial vector remains. For example, using the function `nchar` in the resulting object is still possible.

```
# trying nchar on a na.omit object
print(nchar(my_char))
```

```
R> [1] 1 1 1 1 1 1 1
```

For other objects, however, this property may not hold. Some caution is advised when using `na.omit`. If needed, you can return to the basic class with the `as.*` functions, such as in `as.numeric` or `as.tibble`.

7.6.3 Other Useful Functions

complete.cases - Returns a logical vector indicating whether the lines of a bi-dimensional object are complete (without NA). This function is used exclusively for **dataframes** and matrices.

```
# create matrix
my_mat <- matrix(1:15, nrow = 5)

# set an NA value
my_mat[2,2] <- NA

# print index with rows without NA
print(complete.cases(my_mat))
```

R> [1] TRUE FALSE TRUE TRUE TRUE

7.7 Exercises

All solutions are available at <https://www.msp Merlin.com/afedR>.

01 - Consider the following vectors x and y:

```
set.seed(7)
x <- sample (1:3, size = 5, replace = T)
y <- sample (1:3, size = 5, replace = T)
```

What is the sum of the elements of a new vector resulting from the multiplication between the elements of x and y?

02 - If we performed a cumulative sum of a sequence between 1 and 100, in what element would this sum go beyond the value of 50?

03 - Using R, create an sequence called **seq_1** between -15 and 10, where the interval between values is always equal to 2. What is the sum of the elements of **seq_1**?

04 - Define another object called **seq_2** containing a sequence of size 1000, with values between 0 and 100. What is the standard deviation (functions **sd**) of this sequence?

- a) 45.26
- b) 28.91
- c) 22.19
- d) 12.94
- e) 74.17

05 - Calculate the sum between `seq_1` and `seq_2` vectors (see previous exercises). Did this operation work despite the different size of the vectors? Explain your answer. If it works, what is the highest value of the resulting vector?

- a) 191.5
- b) 150.8
- c) 109.0
- d) 171.2
- e) 130.4

06 - Let's assume that, on a certain date, you bought 100 shares in a company, paying `price_purchase` per share. After some time, you sold 30 shares for a 18 each and the remaining 70 shares were sold for 22 on a later day. Using a *script* in R, structure this financial problem by creating numeric objects. What is the total gross profit from this sequence of transactions on the stock market?

- a) R\$678
- b) R\$904
- c) R\$791
- d) R\$1.017
- e) R\$580

07 - Create a vector `x` according to the following formula, where $i = 1 \dots 100$. What is the sum of the elements of `x`?

$$x_i = \frac{-1^{i+1}}{2i-1}$$

08 - Create a z_i vector according to the following formula where $x_i = 1 \dots 50$ and $y_i = 50 \dots 1$. What is the sum of the elements of z_i ? Tip: check out how the `dplyr::lag` function works.

$$z_i = \frac{y_i - x_{i-1}}{y_{i-2}}$$

- a) -7.116
- b) -65.957
- c) -46.795
- d) -20.343
- e) -33.569

09 - Using `set.seed(10)`, create an object called `x` with random values from the Normal distribution with a mean of 10 and standard deviation of 10. Using the `cut` function, create another object that defines two groups based on values of `x` greater than 15 and lower than 15. How many observations you find in the first group?

10 - Create the following object with the following code:

```
set.seed(15)
my_char <- paste(sample(letters, 5000, replace = T),
                  collapse = '')
```

How many times is the letter 'x' found in the resulting text object?

11 - Based on the `my_char` object created earlier, if we divided it into several smaller pieces using the letter "b", what is the number of characters in the **largest** piece found?

12 - At the address <https://www.gutenberg.org/files/1342/1342-0.txt> it is possible to access a file containing the full text of the book *Pride and Prejudice* by Jane Austen. Use functions `download.file` and `readr::read_lines` to import the entire book as a vector of characters called `my_book` in R. How many lines does the resulting object have?

13 - Bind the vector of characters in `my_book` into a single object called `full_text` using command `paste0(my_book, collapse = '\n')`. Using the `stringr` package, how many times is the word 'King' repeated throughout the text?

14 - For the `full_text` object created earlier, use the `stringr::str_split` function to split the entire text using blanks. Based on the resulting vector, create a frequency table. What is the most used word in the text? Tip: Remove all cases of empty characters.

- a) the
- b) and
- c) you
- d) I
- e) a

15 - Assuming that a person born on 2000-05-12 will live for 100 years, what is the number of birthday days that will fall on a weekend (Saturday or Sunday)? Tip: use operator `%in%` to check for a multiple condition in a vector.

16 - What date and time is found 10000 **seconds** after 2021-02-02 11:50:02?

- a) 2021-02-02 09:39:55
- b) 2021-02-02 14:36:42
- c) 2021-02-02 12:39:23
- d) 2021-02-02 14:22:58
- e) 2021-02-02 13:23:34

Chapter 8

Programming and Data Analysis



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.1 R Functions



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.2 Using for Loops



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.3 Conditional Statements (`if`, `else`, `switch`)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4 Using apply Functions



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4.1 Using lapply



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4.2 Using sapply



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4.3 Using tapply



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4.4 Using `mapply`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4.5 Using `apply`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.4.6 Using `by`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.5 Using package purrr



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.5.1 Function `map_*`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.5.2 Function `safely`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.5.3 Function `pmap`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.6 Data Manipulation with Package `dplyr`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.6.1 Group Operations with `dplyr`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.6.2 Complex Group Operations with dplyr



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.7 Exercises



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

8.8 Exercises

All solutions are available at <https://www.mspferlin.com/afedR>.

01 - Create a function called `say_my_name` that takes a person's name as input and shows the text *Your name is ...* on the screen. Within the scope of the function, use comments to describe the purpose of the function, its inputs and outputs.

02 - Implement a test for the input objects, so that when the input name is not of the `character` class, an error is returned to the user. Test your new function and verify if it is working as expected.

03 - Create a vector with any five names, called `my_names`. Using a *loop*, apply function `say_my_name` to each element of `my_names`.

04 - In the database of Brasil.IO¹ you will find a table with names and genres. Import the data from the file and, using a loop, apply function `say_my_name`

¹<https://data.brasil.io/dataset/genero-nomes/groups.csv.gz>

to 15 random names in the database. Tip: you can read the data directly from the link using `readr::read_csv(LINK)`.

05 - Redo previous exercises using function `sapply` or `purrr::map`.

06 - Use the `BatchGetSymbols` package to download data from the SP500 ('^GSPC'), Ibovespa ('^BVSP'), FTSE ('^ FSTE') and Nikkei 225 ('^N225' index) from '2010-01-01' to the current date. With the imported data, use a loop to calculate the average, maximum and minimum return for each index during the analyzed period. Save all results in a single table and show it in the *prompt* of R.

07 - Redo previous exercise using functions `group_by` and `summarise`, both from package `dplyr`.

08 - On the Rstudio CRAN logs website² you will find data on download statistics for the base distribution of R in the *Daily R downloads* section. Using your programming skills, import all available data between 2020-01-01 and 2020-01-15 and aggregate them into a single table. Which country has the highest download count for R?

- a) PT
- b) VE
- c) DO
- d) US
- e) BE

²<http://cran-logs.rstudio.com/>

Chapter 9

Cleaning and Structuring Data



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.1 The Format of a `dataframe`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.1.1 Converting a `dataframe` Structure (long and wide)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.2 Converting lists into dataframes



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.3 Removing Outliers



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.3.1 Treating Outliers in `dataframes`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.4 Inflation and Price Data



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.5 Modifying Time Frequency and Aggregating Data



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.6 Exercises



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

9.7 Exercises

All solutions are available at <https://www.msperlin.com/afedR>.

01 - Consider the `dataframe` created with the following code:

```
library(tidyverse)
```

```
my_N <- 100
```

```
df <- bind_rows(tibble(ticker = rep('STOCK 1', my_N),
                        ref_date = Sys.Date() + 1:my_N,
                        price = 100 + cumsum(rnorm(my_N))),
                  tibble(ticker = rep('STOCK 2', my_N),
                        ref_date = Sys.Date() + 1:my_N,
                        price = 100 + cumsum(rnorm(my_N))) )

print(df)
```

```
print(df)
```

Is the `dataframe` format long or wide? Explain your answer.

02 - Change the format of the previous `dataframe`, from long to wide or vice versa.

```
library(tidyverse)
```

```
mv N <- 100
```

```
ref_date = Sys.Date() + 1:my_N,
price = 100 + cumsum(rnorm(my_N))) )
```

```
print(df)
```

03 - Consider the following list:

```
library(tidyverse)
```

```
my_l <- list(df1 = tibble(x = 1:100, y = runif(100)),
              df2 = tibble(x = 1:100, y = runif(100), v = runif(100)),
              df3 = tibble(x = 1:100, y = runif(100), z = runif(100)) )
```

Add all `dataframes` in `my_l` to a single object using `do.call` or `dplyr::bind_rows` functions. What happened to the `df1` data where `v` and `z` columns do not exist?

04 - Use the `BatchGetSymbols` package to download the SP500 index data ('^GSPC') from 1950-01-01 to 2021-01-01. What is the sum of the 5 highest positive returns on the index?

- a) 0.5014
- b) 0.7740
- c) 1.2754
- d) 0.2184
- e) 0.3257

05 - Use the `replace_outliers` function (see section 9.3.1) to remove *outliers* from all numeric columns of the SP500 data previously imported with `my_prob = 0.025`. How many lines were **lost** in this cleaning process?

- a) 2977
- b) 16769
- c) 7281
- d) 4649
- e) 9488

06 - Use the `BatchGetSymbols::BatchGetSymbols` function to import the prices of the FTSE index ('^ FTSE') from 2010-01-01 to 2021-01-01. Then, reconstruct the data at the annual frequency, defining each year's value as the last observation of the period. Tip: see the `dplyr::summary_all` function for a functional way to aggregate all the columns of a `dataframe`.

07 - Use the same daily data as the FTSE and reconstruct the data at the monthly frequency, again using the last observation of the period.

08 - For the same daily FTSE data, check the dates and prices of the 20 biggest price drops. If, for each of these cases, an investor bought the index at the price of the biggest drops and kept it for 30 days, what would be his average nominal return per transaction?

- a) 1.40%
- b) 2.53%
- c) 1.91%
- d) 0.88%
- e) 0.37%

Chapter 10

Creating and Saving Figures with ggplot2



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.1 The ggplot2 Package



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.2 Using Graphics Windows



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.3 Creating Figures with Function `qplot`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.4 Creating Figures with Function `ggplot`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.4.1 The US Yield Curve



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.5 Using Themes



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.6 Creating Panels with `facet_wrap`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.7 Using the Pipeline



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.8 Creating Statistical Graphics



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.8.1 Creating Histograms



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.8.2 Creating *boxplot* Figures



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.8.3 Creating *QQ* Plots



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.9 Saving Graphics to a File



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.10 Exercises



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

10.11 Exercises

All solutions are available at <https://www.mspferlin.com/afedR>.

01 - Download TESLA (TSLA) stock data with the `BatchGetSymbols` package for the last 500 days. Display the price line chart adjusted over time using the `ggplot2 :: ggplot` function. Make sure that:

- x and y axes are correctly named;
- The chart has a title (“Prices for TESLA”), subtitle (“Data from YYYY-MM-DD to YYYY-MM-DD”) and a *caption* (“Solution for exercise 01, chapter 10 - afedR”).

02 - Download stock data for:

- APPLE INC. (AAPL)
- MICROSOFT CORPORATION (MSFT)
- AMAZON.COM, INC. (AMZN)
- ALPHABET INC. (GOOG)

using the `BatchGetSymbols` package for the last 1500 days. Display stock prices with different line colors on the same graph. Adapt all other aspects of the graph from previous exercises such as title and axis labels.

03 - For the previous plot, add dots in the lines.

04 - For the same plot, separate the stock prices on different panels with the `ggplot::facet_wrap` function. Use the `scales = 'free'` argument to release the x and y axis of each panel.

05 - Change the theme of the previous graph to a black and white scale, both for the graph area and for the colors of the lines.

06 - For the previous data, present the histogram of the returns of the different stocks in different panels and save the result in a file called '`histograms.png`'.

07 - Use the `BatchGetSymbols::GetSP500Stocks` function to discover all tickers currently belonging to the SP500 index. Using `BatchGetSymbols`, download the annual return data for all stocks in the index, from 2015-01-01 to the current day. After that, create the average/variance map by plotting the average annual return as the y axis and the standard deviation as the x axis. Tip: Use the parallel option in `BatchGetSymbols` to speed up the execution of the code. Also, you will find many outliers in the raw data. Make sure that the graph is visible limiting the x and y axes (see the `ggplot2::xlim` and `ggplot2::ylim` functions).

08 - Head over to the Kaggle data website¹ and choose a particular dataset for your analysis. It need not be related to economics or finance. Feel free to make a decision based on your own interests. After downloading the data, create a visual analysis of the data. Fell free to try out one or more plots of interest.

¹<https://www.kaggle.com/datasets>

Financial Econometrics with R



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.1 Linear Models (OLS)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.1.1 Simulating a Linear Model



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.1.2 Estimating a Linear Model



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.1.3 Statistical Inference in Linear Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.2 Generalized Linear Models (GLM)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.2.1 Simulating a GLM Model



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.2.2 Estimating a GLM Model



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.3 Panel Data Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.3.1 Simulating Panel Data Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.3.2 Estimating Panel Data Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.4 Arima Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.4.1 Simulating Arima Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.4.2 Estimating Arima Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.4.3 Forecasting Arima Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.5 GARCH Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.5.1 Simulating Garch Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.5.2 Estimating Garch Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.5.3 Forecasting Garch Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.6 Regime Switching Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.6.1 Simulating Regime Switching Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.6.2 Estimating Regime Switching Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.6.3 Forecasting Regime Switching Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.7 Dealing with Several Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.7.1 Using `tapply` and `sapply`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.7.2 Using `by`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.7.3 Using `dplyr::group_by`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.8 Exercises



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

11.9 Exercises

All solutions are available at <https://www.msperlin.com/afedR>.

01 - Simulate the following linear process in R:

```
set.seed(5)

# number of obs
n_row <- 100

# set x as Normal (0, 1)
x <- rnorm(n_row)

# set coefficients
my_alpha <- 1.5
my_beta <- 0.5
```

```
# build y  
y <- my_alpha + my_beta*x + rnorm(n_row)
```

Now, estimate a linear model where `x` is the explanatory variable and `y` is the explained variable. Use the `summary` function on the estimation return object to get more details about the model. What is the estimated beta value of the simulated data?

- a) 0.4003
- b) 1.5038
- c) 0.8707
- d) 0.2910
- e) 0.6331

02 - Using the `car` package, test the joint hypothesis that the value of alpha is equal to 1.5 and the value of beta is equal to 0.5. What is the value of the resulting F test?

- a) 10.727
- b) 16.967
- c) 7.799
- d) 40.301
- e) 23.335

03 - Use the `gvlma` package to test the OLS assumptions for the model previously estimated. Does the model pass all tests? If not, increase the value of `n_row` to 1000 and try again. Did the increase in the number of observations of the model impact the assumptions test? In what way?

04 - Use the `BatchGetSymbols::GetSP500Stocks` function to download data for all stocks that are part of the current SP500 index for the last three years. Using the SP500 itself – ticker '`^GSPC`' – as the market index, calculate the beta for each of the stocks. Display the histogram of the estimated *betas*. Note that the SP500 returns are not available in the original database and must be downloaded and added to the original database.

05 - For previously imported data, estimate a panel data version for the market model (beta). In this version, each stock has a different intercept, but they share the same beta. Is the estimated beta significant at 5%?

06 - Using the tidyverse functions, `dplyr::group_by` and `dplyr::do`, estimate an ARIMA model for the returns of each stock, available from the import process of previous exercise. In the same output dataframe, create a

new column with the forecast in $t + 1$ for each model. Which stock has the highest expected return for $t + 1$?

07 - In the same code used for the previous question, add a new column-list with the estimation of an ARMA (1, 0)-GARCH(1, 1) model for the returns of each stock. Add another column with the volatility forecast (standard deviation) at $t + 1$.

By dividing the expected return calculated in the previous item by the expected risk, we have a market direction index, where those stocks with the highest index value have the highest ratio of expected return to risk. Which stock is more attractive and has the highest value of this index?

08 - For the same SP500 database, select 4 stocks at random and estimate a Markov regime switching model with a configuration equivalent to that presented in section 11.6.2. Use the `plot` function to display the graph of the smoothed probabilities and save each figure in a folder called '`fig`'.

Chapter 12

Reporting Results



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

12.1 Reporting Tables



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

12.2 Reporting Models



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

12.3 Creating Reports with *R Markdown*



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

12.4 Exercises



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

12.5 Exercises

All solutions are available at <https://www.mspferlin.com/afedR>.

01 - Observe the data available in the `grunfeld.csv` file. Import the data

into R using package `afedR` and create a descriptive table of the variables. This table should provide enough information for the reader to understand the data. Use `xtable` package to report it in *LaTeX* or Word/Writer format.

02 - Using the `BatchGetSymbols::GetSP500Stocks` function, select 4 stocks at random and download the adjusted price data for the last three years. Estimate an ARIMA(1, 0, 1) model for each stock and report the result on the R screen with the `texreg::screenreg` function.

03 - Create a new *Rmarkdown* report from previous two exercises. Compile the report in html and open it in your *browser*.

Optimizing Code



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.1 Optimizing your Programming Time



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2 Optimizing Code Speed



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.1 Profiling Code



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.2 Simple Strategies to Improve Code Speed



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.2.1 Use Vector Operations



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.2.2 Repetitive binding of `dataframes`



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.3 Using C++ code (package `Rcpp`)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.4 Using cache (package `memoise`)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.2.4.1 Using parallel processing (package `furrr`)



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.3 Exercises



You reached the end of the online version of *Analyzing Financial and Economic Data with R*. The full content of the book can be acquired at Amazon for less than ten dollars. Purchasing this book is a great way of supporting this and other projects of the author. If you are satisfied with the content, please leave your feedback at Amazon or by email (marceloperlin@gmail.com). The book is a lifelong project and I'll keep improving it based on the received feedback.

13.4 Exercises

All solutions are available at <https://www.mspelrin.com/afedR>.

01 - Consider the following code:

```
library(tidyverse)
library(forecast)
library(BatchGetSymbols)

ticker <- '^GSPC'
df_prices <- BatchGetSymbols(tickers = ticker,
                             first.date = '2010-01-01')[[2]]

my_arima <- auto.arima(df_prices$ret.adjusted.prices)
summary(my_arima)
```

Use functions `Rprof` and `profvis` to identify the bottleneck of the code. Which line number is taking the longest execution time?

02 - Use the `Rcpp` package to write a C++ language function that will add elements of two numerical vectors. The output must be another vector of the same size and with elements equivalent to the `x + y` operation. Use the `identical` function to test that all elements of both vectors are equal.

03 - Use the `tictoc` package to compare the performance of the previous function with the native operator `+`, and a loop-based version with the pre-allocation of the result vector. Which alternative has the shortest execution time and why? Does the `Rcpp` version beat the loop version?

04 - Use the `memoise` package to create a memorized version of function `Quandl::Quandl`. Use the new function to import data about the United States Consumer Price Index (quandl code '`'FRED/DDOE01USA086NWDB'`'). How much percentage speed gain do you get from the second call to the memorized version?

Bibliography

- Bache, S. M. and Wickham, H. (2022). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.3.
- Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., and Borges, B. (2021). *shiny: Web Application Framework for R*. R package version 1.6.0.
- Dahl, D. B., Scott, D., Roosen, C., Magnusson, A., and Swinton, J. (2019). *xtable: Export Tables to LaTeX or HTML*. R package version 1.8-4.
- Dancho, M. and Vaughan, D. (2022). *tidyquant: Tidy Quantitative Financial Analysis*. R package version 1.0.6.
- Dowle, M. and Srinivasan, A. (2022). *data.table: Extension of ‘data.frame’*. R package version 1.14.6.
- Dragulescu, A. and Arendt, C. (2020). *xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files*. R package version 0.6.5.
- Eddelbuettel, D., Francois, R., Allaire, J., Ushey, K., Kou, Q., Russell, N., Ucar, I., Bates, D., and Chambers, J. (2022). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.9.
- Freitas, W. (2022). *bizdays: Business Days Calculations and Utilities*. R package version 1.0.12.
- Garmonsway, D. (2022). *tidyxl: Read Untidy Excel Files*. R package version 1.0.8.
- Gentzkow, M., Kelly, B. T., and Taddy, M. (2017). Text as data. Technical report, National Bureau of Economic Research.
- Gorecki, J. (2016). *Rbitcoin: R & Bitcoin integration*. R package version 0.9.7.

- Harrison, J. (2022). *RSelenium: R Bindings for Selenium WebDriver*. R package version 1.7.9.
- James, D. and Hornik, K. (2022). *chron: Chronological Objects which can Handle Dates and Times*. R package version 2.3-58.
- Klik, M. (2022). *fst: Lightning Fast Serialization of Data Frames*. R package version 0.9.8.
- Leifeld, P. (2022). *texreg: Conversion of R Regression Output to LaTeX or HTML Tables*. R package version 1.38.6.
- Mirai Solutions GmbH (2022). *XLCConnect: Excel Connector for R*. R package version 1.0.6.
- Müller, K., Wickham, H., James, D. A., and Falcon, S. (2022). *RSQlite: SQLite Interface for R*. R package version 2.2.18.
- Perlin, M. (2022). *BatchGetSymbols: Downloads and Organizes Financial Data for Multiple Tickers*. R package version 2.6.4.
- Perlin, M. S. (2019). *GetQuandlData: Fast and Cached Import of Data from Quandl Using the json API*. R package version 0.1.0.
- Perlin, M. S. (2021). *simfinR: Import Financial Data from the SimFin Project*. R package version 0.2.4.
- R Core Team (2022). *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...* R package version 0.8-82.
- Raymond McTaggart, Gergely Daroczi, and Clement Leung (2021). *Quandl: API Wrapper for Quandl.com*. R package version 2.11.0.
- Ryan, J. A. and Ulrich, J. M. (2022). *xts: eXtensible Time Series*. R package version 0.12.2.
- Spinu, V., Grolemund, G., and Wickham, H. (2022). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.9.0.
- Teator, P. (2011). *R cookbook*. ” O'Reilly Media, Inc.”.
- Temple Lang, D. (2022). *XML: Tools for Parsing and Generating XML Within R and S-Plus*. R package version 3.99-0.12.
- Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.
- Vaughan, D. and Dancho, M. (2022). *tibbletime: Time Aware Tibbles*. R package version 0.1.7.

- Venables, W. N., Smith, D. M., Team, R. D. C., et al. (2004). An introduction to r.
- Wickham, H. (2019). *Advanced r*. CRC press.
- Wickham, H. (2022a). *rvest: Easily Harvest (Scrape) Web Pages*. R package version 1.0.3.
- Wickham, H. (2022b). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.1.
- Wickham, H. and Bryan, J. (2022). *readxl: Read Excel Files*. R package version 1.4.1.
- Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2022a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.4.0.
- Wickham, H., François, R., Henry, L., and Müller, K. (2022b). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.10.
- Wickham, H., Hester, J., and Bryan, J. (2022c). *readr: Read Rectangular Text Data*. R package version 2.1.3.
- Wuertz, D., Setz, T., and Chalabi, Y. (2022). *timeDate: Rmetrics - Chronological and Calendar Objects*. R package version 4021.106.
- Xie, Y. (2022). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.30.
- Zeileis, A., Gruen, B., Leisch, F., and Umlauf, N. (2022). *exams: Automatic Generation of Exams in R*. R package version 2.4-0.
- Zeileis, A. and the R community (2016). *fortunes: R Fortunes*. R package version 1.5-4.

Index

- [[]], 179
- adding column to dataframe, 162
- assign, 51
- base
 - !=, 230
 - ::, 44
 - <-, 206
 - ==, 230
 - %*%, 189
 - |hyperpage, 231
 - and, 230
 - as.character, 225
 - as.Date, 233
 - as.factor, 228
 - as.list, 186
 - as.matrix, 190, 191
 - as.numeric, 209, 226
 - c, 52
 - cat, 55
 - cbind, 193
 - class, 54, 212
 - colMeans, 193
 - colnames, 159, 188
 - complete.cases, 175, 249
 - cumprod, 211
 - cumsum, 210
 - cut, 207, 245
 - data.frame, 154
 - dim, 59
 - factor, 223
 - format, 57, 58, 240, 241
 - getwd, 66
 - gregexpr, 216
 - gsub, 216
 - head, 175
 - help, 71
 - install.packages, 42
 - installed.packages, 41
 - is.na, 247
 - length, 59
 - levels, 228
 - library, 43
 - list, 177
 - list.dirs, 77
 - list.files, 76, 108
 - load, 108
 - ls, 53
 - matrix, 187, 190
 - matrix multiplication, 189
 - max, 209
 - mean, 33, 184
 - min, 209
 - months, 244
 - na.omit, 176, 248
 - names, 159, 186

nchar, 62, 219
 ncol, 59
 nrow, 59
 numeric, 198
 OlsonNames, 244
 order, 165
 paste, 57, 212
 print, 39
 prod, 211
 quarters, 244
 rbind, 192
 regexpr, 216
 rep, 156, 199
 required, 44
 rm, 65
 rnorm, 200
 rowMeans, 193
 rownames, 188
 runif, 200
 sample, 202
 sapply, 184
 save, 108
 seq, 198, 235
 set.seed, 203
 setwd, 67
 sort, 210
 source, 49
 split, 229
 str, 55
 strrep, 213
 strsplit, 218
 sub, 216
 substr, 215
 sum, 209
 Sys.Date, 129, 242
 Sys.time, 243
 Sys.timezone, 245
 t, 192
 table, 227
 tail, 175
 unique, 176, 209
 unlist, 185
 update.packages, 45
 weekdays, 243
 which, 230
 which.max, 210
 which.min, 210
 write.csv, 101
 BatchGetSymbols, 127
 BatchGetSymbols, 129
 bizdays, 232
 bookdown, 32
 C++, 19
 character, 211
 chron, 232
 code completion, 72
 constants
 LETTERS, 214
 letters, 214
 month.abb, 214
 month.name, 214
 data.table, 174
 dataframes, 154
 dates, 34, 233
 datetime, 238
 decimal, 34
 devtools, 42
 install_github, 43
 difftime, 237
 dplyr
 %>%, 158
 bind_cols, 168, 170
 bind_rows, 168
 filter, 164
 mutate, 162
 select, 162
 environment, 38
 file types
 .R, 35
 .RData, 35, 107
 .Rmd, 35

.csv, 95, 101
.rds, 35, 107
Folder structure, 87
folder structure, 87
forcats
 fct_recode, 226
fst, 110
 read_fst, 110
 write_fst, 110
functions, 33
Julia, 19
latin characters, 34
Latin1, 34
LETTERS, 213
letters, 213
Levels, 223
list, 177
logical matrices, 191
logical operators, 64
lubridate, 232
 dmy, 232
 hour, 242
 mdy, 232
 minute, 242
 now, 243
 today, 242
 ymd, 232
MariaDB, 113
markdown, 36
Matrix, 186
mySQL, 113
NA, 246
NULL, 182
omit, 248
ordering dataframe, 165
prompt, 38
Python, 19
Quandl, 122
Quandl, 122
Quandl.api_key, 122
R consortium, 18
R foundation, 18
R language, 18
RBloggers, 25
readr
 read_csv, 97
 read_lines, 116
 write_lines, 118
readxl, 103
 read_excel, 103
recycling rule, 196
regex, 216
Research scripts, 85
RMarkdown, 20
Rmarkdown, 32, 36
RSelenium, 149
RSQlite, 115
 dbConnect, 115
 dbDisconnect, 115
 dbReadTable, 114
 dbWriteTable, 115
RStudio, 19, 24, 36
RStudio panels, 38
RStudio projects, 36
rvest, 146, 149
 html_nodes, 146
 html_table, 146
 read_html, 146
S language, 18
script editor, 38
shiny, 20
source, 48
splashr, 149
SQLite, 113
stringr, 211
 fixed, 216
 str_c, 212
 str_dup, 213
 str_length, 219

str_locate, 216
str_locate_all, 216
str_replace, 216
str_replace_all, 216
str_split, 218
str_sub, 215

tibble, 155
 data_frame, 155
 tibble, 155

tibbletime, 174

tidyquant, 140
 tq_index, 141
 tq_index_options, 141

timeDate, 232

transpose matrix, 192

UTF-8, 34

utils
 View, 156
 zip, 119

Vector, 186

webscraping, 145

writexl
 write_xlsx, 105

XLConnet, 103

xlsx, 103, 104
 write.xlsx, 105

XML, 149

xts, 172
 xts, 172