

# **The Essentials of R**

**A Toolkit for Data Science**

Thierry Warin, PhD

2025-06-26

# Table of contents

<b>Motivation</b>	<b>7</b>
<b>1 Introduction</b>	<b>10</b>
1.1 From Statistics to Data Science: A Brief History . . . . .	10
1.2 A Tale of Two Languages: R and Python for Data Science . . . . .	12
1.2.1 R: A Language Born for Statistics . . . . .	12
1.2.2 Python: The Versatile Workhorse . . . . .	13
1.2.3 R vs. Python: Complementary Strengths . . . . .	15
1.3 Why Learn Both R and Python? . . . . .	17
1.4 How This Book Is Organized . . . . .	18
1.5 Getting the Most Out of This Book . . . . .	21
1.6 Conclusion of the Introduction . . . . .	22
<b>2 R for the impatient</b>	<b>24</b>
2.1 R and Python Command Reference Table . . . . .	24
2.2 Pedagogical Notes . . . . .	26
2.3 Examples in R and Python . . . . .	26
2.3.1 Load Data . . . . .	26
2.3.2 Descriptive Statistics . . . . .	27
2.3.3 Histogram and Plot . . . . .	27
2.3.4 Arithmetic Operations . . . . .	27
2.3.5 Object Management . . . . .	28
2.3.6 Basic Stats on Vectors . . . . .	29
2.3.7 Plotting . . . . .	29
2.3.8 Frequency Tables and Mosaic Plot . . . . .	31
<b>3 A Toolkit: RStudio, Markdown, Github, Zotero</b>	<b>32</b>
3.1 RStudio IDE and RStudio Cloud . . . . .	33
3.1.1 Setting Up an RStudio Cloud Account and Project . . . . .	33
3.1.2 Understanding the RStudio Interface . . . . .	39
3.2 Writing Documents with R Markdown (and Quarto) . . . . .	50
3.2.1 Creating a New R Markdown Document . . . . .	50
3.2.2 The Structure of an R Markdown Document . . . . .	52
3.2.3 Quarto (.qmd): A Modern Alternative to R Markdown . . . . .	60
3.2.4 Getting Your Hands Dirty: Writing an R Markdown Report . . . . .	64

3.3	Using Git and GitHub for Version Control . . . . .	65
3.3.1	Setting Up GitHub and a New Repository . . . . .	65
3.3.2	Connecting your RStudio Project to GitHub . . . . .	66
3.3.3	Using a Consistent Naming Convention for Files . . . . .	68
3.3.4	The Git Workflow: Pull, Commit, Push . . . . .	68
3.4	Managing References and Citations with Zotero . . . . .	73
3.4.1	Zotero: Installation and Setup . . . . .	73
3.4.2	Better BibTeX for Zotero . . . . .	74
3.4.3	Configuring the R Markdown YAML for Citations . . . . .	75
3.4.4	Inserting Citations in R Markdown (with the citr Add-in) . . . . .	77
3.4.5	Getting Your Hands Dirty: Citing Sources in Your Report . . . . .	82
<b>4</b>	<b>Data Wrangling</b>	<b>84</b>
4.1	R as a Functional Language . . . . .	85
4.2	The Grammar of R: Packages, Libraries, and Pipes . . . . .	85
4.2.1	Using Packages and Libraries . . . . .	86
4.2.2	The Pipe Operator %>% . . . . .	87
4.2.3	Tidy Data Principles . . . . .	88
4.2.4	Data Frames and Types in R . . . . .	88
4.3	Importing Data . . . . .	90
4.3.1	Importing a CSV File . . . . .	90
4.3.2	Importing Data from Google Sheets . . . . .	91
4.3.3	Other Data Formats and Tools . . . . .	92
4.4	Exploring and Preparing Data . . . . .	93
4.5	Manipulating Data Frames . . . . .	95
4.5.1	Adding, Renaming, and Deleting Columns . . . . .	96
4.5.2	Filtering Rows (Subsetting the Data) . . . . .	97
4.5.3	Sorting Data . . . . .	99
4.6	Reshaping Data: Long vs Wide Format . . . . .	100
4.6.1	From Long to Wide . . . . .	100
4.6.2	From Wide to Long . . . . .	102
4.7	Merging Datasets . . . . .	103
4.8	Saving Your Data . . . . .	106
4.9	Summary . . . . .	107
4.10	Exercises: Practice Your Data Wrangling Skills . . . . .	108
<b>5</b>	<b>Creating Beautiful Visuals</b>	<b>112</b>
5.1	Foundations . . . . .	115
5.1.1	What makes an effective visualization? . . . . .	115
5.1.2	What makes an ineffective visualization? . . . . .	119
5.2	Bar chart . . . . .	121
5.3	Line chart . . . . .	123
5.4	Bubble chart . . . . .	125

5.5	Maps . . . . .	127
5.6	Esquisse . . . . .	131
5.7	Interactive Visualizations with D3 . . . . .	136
5.8	Code learned in this chapter . . . . .	141
5.9	Getting your hands dirty . . . . .	142
<b>6</b>	<b>Creating Data Dashboards</b>	<b>146</b>
6.1	Flexdashboard . . . . .	147
6.2	Layout . . . . .	148
6.2.1	Chart Stack (Fill) . . . . .	149
6.2.2	Chart Stack (Scrolling) . . . . .	149
6.2.3	Chart B . . . . .	150
6.2.4	Chart C . . . . .	150
6.3	Row . . . . .	151
6.3.1	Chart 2 . . . . .	151
6.3.2	Chart 3 . . . . .	151
6.4	Column . . . . .	152
6.4.1	Chart 2 . . . . .	152
6.4.2	Chart 3 . . . . .	152
6.5	Column . . . . .	153
6.5.1	Chart X . . . . .	153
6.5.2	Chart Y . . . . .	153
6.5.3	Gauges . . . . .	154
6.6	Sizing and Styling . . . . .	155
6.7	Multiple Pages . . . . .	156
6.8	Storyboards . . . . .	157
6.8.1	Commentary . . . . .	158
<b>7</b>	<b>Automating Data Collection with APIs</b>	<b>160</b>
7.1	WDI . . . . .	161
7.1.1	Database Description . . . . .	161
7.1.2	Functions . . . . .	161
7.1.3	tl;dr . . . . .	165
7.2	OECD . . . . .	165
7.2.1	Database Description . . . . .	165
7.2.2	Functions . . . . .	166
7.2.3	tl;dr . . . . .	171
7.3	spiR . . . . .	172
7.3.1	Database Description . . . . .	172
7.3.2	Functions . . . . .	172
7.3.3	tl;dr . . . . .	176
7.4	statcanR . . . . .	177
7.4.1	Database Description . . . . .	177



7.4.2	Functions	177
7.4.3	tl;dr	180
7.5	EpiBibR	180
7.5.1	Database Description	180
7.5.2	Functions	181
7.5.3	tl;dr	184
7.6	coronavirus	185
7.6.1	Database Description	185
7.6.2	Functions	186
7.6.3	tl;dr	189
<b>8</b>	<b>Debugging: Strategies, Tools, and Best Practices</b>	<b>192</b>
8.1	Learning Objectives	193
8.2	Types of Errors	193
8.2.1	Syntax Errors	193
8.2.2	Runtime Errors	194
8.2.3	Logical Errors	196
8.3	How R Reports Errors and Warnings	197
8.4	Strategies for Debugging	199
8.4.1	1. Read the Error Message (Carefully!)	200
8.4.2	2. Reproduce the Error Consistently	201
8.4.3	3. Simplify and Isolate the Problem	202
8.4.4	4. Use <code>traceback()</code> to Locate the Error	203
8.4.5	5. Pause Execution with <code>browser()</code> for Deeper Inspection	204
8.4.6	6. Use <code>debug()</code> , <code>debugonce()</code> , and <code>recover()</code> for Flexible Debugging	206
8.4.7	7. Leverage RStudio's Debugging Tools (Breakpoints & Step-Through)	208
8.4.8	8. Adopt a Systematic Debugging Workflow	211
8.5	Common R Errors and What They Mean	214
8.6	Practical Debugging Workflows (Putting it All Together)	220
8.7	Exercises – Getting Your Hands Dirty	224
8.7.1	Exercise 1: Sum of Sequence (Logical Error)	224
8.7.2	Exercise 2: Data Frame Binding (Runtime Error)	225
8.7.3	Exercise 3: Missing Library (Runtime Error)	225
8.8	Conclusion	226
<b>9</b>	<b>Conclusion</b>	<b>229</b>
9.1	From Pipeline to Practice – Charting Your Own Data-Science Journey	229
9.1.1	The bigger picture	229
9.1.2	What you should feel comfortable with now	230
9.1.3	Where to go next	230
9.1.4	A note on mindset	230
9.1.5	Your call to action	231
9.1.6	Final words	231

<b>10 Summary</b>	<b>232</b>
Five Cross-Cutting Themes . . . . .	236
What You Can Do Tomorrow . . . . .	236
Final Reflection . . . . .	237
<b>References</b>	<b>238</b>

# Motivation

To cite this book:

```
@book{warin_2025,  
  title = {The Essentials of R/Python},  
  url = {https://warin.ca/essentials/},  
  abstract = {},  
  author = {Warin, Thierry},  
  year = {2025},  
  doi = {}  
}
```

*Why another book on data pipelines, R, and literate documents?*—that is the first question I asked myself before sketching a single outline. The shelves are already crowded with programming manuals and statistics primers; Stack Overflow brims with snippets for every plotting hic-cup imaginable. Yet when I looked at the way analysts actually work inside organisations—marketing teams, city-planning offices, graduate labs—I saw a stubborn, recurring gap between *theory* and *practice*. Textbooks tell you what an *ordinary least squares* model is, but not how to juggle Excel extracts, Git conflicts, dashboard deadlines, and sceptical stakeholders **all in the same afternoon**. My motivation, then, was to write a field guide that sits squarely at the messy intersection of code, narrative, and organisational reality, anchored by three convictions: reproducibility matters, clear stories win minds, and open tools lower the barrier for everyone.

First, **reproducibility is no longer optional**. Ten years ago, you could get away with emailing a PowerPoint full of copied-and-pasted numbers; today, any executive who has lived through a spreadsheet debacle will ask, “*Where did that figure come from, and what happens when the data refreshes?*” Publishing a static PDF is tantamount to delivering stale bread—edible for a moment, useless tomorrow. I wanted a book that treats reproducibility not as a post-script or an academic luxury but as the default posture for commercial and civic work alike. R, with its scriptable grammar, and Quarto, with its “one-button render,” offer the most direct route to living documents. Demonstrating that route end-to-end—showing that the same `.qmd` can emit an HTML microsite, a PDF working paper, and a slide deck—felt like the most practical gift I could offer to practitioners who are drowning in re-work.

Second, **storytelling beats raw computation**. I have met brilliant engineers whose analyses never gained traction because the audience could not follow the logic ladders buried in a

REPL transcript. Conversely, I have watched modest statistical summaries sway corporate strategy simply because the analyst wrapped each number in a compelling, visually clean narrative. The book’s motivation is therefore as literary as it is technical: to persuade analysts that sentences, headings, and figure captions are *first-class citizens* of the data pipeline, not decorative afterthoughts. Markdown’s minimal syntax and RStudio’s live preview close the cognitive gap between writing and reading; you see, instantly, how your words frame your plots. Embedding literate habits early—“*Write the paragraph, then code the chunk that proves the paragraph*”—becomes a force multiplier for any future project. I wanted to illustrate that discipline in concrete, repeat-able templates rather than preach it from afar.

Third, **the toolchain should be democratic**. A chief motivation for choosing R over proprietary platforms is simple economics: zero-cost, community-driven software unlocks opportunity for students in Manila, civil-servants in Nairobi, and epidemiologists in São Paulo as easily as for Silicon Valley hires. By anchoring the book in RStudio Cloud, GitHub, and other free-tier services, I hoped to remove every logistical excuse—no need for admin rights, no licence fees, no GPU clusters—to get started. Too many educational resources assume the reader can install binaries freely or expense a Tableau licence; reality says otherwise. My inbox is full of messages from learners who cannot convince I-T to install a new package on the departmental machine. A browser-based IDE levels that field. The book’s step-by-step screenshots of account setup, YAML scaffolds, and GitHub Pages deployment are motivated by a single goal: let a first-year economics major on a borrowed laptop build and publish a dashboard *today*.

Another spur was **the fragmentation I witnessed inside teams**. Business analysts maintained PowerBI dashboards, data scientists wrote Python notebooks, academics clung to LaTeX, and policy researchers swapped Word docs. Each community reinvented the wheel of data import, cleaning, and charting, yet none could seamlessly share work with the others. Quarto—capable of running R, Python, Julia, and Observable JavaScript in the same file—offers a compelling lingua franca. But lingua francas die without cultural buy-in, so the book devotes a full chapter to *motivation by demonstration*: blending R wrangling with a light Python `scikit-learn` snippet **inside one document**, proving to sceptics that cross-language collaboration need not spawn dependency hell. Even though the rest of the workflow is biased toward R, that cameo signals openness, not tribalism.

I was also motivated by the **psychology of debugging**. Nothing erodes confidence faster than a mysterious NA cascade or a “subscript out of bounds” at 2 am before a grant deadline. While most texts bury debugging in an appendix, I place it centre-stage because it *changes how you write code in the first place*. By teaching `traceback()`, `debugonce()`, and test-driven habits early, the book argues that error-handling is an ingredient of design, not a mop-up operation. This orientation came from mentoring junior colleagues who, when stuck, would comment out half their script and pray. Once they learned to set conditional breakpoints and write a five-line test that fails fast, their productivity—and sleep—improved markedly. Capturing that motivational arc—“from panic to systematic curiosity”—was essential.

**Open science and civic accountability** provide a further motive. The COVID-19 years reminded us how much public trust rides on transparent modelling. Many pandemic dashboards were built in R; many were impossible to audit because the code sat in private silos. By weaving Zotero-driven citations, DOI-based data references, and permissive licences into the workflow, this book nudges readers toward a norm where sharing the *how* matters as much as sharing the *what*. If even a fraction of readers choose to publish their `.qmd` files alongside their HTML reports, the collective verifiability of public analyses will rise. That is a social motivation, not merely technical.

Pedagogically, I was motivated by the **power of narrative scaffolding**. Too many tutorials jump from CSV import to machine learning in sixty seconds, leaving novices with syntactic knowledge but no organiser for *when* to apply which tool. By structuring the book as a single, reusable pipeline—each chapter adding a new stage to the same project—I offer readers a cognitive map they can replay on their own data. The motivation is akin to learning music: practising scales is necessary but playing a full song is what builds confidence. The repeating data set, incrementally enriched and re-visualised across chapters, functions as that song.

The **community ethos** of the R ecosystem was another inspiration. Packages like `{tidyverse}`, `{testthat}`, and `{quarto}` exemplify polite defaults, expressive naming, and considerate documentation. I wanted a book that mirrors those values. Thus, every code chunk is runnable as-is, every figure is generated live rather than pasted, and every chapter ends with a “getting your hands dirty” lab encouraging readers to remix assets in their GitHub forks. The hope is not merely to teach commands but to induct newcomers into *how the community learns aloud*—through reproducible examples, issue threads, and blog posts. Motivation, here, is communal: each new literate report published publicly enlarges the commons.

Finally, I was motivated by **the joy of craftsmanship**. There is a quiet thrill when “Knit” turns raw sensor logs into a responsive dashboard, complete with citations and alt-text, in under a minute. That thrill—*seeing thought crystallise into shareable artefact*—is why many of us fell in love with programming. But joy fades when tooling fights back: when path hell, dependency mismatches, or silent rounding errors sabotage momentum. This book aims to minimise friction so that the joy resurfaces. If a reader, late at night, feels that small zing of satisfaction as their first Quarto site deploys, I will consider the motivational mission accomplished.

The book exists because modern knowledge work demands **always-up-to-date, transparent, and persuasive data narratives**; because the open-source ecosystem finally offers the tools to meet that demand without gatekeepers; and because learners deserve a single, opinionated roadmap that escorts them from blank canvas to public publication without detours into arcane configuration. Reproducibility, storytelling, and accessibility—those are the intertwined motives. Everything else—the screenshots, the debugging detours, the YAML snippets—serves that triple commitment.

# 1 Introduction

We live in an age where **data is ubiquitous** – from personal fitness trackers and social media feeds to scientific research and business analytics, vast amounts of data are being generated every second. Making sense of this data requires a blend of statistical thinking and computational power, which has given rise to the modern field of *data science*. In fact, the term “*Data Science*” can be traced back to the early 1960s, when it was proposed to describe a new profession focused on understanding and interpreting large data sets. Over the decades, as computing technology advanced, data science evolved into a distinct interdisciplinary field, merging traditional statistics with computer science and domain expertise. This introductory chapter provides a narrative of how data science emerged from its roots in statistics, why the programming languages **R** and **Python** have become essential tools for data science, and how this book will give you a head start in acquiring these skills. We will also outline the content of the book – which ranges from quick-start guides to data wrangling, visualization, and more – so you know what to expect in the journey ahead.

## 1.1 From Statistics to Data Science: A Brief History

Statistics as a discipline has a rich history going back centuries, traditionally focused on collecting data and drawing inferences using mathematical techniques. For much of the 20th century, statisticians developed methods on paper and applied them with calculators or primitive computers. However, the latter half of the century witnessed an **explosion of data and computing power** that transformed how we analyze information. In 1962, the influential statistician **John W. Tukey** foreshadowed this transformation in his essay “*The Future of Data Analysis*.” Tukey observed a shift occurring in the world of statistics and argued that statisticians needed to focus not just on theoretical mathematical results but on **data analysis** itself. He famously wrote, “*As I have watched mathematical statistics evolve, I have had cause to wonder and to doubt... I have come to feel that my central interest is in data analysis.*” This marked one of the early calls to merge statistical thinking with the practical analysis of real-world data, a cornerstone of what we now call data science.

By the 1970s, this merging of **statistics and computing** began to formalize. In 1977 the International Association for Statistical Computing (IASC) was founded with the mission to “*link traditional statistical methodology, modern computer technology, and the knowledge of domain experts in order to convert data into information and knowledge.*” Around the same time, Tukey introduced the concept of **exploratory data analysis** (EDA), emphasizing that

exploring data to form hypotheses is just as important as the confirmatory analysis to test hypotheses. These developments underscored a new approach: using computers not only to perform calculations but to **discover patterns and insights** from data.

Fast forward to the 1990s and early 2000s, the digital revolution led to an unprecedented increase in data generation. Personal computers became ubiquitous, the internet connected the world, and businesses began collecting massive datasets. In 2001, statistician **William S. Cleveland** published a seminal article, *“Data Science: An Action Plan for Expanding the Technical Areas of the Field of Statistics.”* In this plan, Cleveland argued that the field of statistics should broaden into what he explicitly called *data science*, by incorporating advances in computing and data management into the training of statisticians. He outlined new areas (such as multidisciplinary investigations and pedagogy of data science) that universities should develop, effectively laying groundwork for the **education of data scientists**.

By the late 2000s, the term *“data scientist”* had entered the lexicon of the tech industry. In 2008, companies like LinkedIn and Facebook began to use the title *data scientist* for roles that involved analyzing large-scale data and building data-driven products. This new role was a natural evolution of the analyst and statistician jobs, now requiring proficiency in programming and machine learning in addition to classical statistics. The importance of these roles grew rapidly – anecdotally underscored when Harvard Business Review in 2012 dubbed *“data scientist”* as *“the sexiest job of the 21st century”*. While the wording was playful, it highlighted that individuals who could wrestle with big data and draw meaningful insights were in extremely high demand.

The growth of data science has been nothing short of explosive. A telling statistic: by 2013, it was estimated that **90% of all the data in the world had been created in the previous two years alone**, illustrating the exponential rise of digital data in the modern era. This deluge of data – often referred to as *“big data”* – came from sources like social networks, sensors (e.g. smartphones, IoT devices), digital transactions, and scientific instruments. Traditional data analysis techniques and tools often struggled to scale up to such volumes, leading to the development of new algorithms, distributed computing frameworks (like Hadoop in 2006, and later Spark), and cloud storage solutions. Data science emerged as a response to these challenges, combining **statistical modeling, computer science (especially algorithms for handling big data and machine learning)**, and **domain knowledge** to extract useful insights from raw data.

Today, data science is a broad umbrella that covers everything from classical statistical analysis to modern machine learning and artificial intelligence. It is applied in virtually every domain: businesses use data science for analytics and decision support, governments use it for policy and public services, scientists use it to make discoveries in fields like genomics and astronomy, and so on. The practice of data science involves a **cycle of tasks**: formulating the right questions, collecting or accessing data, cleaning and **wrangling** that data into a usable form, analyzing the data through statistical methods or machine learning models, visualizing the results, and communicating insights.

Crucially, performing these tasks efficiently requires the right tools. In the early days, a statistician might have relied on **Fortran** or **C** to write custom analysis programs, or used specialized but inflexible software. Over time, more user-friendly and powerful tools emerged. Two of the most important tools in the modern data science toolkit are the programming languages **R** and **Python**. In the next sections, we will see how these two languages – one born in the statistical community, the other in the general programming community – rose to prominence and why learning them is essential for anyone aiming to get a head start in data science.

## 1.2 A Tale of Two Languages: R and Python for Data Science

Data science is inherently multidisciplinary, but at its core it relies on **programming** to handle data. Among the many programming languages available, **R** and **Python** have become the twin pillars of data science. Both are open-source, both have rich ecosystems of libraries/packages for data analysis, and both are widely used by data scientists. Yet they come from very different origins and have distinct strengths. Let's delve into each of these languages and see how they came to play a central role in data science.

### 1.2.1 R: A Language Born for Statistics

**R** is often described as a language *made by statisticians for statisticians*. It was created in the early 1990s by Ross Ihaka and Robert Gentleman, statisticians at the University of Auckland in New Zealand. Frustrated with existing tools for teaching and performing statistical analysis, they set out to develop a better system. R was first conceived as an implementation of the **S language** (a language for statistics developed at Bell Labs in the 1970s) that would be free and open to everyone. The first version of R was released in 1993, and it quickly gathered interest in the statistics community. By 1997, R became a part of the GNU free software project, and a stable 1.0 version was released in 2000. The name “R” itself is a nod to its origins: not only is it the letter after “S” (a successor to the S language), but it also represents the first names of its two creators, Ross and Robert.

From the outset, R was designed with **interactive data analysis** in mind. Unlike lower-level languages such as C or Java, R allows users to quickly perform complex statistical operations with simple commands. For example, R comes with a vast number of built-in statistical functions and also provides the ability to create graphics with a single line of code. It is not just a programming language but also an interactive environment: users can type commands at the R prompt and immediately see results, which is ideal for exploratory work. Over the years, R's capabilities have been massively extended by its community through **packages** – add-on modules that provide additional functions, data, and documentation. The Comprehensive R Archive Network (**CRAN**), R's primary package repository, was established in 1997 to host R source code and user-contributed packages. The growth of CRAN reflects R's popularity:



it started with only a dozen packages, but as of late 2024 it contained over **21,500 packages** covering diverse functionality. These packages enable everything from classical statistical tests and models to cutting-edge machine learning, and from data import tools to interactive web application frameworks.

One of R’s greatest strengths is **data visualization**. Base R graphics were powerful for their time, and the ecosystem expanded further with packages like **ggplot2**, which implemented a “grammar of graphics” for creating elegant and complex plots in a consistent way. In R, producing a publication-quality plot or chart often takes just a few lines of code, which helped establish R as a favorite tool in academia for creating figures to include in research papers. In addition, R has fostered a culture of **reproducible research** through tools like **RMarkdown** and **Quarto** (literate programming frameworks that allow blending text, code, and results in a single document) and integration with version control systems like GitHub. We will explore these topics in this book, as they are essential for collaborative and transparent data science work.

Importantly, R remains **highly popular in academic and research settings**. Its origins in the statistics community made it the go-to language in many university statistics departments. Even as other tools have emerged, R is still deeply embedded in fields like **biostatistics, social sciences, and econometrics**, where many advanced methods have been first developed in R. Surveys indicate that a substantial portion of data professionals continue to use R. For example, one industry report noted that around *35% of data scientists use R* as part of their workflow, particularly in domains requiring careful statistical analysis. R’s emphasis on statistical rigor and the extensive suite of specialized packages (such as those in the **tidyverse** for data manipulation or **Bioconductor** for genomics) make it a **favorite for statisticians and data analysts** who need to dig deep into data.

### 1.2.2 Python: The Versatile Workhorse

**Python**, on the other hand, did not begin as a tool for statisticians – it started as a general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python was conceived as a simple, clean, and **easy-to-read language** that would be approachable for beginners and powerful for experts. Van Rossum named it “Python” not after the snake, but as a playful reference to the British comedy series *Monty Python’s Flying Circus*. Python’s early development was driven by the need for a scripting language that could bridge the gap between the shell (command-line scripts) and more complex languages like C. The language emphasized code readability and productivity, allowing programmers to express concepts in fewer lines of code than would be possible in languages like C++ or Java. Over the 1990s and 2000s, Python steadily grew in popularity, especially among system administrators and software developers, thanks to its clear syntax and the fact that it’s open-source (meaning anyone can use it and contribute to it).

Python’s journey into the realm of data analysis and scientific computing picked up steam in the early 2000s. A pivotal development was the creation of numerical and scientific libraries for Python – most notably **NumPy** (for numerical arrays and linear algebra, first released around 2006) and **SciPy** (for scientific computing routines). These libraries gave Python the ability to handle matrix operations and advanced math, similar to MATLAB or R, thereby attracting scientists and engineers. In 2008, **pandas** was released, adding powerful data manipulation capabilities (particularly for working with tabular data, akin to data frames in R). By the 2010s, Python had a mature ecosystem for data analysis: one could use Python to ingest data, clean and transform it, perform statistical analysis or apply machine learning, and even create visualizations (with libraries like **Matplotlib** and **seaborn**).

The rise of **Jupyter Notebooks** (originally IPython notebooks) further boosted Python’s adoption in data science. Jupyter provides an interactive environment where code, output, visualizations, and narrative text can coexist – an approach very similar to RMarkdown in the R world. This made Python very appealing for exploration and reporting results in research and data analysis projects. As a result, by the late 2010s, Python had firmly established itself as a **leading language for data science** alongside R. In fact, current industry surveys show that Python is *the most widely used programming language among data scientists*. A 2024 data science survey reported that over **78% of data scientists use Python regularly**, making it the top language in the field. Python’s popularity stems from its versatility and the breadth of its **library ecosystem**: for almost any data science task, there is a Python library available. For example, Python offers specialized libraries for machine learning (such as **scikit-learn**, **TensorFlow**, **PyTorch**, and **Keras**), for natural language processing (**NLTK**, **spaCy**), for data visualization (**Matplotlib**, **seaborn**, **Plotly**), and much more. This rich ecosystem allows data scientists to tackle projects end-to-end in Python, from reading raw data and cleaning it, to modeling and deploying results.

Another key reason for Python’s dominance is its **use in production environments**. Python is not only a tool for analysis; it is also used to build data-driven applications and systems. Many companies choose Python for developing web services, automation scripts, and machine learning pipelines. Its general-purpose nature, combined with data science libraries, means a prototype analysis done in Python can more easily be integrated into a larger software system. This has made Python especially popular in industry settings where collaboration between data scientists and software engineers is common. While R can also be used in production (and tools like R Shiny allow deployment of R analyses as interactive web apps), Python’s strength lies in the fact that it is a full-fledged programming language suitable for building large-scale applications.

It’s worth noting that Python’s philosophy emphasizes **readability and simplicity**. Python code often looks closer to pseudocode, which lowers the barrier for newcomers to learn programming concepts. These qualities have made Python a favorite not just in data science, but in introductory programming courses worldwide. In recent years, organizations as well as academia have broadly adopted Python for data science training. Many new data scientists first learn Python and its libraries for tasks like data wrangling and machine learning. Indeed,

Python is sometimes touted as a “*generalist’s language*” – if you need to do a bit of everything (data cleaning, analysis, building a web service, etc.), knowing Python will go a long way.

### 1.2.3 R vs. Python: Complementary Strengths

Given that both R and Python are so capable, one might wonder: **which one should I learn?** The good news is that this is not an either/or proposition – for a budding data scientist, knowing both is ideal. Each language has its **strengths**, and they complement each other:

- **Statistical modeling and analysis:** R has a slight edge due to its origin. Many classical statistical techniques and cutting-edge methods developed by researchers are implemented in R first. If you’re doing academic research in statistics or fields like epidemiology or sociology, you might find that the specific models or tests you need are readily available in an R package. R’s syntax for modeling (e.g., using the `~` formula notation in functions like `lm()` for linear models) is concise and built into the language’s design. Python, while fully capable of statistics (via libraries like `statsmodels` or `scipy.stats`), often requires a bit more work to achieve what might be a single built-in function in R.
- **Machine learning and AI:** Python has become the go-to language in the machine learning community. Frameworks like **TensorFlow** and **PyTorch** (for deep learning) have Python interfaces and are widely used in industry and research for tasks in computer vision, NLP, and other AI fields. While R does have machine learning packages (for example, `caret` or `tidymodels` for traditional ML, and even interfaces to TensorFlow), the cutting-edge development in AI tends to happen in Python first. If your interest in data science leans towards building predictive models at scale or training neural networks, Python’s ecosystem is a major advantage.
- **Data manipulation (wrangling):** Both languages excel here, each with a slightly different flavor. In R, the **tidyverse** collection of packages (notably `dplyr` for data manipulation and `tidyr` for data shaping) provides a fluent, chainable syntax for transforming data. In Python, **pandas** offers similar functionality using DataFrame objects. Many find that complex data transformations are equally achievable in both; it often comes down to which syntax one finds more intuitive. Learning both can even give you insights into data manipulation from two perspectives, strengthening your overall understanding.
- **Data visualization:** R’s `ggplot2` is a powerhouse for statistical graphics, enabling layered, customizable plots that adhere to good visualization principles. Python’s Matplotlib (and its higher-level companions like `seaborn` or `Plotly`) are also very powerful. Some interactive or web-based visualizations might be simpler in Python with tools like Plotly Dash or Bokeh, whereas R has Shiny and interactive HTML widgets. If you want static, publication-quality plots, `ggplot2` is often praised for its aesthetics and ease once the grammar is learned. If you want interactive visualizations embedded in web apps,

Python might offer more options. This book will introduce you to creating visuals in both, which will enhance your ability to communicate data effectively.

- **Integration and Production:** As mentioned, Python integrates smoothly into software development workflows. If you need to **integrate analytics into a production system**, Python might be the better tool (e.g., integrating a predictive model into a web service). R has **RStudio Connect** and other solutions to deploy analyses, but Python's ubiquity in general software engineering means you're more likely to deploy Python code in a production environment. On the other hand, R's interactive tools like **Shiny dashboards** are extremely quick to develop for data-driven storytelling and are often used internally in organizations for reporting.

In practice, many data science teams leverage **both languages**. It's not uncommon to see, for example, an experimental analysis done in R and then translated to Python for operationalization, or initial data cleaning in Python and statistical analysis in R. Both R and Python are open-source and have communities that contribute to bridging them – for instance, there are packages that allow calling Python from R and vice versa. Rather than thinking of R and Python as competitors, this book approaches them as **complementary skills**. By learning both, you equip yourself with a flexible toolkit. You can choose the right tool for the task at hand, and more importantly, you'll understand the concepts of data science more deeply by seeing them implemented in two different ways.

To give a quick taste, here is a simple example of accomplishing a similar task in both R and Python. Suppose we want to get a quick summary of a famous dataset (the **Iris flower dataset**, a classic dataset in statistics and machine learning). In R, the built-in `iris` data frame can be summarized with one function call:

```
# R code: summarize the iris dataset
summary(iris)
```

The `summary()` function in R will output, for each column of the dataset, common statistics (for numerical columns, the minimum, first quartile, median, mean, third quartile, and maximum; for categorical columns like the Iris species, the counts of each category). For example, the output will tell us that the sepal lengths in the dataset range from 4.3 to 7.9 (in centimeters), the median sepal length is 5.8, and so on, along with the frequency of each of the three iris species (setosa, versicolor, virginica).

Now, here is how we could do a similar summary in Python:

```
# Python code: summarize the iris dataset
import seaborn as sns          # seaborn has sample datasets including iris
iris = sns.load_dataset("iris") # load the iris dataset into a pandas DataFrame
iris.describe()
```

In this Python snippet, we use the seaborn library to load the iris dataset (which gives a pandas DataFrame). The `iris.describe()` call then produces summary statistics for each numeric column in the dataset (count, mean, standard deviation, min, quartiles, max). The result is analogous to R's summary, though formatted a bit differently. With these two examples, you can see that **both R and Python allow us to quickly get a sense of data with minimal code**. Throughout this book, we will present code in both languages side by side for major tasks, reinforcing the idea that neither language is “better” universally – each simply has its own syntax and features. By practicing with both, you will become bilingual in the two most important languages of data science.

## 1.3 Why Learn Both R and Python?

For an aspiring data scientist, the recommendation to learn both R and Python might sound daunting at first – isn't it double the work? However, there are compelling reasons why familiarity with both languages will greatly enhance your capabilities (and marketability):

- **Breadth of Opportunities:** In industry job postings, you will often see **R or Python** listed as required or desired skills (and sometimes both). Some companies (or teams) use R extensively, especially in domains like marketing analytics or pharmaceuticals where many analysts come from statistics backgrounds. Others use Python, particularly in tech companies or startups that have machine learning components. If you know both, you can fit into either environment and even act as a bridge between teams. In fact, many large organizations use R for certain projects and Python for others, so being conversant in both is a huge asset.
- **Deeper Understanding:** Learning two different ways to do things can deepen your understanding. For example, working with data frames in R vs. Python: the concept is similar, but the R *tidyverse* approach (e.g., piping data through transformations) contrasts with the Python pandas approach (method chaining or sequential operations). By learning both, you'll gain a better conceptual grasp of data manipulation. Similarly, understanding how statistical modeling is formulated in R can give you insight that translates to using Python's stats libraries, and vice versa.
- **Leveraging Strengths of Each:** As discussed earlier, each language has tasks it's particularly well suited for. If you only know one, you might force it to do something in a suboptimal way, whereas knowing both lets you pick the right tool. For instance, you might prefer to quickly prototype a statistical analysis in R (taking advantage of its rich set of statistical tests and models), and then implement a scalable version of that analysis in Python for deployment. Or you might use Python to scrape data from the web or connect to an API (where its libraries are very handy), then switch to R to explore and plot the data. This book will expose you to such workflows, so you can see how real-world projects might involve multiple tools.

- **Community and Learning Resources:** Both R and Python have vibrant user communities, but sometimes one has better resources for a specific topic. For example, if you're doing a specific type of academic analysis, you might find a ready-made R package and a publication that goes with it. If you're doing something like image recognition, you'll find plenty of Python tutorials and code. By being multilingual, you can tap into both communities and literature. Additionally, sometimes finding a bug or understanding an algorithm is easier in one language's documentation than the other. If you can read code in both, you can learn from a wider pool of examples.
- **Future-proofing your skills:** The tech world evolves quickly. Today's popular tool might be supplanted by something else in a decade. By learning both R and Python, you're essentially learning two ways of thinking about programming (one leaning more towards functional/data-oriented, one towards object-oriented/general-purpose). This makes it easier to pick up new languages or tools in the future. In a sense, you're learning how to learn different syntaxes and approaches. Some of the readers of this book may go on to learn **SQL for databases**, or **Julia** for high-performance numerical computing, or **JavaScript** for interactive visualizations – having started with two languages will make you more adaptable in the long run.

In summary, **learning both R and Python gives you a comprehensive foundation for data science**. Each reinforces the other. Moreover, using both doesn't mean you have to constantly switch between languages for every task – often a project is done in one primary language. But having the flexibility to choose, and the ability to understand code or resources written in either, will accelerate your growth in this field. The goal of this book is to kick-start that process by providing parallel examples and explanations in R and Python, so you can see how a given data science task is accomplished in each. This approach will not only teach you R and Python, but also solidify the underlying *data science concepts* by viewing them through two lenses.

## 1.4 How This Book Is Organized

The chapters in this book are arranged to guide you from basic concepts to more applied techniques, gradually building your proficiency in both R and Python. Each chapter focuses on a specific aspect of the data science journey, often providing side-by-side examples in the two languages. Below is an overview of the chapters and what you can expect to learn from each:

- **Chapter 1: Introduction (this chapter)** – Sets the stage by discussing what data science is and why R and Python are essential tools for it. We covered the historical context of data science and introduced the rationale for learning both languages. By now, you should have a sense of the journey we're about to embark on and why it's worthwhile.

- **Chapter 2: For the Impatient** – This chapter is a quick-start guide for those eager to dive straight into coding. We recognize that many readers may want to see results quickly, so here we provide a *fast-track introduction* to R and Python. You will learn how to install and set up R and Python (and useful development environments like RStudio or Jupyter Notebooks), and you’ll run your first simple data science tasks in each. The idea is to give you a hands-on feel for both languages immediately. We’ll cover just the essentials to get you going: basic syntax, reading a dataset, and doing a simple analysis or plot, all with minimal theory – perfect for the impatient learner who wants to jump in and get their feet wet.
- **Chapter 3: Markdown & GitHub** – Documentation and version control are critical in any scientific or analytical work. In this chapter, we introduce tools and practices for **reproducible research** and collaboration. You will learn about **Markdown**, a lightweight markup language for formatting text, and how it is used within R (RMarkdown/Quarto documents) and Python (Jupyter notebooks) to create reports that blend text, code, and results. We’ll demonstrate how you can create a dynamic report that automatically updates when data or code changes – a key for reproducibility. Additionally, we cover the basics of using **GitHub** for version control, which is essential for collaborative projects and tracking changes in your code or documents. By the end of this chapter, you’ll know how to document your analysis clearly and use GitHub to manage and share your work, ensuring that your projects are transparent and reproducible.
- **Chapter 4: Data Wrangling** – Raw data is rarely ready for analysis. This chapter focuses on the crucial step of **data wrangling**, which involves cleaning, transforming, and preparing data for analysis. We will introduce you to R’s **tidyverse** (with emphasis on **dplyr** for data manipulation and **tidyr** for reshaping data) and the equivalent operations in Python’s **pandas** library. Through examples, you’ll learn how to carry out common tasks like handling missing values, filtering rows, selecting and renaming columns, creating new computed columns, merging datasets, and aggregating data by groups. We’ll also cover some best practices for organizing data (following principles of tidy data). By practicing these tasks in both R and Python, you’ll gain confidence in whipping messy datasets into shape, a skill that analysts and data scientists spend a large portion of time on in real life. This chapter lays the foundation for any analysis by ensuring you know how to get your data cleaned and structured properly.
- **Chapter 5: Visuals** – As the saying goes, “*a picture is worth a thousand words.*” Data visualization is a powerful way to explore data and communicate findings. In this chapter, we delve into creating **visuals** in R and Python. You’ll learn how to make various types of plots: from basic ones like bar charts, histograms, and scatter plots, to more advanced or specialized visuals like boxplots, heatmaps, or interactive charts. In R, we will primarily use **ggplot2**, unraveling its grammar of graphics to build plots layer by layer. In Python, we will use **Matplotlib** and **seaborn** for static plots, and also give a glimpse of interactive plotting libraries. We’ll emphasize not just how to code plots, but also principles of good visualization design – such as choosing appropriate chart

types for the data, using color effectively, and avoiding misleading representations. By comparing R and Python plotting code side by side, you'll appreciate differences in style (for example, the declarative style of ggplot2 vs. the state-machine style of Matplotlib) and become adept at both.

- **Chapter 6: Dashboards** – Data science work often culminates in results that need to be delivered to an audience, and **interactive dashboards** have become a popular way to share insights. In this chapter, we explore how to go from static analysis to interactive applications. In R, we introduce **Shiny**, a framework that allows you to build web-based dashboards using R code (letting users adjust inputs and see results update in real time). In Python, we look at frameworks like **Dash** (from Plotly) or **Streamlit**, which offer similar capabilities to create interactive data apps. You will learn the basics of structuring a dashboard, including UI components (like sliders, dropdowns, etc.) and reactive outputs (like plots or tables that respond to user input). We'll guide you through a simple example of building a dashboard in both R and Python, for instance an interactive data explorer for a dataset. This chapter shows how you can **present your analyses as interactive tools**, which is increasingly in demand in business settings (to let decision-makers play with the data) and also valuable for your own exploratory analysis.
- **Chapter 7: APIs** – Not all data you need will be sitting in a local file or database you have access to. Often, data must be fetched from web services or external sources. These are typically accessed via **APIs (Application Programming Interfaces)**. In this chapter, you will learn how to retrieve data from APIs using R and Python. We'll cover the basics of web requests and JSON data format, which is a common format for API responses. In R, packages like **httr** or **curl** can be used to make web requests, and **jsonlite** to parse JSON. In Python, the ubiquitous **requests** library makes it easy to call web APIs, and the built-in **json** module or **pandas** can help parse JSON into data frames. We will walk through an example, such as querying a real-world API (for example, pulling data from a public API like OpenWeatherMap for weather data, or the GitHub API to get information on repositories, or an economic data API). You'll learn how to authenticate if needed, how to send requests with parameters, and how to handle the returned data. Mastering APIs is important for data science because a lot of interesting data is available via online services – this chapter ensures you know how to tap into those sources programmatically in both languages.
- **Chapter 8: Debugging** – Writing code for data science, like any programming, inevitably leads to errors and bugs. Being able to **debug** effectively is a vital skill that will save you countless hours. In this chapter, we focus on debugging techniques and best practices in both R and Python. We will discuss how to read and interpret error messages in each language (which can initially seem cryptic, but carry important clues). You'll learn strategies for isolating problems: for example, using print statements or logging to understand what your code is doing, or using interactive debugging tools. R provides facilities like the **browser()** function and RStudio's debug mode, and Python



has the built-in **pdb** debugger and IDEs like VS Code or PyCharm with breakpoints. We'll illustrate debugging with a few common scenarios – for instance, a piece of code that isn't doing what's expected, and then walking through a methodical process to find the issue. Additionally, we'll cover simple **testing** practices: writing unit tests for functions, or at least thinking in a way that anticipates potential pitfalls. By the end of this chapter, you should be less intimidated by bugs and equipped with a systematic approach to fixing them, which will make your coding process much smoother.

- **Conclusion** – The concluding section of the book will reflect on the journey you've taken through the chapters. We'll summarize the key lessons learned and discuss the broader perspective – how R and Python fit into the future of data science, the importance of continual learning, and perhaps pointers on what to learn next beyond this book. The conclusion will tie together the narrative of why being well-versed in these tools and concepts gives you a strong foundation in the data science landscape.
- **Summary** – We provide a concise summary of the entire book for quick reference. This may include bullet-point recaps of each chapter's main points, essential commands or syntax in R and Python, and a checklist of skills you've acquired. The summary serves as a handy revision tool – you can skim it to remind yourself of techniques or concepts without going back through full chapters.
- **References** – Throughout the chapters, we cite sources, articles, and documentation (using an academic citation style) to support facts and to point you to further reading. In the references section, you'll find the full list of sources referenced in this book. These include academic papers (for historical notes like Tukey's 1962 paper or Cleveland's article), industry surveys, documentation for R/Python packages, and other books or articles that are relevant. We encourage you to consult these references if you wish to delve deeper into certain topics – for instance, reading the original “The Future of Data Analysis” paper by Tukey, or the official documentation of a library for more details.

## 1.5 Getting the Most Out of This Book

Before we conclude this introduction, a note on how to best use this book. **Hands-on practice is key** in learning programming and data science. We recommend that as you read through the chapters, you actively experiment with the code examples provided. If possible, have R and Python set up on your computer (Chapter 2 will help with that). Try running the example code, modify it, see what happens if you change a parameter or apply it to a different dataset. The real learning happens when you engage with the code, not just read it.

This book is written in an academic tone and provides references and historical context, but it is fundamentally a practical guide. Each chapter's content has been curated to give you both the **why** and the **how**: we discuss why a concept is important, and then show you how to implement it in practice. For undergraduate students or readers new to data science, some

sections may be challenging – don’t be discouraged. Data science is a wide field, and it’s normal if, for example, the first time you see a linear regression code or a complex dplyr chain it feels complex. With repeated exposure and practice, these concepts will become clearer. Feel free to revisit sections, and use the summary chapter as a refresher.

We also encourage a mindset of **curiosity and continuous learning**. The field of data science and the tools (R/Python) are continuously evolving – new packages arrive, new techniques become popular. Consider this book a launch pad. We cover essentials that are unlikely to go out of style soon (like core language features, fundamental packages, and general principles). Once you have the essentials, you will be well equipped to learn new libraries or techniques that come along. Being comfortable with reading documentation and searching for solutions is another skill we quietly hope you build – we provide many references precisely so you know where to look for more information.

Lastly, remember that **data science is a team sport as much as an individual skill**. Engage with others as you learn – discuss with classmates or colleagues, contribute to online forums, share your analysis code on GitHub, or even contribute to open-source packages when you’re ready. The narratives and history included in this introduction highlight that data science grew through collaboration across statistics and computer science, across academia and industry. Embracing that spirit will enhance your learning and open up opportunities.

## 1.6 Conclusion of the Introduction

Data science stands on the shoulders of statistics and computing. In this introduction, we saw how the convergence of these fields has led to a new discipline that is transforming the world. We also introduced **R and Python** as the two essential programming languages driving much of this transformation, each with its unique story and strengths. As an aspiring data scientist (or someone just curious about the field), learning these languages is one of the best investments you can make in your education. **The goal of this book is to make that learning process engaging, comprehensive, and practical.** By weaving together narrative (the “why”), history (the “how we got here”), and code (the “how to do it”), we aim to give you not just knowledge, but also an appreciation for the field of data science and the tools it relies on.

In the chapters ahead, you will progressively build up your skills: from writing your first lines of code, to wrangling real datasets, creating insightful visualizations, and deploying interactive results. You will encounter both R and Python throughout – sometimes one will feel easier than the other for a given task, sometimes vice versa, and that’s okay. The dual exposure is strengthening your versatility. By the end of this book, you should feel comfortable tackling data-centric problems using whichever tool is best suited, and understanding problems from multiple perspectives.

Embarking on this journey, keep in mind that **every data scientist was once a beginner**. The pioneers we mentioned – from Tukey to Cleveland to modern-day professionals – all started by grappling with new concepts and tools. With dedication and practice, you too will gain proficiency and possibly make your own contributions to this ever-evolving field. So, let's get started with the essentials of R and Python, and unlock the world of data science together.

## 2 R for the impatient

This chapter serves as a quick-reference guide and learning bridge for students and professionals who are new to data analysis with **R** or **Python**, or transitioning between the two. It is intended to build **foundational fluency** in using both languages to manipulate data, generate descriptive statistics, and produce visualizations.

By the end of this chapter, you will:

- Understand the basic syntax and logic of R and Python for data analysis
- Learn equivalent commands between R and Python for common data tasks
- Be able to read and write small data workflows in both languages
- Appreciate the different programming paradigms and ecosystems of R and Python
- Cultivate good habits in coding, inspecting, and visualizing data

This chapter is especially designed to be **hands-on** and **applied**. It favors practical functionality over theoretical depth (which will come later). It can be revisited throughout the book as a reference when working through more complex statistical and machine learning models.

### 2.1 R and Python Command Reference Table

Description	R Command	Python Equivalent
Obtain documentation	<code>help()</code>	<code>help(function_name)</code>
View usage examples	<code>example()</code>	<code>import pydoc; pydoc.help()</code>
Manually enter data	<code>c()</code> , <code>scan()</code>	<code>list()</code> , <code>input()</code>
Create a sequence	<code>seq()</code>	<code>range()</code> , <code>numpy.arange()</code>
Repeat values	<code>rep()</code>	<code>itertools.repeat()</code>
Load built-in dataset	<code>data()</code>	<code>from sklearn import datasets</code>
Spreadsheet view	<code>View()</code>	<code>df.head()</code> , <code>df.to_string()</code>
Inspect structure	<code>str()</code>	<code>type()</code> , <code>df.info()</code>
Read CSV file	<code>read.csv()</code>	<code>pandas.read_csv()</code>

Description	R Command	Python Equivalent
Load package	<code>library()</code>	<code>import module_name</code>
Dataset dimensions	<code>dim()</code>	<code>df.shape</code>
Vector length	<code>length()</code>	<code>len()</code>
List objects in memory	<code>ls()</code>	<code>dir(), locals()</code>
Remove object	<code>rm()</code>	<code>del object_name</code>
Variable names	<code>names()</code>	<code>df.columns</code>
Histogram	<code>hist()</code>	<code>matplotlib.pyplot.hist()</code>
Lattice histogram	<code>histogram()</code>	<code>seaborn.histplot()</code>
Stem plot	<code>stem()</code>	<code>matplotlib.pyplot.stem()</code>
Frequencies	<code>table()</code>	<code>collections.Counter(), value_counts()</code>
Cross-tabulation	<code>xtabs()</code>	<code>pandas.crosstab()</code>
Mosaic plot	<code>mosaicplot()</code>	<code>statsmodels.graphics.mosaicplot()</code>
Bin values	<code>cut()</code>	<code>pandas.cut()</code>
Mean	<code>mean()</code>	<code>numpy.mean(), df.mean()</code>
Median	<code>median()</code>	<code>numpy.median(), df.median()</code>
Apply by group	<code>by()</code>	<code>df.groupby().apply()</code>
Summary statistics	<code>summary()</code>	<code>df.describe()</code>
Variance and SD	<code>var(), sd()</code>	<code>numpy.var(), numpy.std()</code>
Sum values	<code>sum()</code>	<code>sum()</code>
Quantiles	<code>quantile()</code>	<code>df.quantile()</code>
Bar graph	<code>barplot()</code>	<code>matplotlib.pyplot.bar()</code>
Lattice barplot	<code>barchart()</code>	<code>seaborn.barplot()</code>
Boxplot	<code>boxplot()</code>	<code>matplotlib.pyplot.boxplot()</code>
Lattice boxplot	<code>bwplot()</code>	<code>seaborn.boxplot()</code>
Scatterplot	<code>plot()</code>	<code>matplotlib.pyplot.plot(), seaborn.scatterplot()</code>
Lattice scatterplot	<code>xyplot()</code>	<code>seaborn.relplot()</code>
Linear regression	<code>lm()</code>	<code>statsmodels.api.OLS(), sklearn.linear_model.LinearRegression()</code>
ANOVA	<code>anova()</code>	<code>statsmodels.api.anova_lm()</code>
Predictions	<code>predict()</code>	<code>model.predict()</code>
Non-linear fit	<code>nls()</code>	<code>scipy.optimize.curve_fit()</code>
Model residuals	<code>residuals()</code>	<code>model.resid</code>
Sampling	<code>sample()</code>	<code>random.sample(), df.sample()</code>
Repeat process	<code>replicate()</code>	<code>list comprehension, numpy.tile()</code>
Cumulative sum	<code>cumsum()</code>	<code>numpy.cumsum()</code>
Empirical CDF	<code>ecdf()</code>	<code>statsmodels.distributions.ECDF()</code>
Binomial distribution	<code>dbinom()</code>	<code>scipy.stats.binom</code>
Poisson distribution	<code>dpois()</code>	<code>scipy.stats.poisson</code>
Normal distribution	<code>pnorm()</code>	<code>scipy.stats.norm</code>

Description	R Command	Python Equivalent
Student t-distribution	<code>pt()</code>	<code>scipy.stats.t</code>
Chi-square	<code>pchisq()</code>	<code>scipy.stats.chi2</code>
Binomial test	<code>binom.test()</code>	<code>scipy.stats.binom_test()</code>
Proportion test	<code>prop.test()</code>	<code>statsmodels.stats.proportion.proportions_ztest()</code>
Chi-square test	<code>chisq.test()</code>	<code>scipy.stats.chi2_contingency()</code>
Fisher's test	<code>fisher.test()</code>	<code>scipy.stats.fisher_exact()</code>
Student t-test	<code>t.test()</code>	<code>scipy.stats.ttest_1samp()</code> , <code>ttest_ind()</code>
Normal QQ plot	<code>qqnorm()</code>	<code>scipy.stats.probplot()</code>
Add margins to table	<code>addmargins()</code>	<code>df.apply()</code> with margins
Proportions from table	<code>prop.table()</code>	<code>df.div(df.sum())</code>
Graphics parameters	<code>par()</code>	<code>matplotlib.rcParams</code>
Power analysis	<code>power.t.test()</code>	<code>statsmodels.stats.power.tt_ind_solve_power()</code>

## 2.2 Pedagogical Notes

- 1. Learn by comparison:** This chapter encourages “cognitive mapping” between languages. Comparing syntax fosters deeper structural understanding and strengthens both retention and flexibility.
- 2. Vocabulary building:** Think of R and Python as two dialects of the same statistical language. Learning the synonyms improves fluency, especially when reading others’ code.
- 3. Practice matters:** Run both versions of simple scripts. Use a dataset like `iris`, `mtcars`, or any CSV to experiment. Code repetition is key to internalizing patterns.
- 4. Expect asymmetry:** Not all commands will have perfect equivalents. That’s part of the learning curve. Focus on **what the function does**, not just how it’s called.

## 2.3 Examples in R and Python

### 2.3.1 Load Data

```
# R
mydata <- read.csv("https://example")
View(mydata)
dim(mydata)
names(mydata)
```

```
# Python
import pandas as pd
mydata = pd.read_csv("https://example")
mydata.head()
mydata.shape
mydata.columns
```

### 2.3.2 Descriptive Statistics

```
# R
mean(mydata$hsgradrate)
median(mydata$hsgradrate)
min(mydata$hsgradrate)
max(mydata$hsgradrate)
```

```
# Python
mydata['hsgradrate'].mean()
mydata['hsgradrate'].median()
mydata['hsgradrate'].min()
mydata['hsgradrate'].max()
```

### 2.3.3 Histogram and Plot

```
# R
hist(mydata$childpov, n=15, freq=FALSE, col="red")
plot(x=mydata$childpov, y=mydata$hsgradrate, col="red")
```

```
# Python
import matplotlib.pyplot as plt
plt.hist(mydata['childpov'], bins=15, color='red', density=True)
plt.show()
plt.scatter(mydata['childpov'], mydata['hsgradrate'], color='red')
plt.show()
```

### 2.3.4 Arithmetic Operations

```
# R
8 + 3
27 / 5
cos(-pi)
abs(-2^3)
sqrt(4068289)
x <- 8 + 3
y <- 3
x + y
x * y
z <- x * y
z
Z # will error
```

```
# Python
import math
8 + 3
27 / 5
math.cos(-math.pi)
abs(-2**3)
math.sqrt(4068289)
x = 8 + 3
y = 3
x + y
x * y
z = x * y
z
Z # will error
```

### 2.3.5 Object Management

```
# R
ls()
rm(y)
ls()
```

```
# Python
dir()
del y
dir()
```



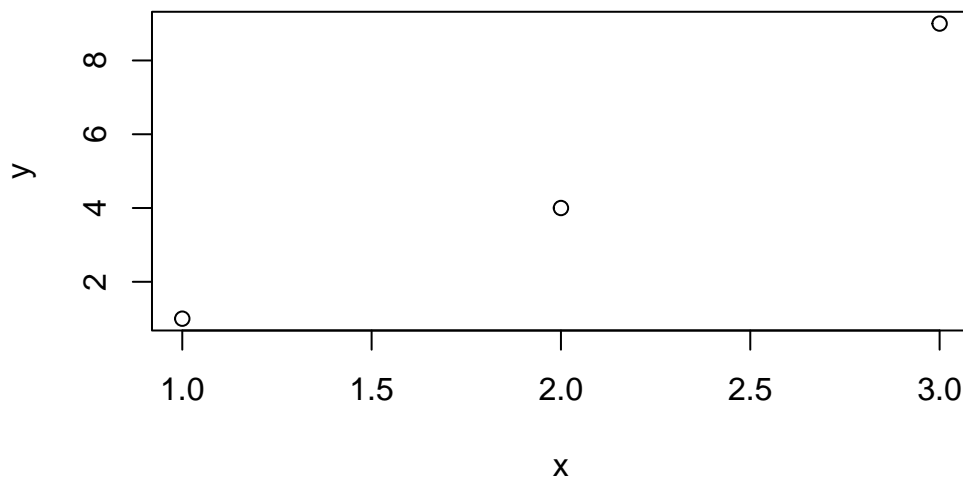
### 2.3.6 Basic Stats on Vectors

```
# R
ourdata <- c(-3,2,0,1.5,4,1,3,8)
length(ourdata)
ourdata[5]
mean(ourdata)
median(ourdata)
range(ourdata)
sd(ourdata)
var(ourdata)
summary(ourdata)
```

```
# Python
import numpy as np
ourdata = np.array([-3,2,0,1.5,4,1,3,8])
len(ourdata)
ourdata[4] # 0-indexed
np.mean(ourdata)
np.median(ourdata)
np.ptp(ourdata)
np.std(ourdata, ddof=1)
np.var(ourdata, ddof=1)
pd.Series(ourdata).describe()
```

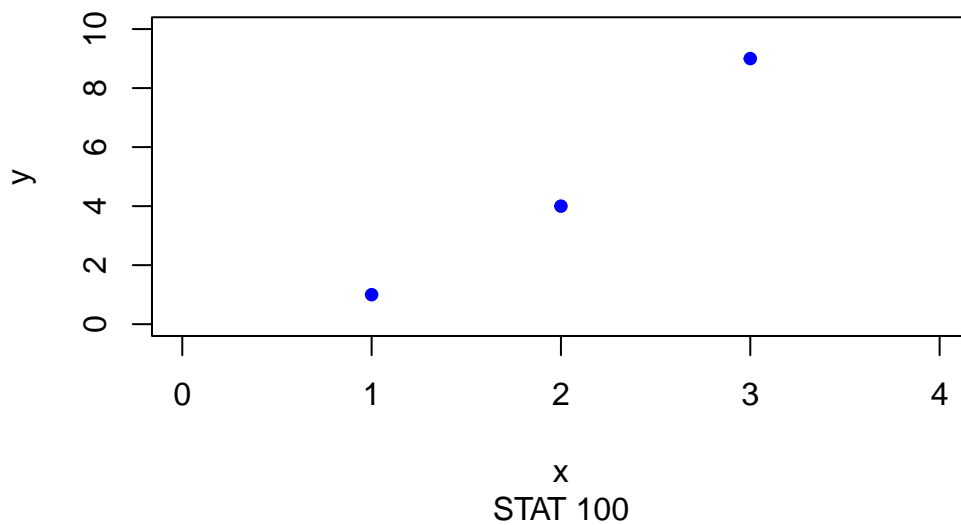
### 2.3.7 Plotting

```
# R
x <- c(1,2,3)
y <- c(1,4,9)
plot(x, y)
```



```
plot(x, y, xlab="x", ylab="y", pch=19, cex=0.8, col="blue", xlim=c(0,4), ylim=c(0,10), main=
```

## Our First Plot!



```
# Python
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [1, 4, 9]
plt.plot(x, y, 'bo-')
plt.xlabel("x")
plt.ylabel("y")
plt.xlim(0, 4)
```

```
plt.ylim(0, 10)
plt.title("Our First Plot!")
plt.suptitle("STAT 100")
plt.show()
```

### 2.3.8 Frequency Tables and Mosaic Plot

```
# R
table(mydata$childpov, mydata$hsgradrate)
xtabs(~hsgradrate, data=mydata)
xtabs(~hsgradrate + childpov, data = mydata)
mosaicplot(~hsgradrate + childpov, data = mydata)
```

```
# Python
import pandas as pd
pd.crosstab(mydata['childpov'], mydata['hsgradrate'])
pd.crosstab(index=mydata['hsgradrate'], columns='count')
pd.crosstab(mydata['hsgradrate'], mydata['childpov'])
from statsmodels.graphics.mosaicplot import mosaic
mosaic(mydata, ['hsgradrate', 'childpov'])
plt.show()
```

## 3 A Toolkit: RStudio, Markdown, Github, Zotero

In this chapter, we will familiarize you with the essential tools for a reproducible data science workflow: the RStudio integrated development environment (IDE), the R Markdown document format, version control with Git and GitHub, and reference management with Zotero. These tools working together will allow you to easily create data analysis reports that integrate code and results, collaborate with others, and ensure your work is transparent and up-to-date.

RStudio is more than just a code editor – it provides a unified interface to write and run code (in R, Python, and more) and to compile documents. R Markdown is a simple formatting language that lets you combine prose, code, and results in one place. Git is a version control system that keeps track of changes to your files and facilitates collaboration via platforms like GitHub. Finally, reference management tools like Zotero, combined with RStudio add-ins, allow you to cite sources and automatically generate bibliographies in your reports. Mastering these will greatly enhance your efficiency and the reproducibility of your work.

**At the end of this chapter, you should be able to:**

1. Understand what an **IDE** is and recognize the key features of the RStudio IDE.
2. Create an account on **RStudio Cloud** (Posit Cloud) and set up a new RStudio project.
3. Navigate the RStudio interface, knowing the purpose of each of its four panels.
4. Create and save your first R Markdown document in RStudio and compile (knit) it to produce a report.
5. Understand the basics of **Markdown syntax** in R Markdown, including YAML headers, text formatting, code chunks, inline code, embedding images, and writing equations.
6. Explain the importance of **Git** for version control and collaboration, and use GitHub to host a project repository.
7. Link an RStudio project with a GitHub repository and perform the typical Git workflow: **Pull**, **Commit**, and **Push** changes.
8. Apply a consistent file naming convention (e.g., *lowerCamelCase*) for your project files.
9. Use **Zotero** (with the Better BibTeX extension) to manage references, and configure an R Markdown document to use citations.
10. Insert citations into your R Markdown document using the **citr** add-in and understand the syntax for citations if the add-in is not available.
11. Compile a bibliography in your report and ensure all cited sources are properly listed.

Throughout this chapter, tips and important points will be emphasized in **bold** or *italics* to catch your attention.

## 3.1 RStudio IDE and RStudio Cloud

RStudio is an example of an **Integrated Development Environment (IDE)**. An IDE provides a graphical interface where you can write code, execute it, and manage the resulting output and files, all in one place. In simpler terms, an IDE connects your point-and-click actions to underlying commands (often the same commands you could run in a terminal) and organizes your workflow. Many IDEs exist, but RStudio is unique in how seamlessly it integrates multiple languages (R, Python, SQL, Markdown, and more) and tools for data science into a single cohesive environment. It lowers the barriers to entry for programming by providing an approachable interface without sacrificing power. In RStudio, a new user can easily mix narrative text and code to produce a report, while an advanced user can harness a full range of coding tools, all within the same window.

Even if you consider yourself a command-line expert, RStudio's interface can boost your productivity by simplifying tasks (like plotting, debugging, or version control) that would otherwise require remembering complex commands. RStudio (developed by Posit, PBC) has undoubtedly helped broaden access to data science, enabling more people to participate in the so-called *fourth industrial revolution* of data and AI by making powerful tools more accessible.

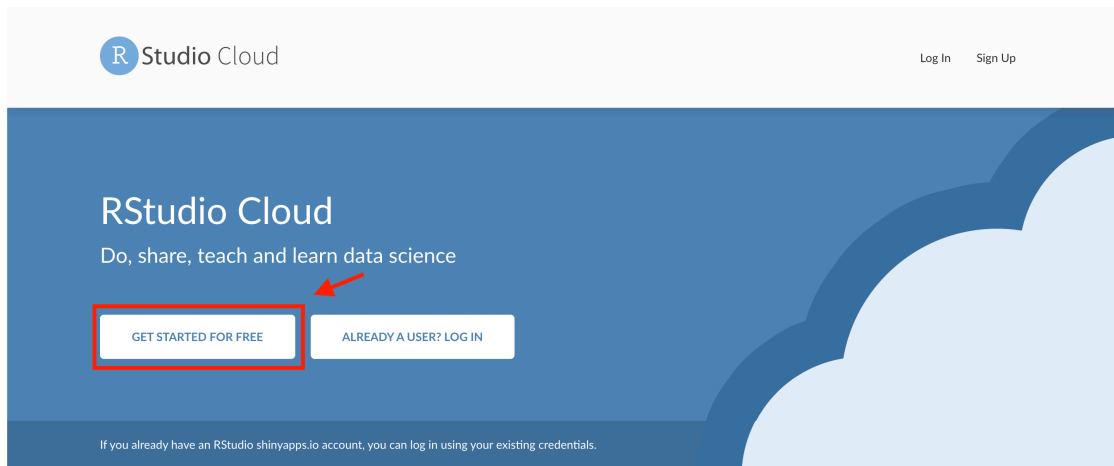
This book is **hands-on**. We encourage you to follow along by actually clicking buttons and typing commands in RStudio to experience the process first-hand. Our focus is on the early stages of a data pipeline: from getting data into a data frame (a tabular data structure in R, akin to a spreadsheet) to organizing and visualizing data, and finally creating a business-oriented report or dashboard. We will not delve into advanced statistical modeling here, but by the end of these chapters you will have a solid understanding of how to build a reproducible data analysis workflow.

To get started, we will use **RStudio Cloud**, a cloud-based instance of RStudio that runs in your web browser. (Note: RStudio Cloud has been rebranded as *Posit Cloud*, but the functionality remains the same. We will refer to it as RStudio Cloud here.) Using RStudio Cloud means you don't have to install anything on your computer initially, and you can access your RStudio environment from any machine with internet access. Later, if you prefer or require, you can install R and RStudio Desktop locally; however, the cloud version ensures everyone has the same setup during learning.

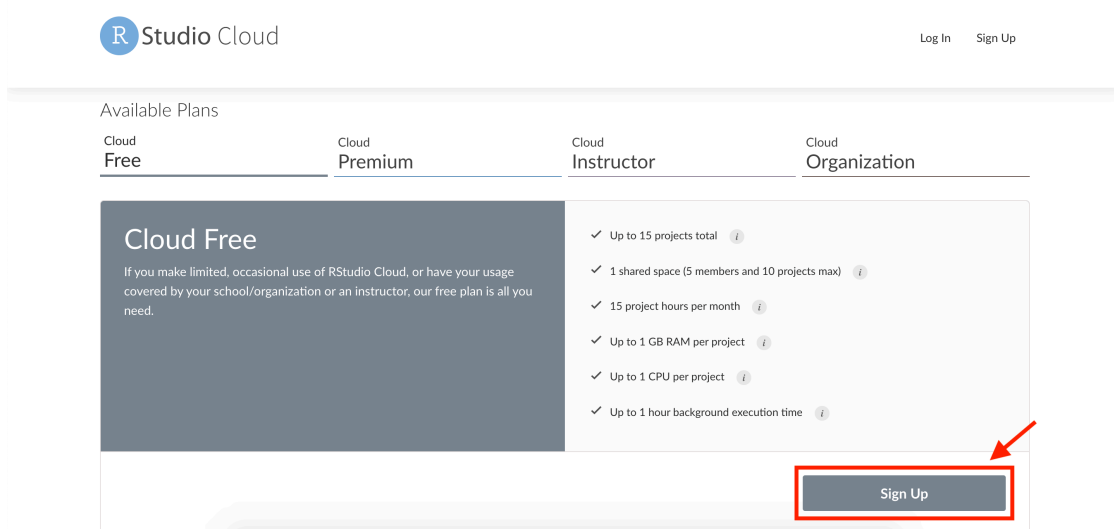
### 3.1.1 Setting Up an RStudio Cloud Account and Project

Follow these steps to create a free RStudio Cloud account and start a new project:

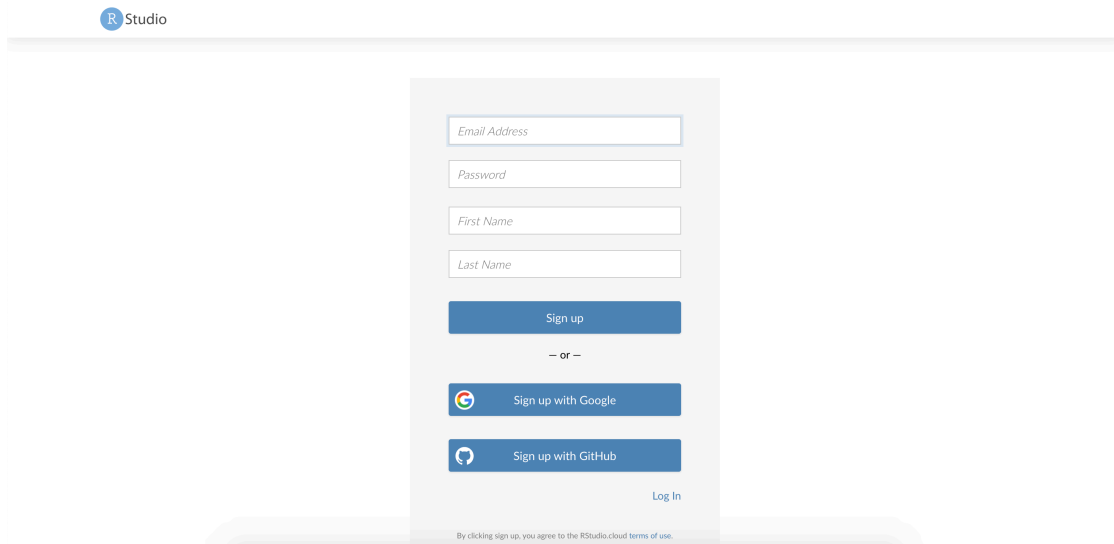
1. **Create an account on RStudio Cloud:** Open your web browser and go to <https://rstudio.cloud/>. Click on the “**GET STARTED FOR FREE**” button.



2. Click on the “**Sign Up**” button.

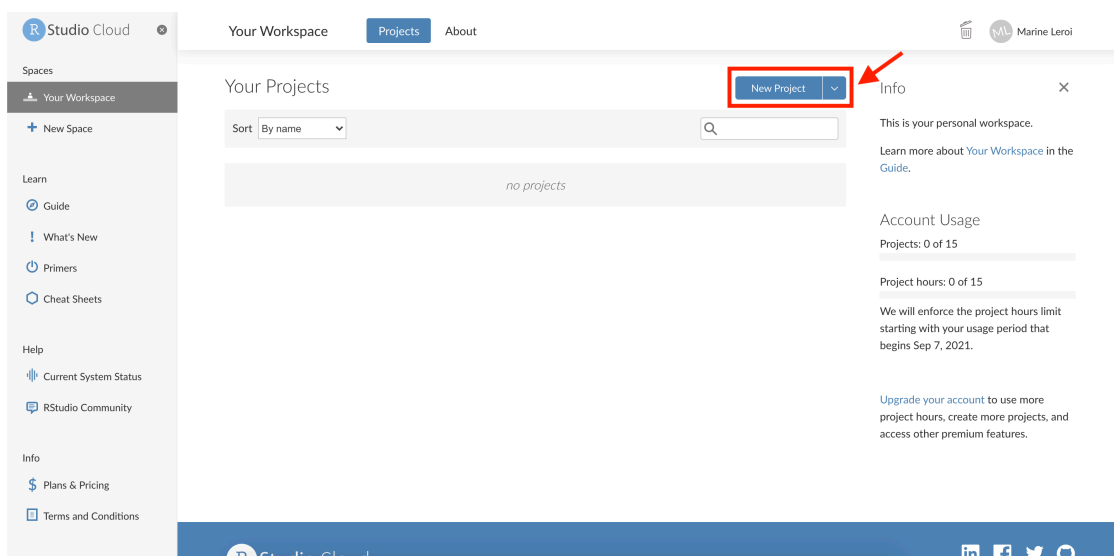


3. You will be presented with options to sign up. You can sign up using a Google account, GitHub account, or an email address. Choose whichever method you prefer and complete any required registration steps (such as verifying your email).

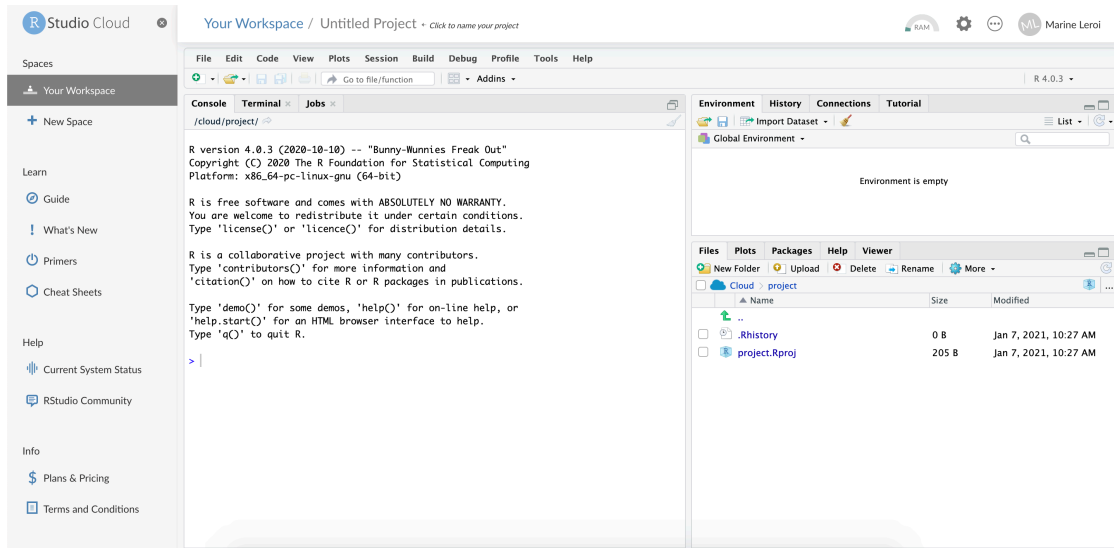


The image shows the RStudio Cloud sign-up page. At the top left is the RStudio logo. The main content area contains a sign-up form with the following fields: Email Address, Password, First Name, and Last Name. Below these fields is a blue 'Sign up' button. Underneath the button is a separator line with '— or —' in the center. Below the separator are two more buttons: 'Sign up with Google' (with the Google logo) and 'Sign up with GitHub' (with the GitHub logo). At the bottom right of the form is a 'Log In' link. At the very bottom, there is a small line of text: 'By clicking sign up, you agree to the RStudio cloud terms of use.'

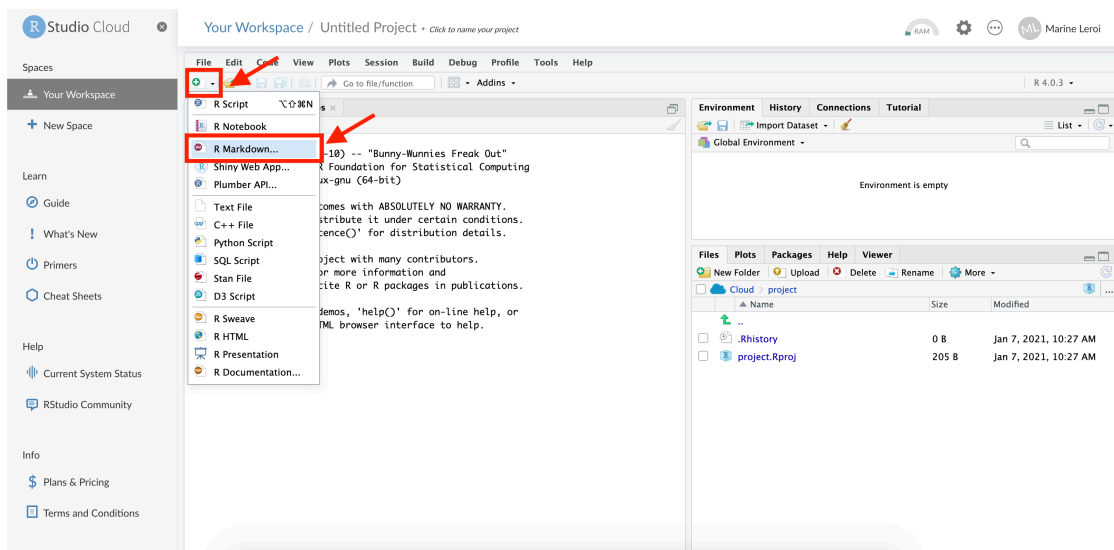
4. **Create a New Project:** Once logged in to RStudio Cloud, you will see a dashboard. Click on the “**New Project**” button to create a new RStudio project space.



5. After a moment, you should see a screen that allows you to configure the new project. You can generally accept the default settings here. The project will be provisioned (this may take a minute as a container with RStudio is started for you).

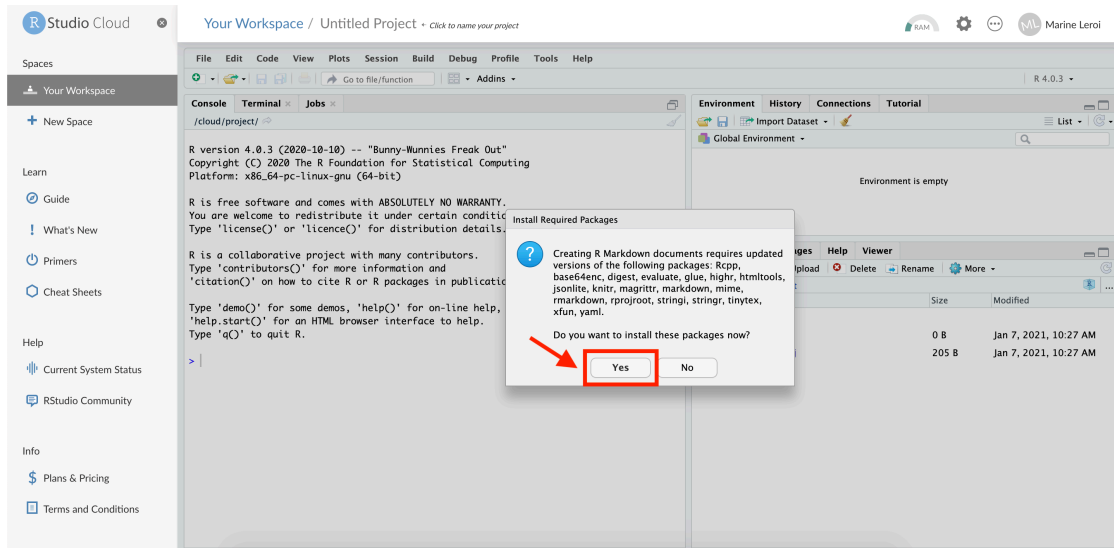


6. **Create a new R Markdown file:** Within your new RStudio Cloud project, you'll be presented with the RStudio IDE interface (which we will describe in detail shortly). To start writing a report, create an R Markdown file. Click on the green “+” button at the top-left of the interface (this button creates a new file), and from the dropdown choose “R Markdown...”.

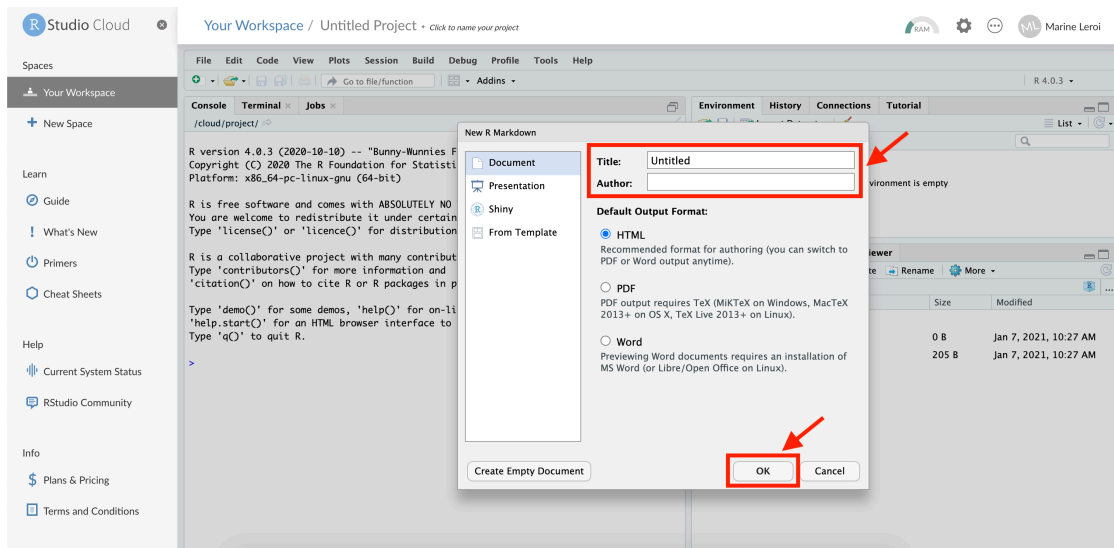


7. A dialog window may appear asking if you want to install necessary packages for R Markdown (if this is your first time using it on the project). Click “Yes” to install any required packages.

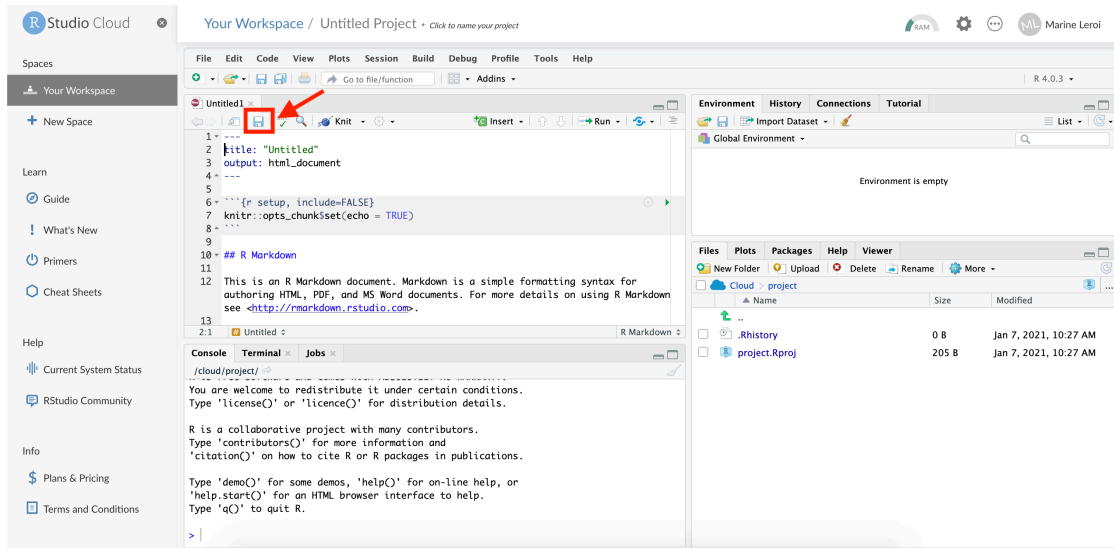




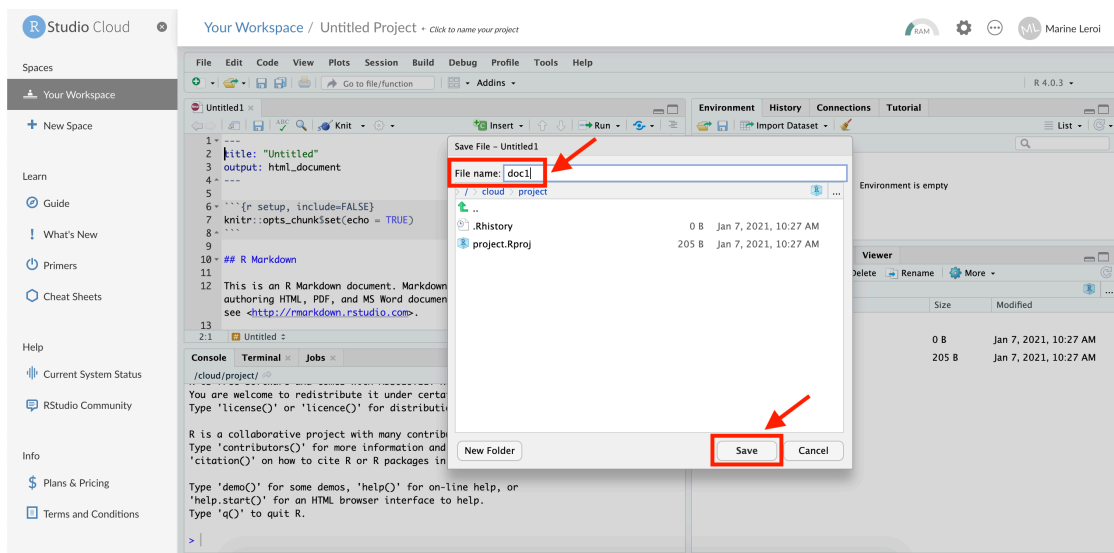
8. After a brief pause (while the system installs packages), another dialog window will appear titled “**New R Markdown**”. Here you can set a title and author for your new document. For now, you can leave the defaults or enter a sample title (e.g., “My First Report”) and your name as the author. Ensure the default output format is **HTML**. Then click “**OK**”.



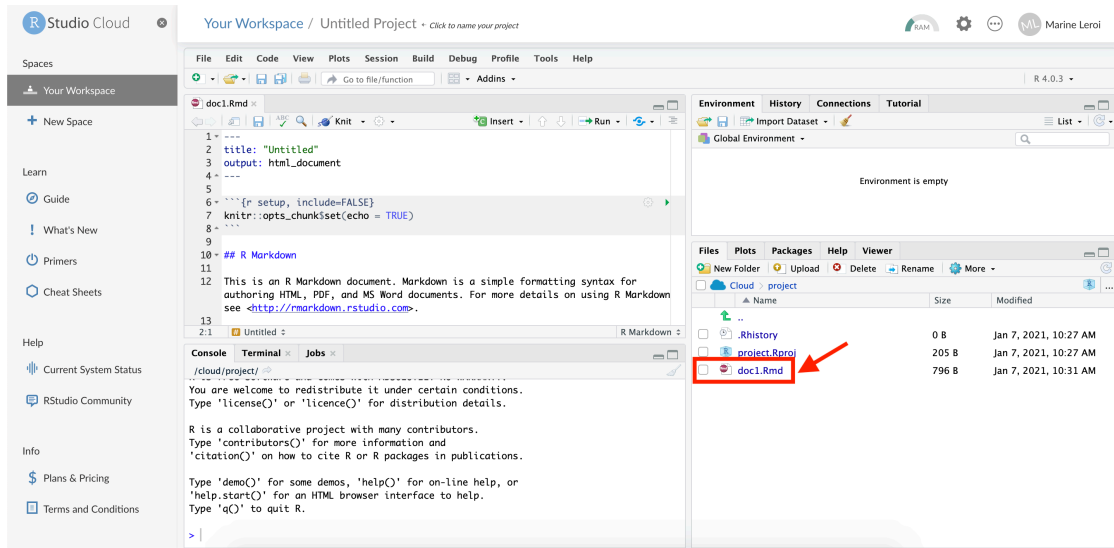
9. RStudio will now open a new file tab in the top-left panel with some example content (this is a template R Markdown document). Before we proceed, **save** this file to your project. Click the **floppy disk icon** ( ) or press **Ctrl+S** (Windows/Linux) or **Cmd+S** (Mac).



- When prompted, give the file a name (for example, “first\_document.Rmd”) and confirm the save. Use a name without spaces or special characters (R Markdown files should have the extension .Rmd).



- The new .Rmd file will now appear in the **Files** pane (usually the bottom right panel in RStudio). You have successfully created and saved an R Markdown document in your RStudio Cloud project.



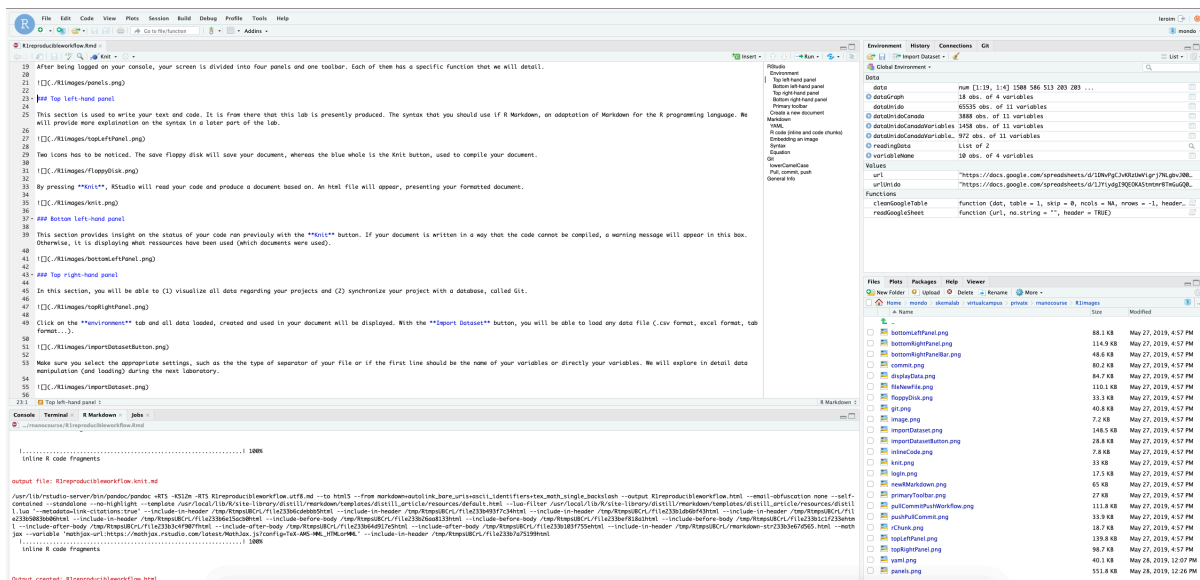
At this point, you have an RStudio Cloud account and a project with a simple R Markdown file ready to go. Next, we will take a closer look at the RStudio IDE interface and learn what each part of the screen is for.

### 3.1.2 Understanding the RStudio Interface

When you open a project in RStudio (whether via RStudio Cloud or the desktop version), the interface is typically divided into **four panels** plus a menu toolbar. Each panel has a specific purpose and knowing what each one does will make you comfortable navigating RStudio as you develop your report.

Think of RStudio as analogous to a suite like Microsoft Office, but all integrated into one window. In one part of the screen you might be writing text (like in Word), in another you might be viewing data (like in Excel), and in another you could be typing commands (like a terminal). The difference is that in RStudio these components talk to each other fluidly. For instance, you can run a snippet of code and immediately see the result (a table or a chart) in another panel. Moreover, every time you re-run or *compile* your R Markdown document, you regenerate the analysis with the latest data and code, ensuring your report is always up-to-date.

Let's break down the components of the RStudio interface (assuming the default layout):

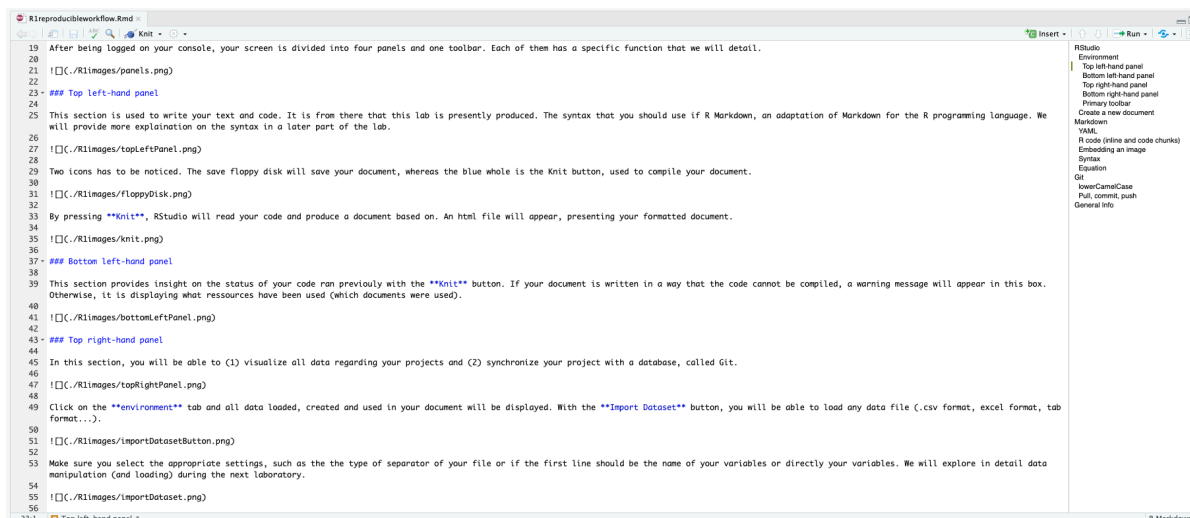


### 3.1.2.1 Top Left Panel: Source Editor

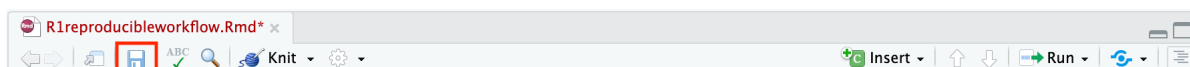
The **top left panel** is the **Source Editor**. This is where you write your scripts and documents. In our case, this is where you will write your R Markdown document (with both text and code). You can think of this area as your text editor or word processor that has programming superpowers. For example, the content of this book was written in the R Markdown source editor panel of RStudio.

In the editor, you can have multiple tabs open (for multiple files). By default, when we created the new R Markdown file, it opened in a tab here. The content is color-coded for easier reading (for instance, R code might be colored differently than plain text).

At the top of this panel, there are a few important buttons:



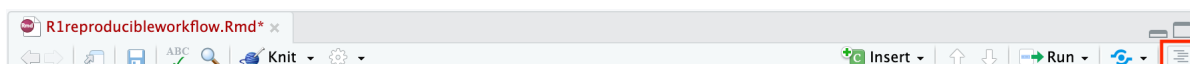
- The **Save** button (floppy disk icon) saves the current file. Remember to save frequently, especially before trying to run or knit your document.
- The **Knit** button (blue icon with a ball of yarn or a circle – it might look like a ball of yarn or a cogwheel depending on your RStudio version) is used to compile your R Markdown document. We often call this “knitting” the document. Pressing **Knit** will tell RStudio to execute all the code in your R Markdown file and produce a formatted output document (by default, an HTML file).



When you click **Knit**, RStudio will ask you to save the file (if you haven’t already) and then it will run through the document. If all goes well, a preview of your compiled document will appear (often in a pop-up window or in a viewer panel). For an HTML output, RStudio will either show it in its Viewer or open it in your web browser. The compiled document will include all your text, as well as the results of any code (e.g., charts, tables) inserted in the appropriate places.



Another useful button in the editor toolbar is the **Document Outline** (it looks like a split rectangle or a bullet list icon). Clicking this toggles an outline view of your document (usually appearing in a pane on the left of the editor). This outline is basically a table of contents for your document, listing all the headings and subheadings. This is very handy for navigation as your documents grow longer – you can click on a section title in the outline to jump directly to that part of the document in the editor.




### 3.1.2.2 Bottom Left Panel: Console and Terminal

The **bottom left panel** is primarily the **Console**. This is where R commands are executed and results printed. When you click Knit to compile your document, you will see RStudio working through each step in the Console, and any messages, output, or errors will appear here. You can also type commands directly into the console prompt `>` to execute R (or Python, etc., if configured) commands interactively.

In the context of knitting an R Markdown document, the console will show you progress and any errors/warnings. For example, if your code has a mistake that causes the document to fail to compile, a red error message will appear in this panel, helping you diagnose the problem. If the document compiles successfully, the console will list the files that were created (for instance, it might say something like “Output created: first\_document.html”).

The bottom left panel may have multiple tabs aside from the Console. By default, you might also see a **Terminal** tab (which gives you access to a shell command line, if needed) and a **Jobs** tab (for background tasks). For most of our needs, you will be looking at the Console tab in this panel.



```
Console | Terminal x | R Markdown x | Jobs x
.../rnanacourse/R1reproducibleworkflow.Rmd
highlight --template /usr/local/lib/R/site-library/distill/rmarkdown/templates/distill_article/resources/default.html --lua-filter /usr/local/lib/R/site-library/distill/rmarkdown/templates/distill_article/resources/distill.lua '--metadata=link-citations:true' --include-in-header /tmp/RtmpaKznXu/file310521897077html --include-in-header /tmp/RtmpaKznXu/file3105242b3fhtml --include-in-header /tmp/RtmpaKznXu/file310570655998html --include-in-header /tmp/RtmpaKznXu/file310535279b92html --include-in-header /tmp/RtmpaKznXu/file31054dfe9dbfhtml --include-before-body /tmp/RtmpaKznXu/file31056f84887bhtml --include-before-body /tmp/RtmpaKznXu/file310511f642d9html --include-before-body /tmp/RtmpaKznXu/file310542babfc7html --include-after-body /tmp/RtmpaKznXu/file310535f54ac6html --include-after-body /tmp/RtmpaKznXu/file310562f42490html --include-after-body /tmp/RtmpaKznXu/file31057a0d2bf4html --include-in-header /tmp/RtmpaKznXu/rmarkdown-str310545a020d1.html --mathjax --variable 'mathjax-url:https://mathjax.rstudio.com/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML' --include-in-header /tmp/RtmpaKznXu/file310569c44c10html
|.....| 100%
inline R code fragments

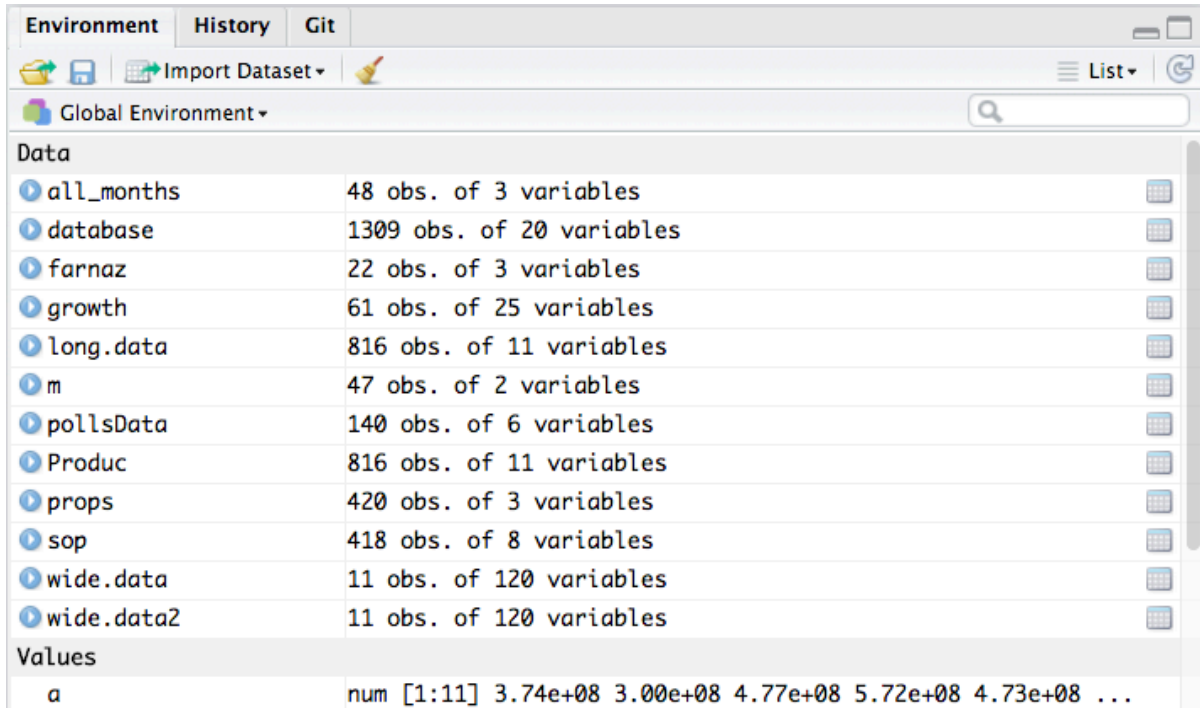
Output created: R1reproducibleworkflow.html
```

### 3.1.2.3 Top Right Panel: Environment and Git

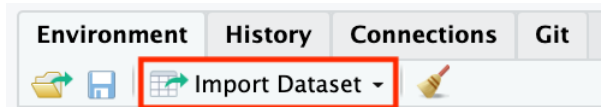
The **top right panel** serves a couple of purposes, primarily data and variable management, and version control. By default, it opens to the **Environment** tab.

- **Environment Tab:** This tab lists all the R objects (data frames, variables, functions, etc.) that are currently loaded in your R session. When you load a dataset or create a new variable in your code, you will see it show up here. It’s like your workspace browser. This gives you a quick way to inspect what data you have in memory. For example, if you read a CSV file into a data frame called `salesData`, once that code runs, you’ll see an entry for `salesData` in the Environment tab along with some details like its type and dimensions. You can click on a data frame in the environment to open it in a spreadsheet-like view for a quick look at the data.

The Environment tab also has an **Import Dataset** button. This provides a GUI wizard to load data (from text files, Excel sheets, etc.) into R without writing code manually. Clicking this will help you browse for a file and set options (like whether the first row is a header, what the field separator is, etc.) and then import it, generating the corresponding R code for you. While we will teach you how to read data with code (which is more reproducible), this tool can be convenient for quick tasks or for beginners.



If you click **Import Dataset**, a dialog will appear where you can select the file and adjust settings. For example, you might choose a CSV file, specify that it's comma-separated, and indicate that the first row contains column names. RStudio will then read the data and show you a preview. When you confirm, it will load the data into your environment (and you'll see the data frame listed in the Environment tab).



Make sure to choose the appropriate settings when importing data: the delimiter (comma, semicolon, tab, etc.), whether the first row is a header (variable names) or actual data, how decimal points are represented, etc. Getting these right ensures your data is read correctly. (We will explore data importing and wrangling in detail in later chapters.)

Import Dataset

Name

database

Encoding

Automatic

Heading

☒ Yes ☐ No

Row names

Use first column

Separator

Semicolon

Decimal

Period

Quote

Double quote (")

Comment

None

na.strings

NA

☒ Strings as factors

Input File

id;day\_code;date;ticker;name;ticker\_jouravant;name\_jouravan  
14;1;41030;171;100;;;55.700325732899;27.3224043715847;;;;;  
14;2;41031;142;110;171;100;46.2540716612378;30.054644808743  
14;3;41032;145;148;142;110;47.2312703583062;40.437158469945  
14;4;41033;184;92;145;148;59.9348534201954;25.1366120218579  
14;5;41036;188;128;117;90;61.2377850162866;34.9726775956284  
14;6;41037;171;100;188;128;55.700325732899;27.3224043715847  
14;7;41038;132;53;171;100;42.9967426710098;14.4808743169399  
14;8;41039;169;143;132;53;55.0488599348534;39.0710382513661  
14;9;41040;169;95;169;143;55.0488599348534;25.9562841530055  
14;10;41043;;;;;-0.0147;0;0;0;0.000771803;-0.0053;-0.  
14;11;41044;;;;;0.0517;1;0;0;1;0;-0.00977756;-0.0147;0.00  
14;12;41045;;;;;0.0039;1;1;0;0;0;-0.022375215;0.0517;-0.0

Data Frame

id	day_code	date	ticker	name	ticker_jouravant	name_jouravan
14	1	41030	171	100	NA	NA
14	2	41031	142	110	171	100
14	3	41032	145	148	142	110
14	4	41033	184	92	145	148
14	5	41036	188	128	117	90
14	6	41037	171	100	188	128
14	7	41038	132	53	171	100
14	8	41039	169	143	132	53
14	9	41040	169	95	169	143
14	10	41043	NA	NA	NA	NA
14	11	41044	NA	NA	NA	NA
14	12	41045	NA	NA	NA	NA

Import

Cancel

After importing, if you click on the name of the dataset in the Environment tab, RStudio will open a viewer (usually in the top left panel) showing the first portion (up to 1,000 rows) of the data in a spreadsheet-like format.



labSession1.Rmd\* x

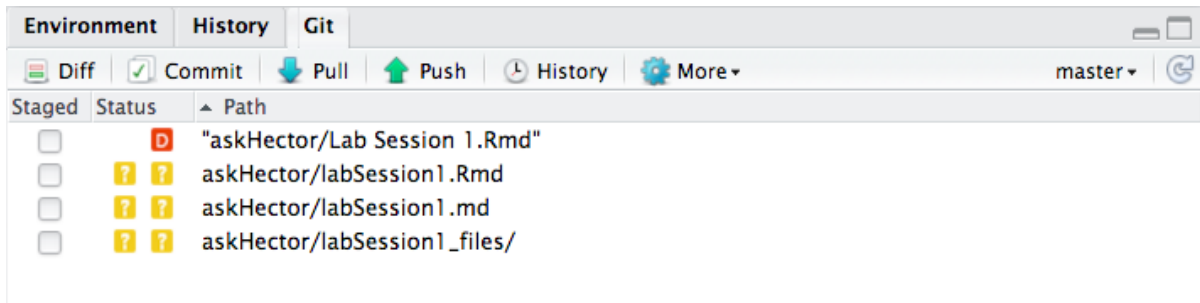
pollsData x

Filter

	date	leave	remain	undecided	source	method
1	16-06-22	45	45	10	YouGov	online
2	16-06-22	45	44	11	Opinium	online
3	16-06-22	43	41	16	TNS	online
4	16-06-22	42	48	11	ComRes	phone
5	16-06-20	44	45	11	Survation	phone
6	16-06-19	44	42	13	YouGov	online
7	16-06-19	47	53	2	ORB	phone
8	16-06-18	42	45	13	Survation	phone
9	16-06-17	43	44	13	YouGov	online
10	16-06-17	44	44	12	Opinium	online
11	16-06-16	44	42	14	YouGov	online
12	16-06-15	45	42	13	Survation	phone
13	16-06-15	43	46	11	BMG	phone
14	16-06-15	51	41	9	BMG	online
15	16-06-14	49	43	8	Ipsos Mori	phone
16	16-06-13	45	46	9	ComRes	phone
17	16-06-13	49	44	7	ICM	online
18	16-06-13	50	45	6	ICM	phone
19	16-06-13	46	39	15	YouGov	online

- **Git Tab:** Next to the Environment tab, you may see a **Git** tab (this will appear if your project is initialized as a Git repository; we will do this soon when we discuss Git). The Git tab will show version control information: which files have been modified, which are staged to commit, etc. From this tab, you can perform Git operations like commit, push, and pull through the RStudio interface. We will cover the specifics in the Git section of

this chapter. For now, just note that this is where Git-related info lives in the RStudio interface.



There may also be a **History** tab in this panel, which logs the commands you have executed in the console.

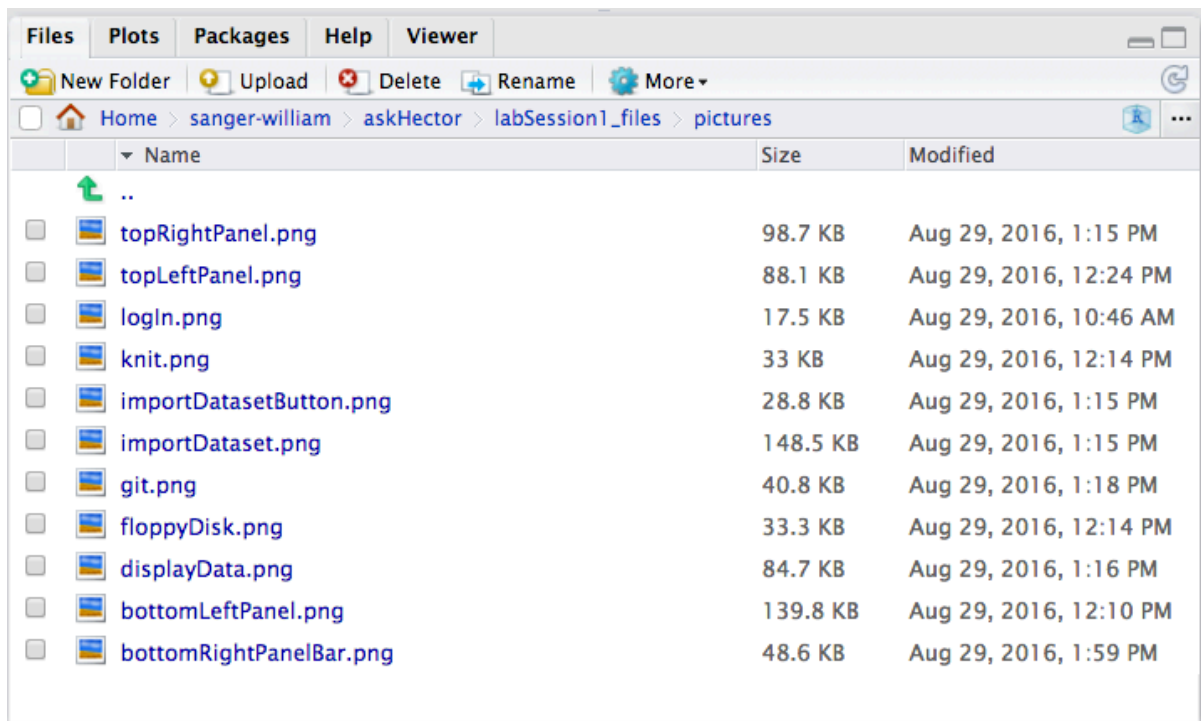
### 3.1.2.4 Bottom Right Panel: Files, Plots, Packages, Help, and Viewer

Finally, the **bottom right panel** is a multipurpose area. By default, it opens to the **Files** tab, which shows the files and folders in your project's working directory (essentially a file browser, similar to Finder on Mac or Explorer on Windows, but limited to your RStudio project directory).

- **Files Tab:** This is where you can see all files associated with your project. You can navigate through sub-folders, and use the buttons provided to manage files. For instance, you can create a New Folder, upload or export files, or delete files. If you select a file, you can use the **More** button (with a gear icon) for additional options like renaming or moving it. One important option here is “**Set As Working Directory**” which tells R (and RStudio) to treat that folder as the base location for relative file paths. By default, when you open an RStudio project, the working directory is set to the project directory, so you usually won't need to change it. (In our image examples, you might see a working directory path like `askHector`, which was an example project name – the notation `./` in front of paths indicates the current working directory.)

When working with R Markdown, it's good practice to keep your data and images within your project and refer to them with relative paths (like `./data/mydata.csv` or `./images/plot1.png`). The Files tab helps you figure out those paths and manage your project's content.

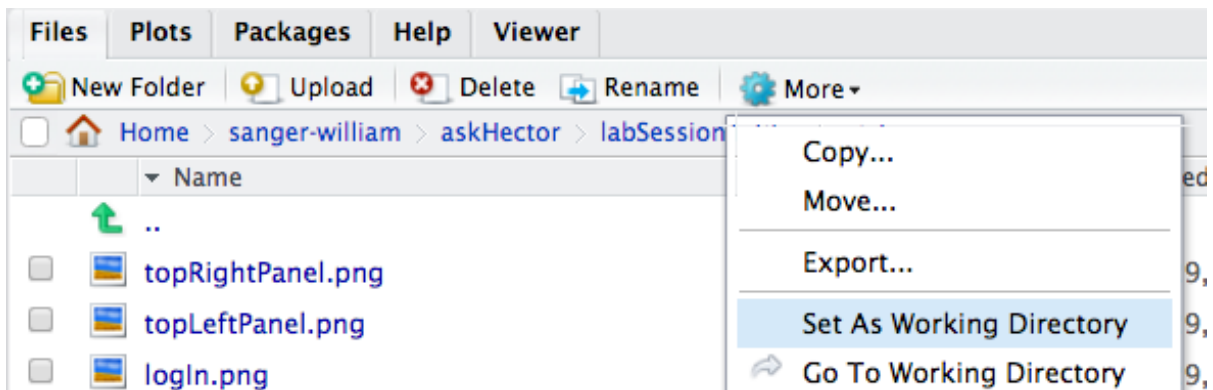
For example, if you have a subfolder called `images` and inside it an image `diagram.png`, the relative path might be `./images/diagram.png`. We use this path when embedding images in our R Markdown to tell RStudio where to find the file.



- **Plots Tab:** Whenever you generate a plot in R (for instance, by calling a plotting function in the console or in a code chunk and running it), it will appear under the **Plots** tab in this panel. You can navigate through previous plots using arrows, zoom into a plot, or export it (save as an image or PDF) from this tab.
- **Packages Tab:** This shows a list of R packages installed in your environment, with checkmarks for those that are currently loaded. You can install new packages or update packages using the buttons here, but often it's just as easy to use `install.packages("packagename")` in the console. Still, the Packages tab provides a quick way to attach a package (by checking its box, which runs `library(packagename)` for you) or see what version is installed.
- **Help Tab:** If you use R's help system (for example, `?mean` or `help(mean)` in the console to get documentation on the `mean` function), the documentation will appear in the Help tab. It's essentially a built-in web browser for R's help files and any other documentation you open.
- **Viewer Tab:** RStudio has an internal viewer for web content. When you create interactive plots (with packages like `plotly` or `leaflet`) or if you preview an HTML widget or a Shiny app, it might appear in the Viewer tab. Also, when you knit an R Markdown to HTML, by default RStudio might show it in this Viewer instead of your external web browser.

Think of the bottom right panel as your miscellaneous toolbox: file manager, plot viewer, package manager, help browser, etc., all in one.

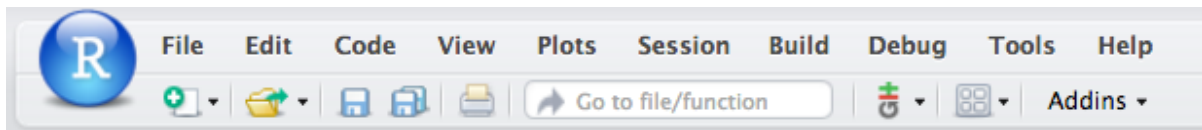
As an example of using the Files tab, consider the path example from above. We mentioned an image with path `./R1images/bottomRightPanel.png` in a description. That path indicates there is a folder named **R1images** in the current project, and inside it a file `bottomRightPanel.png`. Knowing how to read and use such paths is important when you link resources in your R Markdown (like including images or data files). The Files pane can help you verify those file names and paths.



In the Files pane, you have controls to manipulate files:

- **New Folder** button: create a new directory in your project.
- **Upload** (in RStudio Cloud or if enabled): bring files from your local system into the project.
- **More** (gear icon): contains options to Rename, Delete, or Export (download) selected files, and as mentioned, **Set As Working Directory** which changes R's reference point for relative paths.

Generally, you won't need to set the working directory manually if you stick to using RStudio Projects, because the project's main directory is automatically the working directory.



### 3.1.2.5 Primary Toolbar and Menus

At the very top of RStudio (above all panels) is the primary menu bar and toolbar. This includes menus like **File**, **Edit**, **Code**, **View**, **Plots**, **Session**, **Build**, **Git**, **Tools**, **Help** (these may vary slightly if using RStudio Desktop vs Cloud). Many of the functions accessible through buttons in the panels are also available via these menus.

Key items include:

- **File menu:** Create new files, open recent projects, save files, etc.
- **Edit menu:** Text editing functions (undo, copy, paste, find/replace, etc.).
- **Code menu:** Useful code editing shortcuts (comment/uncomment lines, reindent code, etc.).
- **Run menu (or on toolbar):** Buttons to run code from the source editor (like running the current line or selected code, which sends it to the Console).
- **View:** Options to zoom or rearrange panels.
- **Session:** Controls for your R session (restart R, interrupt running code, set working directory, etc.).
- **Git (if a Git repo):** Quick access to version control operations.
- **Tools:** Global options, addins, managing packages, etc.
- **Help:** Access documentation and diagnostics.

In addition to menus, the toolbar typically has icons for common actions (New file, Open file, Save, Knit, Run, etc.). We will explore certain toolbar features (like **Addins**) later in this chapter when we discuss the *cit* add-in for citations.

**To go further:** If you want to set up R and RStudio on your own computer (instead of or in addition to using RStudio Cloud), there is a tutorial available that walks through installing R, RStudio, and necessary packages. You can refer to [this guide](#) for detailed steps on local installation and configuration.

**TL;DR** – *RStudio Interface Overview*:

- **IDE (Integrated Development Environment):** A software application (like RStudio) that provides comprehensive facilities to programmers for software development, combining a source code editor, build automation tools, and more, in one GUI.
- **RStudio environment is split into four main panels:**
  - **Top Left (Source Editor):** Where you write your text and code (R scripts, R Markdown files, etc.). This is your main coding area with a text editor and action buttons like Save and Knit.
  - **Bottom Left (Console/Terminal):** Where code runs and output or error messages appear. You can also type commands here directly. It shows the log and progress when knitting documents.
  - **Top Right (Environment/History/Git):** Shows your data and variables in the Environment tab. Also includes the Git tab for version control (when using Git) and can show command history.
  - **Bottom Right (Files/Plots/Packages/Help/Viewer):** A multipurpose area for browsing project files, viewing plots, managing packages, reading help files, and previewing web content or reports.

By understanding what each panel does, you can efficiently navigate RStudio and make the most of its features while developing your data analysis projects.

Thanks! I'll retain the R Markdown material as-is, then add a new section introducing Quarto (.qmd), including its differences from R Markdown, advantages, and the ability to use other languages like Python or Julia in code chunks. I'll let you know as soon as that section is ready.

## 3.2 Writing Documents with R Markdown (and Quarto)

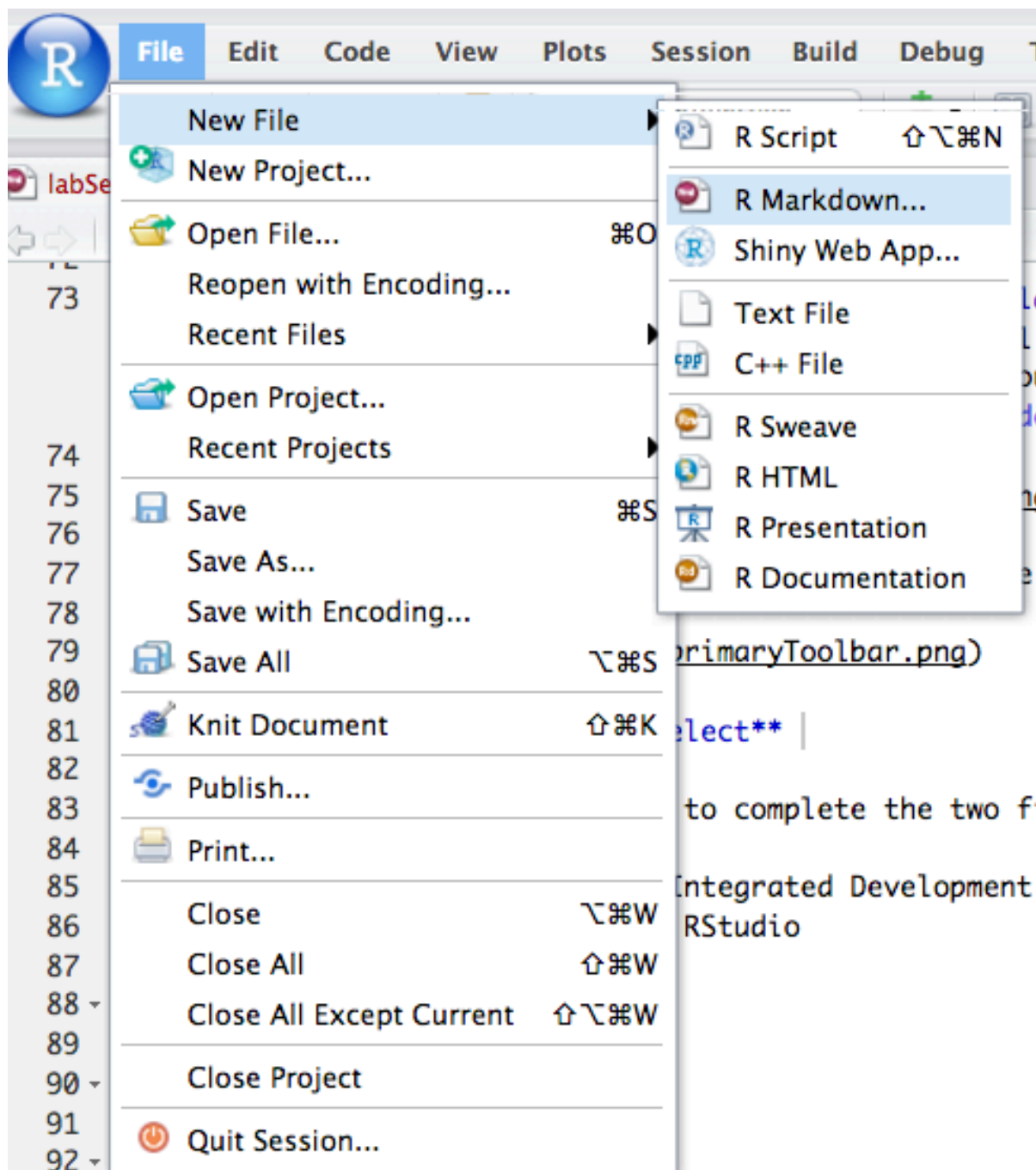
Now that you have set up the RStudio environment, let's focus on creating content using **R Markdown**. (We will also introduce **Quarto**, a newer system similar to R Markdown, later in this section.) R Markdown is one of the two main tools we will use throughout this book (the other being R itself). It allows you to combine regular text (formatted in a simple, readable way) with chunks of R code. When you *knit* an R Markdown document, the code is executed and its output is embedded in the final document, which can be rendered to various formats like HTML, PDF, Word, or even presentation slides.

Using R Markdown is central to the concept of **reproducible research**: your report is reproducible because anyone with your R Markdown file and data can re-run it to get the same results. If the data are updated or the analysis needs to change, you edit the code in one place (the R Markdown file) and knit again to produce an updated report. This is much more efficient and less error-prone than manually updating numbers or plots in a Word document.

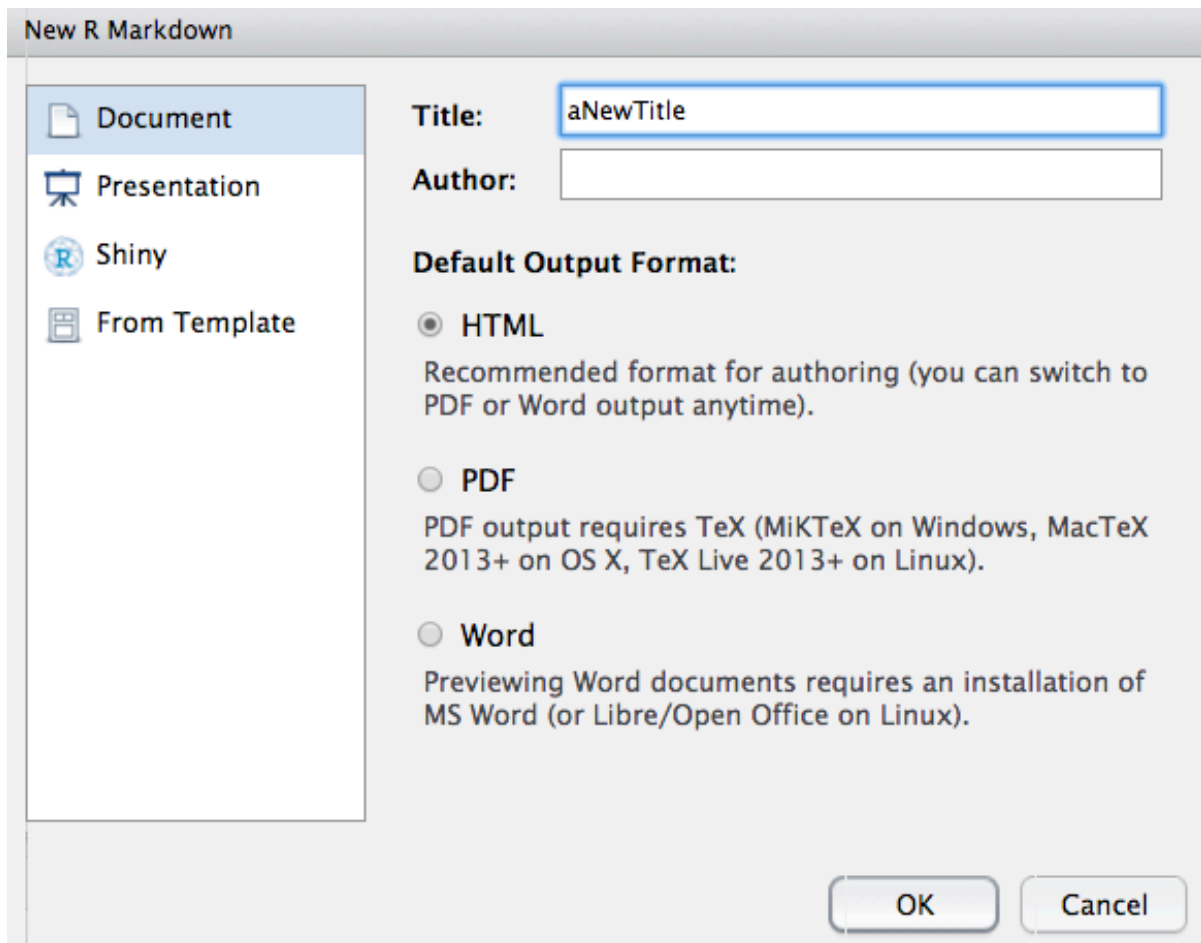
In this section, we will create a simple R Markdown document and learn the basics of the Markdown syntax and structure. *(If you followed the steps in the RStudio Cloud setup, you have already created an R Markdown file with a sample template. We will use that as a starting point. If not, here is how you can create a new R Markdown document in any RStudio session.)*

### 3.2.1 Creating a New R Markdown Document

To create a new R Markdown document in RStudio, use the menu: **File > New File > R Markdown....** This will open a dialog for specifying the title, author, and output format of your new document.



Choose **Document** (the default) as the type of R Markdown (as opposed to Presentation or other specialized formats) and ensure the default output is HTML. Enter a title and author if you'd like (you can change these later in the document's YAML header). Then click **OK**.



RStudio will create a new file (with some example content) and open it in the source editor. Don't forget to **save** this file (with a name ending in `.Rmd`). Once saved, you can proceed to edit the content.

At this point, if you have been following along, you should have achieved a couple of things already:

- Familiarized yourself with the RStudio IDE layout.
- Created (and saved) your first R Markdown document in RStudio.

Now we will dive into how to write in R Markdown.

### 3.2.2 The Structure of an R Markdown Document

An R Markdown document has three basic components:



1. The **YAML header** at the very top (optional but important for specifying document metadata and output options).
2. The **body** of the document, which includes your narrative text mixed with **code chunks** that execute R (or other languages') code.
3. (Possibly) a section for **references** at the end, if you are citing sources (we will cover citations later).

Let's go through these components and some key syntax elements of Markdown.

### 3.2.2.1 YAML Header

YAML stands for "YAML Ain't Markup Language" (a recursive acronym) – essentially, it's a human-readable format for specifying configuration. In an R Markdown file, the YAML header is the section at the very top enclosed by triple dashes `---` at the beginning and end. It provides metadata about the document and instructions for the output format.

Here's an example of a simple YAML header in an R Markdown (.Rmd) file:

```
---
title: "My First Report"
author: "Jane Doe"
date: "26/06/2025"
output: html_document
---
```

This YAML header specifies:

- **title:** The title of the document (appears at the top of the report).
- **author:** The author name (appears below the title in many formats).
- **date:** The date (or any text you want in the date field).
- **output:** The output format. Here `html_document` means we want to knit to an HTML file. Other common options include `pdf_document` for PDF output, `word_document` for a Word .docx file, `beamer_presentation` for a PDF slide deck, `ioslides_presentation` for HTML slides, and `github_document` for a Markdown output suitable for GitHub. You can change this depending on what final format you need.

You can add many other options in YAML. For example, `toc: true` will include a table of contents, `bibliography: references.bib` can specify a file for references (useful when you need to cite sources), or you can specify visual themes for slides and documents. In our simple documents, we'll often stick to just the basic fields shown above.

The YAML header is incredibly convenient because it lets you switch the entire output format or other settings without touching the main body of your document. For instance, if you wrote a report and later need it as a presentation, you can change `output: html_document`

to output: `ioslides_presentation` (and maybe tweak a few things in the content) and knit again to get slides. The same content can be output in different forms with minimal effort.

One thing to note: If you are used to word processors, you might be tempted to adjust the formatting of specific sections manually (like making one heading a different color or a specific line a larger font). In R Markdown (and Quarto), a lot of that fine-grained control is handled by the output format's template or by using custom CSS/LaTeX templates, rather than in the document content itself. The idea is that you focus on content, and let the template handle the style. This can be a mental adjustment: you give up a bit of *immediate* control over appearance in exchange for consistency and speed. The upside is huge – you won't spend time tweaking formatting on every update, and your documents will have a professional, consistent style. The downside is you may need to learn a bit about styling (CSS or LaTeX) if you want to customize beyond the provided templates, but for most cases you can find a template or default that looks good.

*(If you ever need to highly customize appearance, you can. But as a tip: content is king. It's often better to use the default styles unless you have a compelling reason – they are usually chosen by experts in design to be clean and readable.)*

### 3.2.2.2 R Code Chunks

The real magic of R Markdown is its ability to run code and insert the results into the final document. This is done with **code chunks**. A code chunk in R Markdown looks like this in the source:

Everything between the ````{r}` and the closing ````` is interpreted as R code (not as text for the report). When you knit the document, that R code will be executed. Anything the code prints to output (like the summary of a dataset, or a plot) will be captured and inserted into the document at that position.

Let's break down the example chunk above:

```
summary(cars)
```

Here, `cars` is a built-in dataset in R (a simple data frame of car speeds and stopping distances). `summary(cars)` will produce summary statistics for each column (min, max, mean, etc.). When you knit the document, those statistics will appear in the output at the position of the chunk.

If you have multiple lines of code in one chunk, they will all run in sequence. You can also include comments in your code (lines starting with `#`), which are ignored by R when running. Comments in code chunks will not appear in the output document (they're not text for the report, just notes to yourself in the source).

Another example chunk might be:

```
# Plot a scatterplot of the cars data
plot(cars)
```

This chunk, when run, will produce a scatter plot of the `cars` dataset (speed vs stopping distance), and that plot image will be inserted into the document.

By default, code chunks will also **echo** the code (show the code itself) in the output document, followed by the results or plot. However, you can control chunk behavior with **chunk options** inside the curly braces `{ }` that start the chunk. For example:

- `{r, echo=FALSE}` – runs the code *without* showing the code in the output (only the results will appear).
- `{r, include=FALSE}` – runs the code but includes neither the code nor the results in the output (useful if you need to set something up in code that the reader doesn’t need to see).
- `{r, eval=FALSE}` – shows the code in the output but does *not* actually execute it (useful for showing code examples without running them).

There are many such options (like controlling figure size, whether to cache results for speed, etc.), but we won’t overwhelm you with those now. The default behavior is usually fine for learning purposes.

In RStudio’s editor, you can also run individual chunks without knitting the whole document by clicking the little green triangle “play” button that appears to the right of a chunk, or by using the shortcut **Ctrl+Shift+Enter** (Windows) or **Cmd+Shift+Enter** (Mac) when your cursor is inside a chunk. This executes the chunk and shows you the output in the console or plot pane, which is handy for testing and iterative work.

### 3.2.2.3 Inline Code

Sometimes you want to embed a single value or a small result directly in your text, rather than showing it as a separate block. For example, you might want to write: “The average speed is 15.4 mph” where that number is calculated from data. Hard-coding such numbers in text is not reproducible (if the data updates, your text would be wrong unless you remember to update it too), but R Markdown allows **inline code** to solve this.

Inline R code is written using a single backtick ``` followed by `r` and the code, then another backtick. For example: `15.4` in your text will be replaced by the result of the R expression `mean(cars$speed)` when the document is knit.

So you could write in your R Markdown file:

```
The average speed of the cars dataset is 15.4 miles per hour.
```

When you knit, that will become something like:

*“The average speed of the cars dataset is 15.4 miles per hour.”* (assuming 15.4 is the actual mean of that dataset).

Inline code is extremely useful for embedding statistics or results in your narrative. It ensures consistency between what your analysis calculates and what you describe in the text, because the value is generated dynamically. If your data changes or you do a different analysis, the narrative text will update on the next knit.

One thing to note: inline code should be brief. It’s not meant for long computations or producing plots; it’s for a single number or a short piece of text. Also, by default inline results are inserted as plain text (if they’re numeric, they’ll be formatted as numbers). You can control formatting (like number of decimal places) using R functions or formatting options if needed (for example, using the `round()` function inside the inline code to round a number).

#### 3.2.2.4 Basic Markdown Syntax for Text

Now let’s cover how to format the text itself in your R Markdown document using Markdown syntax. Markdown is designed to be simple and readable as plain text, while allowing for basic formatting. Here are some common elements:

- **Headings:** Use the `#` symbol to denote headings. The number of `#` you use indicates the level of the heading. For example:
  - `#` **Heading 1** (usually the document title; if you use the YAML title, you typically don’t need to put a level-1 heading in the body).
  - `##` **Heading 2** (a major section heading).
  - `###` **Heading 3** (a subsection).
  - You can continue to smaller sub-sections with `####`, `#####`, etc., but rarely do you need more than 3-4 levels in a short report.

In the output, these will be formatted with decreasing font sizes or emphasis. They will also be used to build a table of contents if you enabled `toc: true` in the YAML.

- **Bold and Italic text:**
  - To make text **bold**, wrap it in double asterisks, like `**this**`, or in double underscores `__this__`.
  - To make text *italic*, wrap it in single asterisks `*this*` or single underscores `_this_`.
  - You can combine for ***bold and italic*** with triple asterisks/underscores, but that’s less common.

- **Lists:**

- **Unordered lists (bulleted lists):** Start a line with a dash `-` or an asterisk `*`. Indent by 2 spaces (or a tab) to create sub-lists. For example:

```
- First bullet item
- Second bullet item
  - Sub-item
  - Sub-item
- Third bullet item
```

will produce a bulleted list with a sub-list under the second item.

- **Ordered lists (numbered lists):** Start lines with `1.`, `2.`, etc. The numbers will automatically display in order when rendered (you can actually just put `1.` for each and Markdown will number them correctly).

```
1. Step one
2. Step two
3. Step three
```

will produce an ordered list of steps.

- You can mix lists with paragraphs or sublists, but be careful with indentation so that Markdown knows what is part of a list versus a new paragraph.
- **Links:** To insert a hyperlink, use the format `[link text] (URL)`. For example: `[RStudio website] (https://www.rstudio.com){target="_blank"}` will render as a clickable link: [RStudio website](https://www.rstudio.com). (The `{target="_blank"}` part is optional; it makes the link open in a new tab for HTML output.)
- **Images:** In Markdown, images are inserted similar to links but with an exclamation mark prefix. The syntax is `![] (path/to/image.png)`. You can include alternative text inside the brackets for accessibility (in case the image doesn't load). For example:

```
![RStudio logo] (./images/rstudio-logo.png){width=200}
```

will include the image at that path, scaled to a width of 200 pixels (the `{width=...}` is a way to suggest a display width; it's especially useful for LaTeX/PDF output for controlling image size). In our documents, we often provide images with a specific width for consistency.

In this chapter, we have an `images` folder in our project, and we reference images by their relative path. This keeps the document self-contained (as long as the images folder accompanies the document).

- **Code formatting in text:** If you want to refer to a piece of code or a filename in your narrative (without executing it), you can format it as code by wrapping it in single backticks. For example, “Use the function `mean()` to calculate the average” or “Open the file `analysis.Rmd` for editing.” This will display the text in a monospaced font and distinguish it as code.
- **Blockquotes:** If you want to quote a passage or provide a highlighted note, start the line with `>`. For example:

```
> Data science is the art of turning data into actions.
```

will appear as an indented block quote. (We won’t use blockquotes often for our purposes, but it’s good to know.)

These are the most frequently used Markdown elements for basic writing. There are others (like tables, which we’ll see later when discussing references and citations, or horizontal rules using `---` or `***`), but the above covers the essentials. You can always refer to the RStudio Markdown cheatsheet for a quick reference on syntax ([R Markdown Cheat Sheet \(PDF\)](#)).

### 3.2.2.5 Adding Mathematical Notation

If you need to include mathematical equations or symbols, R Markdown (via Pandoc’s Markdown) allows you to use LaTeX math notation:

- For **inline math**, wrap the LaTeX in single dollar signs. Example:

```
The formula for the line is $y = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \epsilon$.
```

This will render inline, for example: *The formula for the line is  $y = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \epsilon$ .*

- For **display (centered) equations** that appear on their own line (typically numbered in scholarly writing), use double dollar signs on separate lines:

```
$$
y_i = \alpha_0 + \alpha_1 x_{i1} + \alpha_2 x_{i2} + \alpha_3 x_{i3} + \epsilon_i
$$
```

This will display the equation centered on its own line:  $y_i = \alpha_0 + \alpha_1 x_{i1} + \alpha_2 x_{i2} + \alpha_3 x_{i3} + \epsilon_i$

You can include Greek letters (e.g., `\alpha` for  $\alpha$ ), subscripts with `_` (e.g., `x_{i1}` for  $x_{i1}$ ), superscripts with `^` (e.g., `x^2` for  $x^2$ ), fractions with `\frac{a}{b}`, summation symbols `\sum`, and so on. Essentially, anything you could write in LaTeX math mode can be done in R Markdown. This is extremely powerful for reports that include statistical or mathematical formulas.

### 3.2.2.6 Knitting the Document

Once you have some content in your R Markdown file – a combination of text, code chunks, etc. – you can generate the output by clicking the **Knit** button in RStudio. It's the button with a ball of yarn icon and a needle, at the top of the source pane. If you're not using RStudio, you can also knit by calling a function in R (e.g., `rmarkdown::render("your_file.Rmd")`), but in RStudio the button is easiest.

When you click **Knit**, RStudio will run through the document, executing each code chunk in a clean R session, capturing the output and plots, and then assembling everything into the final document (HTML, PDF, etc., depending on your YAML). If there are errors (for example, a chunk of R code has a mistake), the knitting process will stop and you'll see an error message in the R console. You would then fix the issue and knit again. If all goes well, you'll get an output document in your chosen format.

For example, if you knit the default sample R Markdown that RStudio created when you made a new file, you'll get an HTML page with a title, some introductory text, a plot of the **pressure** dataset, and a few other examples. RStudio will likely open this in its Viewer pane (or you can open the HTML file in a web browser).

Every time you knit, you're running a fresh execution of the document. That means it's **reproducible** – the output document doesn't rely on anything you did before by hand (you're not copy-pasting results; the code in the document always generates the results from scratch). It also means that if you change the data or update a parameter in your analysis, those changes will be reflected the next time you knit. This ensures your report is always in sync with your latest analysis or data.

By this point, you should be comfortable creating an R Markdown file, writing basic content with Markdown formatting, and including R code to perform computations or generate plots. You should also be able to click Knit and produce a nicely formatted output document that combines text and code results.

**Reproducible Research Note:** We emphasize reproducibility because it saves time and avoids errors. For instance, imagine using R Markdown for a business report: next month you get new data, so you update the data file and re-knit the report. All the tables and figures update automatically based on the new data. Contrast this with a manual process (like updating numbers in Excel and copying them into Word) – that approach is tedious and prone to mistakes. Embracing tools like R Markdown means your work is not only more efficient for yourself, but also easier to hand off to someone else or to revisit after time, since everything needed to regenerate the results is in one place (the script and the data).

As a real-world example of reproducible research and transparency, consider an initiative in academia: many journals encourage or require authors to publish replication materials. For example, there is a repository of academic articles in economics where each paper's data and code are made available so others can reproduce the findings. You can explore one such

repository here: [Economic Articles – Reproducible Research](#). This site lists economics papers that come with code and data, allowing anyone to rerun the analysis. It’s a good reminder of the growing expectation in many fields that results should be reproducible.

### TL;DR – *Key R Markdown Elements*:

- YAML header at the top (between --- lines) defines document title, author, date, and output format (among other options).
- Write narrative text in Markdown: use # for headings, **\*\*bold\*\*** or *\*italic\** for emphasis, - for bullet lists, 1. for numbered lists, etc.
- Include R code in **code chunks** like:  
This will run the code and include its output (or plot) in the document.
- Include **inline R code** with ``r`` to insert computed values into your text.
- Embed an image with Markdown syntax, e.g. `` (optionally with size attributes like `{width=400}`).
- Write equations with LaTeX syntax:  $E = mc^2$  for inline or  $E = mc^2$  for a standalone formula.
- Click **Knit** to compile the document and see the output. If using a different environment, run the render function to produce the output file.

Some R functions we used in the examples above:

- `summary(object)` – produces summary statistics of a dataset or R object.
- `plot(object)` – produces a plot (the output depends on the type of object; for a data frame like `cars`, it will make a quick scatter plot).
- `mean(vector)` – calculates the mean of a numeric vector.

### 3.2.3 Quarto (.qmd): A Modern Alternative to R Markdown

By the way, there’s a newer tool called **Quarto** that you might hear about. Quarto (with file extension `.qmd`) is essentially the next generation of R Markdown, introduced by Posit (formerly RStudio) in 2022. It extends the ideas of R Markdown to be **multi-language and multi-engine**. In practical terms, a Quarto document looks and feels very much like an R Markdown document: you write Markdown text, you include code chunks, and you render the document to formats like HTML, PDF, Word, slides, etc. The big difference is that Quarto isn’t limited to R – it allows you to use Python, Julia, JavaScript, and other languages in your code chunks, in addition to (or instead of) R.

If you are only using R, Quarto will behave almost identically to R Markdown for your purposes. In fact, Quarto can render most existing `.Rmd` files without modification. But Quarto provides



a unified framework for doing more, and it's likely to be the focus of new features going forward. Here, we'll give a brief overview of Quarto and how it compares, so you're aware of it.

**Creating a New Quarto Document:** To create a Quarto document in RStudio (version 2022.07 or later), you can use the menu **File > New File > Quarto Document**. This is analogous to creating an R Markdown file. When you do this, you'll get a new file with extension `.qmd`. The RStudio IDE will recognize it as a Quarto file. The sample content of a new Quarto document is very similar to the R Markdown template, just with slight differences in the YAML and perhaps an example of a Python chunk. Don't forget to **save** the file (with a `.qmd` extension).

The RStudio IDE shows a **Render** button (often a blue circle icon with a white "Q") when you have a Quarto document open. This replaces the Knit button for `.qmd` files. Clicking **Render** will execute the Quarto workflow (which, behind the scenes, runs the code and uses Pandoc to create the output). The experience of writing and rendering is otherwise quite similar to R Markdown. You can also render Quarto documents from the command line using `quarto render mydocument.qmd` or from R by using the Quarto R package (`quarto::quarto_render("mydocument.qmd")`), but using the RStudio button is easiest when you're working interactively.

**Structure of a Quarto Document:** A `.qmd` file also has three main parts, much like R Markdown:

1. A YAML header at the top (between `---` lines) for title, author, output format, etc.
2. The body of the document with Markdown text and code chunks.
3. An optional references/bibliography section at the end (if you need to cite sources).

Everything you learned about writing Markdown text (headings, lists, bold/italic, links, images, etc.) and including mathematical notation applies equally to Quarto. The Markdown syntax is the same. The idea of **code chunks** is also the same, but with Quarto you are not restricted to R for those chunks.

**YAML in Quarto:** The YAML fields in a Quarto document are largely the same, but the **output format** is specified a bit differently. Instead of `output: html_document` (which was specific to the R Markdown system), Quarto uses `format` to specify output. For example, a simple Quarto YAML might be:

```
---
title: "My First Report (Quarto)"
author: "Jane Doe"
date: "26/06/2025"
format: html
---
```

This tells Quarto to produce an HTML document. If you wanted a PDF, you would use `format: pdf`. For a Word document, `format: docx`. Quarto has short, intuitive names for formats (it consolidates what was a variety of output functions in R Markdown into a single system). You can even list multiple formats if you want to render to, say, HTML *and* PDF at the same time (Quarto can target multiple outputs in one render, using a list under `format` field). For example:

```
---
title: "Report"
format:
  html: default
  pdf: default
---
```

would produce both HTML and PDF outputs with one command. But for now, you can stick to a single format like HTML.

All the other YAML options (title, author, date, toc, etc.) work similarly in Quarto. In fact, you'll often find that you can copy an R Markdown YAML header into a Quarto doc and just adjust the `output/format` field and it works.

**Code Chunks in Quarto:** Quarto's code chunks are very flexible. You still delineate a code chunk with triple backticks and curly braces, but instead of always `{r}`, you indicate the language. For example:

- An R chunk: ````${r}```` (just like before).
- A Python chunk: ````${python}````.
- A Julia chunk: ````${julia}````.
- Even an Observable JavaScript chunk: ````${ojs}```` (for advanced interactive JS, if needed).

Within one Quarto document, you can have some chunks in R and others in Python (or other languages). Quarto will automatically use the right engine to run each chunk. For R, it uses **knitr** (just as R Markdown does). For Python, Quarto can either use an embedded Python session or Jupyter behind the scenes. The result is that you can mix R and Python in one report seamlessly.

For example, here's a tiny Quarto document body with mixed languages:

When you render this Quarto document, the first chunk will execute in R (calculating the mean of 1 through 5, which is 3), and the second chunk will execute in Python (doing the same computation in Python). The results from each chunk will be inserted in the final document. In this case, both would output the value 3 (plus any printed output if present).

Just like in R Markdown, you can include **comments** in your code with `#` (for both R and Python), and you can control whether code is shown or hidden, executed or not, using chunk options. Quarto actually introduces a slightly different way to specify chunk options: you can

write them as lines starting with `#|` inside the chunk (this is a YAML-like syntax for options). For example:

In this Quarto R chunk, we set `echo: false` (so the R code won't be shown) and `fig-width: 6` (inches) for the resulting plot. This new `#|` syntax is optional; Quarto will also understand the old way (`{r, echo=FALSE}`) for compatibility. Use whichever style you prefer. The key point is that chunk options and their effects (hiding code, figure size, etc.) behave the same in Quarto as they do in R Markdown.

**Inline code** in Quarto is also supported and works the same way: use `4` or even ``python 2+2`` inside your text to evaluate expressions. By default, Quarto assumes an unlabeled inline code chunk is in R (since it originated in R Markdown), so to be safe when using other languages inline, you prefix the language like ``python``. In practice, most people use inline code for simple numerical results and typically with R, but it's good to know you have options.

**Rendering a Quarto document:** As mentioned, if you're using RStudio, click the **Render** button (which has a little Quarto icon on it) to render the `.qmd` file. The output (HTML or whatever format you chose) will be produced and opened in the Viewer or browser, just like with knitr. You can re-render anytime as you edit the document. RStudio even has an option to Render on Save, which will automatically update the output each time you save the file, giving you a live preview.

If you are not using RStudio, you can render Quarto from a terminal by navigating to your project folder and running `quarto render document.qmd`. This is one advantage of Quarto being a stand-alone tool: you don't need R to render a document (unless it contains R code, of course, in which case R needs to be installed, but you wouldn't need the RStudio IDE or even the R `rmarkdown` package). Quarto can also be used in other editors like VS Code, etc., but we'll stick with RStudio here.

**Reproducibility and Quarto:** The philosophy of Quarto is the same as R Markdown in terms of reproducibility. A Quarto document is a plain text, source-of-truth for your analysis. Anyone with your `.qmd` file and data can rerun it (provided they have the required software like R or Python available) to get the same results. Quarto was created to broaden this capability beyond R. So if down the line you collaborate with someone who uses Python, you could integrate your work in one Quarto report, with some chunks in R and some in Python, instead of juggling separate documents.

Quarto also consolidates many extensions of R Markdown (like bookdown for books, revealjs/xaringan for slides, etc.) into one system. So you can create books or slides by just changing the format in YAML or using Quarto project configurations, without needing additional R packages. This won't matter for us until later (if at all), but it's nice to know that Quarto can handle larger projects too (websites, blogs, books, presentations, dashboards, etc.) in a unified way.

**Should you use Quarto or R Markdown?** If you are comfortable with R Markdown and just starting out, you can continue with R Markdown for now – everything you learn will be

directly applicable to Quarto. If you're feeling adventurous or are already familiar with these concepts, you might try using Quarto for your new documents. The learning curve is about the same, and the content we cover (writing text, using code chunks, etc.) applies equally. The main differences you'd encounter are the slight change in YAML (using `format:`) and the use of the Render button instead of Knit. Both R Markdown and Quarto are actively supported (Posit has stated that R Markdown will continue to work and be maintained, and you can choose either system). For this book, we'll primarily demonstrate with R Markdown (.Rmd) since it's what RStudio Cloud had set up initially, but feel free to use Quarto (.qmd) if you prefer – the steps and results should be nearly identical.

If you want to learn more about Quarto, the official website [quarto.org](https://quarto.org) has excellent documentation and tutorials. As Quarto is relatively new, keep an eye on its development and community examples. It's quickly becoming a standard for reproducible reporting in data science, just as R Markdown has been.

### 3.2.4 Getting Your Hands Dirty: Writing an R Markdown Report

The best way to become comfortable with Markdown (and the R Markdown workflow) is to practice. As an exercise, try to reproduce a given report using R Markdown. We have a sample report available here: [Sample Markdown Report](#). Your task is to create an R Markdown file that generates a report exactly like that sample.

Some tips for this exercise:

- Pay attention to the **headings** levels and formatting in the sample report (it contains multiple sections and maybe sub-sections). Make sure your document's headings (`#`, `##`, `###`, etc.) match the structure.
- Reproduce the **bold/italic text**, **lists**, and any other formatting exactly as shown.
- Don't forget to add any **images** that are in the sample. (You can find the images in the `chapter2` folder of the [GitHub repository](#) for this book's materials.) Use the correct relative paths to include them in your document (e.g., `` with appropriate width if needed).
- Include the HTML link as shown in the sample (practice writing a hyperlink in Markdown with the correct text).
- Essentially, your output should be as close to pixel-perfect as possible compared to the sample.

This exercise will test your understanding of Markdown syntax and the use of RStudio to create a document. If you get stuck, refer back to this chapter or use the R Markdown Cheat Sheet mentioned earlier for quick reference.

Once you're done writing your R Markdown file, click **Knit** to produce the HTML output, and compare it to the sample report. If they match closely, congratulations! You've successfully written a report with R Markdown. You've learned how to format text, include code and

output, and generate a polished document. Keep this file—you’ll continue to build on these skills in subsequent chapters. (And if you’re curious, you could also try doing the same with a Quarto document for practice, but that’s optional.)

### 3.3 Using Git and GitHub for Version Control

Now that you have the basics of creating content with RStudio and R Markdown, it’s time to address an important aspect of professional and academic work: **version control** and collaboration using Git.

Think of Git as a “save game” system for your project, but much more powerful. With Git, every time you reach a milestone or make a set of changes, you can save a version (commit) of your work. You can later review what changed, revert to a previous version if something breaks, or branch off to try an alternative approach without losing your original work. Moreover, when multiple people are collaborating, Git helps merge changes and ensures that nothing important is overwritten.

GitHub, GitLab, Bitbucket, and similar platforms host Git repositories online, enabling collaboration and off-site backup. In this book, we will primarily refer to GitHub, as it’s a popular choice, but the concepts apply generally.

Why bother with Git? A few key reasons:

- **Track Changes:** You can always see what was changed, when, and by whom. This is like “Track Changes” in Word, but for code and with a permanent history.
- **Collaboration:** Teams can work on the same project simultaneously. Git will help integrate their contributions.
- **Backup:** Your work is stored in the cloud repository, so even if your computer fails, your code (and possibly data, if included) is safe.
- **Reproducibility:** You can tag specific versions of your analysis (for example, the code as it was when you submitted a report or published a paper). Later, you or others can retrieve that exact version to reproduce the results.

In the context of our class or book, we will also use GitHub as a means for instructors and students to share materials. It’s an essential skill in modern data science workflows.

#### 3.3.1 Setting Up GitHub and a New Repository

Let’s start by creating a GitHub account and a repository for your project:

1. **Create a GitHub account:** If you don't already have one, go to [GitHub.com](https://github.com) and sign up for a free account. Choose a username (this will be part of the URL for anything you share, so pick something professional). Confirm your email, etc., as prompted by GitHub.
2. **Create a new repository on GitHub:** Once logged in to GitHub, look for a “+” icon at the top right and select “**New repository**”, or click the “**New**” button on your profile's Repositories tab. We'll make a test repository:
  - **Repository name:** You can name it anything, e.g., `myrepo` or `dpr-project` (avoid spaces, and case doesn't matter but typically we use lowercase for repo names).
  - **Description:** (Optional) e.g., “Testing my setup” or “Data Pipeline Project repository”.
  - **Privacy:** Choose **Public** (since it's a test and for learning; you can make it private if you want only you and invited collaborators to see it, but public is fine for now).
  - **Initialize with a README: Yes.** Check the box for “Add a README file”. This will create a default README that you can edit later. Initializing with a README is convenient because it also initializes the repository with a main branch.
  - You can skip adding .gitignore or license for this test.
  - Click “**Create repository**”.

Congratulations, you now have a repository on GitHub. On the repository page, you should see your README file and some info.

3. **Get the repository URL:** On GitHub, there will be a green “**Code**” button (usually top right on the repo page). Click it and ensure “HTTPS” is selected (not SSH, for simplicity). You'll see a URL like `https://github.com/YourUsername/myrepo.git`. Copy that URL – we will need it to connect RStudio to this GitHub repo.

### 3.3.2 Connecting your RStudio Project to GitHub

Now that you have a remote repository on GitHub, let's connect your RStudio project to it. RStudio has built-in support for Git, which makes this relatively easy:

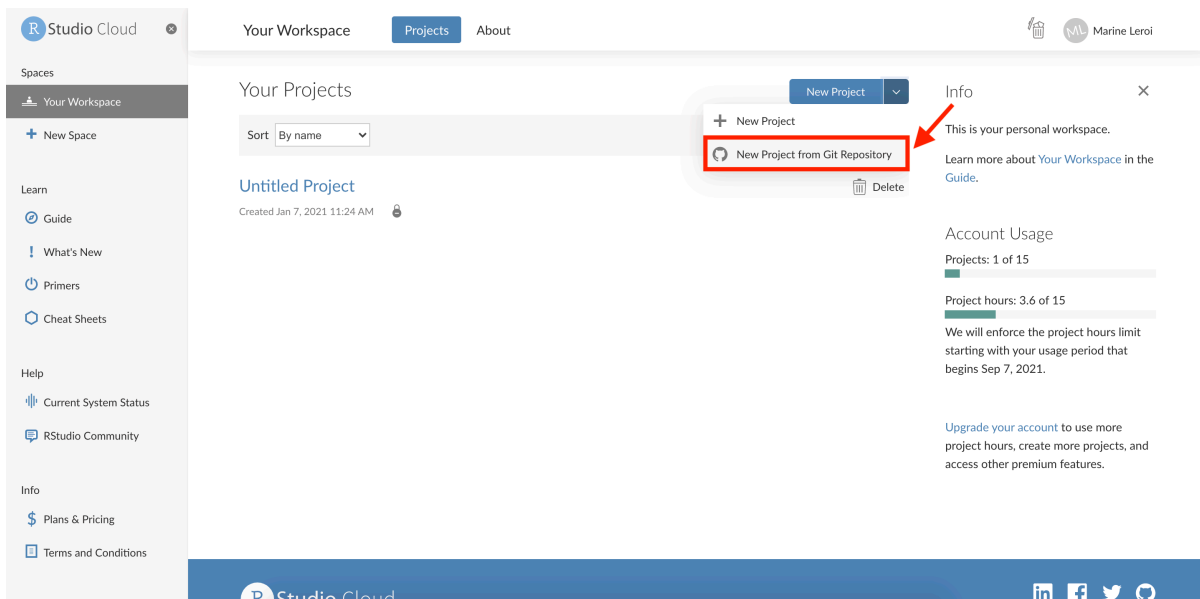
If you created a project earlier without Git and want to link it, a straightforward way is to create a new project from Git (which essentially clones the GitHub repository into RStudio). For learning, we can do this:

- In RStudio (Cloud or Desktop), go to **Project** menu (if you have a project open, it might show as the project name in the top-right, click that and choose “New Project”, or from the Start page choose New Project).
- Choose “**Version Control**”, then “**Git**” as the type of new project.
- It will prompt for a repository URL. Paste the URL of the GitHub repo you copied (e.g., `https://github.com/YourUsername/myrepo.git`).

- It will also ask for a **Directory name** – by default this will fill in the repo name. You can leave it as is (myrepo).
- Choose where to create this project directory on your computer or cloud space. (On RStudio Cloud, it will just create it in your workspace; on Desktop, you'd choose a location on your disk).
- Click **“Create Project”**.

What happens now is RStudio will **clone** the repository from GitHub into a new project. Cloning means it brings a copy of the repository (all files and the full Git history) to your local environment. Since the repo currently only has a README, you'll soon see that in your Files pane.

If you do this on RStudio Cloud, it might look like:



After creation, in the top-right of RStudio, you should see that Git tab we mentioned becomes active (because this project is now a Git repository). The README.md file should be listed in the Files pane.

Now your RStudio project is linked with GitHub. The next steps are to bring your existing work into this repo, and then learn the Pull/Commit/Push cycle.

**(If you started from scratch by cloning, you may not have your R Markdown file here yet. You can upload it or create a new one in this project and copy over content. Alternatively, you could have initiated Git in an existing project and connected to GitHub – but that’s a slightly more advanced workflow. For now, it’s fine to bring your work into this new Git project manually.)**

### 3.3.3 Using a Consistent Naming Convention for Files

Before we commit files to Git, one best practice: adopt a consistent naming scheme for your files. The book (and many developers) recommend **lower camel case** (also known as **lower-CamelCase**) for file names. This means:

- Use all lowercase for the beginning of the filename.
- If the name has multiple words, do not use spaces. Instead, concatenate the words and capitalize each subsequent word.
- For example, instead of naming a file “*Reproducible Document 1.Rmd*” (which has spaces and capital letters scattered), name it `reproducibleDocument1.Rmd`. Another example: `dataCleanup.R` or `salesAnalysis.Rmd`.

Why? Spaces in filenames can cause issues in URLs or require quoting in code. Using a consistent style like lowerCamelCase or snake\_case (words\_separated\_by\_underscores) avoids these problems. The key is no spaces, no special characters (stick to letters, numbers, underscores, hyphens, or camelCase caps), and make it readable. LowerCamelCase has the advantage that each word boundary is still clear (because of the capital), without needing an underscore.

So as you create files (scripts, data files, images, etc.), name them clearly and consistently. This will pay off when referring to them in code and when collaborating, because there’s no ambiguity or need for awkward quotes around file paths.

### 3.3.4 The Git Workflow: Pull, Commit, Push

With your project now under Git version control and linked to GitHub, let’s outline the basic workflow. The three main operations you will use constantly are:

- **Pull:** Get the latest changes from the remote GitHub repository down to your local project.
- **Commit:** Save (record) a snapshot of your changes in the local repository.
- **Push:** Send your committed changes up to the remote GitHub repository.

Think of it this way: Git is like a journal of changes. You write in the journal locally (commit) and later publish those changes to the world (push). Conversely, if others have written new entries (commits) to the shared journal on GitHub, you pull them to get up-to-date.

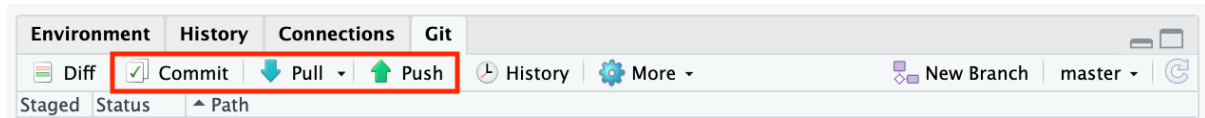
The mantra to remember, especially when collaborating or even if you work on multiple computers, is: **Pull first, then commit and push**. Always pull at the start of your session to make sure you have the latest version, then do your work, commit your changes, and push them back.

Let’s go step by step, using RStudio’s Git interface:

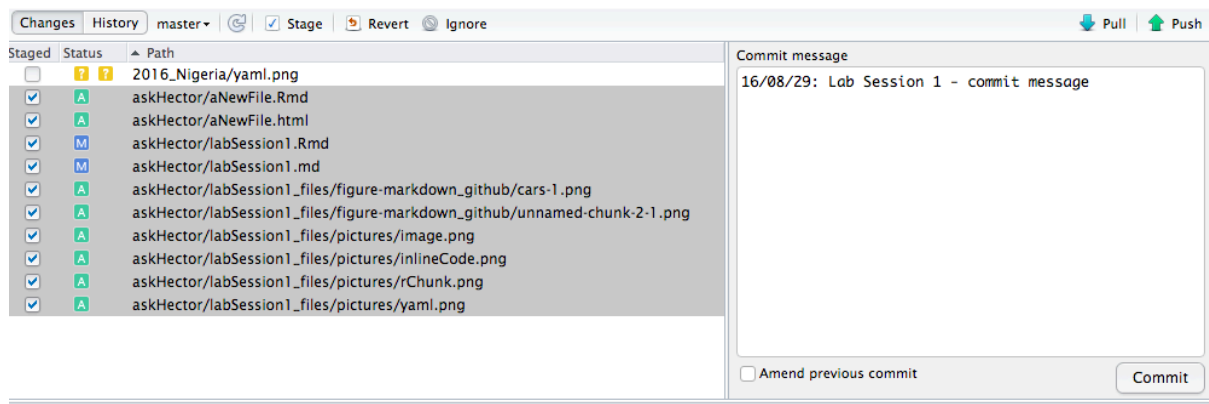


1. When you open your RStudio project (e.g., at the start of the day or when you know someone else may have pushed changes), click the **Pull** button. This fetches any changes from GitHub and merges them into your local files. If no one else changed anything, this might not bring any new changes, but it never hurts to pull. If there were changes (for example, maybe the instructor pushed a correction to an exercise or your teammate added data), those files will be updated in your project.

In RStudio's Git tab, the Pull button usually looks like a downward blue arrow.



2. After pulling (or if no changes needed pulling), you work on your files as usual: edit the R Markdown, maybe add some data or images to your project folder, etc.
3. Once you have made some progress and want to save a version, you will **commit** your changes. In the Git tab, you should see a list of files that have been modified, added, or deleted. New files will be marked with ? (untracked), modified ones with M, etc.
  - Click the checkboxes next to the files you want to include in this commit (or click the **Stage** button to stage selected files). Staging just means “prepare these files to be committed”. Typically, you stage all the files relevant to the change you are committing.
  - Once staged, those files will move into the “Staged” section in RStudio's Git interface.
  - Now click **Commit**. A new window will pop up showing the changes (it may show a diff – lines added/removed in each file).
  - Enter a **commit message** in the box. This is a short description of what you did, like “Drafted introduction and added images” or “Fixed typo in Markdown section” or “Added ggplot2 visualization of sales data”. A good commit message is clear and specific to the changes made.
  - Then confirm the commit. The changes are now recorded in your local repository's history. (At this point, it's not yet on GitHub – that requires a push.)



After committing, the files will no longer show as modified in the Git tab (until you edit them again).

4. Finally, **Push** your commit(s) to GitHub. Click the **Push** button (it's usually an upward arrow). If this is the first time pushing to this repository from your machine, you might be prompted for your GitHub credentials or a PAT (Personal Access Token) since GitHub requires authentication. Follow the prompts (you might need to use a token instead of your password if asked – GitHub has guidance on that, but RStudio might handle it via a one-time setup). On RStudio Cloud, the authentication might be managed automatically via your linked account.

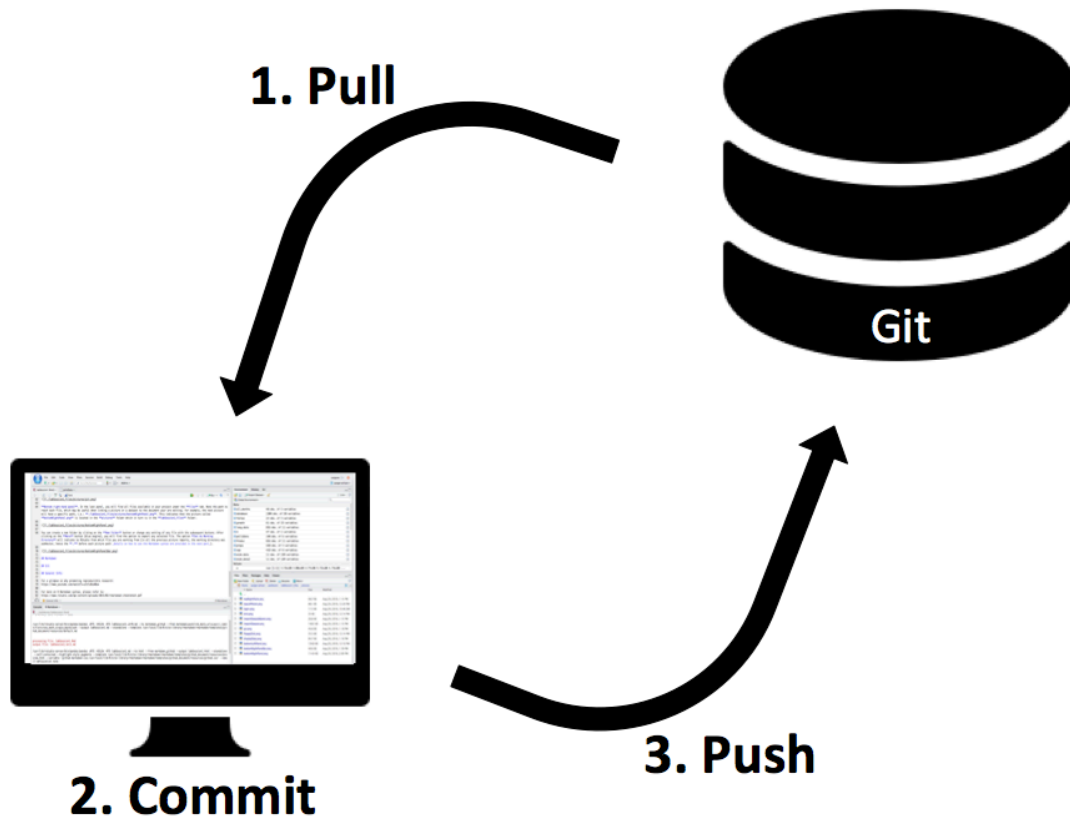
When you push, your commits are sent to the GitHub server and incorporated into the remote repository. If someone looks at the GitHub repo page now, they will see your new commits and updated files.

If at any point someone else pushed changes while you were working, your push might be rejected because your local is behind. In that case, you should Pull again (to merge their changes) and then push. Usually if you commit first and then try to push and it fails, doing a Pull will bring the other changes and often merge automatically (unless you both edited the same lines, causing a conflict). Git will notify you of conflicts if any, which you'd have to resolve manually (outside the scope of this intro, but basically you'd open the conflicted file, decide which version of lines to keep, then commit the resolved file).

For our use (student syncing with instructor's repository, for example), the typical pattern is:

- Always **Pull** when you start working or before you make big changes, to get any updates (e.g., maybe we provided a new dataset or corrected a typo in the starter code).
- Work on your tasks.
- **Commit** your changes locally with a message about what you did.
- **Push** to upload your work to GitHub (so the instructor can see it, or just to back it up for yourself).

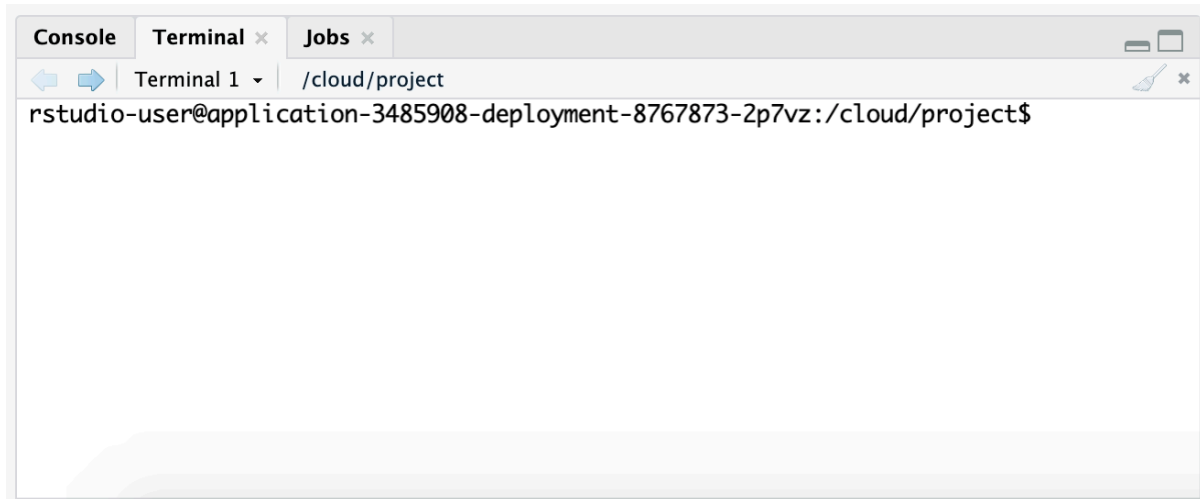
If you remember “**Pull > Commit > Push**” as a habit, you’ll avoid many common pitfalls like merge conflicts or accidentally diverging from the main repository.



One more thing: The first time you try to commit in a new Git repo on RStudio, you might get a message that your identity is not set. Git needs to know a name and email to associate with your commits (this can be anything, but typically you use the same email as your GitHub account and your actual name or alias). You set this up once:

**Git config (only needed once):** In RStudio, you can open the Terminal (there’s a Terminal tab next to Console, or use Tools -> Terminal -> New Terminal). At the \$ prompt, type the following (replace with your details):

```
git config --global user.email "your-email@example.com"
git config --global user.name "Your Name"
```



Hit Enter after each. This stores your name and email in Git’s global config so it will attach them to commits. (The email is what ties commits to your GitHub account if it matches, but even if not, it’s fine.)

After configuring, you can proceed to commit and push normally. You shouldn’t have to do this again on the same system.

To summarize the **Git workflow**:

- **Pull**: Download and integrate changes from GitHub to your local project.
- **Stage + Commit**: Select the changes you made and record them as a new version in your local repository with a message.
- **Push**: Upload your new commits to the GitHub repository so others (or your other devices) can see them.

This will ensure your work is versioned and backed up. No more “final\_report\_v7\_final\_FINAL.docx” files – Git will handle versioning seamlessly.

**TL;DR** – *Git and GitHub Basics*:

- **Git** is a version control system for tracking changes in your files, and **GitHub** is an online hosting service for Git repositories.
- Use a consistent naming scheme (like lowerCamelCase) for files to avoid issues and keep things tidy.
- The main commands in daily use:
  - **Pull**: always do this first to get the latest changes from the remote (GitHub).
  - **Commit**: save your changes locally with a descriptive message.
  - **Push**: send your committed changes to the remote repository.

- Remember the order **Pull** → **Commit** → **Push** whenever you start and finish a work session.

With your project now under version control, you can collaborate easily and have peace of mind that your work is safe and trackable. Next, we will ensure that even your references and citations in the report are handled in a reproducible way!

## 3.4 Managing References and Citations with Zotero

Finally, we come to a crucial aspect of report writing: citing sources and managing references. In academic or professional reports, you often need to refer to articles, papers, websites, or other sources. Keeping track of these manually can become tedious and error-prone, especially when formatting citations and bibliographies according to specific styles.

We will use **Zotero**, a popular free and open-source reference management tool, to handle our references. Zotero allows you to collect references (from academic papers, books, web pages, etc.), organize them, and then easily insert citations into your documents. Combined with an RStudio add-in called **citr** and the **BibTeX/BibLaTeX** system, this becomes a powerful, reproducible way to manage references.

Why manage references programmatically? Two big reasons:

1. **Efficiency:** Once you have a reference in Zotero, you can cite it in any document with a couple of clicks, and Zotero will handle the heavy lifting of creating and formatting the bibliography. If you need to switch citation styles (say from APA to Chicago), it's a matter of changing a setting, not retyping everything.
2. **Reproducibility:** By keeping a bibliography file (usually `.bib` for BibTeX) under version control with your project, anyone else with your project can compile your document and get the same references and citations. It also means you can regenerate the document at any time and have the citations update or remain consistent.

Let's walk through setting up Zotero and integrating it with R Markdown.

### 3.4.1 Zotero: Installation and Setup

If you haven't already, download and install Zotero from the official site: [Zotero Download](#). Zotero is available for Windows, Mac, and Linux. Install the application.

Also, install the **Zotero Connector** for your web browser (available on the same download page). This is an extension that allows you to quickly save references to Zotero as you browse (for example, when you're viewing a journal article, you can click the connector button and it will grab the citation info and even the PDF, if available, into your Zotero library).

Open Zotero on your computer. It has a left pane (collections and library organization), a middle pane (list of references), and a right pane (details of the selected reference). You might want to create a new **Collection** for this project or course (think of collections as folders or playlists of references). For example, make a collection called “DPR Project References”.

You can add references to Zotero manually, or via the connector while browsing. For now, just ensure Zotero is installed and running.

### 3.4.2 Better BibTeX for Zotero

Better BibTeX (often abbreviated as **BBT**) is a Zotero plugin that supercharges Zotero’s ability to work with LaTeX/BibTeX and by extension R Markdown. It offers features like stable citation keys (so that the keys used to cite items don’t randomly change), and an auto-export function that keeps a `.bib` file updated with your library or a specific collection.

Install Better BibTeX by downloading the latest release (`.xpi` file) from its GitHub releases page: [Better BibTeX latest release](#). Look for an `.xpi` file (Zotero plugins use this extension).

Once downloaded, in Zotero go to **Tools > Add-ons**. In the Add-ons Manager, click the gear icon and choose “**Install Add-on From File...**”, then select the `.xpi` file you downloaded. Confirm to install, and then restart Zotero when prompted.

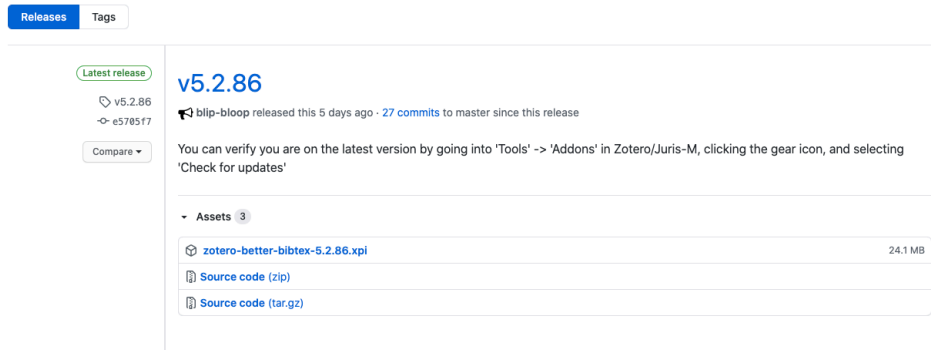
After restarting, Zotero will have Better BibTeX enabled. You can check its presence by going to **Edit > Preferences > Better BibTeX** (or sometimes in a separate tab in the Preferences). There you can configure things like citation key format. The default key format might be something like `[auth] [year]` which results in keys like `Wickham2016` for a reference by Wickham in 2016, for example. You can customize it, but default is fine to start.

One important thing to do now: set up auto-export of a BibTeX file. This will allow your R Markdown document to always pull citations from your Zotero library without manual exporting each time.

- In Zotero, go to **File > Export Library...** (or right-click a specific collection and choose Export).
- In the export dialog, choose **Better BibTeX** as the format.
- Check the box “**Keep updated**” if available (this is a BBT feature; it will keep the exported file updated as your library changes).
- Choose a location and name for the `.bib` file. For example, you might create a file called `references.bib` in your project folder (you can save it directly to your project directory on your computer). If you are on RStudio Cloud, a bit trickier – you might instead periodically export and upload the bib file, or use a synced storage like Dropbox. But let’s assume local for now.
- Now, whenever you add or modify references in Zotero, this `references.bib` will be automatically updated by BBT.

If “Keep updated” isn’t visible (it might be only when exporting a Collection vs the whole library, can vary by version), alternatively you can periodically re-export. But BBT usually has an option under Preferences -> Automatic Export where you can configure it.

Why all this? Because our R Markdown document will use that `.bib` file to retrieve citation info for rendering the bibliography.



*(The image above might show the GitHub release page where the .xpi can be downloaded.)*

### 3.4.3 Configuring the R Markdown YAML for Citations

Earlier we discussed YAML for title, author, etc. To enable citations, we need to add a couple of fields to the YAML:

- **bibliography:** – this should point to your `.bib` file (for example, **bibliography: references.bib**).
- **csl:** or **biblio-style:** – this is for specifying the citation style. There are two ways to handle style:
  - If using the default Pandoc citation processor (which is sufficient for most cases), you can specify a CSL (Citation Style Language) file. For example, you might have **csl: apa.csl** for APA style (you’d need to have that csl file downloaded). Alternatively,
  - If using BibLaTeX via the LaTeX engine (which happens when outputting to PDF with **citation\_package: biblatex**), you can use **biblio-style:** to name a BibLaTeX style.
- **citation\_package: biblatex** – if outputting to PDF, many recommend using the biblatex package for better Unicode support, etc. In our example, we will indeed use PDF as an example and BibLaTeX.

Basically, to cover all bases, you can include:

```
bibliography: references.bib
biblio-style: apa
output:
  pdf_document:
    citation_package: biblatex
```

This YAML snippet says:

- Use `references.bib` for citation data.
- When using `biblatex` (which we will for PDF), use the “`apa`” style (APA style citations). We could put another style name if desired (like `chicago-authordate` etc., if the LaTeX package is available).
- Use `biblatex` for handling citations in the PDF output.

For HTML output, the `biblio-style` might not do anything; instead, Pandoc will look for a CSL file if provided, or default to something like Chicago author-date. If you want a specific style in HTML, you would use a CSL. You could add `cs1: apa.csl` (after obtaining the appropriate CSL file from somewhere like Zotero’s style repository).

For simplicity, let’s assume APA style for now. If you don’t have the CSL, the above might still produce a default author-year style.

So your YAML might now look like:

```
---
title: "My Analysis Report"
author: "Jane Doe"
date: "2025-06-26"
output:
  pdf_document:
    citation_package: biblatex
bibliography: references.bib
biblio-style: apa
---
```

*(If you’re knitting to HTML, you could instead do `output: html_document` and maybe include a `cs1` line. But to keep things consistent, we can keep the YAML as above; it won’t break HTML output, it just might not use the `biblio-style` in that case.)*

This YAML setup ensures that when you knit your document:

- It knows to look into `references.bib` for any citation keys you use.
- It will format the bibliography at the end according to APA style (for PDF via `biblatex`).
- It will include in-text citations in APA format (e.g., parenthetical author-year).



### 3.4.4 Inserting Citations in R Markdown (with the citr Add-in)

Now comes the fun part: actually citing something in your text. You could do this manually by knowing the citation key for an item and typing it, but there's a handy tool to avoid leaving the RStudio environment: the **citr** add-in.

**Installing citr:** citr is an R package that provides an RStudio Add-in. To install it, we actually need the development version from GitHub (though CRAN might have a version; the instructions given in our material use GitHub). Let's follow those:

Open the RStudio Console (or a chunk) and run:

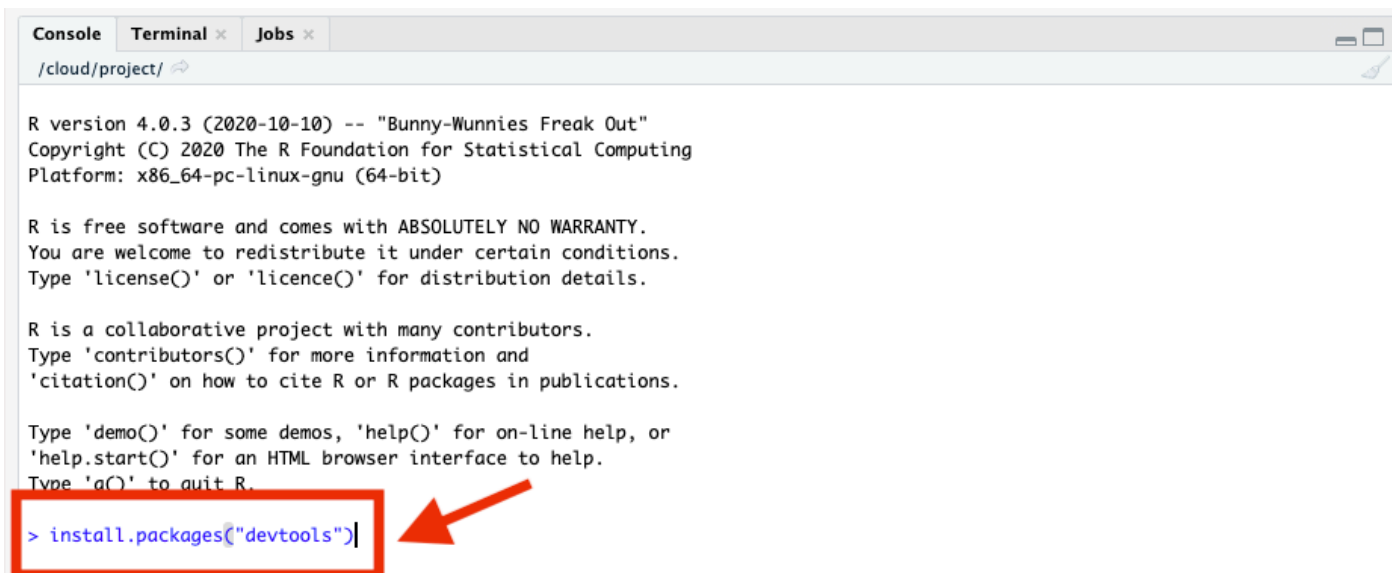
```
install.packages("devtools")
```

(This installs the devtools package if you don't have it, which lets you install packages from GitHub.)

Then run:

```
devtools::install_github("crsh/citr")
```

This will download and install the citr package from GitHub. It might ask you to update some packages; if you see a prompt like "These packages need updating, enter 1 for All, etc.", you can enter 1 to accept updating all needed packages.



```
Console Terminal x Jobs x
/cloud/project/
* installing *binary* package 'pkgload' ...
* DONE (pkgload)
* installing *binary* package 'rcmdcheck' ...
* DONE (rcmdcheck)
* installing *binary* package 'waldo' ...
* DONE (waldo)
* installing *binary* package 'usethis' ...
* DONE (usethis)
* installing *binary* package 'roxygen2' ...
* DONE (roxygen2)
* installing *binary* package 'testthat' ...
* DONE (testthat)
* installing *binary* package 'devtools' ...
* DONE (devtools)

The downloaded source packages are in
'/tmp/Rtmpgr9Ewu/downloaded_packages'
> devtools::install_github("crsh/citr")
```

If during installation you get a prompt in the console (often something about building vignettes or updating packages) like this:

- 1: All
- 2: CRAN packages only
- 3: None

Just type 1 and press enter to select All, as instructed:

```
Console Terminal x Jobs x
/cloud/project/
* installing *binary* package 'testthat' ...
* DONE (testthat)
* installing *binary* package 'devtools' ...
* DONE (devtools)

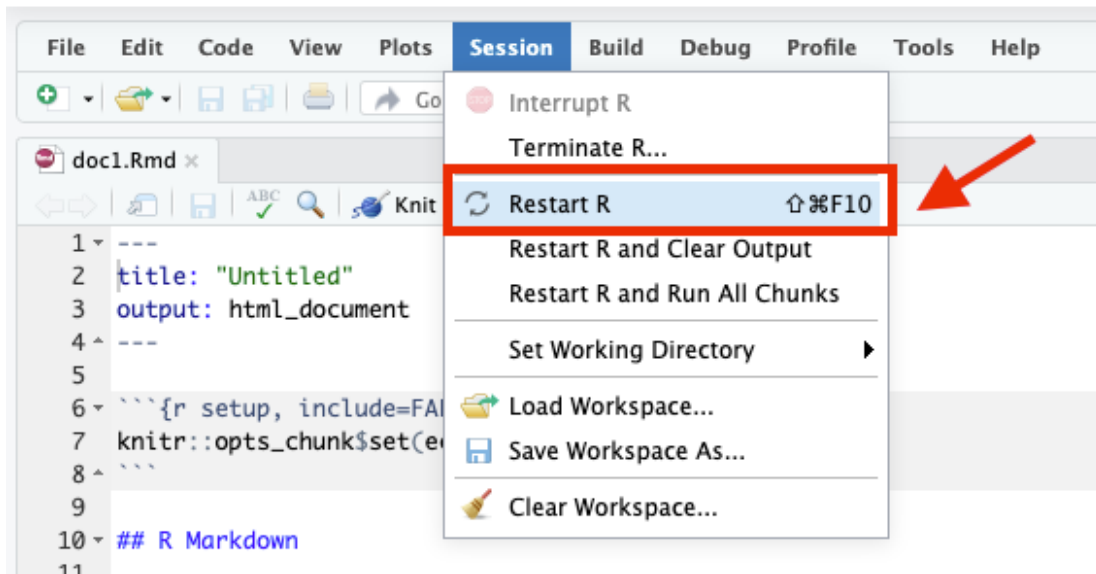
The downloaded source packages are in
'/tmp/Rtmpgr9Ewu/downloaded_packages'
> devtools::install_github("crsh/citr")
Downloading GitHub repo crsh/citr@v0.0.0

These packages have more recent versions available.
It is recommended to update all of them.
Which would you like to update?

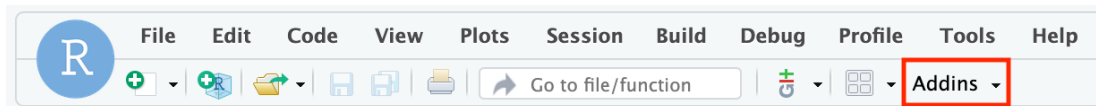
1: All
2: CRAN packages only
3: None
4: htmtltools (0.5.0 -> 0.5.1) [CRAN]

Enter one or more numbers, or an empty line to skip updates:1
```

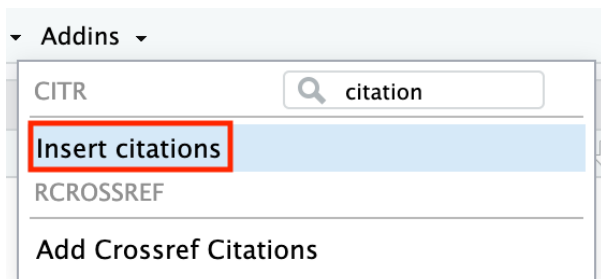
After installation, it's a good idea to restart your R session (because you installed new packages). In RStudio, go to **Session > Restart R** (or use the keyboard shortcut: Ctrl+Shift+F10). This ensures the new add-in is registered.



Now, in RStudio, check the **Addins** menu (it's typically at the top toolbar, to the left of Help, appearing when you have an R Markdown open or just in general). Click **Addins**, and find **Insert Citations** (you can use the search bar in the addins list to type “citation” and it should filter). This “Insert Citations” add-in is provided by citr.



When you click **Insert Citations**, a window should pop up. (Note: Zotero must be running, because citr will try to communicate with Zotero to get the references.) In this window, you should see the list of references from your Zotero library or at least those in the exported Bib file. There will be a search bar where you can type an author name, year, title, etc., to filter your references.

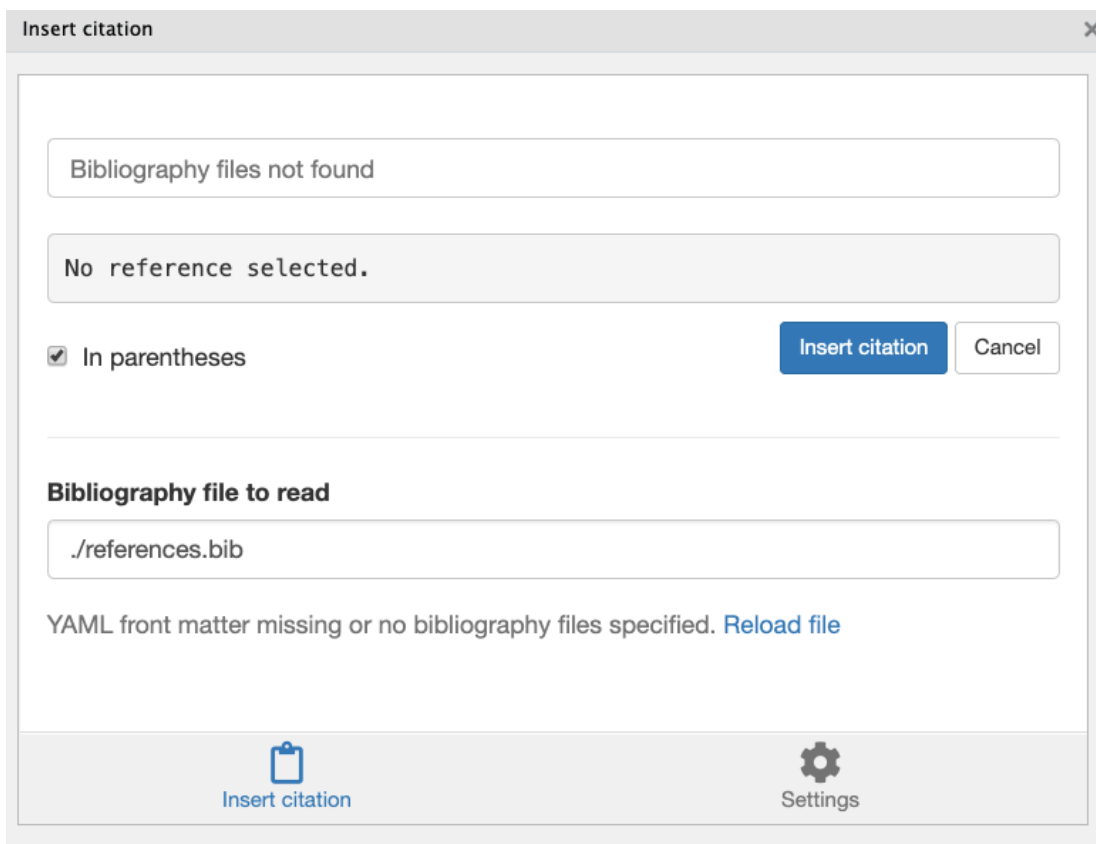


Select the reference(s) you want to cite (you can usually select multiple by holding Ctrl or Shift, or add them one by one). Once selected, you can press Enter or click “Insert”. The add-in will then insert citation keys into your R Markdown document at the cursor position.

The syntax that appears will be like `[@citationKey]` for a parenthetical citation or `@citationKey` for an in-text citation, depending on what you chose. By default, `citr` likely inserts `[@key]` (parenthetical). You can always edit for narrative citations.

For example, if you selected an item by James (1890) with citation key `james_1890`, the add-in might insert `[@james_1890]`. In your text, this will render as (James, 1890) in APA style. If you wanted “James (1890)” as part of the sentence, you would remove the brackets: just `@james_1890` in the Rmd will produce James (1890).

The add-in is nice because you don’t have to remember or look up the keys; you just search your library.



The pop-up window (image above) shows the references; you can likely double-click or use buttons to insert.

After inserting the citation in your R Markdown text, you’ll see the placeholder like `[(james_1890?)]`. If you now knit your document, it will replace that with a formatted

citation and add the full reference in a “References” or “Bibliography” section at the end of the document (usually automatically, unless you specify otherwise).

By default, the bibliography will appear under a heading like “References” at the end of the document.

If for some reason you are not using `citr` or can’t (maybe working on a machine where you can’t install it), you can still cite by typing the keys manually. The syntax to remember:

- `[@key]` for parenthetical (author, year).
- `@key` for in-text (Author (Year)).
- You can add page numbers or prefixes/suffixes, for example: `[@key, p. 123]` or `[@key, see Chapter 1]`. Also, if you want to suppress the author in parenthetical (say you already mentioned the author’s name in text and just want the year in parentheses), you prefix the key with a minus: `[-@key]` will give just (Year).

Here’s a quick reference of citation syntax (assuming your Bib has entries for James 1890 and Bem 2011 with keys as shown):

Citation type	Syntax	Rendered citation
Citation within parentheses	<code>[@james_1890]</code>	James, 1890)
Multiple citations	<code>[@james_1890; @bem_2011]</code>	Bem, 2011; James, 1890) <i>(order will be automatically sorted by citation style rules, often alphabetically by author)</i>
In-text citation (no parentheses around author name)	<code>@james_1890</code>	James (1890)
Year-only, for narrative citations where author is in text	<code>[-@bem_2011]</code>	(2011) <i>i.e., just the year in parentheses, assuming you wrote “Bem” in text already</i>

As mentioned, you can add a prefix or suffix within the square brackets: For example, `[@bem_2011, p. 5]` would yield (Bem, 2011, p. 5). Or, `[@bem_2011; @james_1890, Chapter 2]` yields (Bem, 2011; James, 1890, Chapter 2). Each `;` separates citations, and a comma after a key separates the citation from its suffix.

One thing to be cautious about: When multiple citations are inside one set of brackets, they will be sorted by the processor. So if you write `[e.g., @james_1890; @bem_2011]`, it may output as (e.g., James, 1890; Bem, 2011) or with authors sorted alphabetically after the text “e.g.,” stays tied to James (maybe not what you intended because you wanted e.g. to apply to both). There’s no easy fix to that except to word your sentences carefully or cite separately. Just a nuance to keep in mind: prefixes belong to the specific citation immediately following them within the brackets.

Now, after adding a few citations in your text, try knitting your document. If everything is set up correctly, you should see in-text citations and a reference list at the bottom. If you encounter any errors:

- Make sure the YAML is correct (especially indentation – YAML is picky about indentation).
- Make sure the .bib file path is correct. If the .bib file is not in the same directory as the .Rmd, adjust the path accordingly, e.g., `bibliography: refs/references.bib` if it's in a subfolder.
- Ensure Zotero is running and BBT is keeping the bib updated (or that you exported the bib after adding references).
- If using PDF output and you get an error about biber or something, ensure you have a TeX distribution installed with biber (TinyTeX for example if not already).

Once it works, you'll have fully automated citations in your report!

**TL;DR** – *Reference Management Recap*:

- Use **Zotero** to collect and manage references for your project. It's free and helps keep all citation info handy.
- Install the **Better BibTeX** plugin for Zotero to streamline integration with R Markdown (ensures stable citation keys and easy export).
- Configure your R Markdown document's YAML to point to a .bib file (`bibliography:`) and use a citation style (`biblio-style:` or `csl:` and possibly `citation_package: biblatex` for PDF).
- Install and use the **citr** add-in in RStudio to search and insert citations without leaving the editor.
- In citations: use `@citekey` for in-text and `[@citekey]` for parenthetical. Add multiple inside one set of brackets separated by semicolons for combined parentheses.
- The reference list will be generated automatically upon knitting, listing all sources you cited.

### 3.4.5 Getting Your Hands Dirty: Citing Sources in Your Report

Let's extend the earlier exercise. You had reproduced a sample report in Markdown. Now, suppose we provide you with a BibTeX file of references (say `sample_refs.bib`) and an updated version of the report that includes citations. Your task:

1. Add the necessary YAML fields to your document to enable citations (point to the provided .bib file, specify the style).
2. Insert the citations in the text at the appropriate places exactly as in the target report (you can use `citr` with the provided bib, or if you know the keys, type them).
3. Include the bibliography in the output (it will happen automatically when you knit if YAML is correct).

The folder named *chapter4* on the [GitHub repository](#) contains the `.bib` file and an example of the report with references. Use those to guide you.

After knitting, your output should match the **report with references** (which presumably shows the citations and reference list).

By completing this, you will have experienced the full workflow: writing in R Markdown, using Git for version control, and adding citations for a polished, professional document. You are now well-equipped to produce a reproducible research report from start to finish!

## 4 Data Wrangling

Data wrangling is a crucial step in any data science project, involving collecting, cleaning, transforming, and preparing data for analysis. In a typical data pipeline, raw data must be gathered from various sources and then manipulated into a structured format suitable for visualization or modeling. In this chapter, we focus on using R for effective data wrangling. R is a powerful functional programming language that excels at data manipulation and analysis, making it a popular choice in fields such as biology, medicine, economics, finance, psychology, sociology, and the humanities. Our goal is to harness these tools for business analytics in a **reproducible** way – meaning that all data operations are documented in code (for example, using RStudio and R Markdown) so that results can be easily reproduced and updated.

Throughout this chapter, we will highlight important tips in **bold** or *italics* and demonstrate code examples in R. By the end, you should be comfortable performing common data wrangling tasks in R.

**At the end of the chapter, you should be able to:**

- Import data from various sources (e.g., CSV files and Google Sheets) into R.
- Understand what a *data frame* is in R and how to inspect its contents.
- Understand the difference between using `package::function()` and `library(package)` when working with R packages.
- Use the pipe operator (`%>%`) to write clear, readable sequences of data manipulation steps.
- Add new columns to a data frame, rename or delete existing columns.
- Subset rows of a data frame based on conditions (filtering).
- Sort (arrange) data frame rows in ascending or descending order.
- Transform data between *long* and *wide* formats (tidy data principles using **tidyr** functions).
- Merge multiple data frames based on common keys (joining datasets).
- Save your cleaned or transformed data to a file for later use.

We will illustrate these techniques using a real dataset from the United Nations Industrial Development Organization (UNIDO), which contains various industrial statistics for different countries over time. Originally, this database has 655,350 data points, but through wrangling we will distill it down to a small summary for visualization. The concepts and functions covered, however, are applicable to any dataset.



## 4.1 R as a Functional Language

R is a *functional programming language*, meaning it is built around functions as the primary way to operate on data. In a functional language, you typically call specific functions (which may internally execute many operations) to accomplish tasks. This approach allows you to perform complex operations with relatively simple and human-readable code. Other examples of popular functional or scripting languages used in data science are Python and Julia.

Functional languages offer significant benefits: they make it straightforward to express complex transformations in just a few lines of code. For instance, with the right functions, you can programmatically collect all patent records worldwide and analyze innovation trends, or gather thousands of epidemiological research articles and identify leading research teams—all tasks that can be done on a personal computer with surprisingly little code. (Of course, such intensive tasks might be slow or require substantial computing time, but they are feasible.) Modern businesses can leverage these tools to transform their operations, turning vast raw datasets into actionable insights. In fact, the availability of powerful open-source tools like R compels businesses to adapt and become more data-driven.

At the same time, using a functional language means you must remember or look up the appropriate functions to use for each task. The R ecosystem provides thousands of functions, both in base R and in contributed packages, to handle nearly any data manipulation or analysis need. This open-source ecosystem is a double-edged sword: it fosters rapid innovation (anyone can create a new package or function and share it), but it also means there can be overlapping or incompatible functions. The community mitigates these issues through documentation, forums like StackOverflow, and continuous package development to fix bugs and improve compatibility.

**R, Python, or Julia?** Once you learn one high-level language, it becomes easier to learn others. R, Python, and Julia share many concepts and have similar capabilities for data science tasks. R has traditionally been favored in statistics and fields requiring complex data analysis and reporting; Python is often favored in engineering and general software development contexts; Julia is newer but offers speed advantages for heavy computations. In this book we focus on R, but keep in mind that the core data manipulation concepts are transferable across languages.

## 4.2 The Grammar of R: Packages, Libraries, and Pipes

Before diving into data manipulation in R, it's important to understand some basic “grammar” of how R code is structured, especially when using add-on packages.

### 4.2.1 Using Packages and Libraries

The real power of R comes from its packages – collections of functions, data, and documentation that extend R’s capabilities. There are thousands of packages available for everything from data wrangling (e.g., **dplyr**, **tidyr**) to visualization (**ggplot2**) to specialized analyses. In this chapter, we will use several packages, including **readr** (for reading data), **gsheet** (for Google Sheets access), **dplyr** (for data frame manipulation), **tidyr** (for reshaping data), and **tidyverse** (which actually bundles many data science packages together).

**Attaching a package with `library()`:** To use a package’s functions in your R session, you usually need to *attach* it by calling `library(packageName)`. For example, to use functions from the **dplyr** package, you would run: `library(dplyr)`. Once a package is attached, you can call its functions by name without any special prefix.

**Using a function with `package::function()`:** Alternatively, you can use a function from a package without attaching the entire package by using the syntax `packageName::functionName()`. For instance, `gsheet::gsheet2tbl()` calls the function `gsheet2tbl` from the **gsheet** package directly. This approach is convenient when you only need one function from a package or want to avoid name conflicts between functions in different packages. The downside is that you must prepend the package name each time you call the function, which can make code verbose if you use many functions from the same package.

**Example – reading from Google Sheets:** Below we demonstrate both methods. We use a Google Sheets URL for the UNIDO dataset as an example (more on data importing in the next section):

```
# Using package::function() without attaching the package
data_unido <- gsheet::gsheet2tbl("https://docs.google.com/spreadsheets/d/....edit#gid=416085055")
```

In the code above, `gsheet2tbl()` is explicitly called from the **gsheet** package. Alternatively, we could attach the package first:

```
# Attaching the package, then calling the function directly
library(gsheet)
data_unido <- gsheet2tbl("https://docs.google.com/spreadsheets/d/....edit#gid=416085055")
```

After running `library(gsheet)`, any function defined in the **gsheet** package (such as `gsheet2tbl`) can be used without the `gsheet::` prefix. Attaching packages at the start of your script (or R Markdown document) is often a good practice if you will use many functions from those packages, as it keeps the code cleaner and easier to read.

**Tip:** Many R users include a block at the top of their script or notebook to load all necessary libraries at once. For example:

```
library(gsheet)
library(dplyr)
library(tidyr)
library(readr)
```

This ensures all functions you need are available throughout your analysis.

### 4.2.2 The Pipe Operator %>%

When writing a sequence of data manipulation steps, R's *pipe* operator (`%>%`) from the **magrittr** package (which is included in the **tidyverse**) can make code more readable. The pipe passes the output of one function as the input to the next function, allowing you to chain operations in a left-to-right style that often mimics natural language.

Without pipes, you might nest function calls or create intermediate variables for each step. With pipes, you can instead write a clear linear sequence. For example, calling the logarithm function on a variable `x` normally looks like `log(x)`. Using the pipe, you can rewrite this as:

```
x %>% log()
```

This statement says “take `x`, then apply `log()` to it.” It’s a trivial example, but pipes shine in more complex sequences. Consider using the **dplyr** function `glimpse()` to get a quick overview of a dataset (we’ll discuss `glimpse()` in detail later). We can pipe our data frame into `glimpse()` as follows:

```
library(dplyr)
data_unido %>%
  glimpse()
```

This code takes the `data_unido` data frame and pipes it into `glimpse()`, which then prints an informative summary of the data frame’s structure. Using the pipe here saves us from having to write `glimpse(data_unido)`. More importantly, if we want to perform multiple operations in sequence (filtering, mutating, summarizing, etc.), the pipe allows us to write these steps in a clear order, instead of having to read from the inside out as with nested function calls. We will use `%>%` frequently in our examples as it is a cornerstone of readable tidyverse code.

### 4.2.3 Tidy Data Principles

In R (and data science in general), there is a recommended structure for datasets called **tidy data**. Hadley Wickham, a chief scientist at RStudio, coined the principles of tidy data which make data easier to manipulate and analyze. A dataset is tidy if:

- **Each variable is in its own column.** For example, if you have variables like “Country”, “Year”, and “Population”, each of these should be a separate column in your data frame rather than combined in one column.
- **Each observation is in its own row.** Each row typically represents one observation or record (e.g., a single country-year combination in a country-year dataset).
- **Each value is in its own cell.** A cell is the intersection of a row and a column, and it should hold only a single value (not a list of values or a combination of multiple pieces of information).

Following these rules makes it much easier to perform analysis, because most of R’s functions (especially in the **tidyverse**) are designed with tidy data in mind. If your data is not tidy, you might need to reshape it (for instance, converting from a “wide” format to a “long” format or vice versa) before analysis. We will practice this when we use **tidyr** functions like `pivot_longer()` and `pivot_wider()` later in this chapter.

### 4.2.4 Data Frames and Types in R

The primary data structure we will work with is the **data frame** (or its modern equivalent, the **tibble**). A data frame is essentially a table of data where each column is a variable and each row is an observation, consistent with the tidy data format described above. Columns in a data frame can be of different types:

- Numeric (numbers, e.g., 42 or 3.14)
- Character (strings of text, e.g., "Canada")
- Logical (Boolean values TRUE or FALSE)
- Factors (categorical variables with a fixed set of possible values)
- Others (dates, times, etc., which are usually handled by specialized classes)

When you import data into R, the columns might not always be in the desired type. For example, numbers might be read in as strings (character) due to formatting issues. It’s important to inspect your data and ensure each column is encoded with an appropriate type. You can convert column types using functions like `as.numeric()`, `as.character()`, `as.factor()`, etc.

RStudio’s interface can help you keep track of data frames and their columns. When you load a data frame, it appears in the **Environment** tab (typically in the upper right panel of

RStudio) listing the object's name, type, and a preview of its contents. If you click on a data frame in the Environment, it will open in a spreadsheet-like viewer for you to browse.

The screenshot shows the RStudio interface with the following components:

- Script Editor:** Contains R code for loading data from a Google Sheet and creating a plot. The code includes comments and function calls like `library(gsheet)`, `gsheet2tbl()`, and `geom_line()`.
- Console:** Shows the execution of the R code, including an error message: "Error: Unknown parameters: width".
- Environment Pane:** Lists the objects in the global environment, including `dataCanada`, `dataCanada181`, `dataCanada2`, `tableCode`, `countryCode`, `year`, `isicCode`, `value`, `dataCanada221`, `dataCanadaFull`, `dataCanadaFullLong`, `dataReverse`, `dataSorted`, `dataUnido`, `dataUnidoCanada`, `dataUnidoCanadaVariables`, and `tableCode`.
- Plot Pane:** Displays a line plot with the x-axis representing years (2009, 2010, 2011, 2012) and the y-axis representing values (1800, 1900, 2000, 2100). The plot shows two lines: a blue line and a black line.

Let's say we have loaded a data frame called `data_unido` containing the UNIDO dataset. We can check its structure and contents with a few useful commands:

- `colnames(data_unido)` will show all column names.
- `dim(data_unido)` will show the dimensions (number of rows, number of columns).
- `head(data_unido)` shows the first six rows; `tail(data_unido)` shows the last six.
- `str(data_unido)` or the **dplyr** alternative `glimpse(data_unido)` shows the structure of the data frame, including each column's name, type, and a preview of values.
- `summary(data_unido)` gives summary statistics for each column (for numeric columns, min/median/mean/max, etc., and for categorical columns, frequency of each category).

As an example, after loading `data_unido`, we might want to ensure certain columns are numeric. In the UNIDO dataset, columns like country codes, year, and indicator codes were read in as text. We can convert them to numeric as follows:

```
# Suppose data_unido is already loaded as a data frame.
# Convert some columns from character to numeric type:
data_unido$tableCode <- as.numeric(data_unido$tableCode)
data_unido$countryCode <- as.numeric(data_unido$countryCode)
data_unido$year <- as.numeric(data_unido$year)
data_unido$isicCode <- as.numeric(data_unido$isicCode)
```

```
data_unido$value      <- as.numeric(data_unido$value)

# Now check the structure and summary to confirm changes:
str(data_unido)
summary(data_unido)
```

In this code, `data_unido$year <- as.numeric(data_unido$year)` takes the `year` column of `data_unido`, converts it to numeric, and overwrites the original column with the numeric version. We repeat this for other columns that should be numeric. After conversion, `summary(data_unido)` will show, for example, the minimum and maximum year, which confirms that the `year` column is numeric and gives an idea of the time range.

**Note:** If a column contains non-numeric characters (e.g., “N/A”, or mixed text and numbers), using `as.numeric()` on it will result in `NA` (missing values) for those entries and a warning. Always make sure the data is clean or handle such cases (perhaps using functions like `parse_number()` from **readr** which can extract numbers from strings).

Now that we have covered the basics of R’s grammar and how to inspect data, let’s move on to actually getting data into R.

## 4.3 Importing Data

Real-world data can come in many formats and from many sources. Common sources include local files (like CSVs or Excel spreadsheets), databases, web APIs, and online data repositories. R provides tools to import data from all these sources. In this section, we will focus on two straightforward cases: importing a CSV file and importing data from a Google Sheet. (We will also briefly mention other formats like SPSS/Stata and the use of the **data.table** package for large data.)

### 4.3.1 Importing a CSV File

A **CSV (Comma-Separated Values)** file is a plain text file that stores tabular data. Each line of the file is a row, and columns are separated by commas (or sometimes other delimiters like tabs or semicolons). CSVs are one of the most common formats for sharing data.

To import a CSV file in R, you can use the base R function `read.csv()`. However, we will use `read_csv()` from the **readr** package (part of the tidyverse), which is typically faster and produces tibbles (a modern form of data frame). The usage is similar.

Before reading the file, make sure the file is accessible by R. If you're working in RStudio, you can upload the file to your RStudio workspace or ensure your working directory is set to where the file is located. In RStudio Server or Cloud, you might upload via the Files pane. For this example, suppose we have a CSV file named `gdpCountry.csv` located in a folder called `data` within our project.

Here's how we load the CSV into a data frame:

```
# Loading the readr package for read_csv (if not already loaded via tidyverse)
library(readr)
# Read the CSV file into a data frame (tibble)
gdpCountryData <- read_csv("./data/gdpCountry.csv")
```

After this, `gdpCountryData` will contain the data from the CSV. In this code:

- `"./data/gdpCountry.csv"` is the **path** to the file. `./data/` means there is a subfolder named “data” in our current working directory, and the file is inside it.
- We assign the result of `read_csv()` to a variable called `gdpCountryData`. This new variable is a data frame that holds all the information from the CSV file. We can now manipulate `gdpCountryData` in R (filter it, summarize it, etc.).

If the CSV file has a header row (column names) at the top, `read_csv` will automatically use those as column names. If not, you may need to specify `col_names = FALSE` and assign names later. In our case, we assume the CSV had headers.

After loading, it's good to do a quick check:

```
head(gdpCountryData)
dim(gdpCountryData)
```

This will show the first few rows and the dimensions (to see how many rows and columns were read in).

### 4.3.2 Importing Data from Google Sheets

Another common scenario is collaborating on data via Google Sheets. Rather than exporting the data to CSV manually, R can directly read from a Google Sheet URL (if the sheet is shared appropriately). We will use the **gsheet** package, which provides a convenient `gsheet2tbl()` function to read Google Sheets into R.

**Setup:** First, make sure your Google Sheet is shareable. Open the sheet in your browser, click the **Share** button, and set it so that “Anyone with the link can view” (or

edit, if you want others to collaborate). Copy the *sharing URL*. It will look something like:

<https://docs.google.com/spreadsheets/d/1uLaXke-KPN28-ESPPoihk8TiXVWp5xuNGHW7w7yqLCc/edit#gid=>

Notice it contains a long ID string (in this example, `1uLaXke-KPN28-ESPPoihk8TiXVWp5xuNGHW7w7yqLCc`). This identifies the document.

**Reading the Google Sheet:** Use `gsheet2tbl()` with the sheet's URL. For example:

```
# Load the gsheet package
library(gsheet)

# Use gsheet2tbl to read the Google Sheet into a data frame
data_unido <- gsheet::gsheet2tbl("https://docs.google.com/spreadsheets/d/1uLaXke-KPN...LCc/e
```

This will reach out to Google's servers and pull the spreadsheet data into R as a data frame (actually a tibble, which is essentially the same for our purposes). The object `data_unido` now holds the UNIDO dataset we will work with. We can inspect it just as we would any imported data:

```
dim(data_unido)
colnames(data_unido)
```

Often, we may also have a separate sheet (or CSV) that contains metadata, such as variable definitions or codes. In the example above, `variablesUnido` was mentioned, which likely contains descriptions of each variable in the UNIDO dataset. We could load that as well if needed:

```
variables_unido <- gsheet2tbl("https://docs.google.com/spreadsheets/d/1JYiydgI9QE0...MD8/edit
```

Now `variables_unido` might contain a lookup table explaining what each code in the dataset means (for instance, that `tableCode == 4` corresponds to “Number of Employees”, etc.). This can be useful for reference.

### 4.3.3 Other Data Formats and Tools

R can handle a variety of data formats:

- **Excel files:** Use the `readxl` package (functions `read_excel()`, etc.) or export Excel to CSV and use `read_csv`.
- **Statistical software files** (SPSS `.sav`, Stata `.dta`, SAS `.sas7bdat`): Use the **haven** package, which provides `read_spss()`, `read_stata()`, `read_sas()`, etc., to import these files.



- **Databases:** Use packages like **DBI** along with database-specific drivers (RMySQL, RPostgreSQL, etc.) to run SQL queries and import data from databases.
- **APIs and Web Data:** Packages like **httr** and **jsonlite** can be used to retrieve data from web APIs (often returning JSON or XML). Web scraping might involve packages like **rvest**.
- **Specialized formats:** For example, **readr** has **read\_tsv()** for tab-separated data, and **data.table** has **fread()** which can auto-detect formats and is extremely fast.

One notable package is **data.table**. It provides an alternative data frame structure and syntax in R that is highly optimized for speed and memory efficiency (often approaching the speed of comparable tasks in lower-level languages like C++). For very large datasets (millions of rows), **data.table::fread()** and the **data.table** syntax can significantly outperform base R or **dplyr** in reading and manipulating data. However, **data.table** has its own syntax which is a bit different from the tidyverse style we focus on here. If you are working with huge data and need speed, you might explore **data.table**, but for this chapter we will stick with tidyverse tools.

**Recap – Importing:** To summarize, here are the key functions for importing data we’ve seen:

- **read\_csv("path/to/file.csv")** – Read a CSV file from disk (from **readr**).
- **gsheet2tbl("sheet\_url")** – Read a Google Sheet via URL (from **gsheet**).
- (Plus mention of others like **read\_excel**, **read\_spss**, etc., for completeness, but not demonstrated in code here.)

After importing data, the next steps are to explore and clean it. We’ll use the UNIDO data (**data\_unido**) as our working example for the rest of this chapter.

## 4.4 Exploring and Preparing Data

Once data is loaded into R, the first thing to do is often to explore it: check its dimensions, peek at the first few rows, and get summaries of each column. This helps us verify that the import worked correctly and that we understand the structure of the data. We have already touched on some of the functions for this, but let’s go through a structured example using our **data\_unido** dataset.

Assume **data\_unido** is a data frame with the following columns (based on context from UNIDO data):

- **countryCode:** a numeric or character code for countries (e.g., 124 for Canada).
- **year:** the year of observation (e.g., 2005, 2006, ..., 2012).
- **isicCode:** a code representing an industry (using the ISIC classification, e.g., 131 might be textiles, 181 might be printing).

- **tableCode**: a code for the indicator being measured (e.g., 1 = number of establishments, 4 = number of employees, etc.).
- **value**: the value of the indicator (e.g., the number of employees in that industry, country, and year).

There may be other columns (the original data had up to 10 columns including some definitions and source info), but in our examples we will eventually narrow down to these key ones.

Let's explore `data_unido`:

```
# View column names
colnames(data_unido)

# View first 6 rows
head(data_unido)

# View the last 6 rows
tail(data_unido)

# Check the size of the data frame (rows, columns)
dim(data_unido)

# Get a detailed structure of the data frame
glimpse(data_unido)  # from dplyr, similar to str(data_unido)
# Or use base R:
str(data_unido)

# Get summary statistics for each column
summary(data_unido)
```

The output of these commands will tell us:

- How many rows and columns (`dim`) – we expect something like 655350 rows and 10 columns in the full UNIDO dataset.
- The names of the columns (`colnames`).
- The first few entries (`head`) to spot-check what the data looks like (are there headers? Does it seem to be reading correctly?).
- The structure (`glimpse/str`) – this is very useful as it lists each column with its type and a sample of values. We might see that some columns that should be numeric are currently character, etc.
- The summary (`summary`) – which for numeric columns will give min, median, mean, max, etc., and for factor/char will either list frequencies or say it's not meaningful.

From earlier, we already converted some columns to numeric (countryCode, year, etc.). If we had not, `glimpse` might show them as `<chr>` (character). Assuming we did convert them, `glimpse` should show them as `<dbl>` (numeric type for doubles) now, which is what we want.

It's common to drop some unnecessary columns at this stage, especially if the dataset has metadata or extra identifiers that we won't use. For example, the UNIDO dataset had columns like `isicCodeCombinaison`, `tableDefinitionCode`, `sourceCode`, `updateYear`, `unit` which might not be needed for analysis after initial filtering. We can remove columns by assigning them `NULL`. For instance:

```
# Remove some unneeded columns to simplify the data frame
data_unido$isicCodeCombinaison <- NULL
data_unido$tableDefinitionCode <- NULL
data_unido$sourceCode <- NULL
data_unido$updateYear <- NULL
data_unido$unit <- NULL

# Verify the columns have been removed
colnames(data_unido)
```

After this, `data_unido` should have fewer columns. In our case, it would now have columns like `countryCode`, `year`, `isicCode`, `tableCode`, and `value` (5 columns remaining). The data frame might be about 65,535 rows  $\times$  5 columns at this point (if we've filtered some, as we'll do next). We can confirm with `dim(data_unido)`.

Cleaning the data might also involve renaming columns to something more human-readable. The `dplyr` function `rename()` is useful for this. For example, we might rename `tableCode` to `IndicatorCode` or even directly to what it represents (if `tableCode` 4 = "Employees", maybe we create an `Employees` column). We'll see an example of renaming in the exercises.

Now that the data is imported and we have a sense of what's inside, we can start wrangling the data to answer specific questions or prepare for analysis.

## 4.5 Manipulating Data Frames

The real work of data wrangling involves transforming the data: adding or removing columns, filtering for certain rows, sorting, combining data from multiple sources, and so on. R provides multiple ways to do these tasks; we will primarily use functions from `dplyr` for a consistent and readable approach. Recall that we should run `library(dplyr)` to have these functions available (if not done already).

### 4.5.1 Adding, Renaming, and Deleting Columns

**Adding a new column:** To add a new column to a data frame, you can use the assignment operator `<-` with the data frame and new column name. For example, suppose we want to add a column `newColumn` to `data_unido` where every value is 42 (perhaps as a placeholder or dummy variable):

```
# Add a new column named 'newColumn' with a constant value
data_unido$newColumn <- 42
```

This creates `newColumn` as a column in `data_unido` and assigns 42 to every row. We can verify the addition by checking the column names or a slice of the data:

```
colnames(data_unido)      # to see if 'newColumn' is listed
head(data_unido[, "newColumn"]) # to see the first values in newColumn
```

We can also create columns based on existing data. For example, we might scale a column or combine information from multiple columns. Say `newColumn` should actually be computed as double of `value` plus 5:

```
data_unido$newColumn <- data_unido$newColumn * 2 + 5
head(data_unido[, "newColumn"])
```

If originally `newColumn` was 42 for all rows, now it will be  $42 * 2 + 5 = 89$  for all rows (since we haven't varied it by row). This is just a trivial example to illustrate operations on columns.

**Renaming columns:** If you decide to change the name of a column (maybe `newColumn` isn't descriptive), you can use `dplyr::rename()`. The syntax is `rename(dataframe, newName = oldName)`. For instance:

```
library(dplyr)
data_unido <- data_unido %>%
  rename(newColumnRenamed = newColumn)
colnames(data_unido)
```

Now, the column that was `newColumn` is called `newColumnRenamed`. We assigned the result back to `data_unido` (although `rename` will return the renamed data frame even if you don't, it's good practice to explicitly assign or use the pipe as we did).

**Deleting columns:** To remove a column, assign `NULL` to it as we showed earlier. For example, to remove `newColumnRenamed` now:

```
data_unido$newColumnRenamed <- NULL
colnames(data_unido)
```

This will drop that column entirely from the data frame. (Another way to drop columns is using the `select()` function in `dplyr`, but assigning `NULL` is straightforward for removing single columns by name.)

After these operations, our `data_unido` is back to having only the original columns we care about. We now have a cleaner dataset with which to work.

### 4.5.2 Filtering Rows (Subsetting the Data)

Often, we don't need all the data that we imported. We might be interested in a specific country, a specific time range, or a subset of indicators. *Filtering* refers to subsetting the rows of the data frame based on some condition.

The `dplyr` function for filtering is aptly named `filter()`. You provide it with a data frame and one or more conditions, and it returns only the rows that satisfy those conditions.

**Example scenario:** Using the UNIDO data, suppose we want to focus on Canada's data, specifically on two indicators: number of establishments and number of employees, for years after 2009. Concretely:

- Canada is identified by `countryCode == 124` (assuming the country codes follow the ISO numeric standard where 124 = Canada).
- The indicator for “number of establishments” might be coded as 1, and “number of employees” as 4 (this is inferred from context).
- “After 2009” means `year > 2009` (which would include 2010, 2011, 2012 given our dataset goes up to 2012).

We can apply these filters step by step or in one go. First, let's filter the data for Canada:

```
library(dplyr)
data_unido_canada <- filter(data_unido, countryCode == 124)
dim(data_unido_canada)
head(data_unido_canada)
```

Now `data_unido_canada` contains only rows where `countryCode` is 124. The dimension might be, say, 3888 rows and 5 columns (just as a reference from the dataset). Next, filter this down to only the desired indicators (tableCode 1 or 4):

```
data_unido_canada_vars <- filter(data_unido_canada, tableCode == 1 | tableCode == 4)
dim(data_unido_canada_vars)
head(data_unido_canada_vars)
tail(data_unido_canada_vars)
```

Here we used the R logical operator `|` which means OR. The condition `tableCode == 1 | tableCode == 4` keeps rows where the `tableCode` is either 1 **or** 4. Now `data_unido_canada_vars` should contain only the rows for Canada and for those two indicators. We can check the first and last rows to see years and ensure we have both indicators present.

Finally, filter for `year > 2009`:

```
data_unido_canada_post2009 <- filter(data_unido_canada_vars, year > 2009)
dim(data_unido_canada_post2009)
head(data_unido_canada_post2009)
```

This should leave us with data for Canada, indicators 1 and 4, for years 2010, 2011, 2012 (since the original data likely covers 2005–2012). The resulting number of rows might be around 972 (because it's 3 years  $\times$  2 indicators  $\times$  some number of industries, if each industry is a row; or if each row was an industry-year-indicator combination).

We could have combined all those filters into a single `filter()` call, like so:

```
data_unido_canada_post2009 <- filter(data_unido,
                                     countryCode == 124,
                                     (tableCode == 1 | tableCode == 4),
                                     year > 2009)
```

Dplyr `filter()` allows you to put multiple conditions separated by commas, which are treated as AND (all must be true). The above is equivalent to the step-by-step approach but does it in one shot.

At this point, we have drastically pared down the dataset to a manageable subset. This is often necessary before reshaping or plotting, because the full dataset might be too large or not relevant to the question at hand. In our case, going from 655k points to a few hundred or a thousand is useful for illustration and perhaps for focusing on a particular story (like how the number of establishments and employees in Canada changed post-2009).

The example above filtered by a specific country and indicators. You can filter by any condition needed:

- A range of years (e.g., `year >= 2009 & year <= 2012` for between 2009 and 2012 inclusive).

- A set of countries (using `%in%`, e.g., `countryCode %in% c(124, 840, 392)` for Canada, USA, Japan if those are the codes).
- Excluding missing values (`!is.na(column)`).
- Etc.

**Logical operators recap:** In R, `&` means AND, `|` means OR, and `!` means NOT. When combining multiple conditions in `filter()`, commas act like AND. So `filter(df, cond1, cond2)` is the same as `filter(df, cond1 & cond2)`.

Now that we have filtered data, let's look at sorting.

### 4.5.3 Sorting Data

Sorting (ordering) the data can help in viewing it or preparing it for output. For example, we might want to sort the filtered Canadian data by the value column to see which industry-year had the highest number of employees or establishments.

The **dplyr** function for sorting is `arrange()`. By default, `arrange(data, columnName)` will sort the data frame by that column in ascending order (smallest to largest). For descending order, you wrap the column in `desc()`. You can also sort by multiple columns (it will sort by the first, and break ties by the second, and so on).

Using our `data_unido_canada_post2009` example (which has columns `countryCode`, `year`, `isicCode`, `tableCode`, `value`):

- If we want to sort by `value` ascending:

```
data_sorted <- arrange(data_unido_canada_post2009, value)
head(data_sorted)
```

- If we want the largest values first (descending order):

```
data_reverse <- arrange(data_unido_canada_post2009, desc(value))
head(data_reverse)
```

Comparing `head(data_sorted)` and `head(data_reverse)` would show that the first rows of one are the smallest values and of the other are the largest values. Sorting can be useful before saving data to a file or just to quickly identify extremes.

We could also arrange by multiple keys. For example, maybe we want data sorted by year, then by value within each year:

```
data_year_val <- arrange(data_unido_canada_post2009, year, desc(value))
```

This would sort the data by year ascending, and for the same year, sort by value descending (so the largest value in each year comes first for that year).

At this point, we’ve covered how to get a subset of the data that we need and how to sort it. Next, we’ll discuss reshaping the data between long and wide formats, which is often needed to meet the “tidy data” requirements or to prepare for certain analyses or visualizations.

## 4.6 Reshaping Data: Long vs Wide Format

Data can be structured in different ways. Two common structures are **wide format** and **long format**:

- **Wide format:** Each distinct variable has its own column, and each row might represent an entity with multiple observations across those columns. For example, a wide format could have one row per country, and separate columns for each year’s GDP.
- **Long format:** There is typically one column that holds values and another column that indicates what kind of value it is (with each observation occupying one row). In long format, you might have one row per country-year, and a column indicating which year it is and another column for the GDP value.

Tidy data, as discussed, often corresponds to a long format: each observation is one row. However, data is often recorded or presented in wide format. We need to be able to convert between these formats. In R, the **tidyr** package provides the functions `pivot_wider()` and `pivot_longer()` to do this transformation (these replace the older `spread()` and `gather()` functions).

### 4.6.1 From Long to Wide

Let’s use our filtered Canadian data as an example. Currently, `data_unido_canada_post2009` is in a *long* format: each row is an observation (with country, year, indicator code, and value). Suppose we want to create a *wide* format where we have separate columns for each industry’s value. For example, one column for the textile sector (ISIC 131) employees, and one for the printing sector (ISIC 181) employees, for each year.

Why might we want to do this? Perhaps to compare those two industries side by side over years for Canada. A wide format could have:

- Columns: year, tableCode (indicator), countryCode, and then separate columns for each selected `isicCode` (131, 181).
- Rows: each row might correspond to a unique combination of year, tableCode, countryCode (in our filtered data, countryCode is constant 124 for Canada, tableCode might be 1 or 4 depending on if we included both employees and establishments – we did).



- The cells in the new industry-specific columns would be the **value** of that indicator for that industry.

Our filtered data `data_unido_canada_post2009` currently includes two tableCodes (1 and 4). To keep it simple, let's further filter to just one indicator, say tableCode 4 (number of employees), before widening, because mixing two different indicators in a single wide table can be confusing (they would become separate rows anyway if tableCode remains a column). Alternatively, we could include tableCode as part of the key so that employees and establishments don't mix.

For demonstration, assume we want to compare two industries: ISIC 131 vs ISIC 181 (textiles vs printing) for the number of employees (tableCode 4) in Canada, years 2010–2012. We can create two data subsets and then merge, or directly pivot. Let's try pivoting directly on the dataset that contains both industries.

We will use `pivot_wider()`. Its key arguments are:

- `names_from`: the column whose values will become new column names.
- `values_from`: the column whose values will fill those new columns.

In our case:

- `names_from = isicCode` (because we want a column for each distinct isicCode).
- `values_from = value` (because those are the numbers we want in the cells).

We should ensure that the combination of the other columns (year, countryCode, tableCode) is unique for each isicCode in the data we pivot. In our dataset, for each year and tableCode, there might be multiple industries. When we pivot wider, each row of the wide table will correspond to one combination of the non-pivoted columns.

Let's do it:

```
library(tidyr)
wide_data <- data_unido_canada_post2009 %>%
  pivot_wider(names_from = isicCode, values_from = value)
head(wide_data)
dim(wide_data)
```

After this pivot:

- The new `wide_data` should have columns: year, countryCode, tableCode, and then one column for each unique isicCode present in `data_unido_canada_post2009` (in our filtered case, it might be many industries unless we filtered to just a couple of ISIC codes; let's assume we only kept a couple for simplicity). If we hadn't filtered, it could be a lot of columns (the example given said 165 columns, which suggests 163 industry codes plus year and a couple other keys).

- Each row of `wide_data` represents a single year-country-indicator combination. In our case, country is fixed as Canada (124) and indicator could be 1 or 4 if we left both. If both 1 and 4 are present, then for each year we would have two rows (one for tableCode 1, one for 4).
- For each such row, the value for a particular industry code will appear in that industry's column, and presumably the other industry columns will be NA for that row (because a given row is one observation of one indicator in one industry).

The dimension output (`dim(wide_data)`) might show, for example, 6 rows and (some number of columns). The original text said 6 lines and 165 columns for a similar operation, which likely means they had 6 rows (maybe 3 years  $\times$  2 indicators?) and 165 columns (for each industry code present).

Our wide format is not tidy (because each industry's values are split into separate columns), but sometimes wide format is needed, for instance, for certain types of analysis or simply for readability or exporting to another program.

#### 4.6.2 From Wide to Long

Now, suppose we have a wide format data frame (like `wide_data` we just created) and we want to go back to a long format (tidy format). We use `pivot_longer()` for this. The arguments for `pivot_longer()` are essentially the opposite of `pivot_wider()`:

- We need to specify which columns to gather into key-value pairs. Often, it's easier to specify which columns *not* to pivot (like the ID or key columns that should remain as is). We can use the special syntax `!c(col1, col2, ...)` to indicate “all columns except these”.
- We specify `names_to` for the name of the new column that will contain what was formerly column names (e.g., “isicCode”), and `values_to` for the new column that will contain the values from those columns.

Continuing our example, to pivot `wide_data` back to long format:

```
long_data <- wide_data %>%
  pivot_longer(
    cols = !c(year, countryCode, tableCode),    # all columns except these key columns
    names_to = "isicCode",
    values_to = "value"
  )
dim(long_data)
head(long_data)
```

A couple of things to note:

- We used `!c(year, countryCode, tableCode)` in `cols`. The exclamation mark `!` in front of `c(...)` means “not these columns”. So we are telling `pivot_longer` to gather all columns *except* `year`, `countryCode`, and `tableCode`. These excepted columns will remain as they are (they are the identifier columns that stay in each row).
- The result will have a column named `isicCode` (as we specified) and a column named `value`. Every row of `long_data` is now a single observation: it has a `year`, a `countryCode`, a `tableCode`, an `isicCode`, and a `value`. In fact, if we did everything correctly, `long_data` should look identical to our original `data_unido_canada_post2009` (except maybe the order of rows might differ, and the type of `isicCode` might be character now because it came from column names, but we could convert it back to numeric).

The dimension (`dim(long_data)`) should match the original number of filtered observations we had before widening. The `head(long_data)` will show a few rows and we expect to see columns for `year`, `countryCode`, `tableCode`, `isicCode`, `value`.

**Important:** For creating plots and running most analyses in R, the long format is usually required. For example, **ggplot2** (for visualization) expects data in long format: each row is one observation. If your data is in wide format, you often need to convert it to long before using such tools.

So, why do we ever use wide format? Wide format can be useful for:

- Presenting data in tables (reports, spreadsheets) where each column is an easy-to-understand category.
- Performing certain operations that are easier in wide form (e.g., matrix operations, or calculating differences between two specific columns).
- Exporting to formats or software that expect wide data (Excel users often prefer wide format tables).

In R, though, analysis tends to favor long format, and we will proceed with data in long format for the remainder of the chapter.

We have now covered how to reshape data, which is a key part of wrangling when dealing with multiple variables that could be structured in different ways.

## 4.7 Merging Datasets

Data often comes split across multiple tables. For example, you might have one table with economic indicators and another with geographic information; or, as our earlier anecdote suggested, you might have a customer information table and another table with customer transactions. Combining data from different sources is another fundamental operation in data wrangling.

R provides *join* functions to merge data frames. The most common is `left_join()` from **dplyr**, which merges two data frames based on a common key, keeping all observations from the “left” data frame and adding matching information from the “right” data frame.

Before we demonstrate merging in R, let’s motivate it with an example scenario: Imagine you have two hacked datasets (to borrow the chapter’s playful scenario): one from a bank containing customer account details, and another from a credit bureau containing social security numbers. By joining them on a common identifier (like name or SSN), you could suddenly have a much richer dataset about each customer. This is exactly what companies do legitimately when they integrate data from different departments or partner organizations: merging datasets gives new insights by combining information.

In our case, we will merge subsets of the UNIDO data. Let’s create two small data frames from `data_unido`:

- `data_canada_131`: Data for Canada (`countryCode == 124`), for the textile sector (`isicCode == 131`), for the number of employees (`tableCode == 4`), and for years after 2008.
- `data_canada_181`: Similar, but for the printing sector (`isicCode == 181`).

These two data frames represent two different industries in Canada. Each one, if filtered to `tableCode 4` and after 2008, should have data for years 2009–2012 (assuming data exists for those years) on number of employees. Our goal is to merge these so that we can directly compare the values side by side.

First, let’s obtain these datasets (we assume `data_unido` is still our full dataset loaded from GSheet but possibly with extraneous columns removed and types set):

```
# Ensure dplyr and tidyr are loaded
library(dplyr)
library(tidyr)

# Create first dataset: Canada, ISIC 131 (textiles), employees (tableCode 4), year > 2008
data_canada_131 <- data_unido %>%
  filter(countryCode == 124, isicCode == 131, tableCode == 4, year > 2008) %>%
  pivot_wider(names_from = isicCode, values_from = value)
head(data_canada_131)

# Create second dataset: Canada, ISIC 181 (printing), employees (tableCode 4), year > 2008
data_canada_181 <- data_unido %>%
  filter(countryCode == 124, isicCode == 181, tableCode == 4, year > 2008) %>%
  pivot_wider(names_from = isicCode, values_from = value)
head(data_canada_181)
```

A few things to note in this code:

- We applied the filters inside the pipe for clarity. `data_canada_131` now should contain only data where `countryCode` is 124, `isicCode` 131, etc. After filtering, we used `pivot_wider`. Since in each of those filtered sets there's only one `isicCode` value, `pivot_wider` will essentially turn the `value` column into a column named "131" (or "181" for the second dataset). It's a way to prepare for the join: each dataset now has a column with the industry code as the name and the employee count as the value.
- The key columns that remain in each (besides the new "131" or "181" column) should be `year`, `countryCode`, and `tableCode`. For our filtered data, `countryCode` is constant (124) and `tableCode` is constant (4) within each data frame, but they are still present as columns. `Year` will vary (2009, 2010, 2011, 2012 for instance).
- We could have dropped `countryCode` and `tableCode` since they are constant (for these particular subsets), but it won't hurt to keep them as keys for the merge.

Now, to merge `data_canada_131` and `data_canada_181`, we need to have common key columns. By inspecting, both should have `year`, `countryCode`, and `tableCode`. Those make sense as keys: we want to align the data by year (and it's the same country and indicator in both). We will use `left_join` to combine them:

```
data_canada_full <- left_join(data_canada_131, data_canada_181, by = c("year", "countryCode"))
head(data_canada_full)
```

In `left_join(X, Y, by = c("col1", "col2", ...))`, the first argument `X` is the left table (in our case `data_canada_131`), the second argument `Y` is the right table (`data_canada_181`), and `by` specifies the columns that should match between the two. We used the vector of column names `c("year", "countryCode", "tableCode")` as the key. We could also have left out `by` and, if the column names are identical in both data frames, `left_join` would guess them; but it's clearer to be explicit.

The result `data_canada_full` will have:

- All rows from `data_canada_131` (the left table). That means all years after 2008 that had textile (131) data.
- It will add the matching columns from `data_canada_181` where the year, country, and `tableCode` match.
- If there was a year present in `data_canada_131` that isn't in `data_canada_181`, the new columns from `data_canada_181` (i.e., the "181" column) will be NA for that year. If `data_canada_181` had an extra year that 131 didn't, those would not appear because `left_join` keeps only rows from the left side. (An alternative would be `full_join` to keep all, or `inner_join` to keep only matching rows; but we chose `left_join` as a common approach.)

Given our filtering, both likely cover 2009–2012, so they should align perfectly, and `data_canada_full` should have one row per year (2009, 2010, 2011, 2012) for Canada, with `tableCode` 4, and two additional columns: one named "131" and one named "181",

containing the employee counts in those industries. We can confirm by looking at `head(data_canada_full)` (though that will show all since there are only a few rows).

Now, `data_canada_full` is in a wide format (year, countryCode, tableCode, 131, 181). If we want to get it back to a tidy long format (e.g., for plotting), we can pivot longer:

```
data_canada_full_long <- data_canada_full %>%
  pivot_longer(
    cols = !c(year, countryCode, tableCode),
    names_to = "isicCode",
    values_to = "value"
  )
head(data_canada_full_long)
```

After this, `data_canada_full_long` will have columns: year, countryCode, tableCode, isicCode, value, and each row will be a single observation (e.g., Canada, 2010, tableCode 4, isicCode 131, value X; then Canada, 2010, tableCode 4, isicCode 181, value Y; etc.). Essentially, we're back to a long format where the two industries' data are stacked, but this time we only have those two industries in the dataset.

This `data_canada_full_long` is now ready for further analysis or visualization (for instance, we could create a plot of employees in Textile vs Printing in Canada over 2009–2012).

Finally, if we want to save this prepared dataset to a file (to perhaps use in a visualization chapter or share), we can write it to a CSV:

```
library(readr)
write_csv(data_canada_full_long, "dataCanadaFullLong.csv")
```

This will create a file `dataCanadaFullLong.csv` in your working directory (or specified path) containing the year, countryCode, tableCode, isicCode, and value columns for the two industries we merged. We now have a clean, merged dataset that came from two different subsets originally.

## 4.8 Saving Your Data

It's worth noting that after wrangling data, you often want to save the clean or transformed version for later use. We just saw an example of using `write_csv()` from **readr** to save a data frame to a CSV file. The base R alternative is `write.csv()`, but `write_csv` is a bit faster and doesn't add row numbers by default. The usage is straightforward: give it the data frame and the desired file path.

Other functions for exporting:

- `write_rds()` (from **readr**) can save an R object in a binary RDS format, which preserves data types and is often more compact; you load it back with `read_rds()`.
- If collaborating with Excel users, the **writexl** or **openxlsx** packages can write `.xlsx` files.
- For databases, you'd use DBI functions to append or copy data into database tables.

For now, saving to CSV is a simple and effective way to export the results of our wrangling.

## 4.9 Summary

In this chapter, we covered the essential steps of data wrangling in R:

- We learned about R's approach to functions and packages, and how to use `library()` and `package::function()` to access the tools we need.
- The pipe operator `%>%` was introduced as a way to make code more readable by chaining operations.
- We reviewed the concept of tidy data (each variable in a column, each observation in a row, each value in a cell) and saw how this principle guides data structuring in R.
- We practiced importing data from CSV files and Google Sheets into R data frames, and discussed other formats briefly.
- Using a real dataset, we explored how to inspect data frames (viewing columns, sizes, and summaries) and ensure the data types are correct, converting types when necessary.
- We added new columns to a data frame, renamed and removed columns, using base R and dplyr approaches.
- We filtered rows based on conditions with `filter()`, allowing us to focus on subsets of the data.
- We sorted data frames with `arrange()` to order the observations.
- We reshaped data between long and wide formats using `pivot_wider()` and `pivot_longer()`, and reinforced why long format (tidy format) is often preferred for analysis.
- We merged data from two different subsets using `left_join()`, demonstrating how to align data on keys and combine it into one data frame.
- Finally, we showed how to save the cleaned/wrangled data to a CSV file for future use.

Below is a table of some key R functions we encountered and their purposes:

Function	Purpose
<code>read_csv()</code>	Read data from a CSV file into a data frame (tibble). <i>(From readr)</i>
<code>gsheet2tbl()</code>	Read data from a Google Sheets URL into a data frame. <i>(From gsheet)</i>
<code>head()</code> / <code>tail()</code>	Display the first / last rows of a data frame (default 6 rows).
<code>colnames()</code>	Show or set the column names of a data frame (or matrix).

Function	Purpose
<code>dim()</code>	Show the dimensions (number of rows and columns) of a data frame.
<code>str()</code> / <code>glimpse()</code>	Display the structure of a data frame (columns types and examples). <code>glimpse</code> is from <code>dplyr</code> and is similar to <code>str</code> but formatted nicely for tibbles.
<code>summary()</code>	Produce summary statistics of each column in a data frame.
<code>as.numeric()</code> / <code>as.character()</code> / <code>as.factor()</code>	Convert a vector (or data frame column) to numeric, character, or factor type, respectively.
<code>\$</code> (dollar sign)	Extract or create a column in a data frame ( <code>df\$col</code> ), also used to assign to a column ( <code>df\$newcol &lt;- ...</code> ).
<code>library()</code>	Attach an R package, making all its functions available by name.
<code>%&gt;%</code> (pipe)	Pass the output of one expression as the input to the next, enabling chaining of commands. ( <i>From magrittr/dplyr</i> )
<code>filter()</code>	Subset rows of a data frame based on condition(s). ( <i>From dplyr</i> )
<code>rename()</code>	Rename columns in a data frame. ( <i>From dplyr</i> )
<code>arrange()</code>	Sort a data frame's rows by the values of one or more columns. ( <i>From dplyr</i> )
<code>desc()</code>	Helper function to sort in descending order (used inside <code>arrange</code> ). ( <i>From dplyr</i> )
<code>pivot_wider()</code>	Convert data from long to wide format, creating new columns for levels of a key variable. ( <i>From tidyr</i> )
<code>pivot_longer()</code>	Convert data from wide to long format, gathering columns into key-value pairs. ( <i>From tidyr</i> )
<code>left_join()</code>	Merge two data frames, keeping all rows from the left and adding matching data from the right. ( <i>From dplyr</i> )
<code>write_csv()</code>	Write a data frame to a CSV file. ( <i>From readr</i> )

With these tools, you can accomplish a vast array of data cleaning and transformation tasks. Data wrangling often requires creativity and problem-solving, as every dataset can have its own quirks. The functions above are like a toolkit—you will combine them in different ways depending on what the data looks like and what form you need it to be in for your analysis.

In the next chapter, we will build on this foundation and explore data visualization, where we take our tidy, wrangled data and create insightful graphs.

## 4.10 Exercises: Practice Your Data Wrangling Skills

It's time to get your hands dirty and practice the concepts from this chapter. In this exercise, we will work step-by-step through a data wrangling task using a sample dataset. The exercise



is broken down into parts, and by the end you will have applied importing, filtering, reshaping, and joining on the data.

For these exercises, assume we have:

- A CSV file containing GDP data (`chapter6data.csv`).
- A Google Sheet containing geographical coordinates (`locations` data).
- We will combine these to get a final dataset.

**Step 1: Import a CSV file** Import the CSV file called `chapter6data.csv` (which contains GDP data for various countries and years) into R and store it in a data frame `gdp`. Use `readr::read_csv()` to do this.

1. Import the CSV file:

```
library(readr)
gdp <- read_csv("chapter6data.csv")
```

(Fill in the path or ensure your working directory is set so that the file can be found.)

**Step 2: Import a Google Sheet** Import a dataset containing longitude and latitude for countries from the Google Sheet available at the given URL. Store this in a data frame `locations`. Use the `gsheet2tbl()` function for this.

2. Import the Google Sheet:

```
library(gsheet)
locations <- gsheet2tbl("https://docs.google.com/spreadsheets/d/1nehKEBKTQx11LZuo5ZJFKTV")
```

Make sure the Google Sheet is shared publicly (or at least accessible via the link). The `locations` data might have columns like Country, Longitude, Latitude.

**Step 3: Delete an unnecessary column** The `gdp` data frame may have an extra index column (for example, sometimes CSVs exported from certain systems include an unnamed index as the first column, which might appear as `X1` in R). Remove the column named `X1` from the `gdp` data frame if it exists.

3. Remove the `X1` column from `gdp`:

```
gdp$X1 <- NULL
```

Confirm that the column is gone by checking `colnames(gdp)`.

**Step 4: Filter the data** We only want to focus on a few countries in this analysis. Filter the `gdp` data to keep only the following countries: **United States, Canada, Japan, Belgium, and France**.

4. Filter `gdp` for specific countries:

```
library(dplyr)
gdp2 <- filter(gdp, country %in% c("United States", "Canada", "Japan", "Belgium", "France"))
```

Here we assume the GDP data frame has a column named `country`. We used the `%in%` operator to match a set of values. The result `gdp2` contains data only for those five countries.

**Step 5: Reshape data from wide to long** Suppose the `gdp2` data is in a wide format, where each year might be a separate column (e.g., columns for 2015, 2016, 2017, etc., containing GDP values). We want to convert this to a long format with three columns: “country”, “year”, and “gdp”. Use `pivot_longer()` to gather the year columns.

5. Convert `gdp2` to long format:

```
library(tidyr)
gdp3 <- gdp2 %>%
  pivot_longer(
    cols = -country,          # all columns except 'country'
    names_to = "year",
    values_to = "gdp"
  )
```

In this code, we assume `gdp2` has a column named `country` and other columns that are year names (like “2015”, “2016”, etc.). The `-country` in `cols` means we take all columns except `country` (thus all the year columns) to pivot into long format. After this, `gdp3` will have three columns: `country`, `year`, and `gdp`, where each row is the GDP of a country in a particular year.

**Step 6: Merge the datasets** Now we have `gdp3` (country-year-GDP data) and `locations` (country-longitude-latitude data). We want to combine these so that for each country and year, we also have the location information. The `locations` data likely has one row per country (since coordinates don’t change by year). We will join `locations` to `gdp3` by the `country` column.

6. Join `locations` with `gdp3`:

```
gdp4 <- left_join(gdp3, locations, by = "country")
```

This will add the longitude and latitude columns from `locations` to the `gdp3` data frame, matching on the country name. We use `left_join` with `gdp3` as the left table so that all country-year combinations remain, and we bring in the coordinates for those countries. If a country in `gdp3` wasn’t found in `locations`, the coordinates would be NA (but since we filtered to specific countries that hopefully are in the `locations` data, we should get matches for all).

After completing these steps, `gdp4` should contain the country, year, GDP, and location (longitude, latitude) for the five selected countries. This dataset is now ready for any geographical visualization or further analysis you might want to do (for example, mapping GDP values on a world map, etc.).

Make sure to test each step and inspect the results using functions like `head()`, `dim()`, and `summary()` to ensure the transformations are correct. Happy wrangling!

## 5 Creating Beautiful Visuals

In this chapter, we learn to create several types of data visualizations in R: bar charts, line charts, bubble charts, and maps. We will also explore some options to improve your graphics, including an introduction to using D3 (a powerful JavaScript visualization library) through its R wrapper. By the end of the chapter, you should be able to:

1. Create bar charts
2. Produce line charts
3. Generate bubble charts
4. Create maps
5. Integrate D3 visualizations in R for interactive graphics

Data visualization is broadly defined as the act of conveying information through graphical representations of data. But why do we visualize data? One reason is the sheer volume and complexity of data in the modern world. According to a Forbes report, **2.5 quintillion bytes of data are created each day**, and an estimated **90% of the world’s data was generated in just the last two years**. This explosion of data calls for better tools and techniques to **understand**, **analyze**, and **communicate** information effectively. As Google’s chief economist Hal Varian famously said, *“The ability to take data – to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it – that’s going to be a hugely important skill in the next decades...”*. Data visualization is a cornerstone of that skillset, because it leverages our visual perception to make sense of complex information.

We can identify several **goals of data visualization**:

- **Record:** Visualizations can be used to record information (for example, ancient maps or scientific illustrations documented knowledge for future reference).
- **Analyze and Explore:** We use visualizations to explore data, find patterns, and gain insights (often called *exploratory data analysis*).
- **Communicate:** Perhaps most importantly, visualizations help communicate data and findings to others in a clear and compelling way.

To illustrate these goals, consider some historical examples. In the era of exploration, maps were a crucial form of data visualization to record geographical information. One famous example is Alfred E. Pease’s *“Road Map”* (1901) of regions in East Africa (countries of Jidda, Jeel , Liban, Adda, Chor , Wata, Wargi, Arusi, and Koreyu Gallas), which recorded travel

routes and geographic features in great detail. Similarly, the missionary-explorer David Livingstone created maps of central and southern Africa in the mid-19th century; his **surveys enabled large regions to be mapped that had previously been blank on European maps**, thus recording new geographical data and aiding future exploration. Another example of recording information visually is the famous notebooks of Leonardo da Vinci. Leonardo's meticulous sketches of human anatomy and designs of machines are early data visualizations – he used drawings to document observations from nature and engineering. These historical cases show how visualization has long been used to **capture and preserve knowledge**.

Visualization is also fundamental for analysis and invention. In 1786, William Playfair, a Scottish engineer and economist, introduced some of the first *statistical graphics*. He is credited with inventing the **line graph**, **bar chart**, and **pie chart** – basic visual forms that we still use to analyze data today. Playfair's charts allowed people to see economic data (such as imports and exports) over time and compare values at a glance, something not easily done with raw tables of numbers. By translating data into visuals, Playfair essentially created a new language for **analyzing statistical information**.

Visualizations are extremely effective at **communicating** ideas to an audience. A great modern example is the work of **Hans Rosling**, who became famous for turning global statistics into animated visual stories. In his 2006 TED Talk “*The Best Stats You’ve Ever Seen*”, Rosling used interactive bubble charts (through the Gapminder software) to debunk myths about global development. It was “*the first time [many had] seen somebody tell such a compelling story based on numbers... using visualization to present, to educate, and to entertain*”. Rosling's lively presentations — described as having “*the drama and urgency of a sportscaster*” — showed how powerful a well-designed visualization can be in communicating data-driven insights to the general public.

Why else is visualization so important? Human beings are highly visual creatures: a significant portion of our brain is devoted to processing visual information. A well-designed chart can convey a message more clearly and quickly than a table of numbers or a dense paragraph of text. For example, consider a simple data comparison like cholesterol levels over time. You could present the numbers in a table (see Figure 1 below), or you could plot them as a line chart (Figure 2). The line chart immediately reveals the trend — perhaps cholesterol spiking and then falling — far more intuitively than scanning through rows of figures. In general, **graphs often reveal patterns that might be difficult to discern from raw data alone**, as demonstrated in this cholesterol example.

	Males		Females	
Income Group	Under 65	65 or Over	Under 65	65 or Over
0–\$24,999	250	200	375	550
\$25,000+	430	300	700	500

Figure 1: Cholesterol measurements presented as a table of numbers.

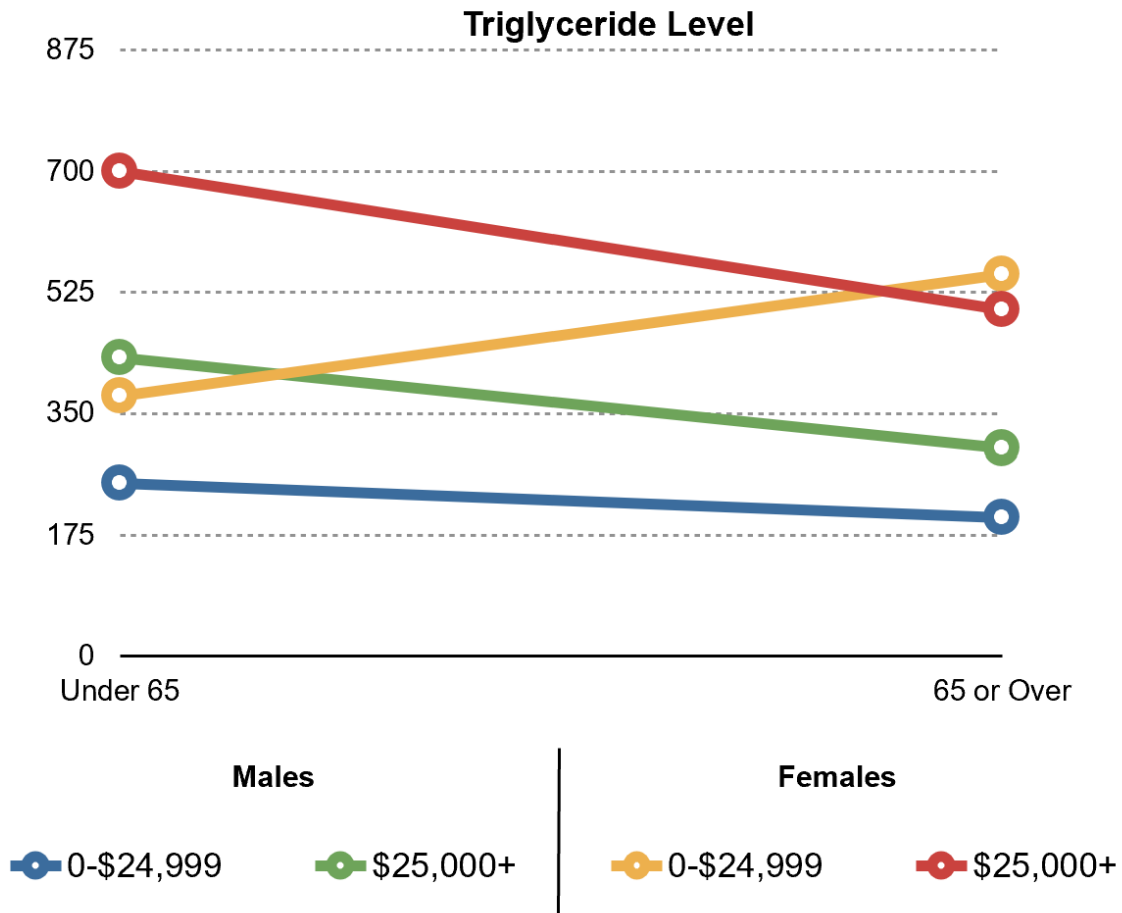


Figure 2: The same cholesterol data presented as a line graph. The visual format makes it easier to spot trends, such as the rise and fall of cholesterol level over time.

Visualization leverages the concept of **external cognition** – using resources outside our mind to enhance our thinking. As cognitive scientist Don Norman aptly put it, “*It is things that make us smart.*” By creating external visual representations (graphs, maps, diagrams), we offload some cognitive work to these artifacts, effectively *augmenting our cognitive capabilities*. In the words of information visualization researcher Stuart Card, “*Visualization is really about external cognition, that is, how resources outside the mind can be used to boost the cognitive capabilities of the mind.*”. In practice, this means a chart or diagram can help us think through a problem by revealing patterns and relationships in the data, freeing us from having to imagine everything in our head.

In summary, data visualization serves as a bridge between **data** and **understanding**. Next, we’ll discuss what makes a visualization effective or ineffective, and outline some key principles

to guide us in creating our own charts.

## 5.1 Foundations

For further reference on the concepts in this section, you may consult the *Data Pipeline with R* book (Warin, 2023). In this chapter, we will get hands-on with creating graphics while also keeping in mind some of the latest thinking in data visualization. There are a few fundamental principles to remember:

- **Graphical Integrity:** The visualization should tell the truth about the data.
- **Use the Right Display:** Choose an appropriate type of chart for the data and message.
- **Keep It Simple:** Avoid unnecessary clutter so the data stands out.
- **Use Color Strategically:** Use color purposefully to highlight or group data, not to decorate.
- **Tell a Story:** Aim to convey a message or insight; don't just show data for data's sake.

We will expand on each of these principles shortly. In essence, an effective visualization **reveals patterns and communicates ideas clearly**, taking advantage of human perception to offload cognitive effort. A poor visualization, by contrast, can obscure the data or even mislead the viewer. Let's explore what makes visualizations effective, then look at some common pitfalls.

### 5.1.1 What makes an effective visualization?

Effective visualizations adhere to the principles listed above, and they leverage our perceptual strengths. They maximize the information communicated, while minimizing distractions or distortions. Here are some guidelines and concepts that help define effective data graphics:

- **Show the data clearly:** As Edward Tufte famously emphasized, “*Above all else show the data.*” Every element in a graph should serve a purpose in presenting the data or supporting interpretation. This leads to the idea of the **data-ink ratio** – the proportion of ink (or pixels) in the graphic that actually represents data. An effective chart **maximizes the data-ink ratio**, meaning most of what you see corresponds to actual data, and very little is non-essential decoration. For example, heavy gridlines, bold borders, or cute background images usually add **no new information**; they are “non-data ink” that can be removed to let the data shine through. Good visualizations aim for a high data-ink ratio (ideally close to 1.0) without sacrificing clarity.
- **Avoid distortion:** The visual representation must accurately reflect the data. This is part of “graphical integrity.” For instance, if one value is twice as large as another, it should appear twice as large on the graph (length of a bar, position on an axis, etc.). Using inconsistent scales or truncated axes can mislead the audience about the true

magnitudes. Tufte introduced the concept of the *Lie Factor* to measure distortion: it's the ratio of the effect shown in the graphic to the actual effect in the data. A Lie Factor far from 1.0 indicates a discrepancy between the visualization and reality. Effective visualizations strive for a Lie Factor of 1 — in other words, no distortion of the data's message.

- **Choose the right chart for the data:** Different displays are suited for different data and questions. For example, if you want to show parts of a whole, a pie chart might be appropriate; to show trends over time, a line chart is usually better. There are many types of charts (bar, line, scatter, histogram, map, etc.), and using the wrong one can confuse the audience. An effective visualization matches the data and the message to an appropriate visual form. A classic mistake, for instance, is using 3D pie charts or fancy donut charts when a simple bar chart would be clearer. Simpler is often better. A good rule of thumb is **show data variation, not design variation** — meaning the visualization's form should be dictated by the data, not by decorative whims.
- **Keep it simple and focus on what matters:** This principle encompasses Tufte's advice to eliminate *chartjunk*. **Chartjunk** refers to all the extra visual elements that do not improve understanding, such as unnecessary illustrations, overly heavy or decorative fonts, gratuitous 3D effects, and the like. Such elements can distract or even mislead the viewer. An effective visualization usually has a clean, minimalistic design: just enough visual elements to communicate the data clearly, and no more. Minimalism for its own sake isn't the goal; the goal is **clarity**. For example, you might lighten or remove grid lines if they aren't needed, or avoid elaborate textures and icons that don't convey new information. Every pixel on the chart should earn its keep. When you look at an excellent chart, your attention should go immediately to the data patterns, not to flashy graphics or irrelevant embellishments.
- **Use color and styling with purpose:** Color is a powerful tool in visualization — it can group related items, highlight important points, or represent additional variables. However, color should be used *strategically*. Too many colors or ill-chosen colors can confuse or distort the message (for instance, using very similar colors for different categories, or choosing colors with unintended emotional connotations). Effective visualizations often use a limited, harmonious color palette and leverage color to guide the eye. For example, using a bold color to highlight one key series in a line chart, while keeping other series in muted tones, can emphasize the story you want to tell. Use color to encode data (categorical or sequential palettes as appropriate) and to draw attention, not just to decorate. The same goes for other stylistic choices like fonts or line widths — they should enhance readability and comprehension. Simpler fonts and clear labels typically work better than ornate designs.
- **Tell a story with the data:** The best visualizations often have a narrative or message. They are *opinionated* in the sense that the designer has a point they want to communicate (while still being truthful to the data). This could be a trend, a comparison, a correlation, or an outlier that forms the “*story*” of the chart. For example, in a time series of



economic data, the story might be that a particular policy change coincided with a rise in employment. A neutral presentation would just show the line, but a storytelling approach might annotate the chart with the date of the policy change and perhaps use a highlight color to mark the post-policy period. Another example: if you're visualizing population growth, you might structure the graphic to build up to a surprising finding (this is analogous to a story's climax). As story expert Robert McKee wrote, "*A story is not an accumulation of information strung into a narrative, but a design of events to carry us to a meaningful climax.*" In visualization, this means designing the graphic (and any accompanying text or animation) to make a meaningful point, rather than just plotting data without context.

Let's consider an example of storytelling in visualization versus a neutral approach. The figure below (Figure 3) shows two charts of the **same data** (annual fatalities in a certain conflict), taken from a blog discussion by data visualization expert Andy Cotgreave. The **left chart** is titled "Iraq's Bloody Toll" and uses bold red coloring and an inverted y-axis (bars falling from the top), whereas the **right chart** has a neutral title and uses a calm blue color. Both charts plot the same numbers, but they evoke very different emotional responses and even suggest different narratives. The left one, with its alarming red downward bars and dramatic title, emphasizes the magnitude of deaths and gives a sense of tragedy. The right one, with blue upward bars and a more neutral framing (focusing on decline of violence), conveys a more subdued or even optimistic view. Cotgreave's point was that by simply changing the **title**, **color**, and **orientation** of a chart, you can **tell very different stories with the same data**. Neither version is lying — they both show the data accurately — but each is *opinionated* in its own way. This example underscores the importance of being mindful of how design choices (like color and wording) influence the message of a visualization.

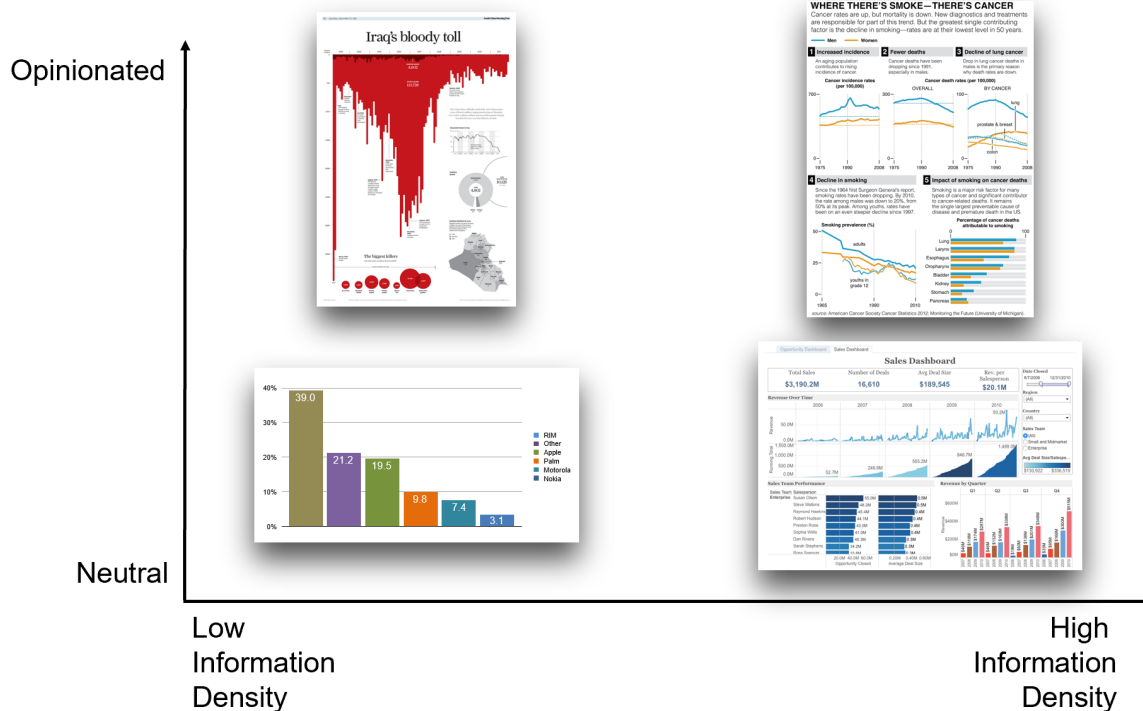


Figure 3: Two visualizations of the same data (annual fatalities in a conflict) leading to different impressions. The left uses red bars dropping downward from the title “Iraq’s Bloody Toll,” creating a dramatic, alarming narrative. The right uses blue bars rising upward and a neutral title highlighting the decrease in fatalities, creating a calmer, more optimistic impression. This demonstrates how color, orientation, and titling can be used strategically to tell different stories with the same underlying data.

In designing visualizations, always consider your **audience**. An effective visualization for a group of domain experts might look different from one aimed at the general public. Experts might appreciate more detail, precise annotations, or novel visualization forms, whereas non-experts might need clearer labeling, more context, or simpler chart types. Think about what background knowledge your audience has, and what you want them to take away from the chart. Adapt the level of complexity and the framing of the story accordingly. For instance, if you’re showing a chart of financial data to an audience of economists, you might include inflation-adjusted values and detailed axis labels; but if you’re showing it to a general audience, you might use a straightforward nominal trend line and a brief note in plain language about what happened in certain years. In all cases, the visualization should aim to **engage** the audience – through clarity, through visuals that draw interest (perhaps using color or interactivity) – and to communicate the intended insight effectively.

Aesthetics do play a role in engagement. Research in design suggests that **attractive things are often perceived as more useful** and can make users more tolerant of minor difficulties. A visually appealing chart (one that is clean, nicely formatted, with a good use of color and

layout) can invite the audience in. Elements of style communicate subtle messages: they reflect the **tone** (professional, playful, academic, etc.), and can even signal the identity or brand of the creator. For example, a tech company’s data dashboard might use a sleek modern design with the company’s branding colors, whereas a data journalism piece might use a design that matches the style of the publication. **Playfulness** in design – such as interactive features or imaginative illustrations – can encourage exploration and keep viewers engaged. **Vividness** (using a striking visual metaphor or memorable imagery) can make a visualization more memorable. These aesthetic considerations should complement, not overshadow, the data. The ultimate judge of an effective visualization is whether it **successfully communicates the intended information** and leaves a clear impression with the audience.

To recap the characteristics of effective visualizations:

- They have **graphical integrity** (no misleading representations; faithful to the data).
- They use an appropriate and **simple display** (right chart type, minimal extraneous elements).
- They have a high **data-ink ratio** and minimal **chartjunk** (no unnecessary clutter).
- They **use color and design thoughtfully** to enhance understanding (grouping, highlighting, consistent styles).
- They often **tell a story or highlight an insight**, guiding the viewer to the important points.

Now that we’ve seen what makes a visualization good, let’s consider how they can go wrong.

### 5.1.2 What makes an ineffective visualization?

Just as good visualizations can enlighten, bad ones can confuse or mislead. Here are some common pitfalls that lead to ineffective (or even infamous) visualizations:

- **Misleading scales or proportions:** One of the worst sins is manipulating the axes or visual proportions to exaggerate or downplay changes. For example, starting a bar chart’s y-axis at a high value (instead of zero) can make differences look much larger than they are, violating the proportionality principle of graphical integrity. Another example is using pictograms or 3D objects where the area or volume doesn’t match the data (e.g., drawing people icons to represent quantities but scaling their height in a way that also changes their width, thus visually inflating the difference). These practices distort the viewer’s perception of the data.
- **Excessive chartjunk and clutter:** When a graph is loaded with unnecessary elements, the core message gets lost. Busy backgrounds, too many grid lines, decorative illustrations, or overly complex legends can overwhelm the viewer. Ineffective visuals often try to be *too fancy*, using 3D effects, shadows, or bright clashing colors that add no value (or even impede comprehension). Remember that **chartjunk consists of all visual elements not necessary to comprehend the information**. A cluttered chart

forces the reader to spend extra mental effort filtering out noise, which is frustrating and counterproductive.

- **Wrong choice of visualization:** If the visual form doesn't fit the data, the result can be confusing. Imagine using a pie chart to show a trend over time (pie charts are not meant for time series), or using a line chart for categorical comparisons that have no inherent order. The viewer might struggle to grasp the point because the visualization isn't suited to the message. Each type of chart has strengths and limitations; using them inappropriately is a recipe for an ineffective graphic.
- **Lack of context or labeling:** A chart without clear labels, title, or explanation can be misleading or meaningless. If viewers have to guess what the axes represent or what units are used, the visualization fails to communicate. An ineffective visualization might show data but omit important context (such as labeling a spike in a time series with the event that caused it), leaving the audience puzzled about significance. Always include enough context (titles, axis labels, legends, annotations) to make the data understandable.
- **Overcomplicating when simplicity would do:** Sometimes designers make a chart overly complex by introducing too many variables or combining too many chart types, when splitting it into multiple simpler charts would communicate more clearly. Stacking too much information into one visual can overload the reader's working memory.

A notorious example of an ineffective visualization was a chart once aired on a news network that showed a **non-sensical y-axis** (with percentages that added up to far above 100%, for instance) or distorted scaling to exaggerate a political point. This chart became an internet meme for how blatantly it violated basic principles of bar chart design. In general, any chart that makes a reader say “*Wait, that doesn't look right...*” is likely ineffective.

For a dose of humor (and caution), you can visit the website **viz.wtf**, which curates real-world examples of “bad” visualizations. These include gems like bar charts where the bars have been rotated or reordered in bizarre ways, pie charts with too many slices and garish colors, maps with meaningless coloring, and other mishaps. Such examples serve as a reminder of what *not* to do. They underscore how easy it is to be misled by poor design, and why adhering to the principles of integrity and simplicity is so important. When designing your own visualizations, keep these pitfalls in mind and always ask yourself: *Is my chart clear? Is it truthful? Is it necessary and focused on the data?* If so, you're on the right track.

Having covered the theoretical foundation, we will now **get our hands dirty with practical visualization in R**. We'll start with some basic chart types (bar, line, bubble) using the popular **ggplot2** package, then look at creating maps, and later explore an interactive visualization with D3. Throughout, remember the principles we discussed and try to apply them in your plotting choices.

## 5.2 Bar chart

Let's begin with a simple bar chart in R using **ggplot2** (a powerful visualization package based on the *Grammar of Graphics*). Recall that in Lab 2 we created a dataset called **dataCanadaFull** (containing, say, number of employees by industry and year in Canada). We also transformed it into a “long” format suitable for ggplot2. Here we'll use that long-form dataset, **dataCanadaFullLong**. (If you have the data in wide format, with separate columns for each industry or category, you would first need to reshape it into a long format, but in our case we already did that in the previous lab.)

First, load the data (assuming it's stored as a CSV file):

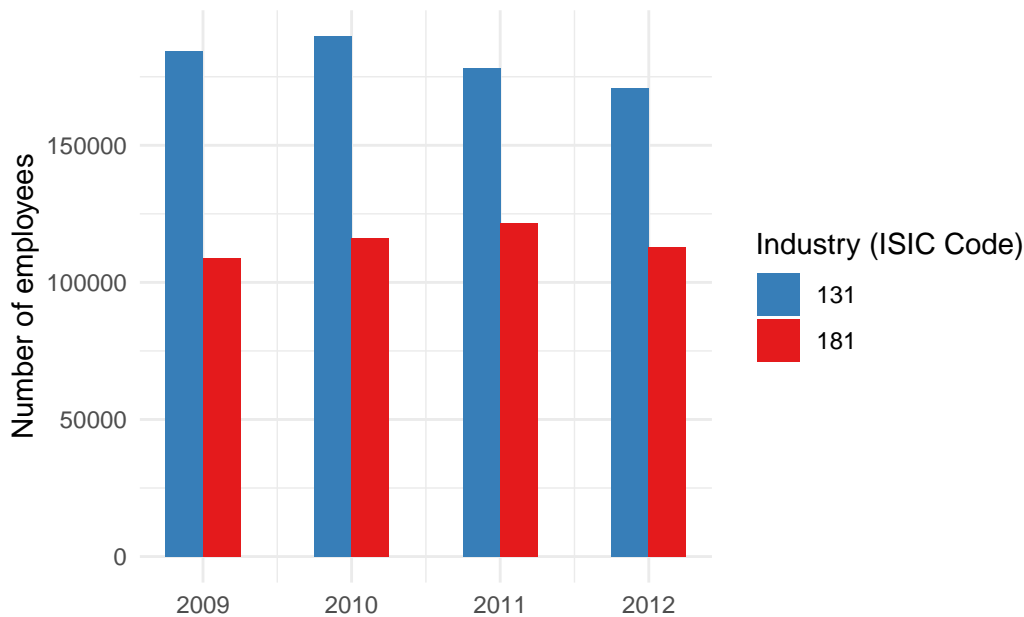
```
dataCanadaFullLong <- readr::read_csv("./data/dataCanadaFullLong.csv")
```

Our dataset has columns such as **year**, **isicCode** (an industry code), and **value** (number of employees). We should ensure that **isicCode** is treated as a categorical variable (factor or character) rather than numeric, otherwise ggplot might treat it as a continuous variable. We can convert **isicCode** from numeric to character:

```
dataCanadaFullLong$isicCode <- as.character(dataCanadaFullLong$isicCode)
```

Now, let's produce a bar chart of the number of employees by year, with different bars for each industry (identified by ISIC code). We will map **year** to the x-axis and **value** (number of employees) to the y-axis. Different industries will be indicated by different colors (fill) of the bars. We'll use a grouped bar chart (also known as side-by-side bars) so that for each year, we see a cluster of bars (one per industry). Here is the ggplot2 code to do this:

```
# Produce a bar chart
library(ggplot2)
library(ggthemes)
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, fill = isicCode)) +
  geom_bar(stat = "identity", width = 0.5, position = "dodge") +
  xlab("") +
  ylab("Number of employees") +
  labs(fill = "Industry (ISIC Code)") +
  theme_minimal() +
  scale_fill_brewer(palette = "Set1", direction = -1)
```



Let's break down this code:

- `ggplot(data = dataCanadaFullLong, aes(...))` initializes the plot with our dataset and specifies the **aesthetic mappings**. We map `x = year`, `y = value`, and `fill = isicCode`. This means year will determine the horizontal position of bars, value will determine their height, and the fill color will correspond to the industry code.
- `geom_bar(stat = "identity", ...)` adds the bar geometry. By default, `geom_bar` in `ggplot` uses `stat="count"` (which counts rows), but here we already have a numeric value to plot, so we use `stat="identity"` to tell `ggplot` to use the values as they are. We set `width = 0.5` to make the bars a bit thinner (half the default width), and `position = "dodge"` to place bars for different industries side by side rather than stacked. "Dodge" separates the bars along the x-axis so that within each year, bars for each industry are next to each other.
- `xlab("")` and `ylab("Number of employees")` set the x-axis label and y-axis label. We leave the x-axis label blank in this case (since "year" might be self-evident or we might rely on the axis tick labels).
- `labs(fill = "Industry (ISIC Code)")` changes the legend title for the fill aesthetic to something more descriptive (instead of the default "isicCode").
- `theme_minimal()` applies a clean minimal theme to remove distractions like background grids. This theme gives a nice, uncluttered look by default.
- `scale_fill_brewer(palette = "Set1", direction = -1)` applies a ColorBrewer palette for the fill colors. "Set1" is a qualitative palette with distinct colors, and `direction = -1` reverses the palette (this is optional). ColorBrewer palettes are well-designed sets of colors for categorical data, ensuring enough contrast and aesthetic appeal.

The result should be a bar chart where the x-axis has years, y-axis shows the number of employees, and within each year there are multiple colored bars (each color corresponding to an industry code). The legend indicates which color corresponds to which industry code. This bar chart allows us to compare values across industries for each year, and also see trends for each industry over the years by comparing the bars of the same color across the x-axis.

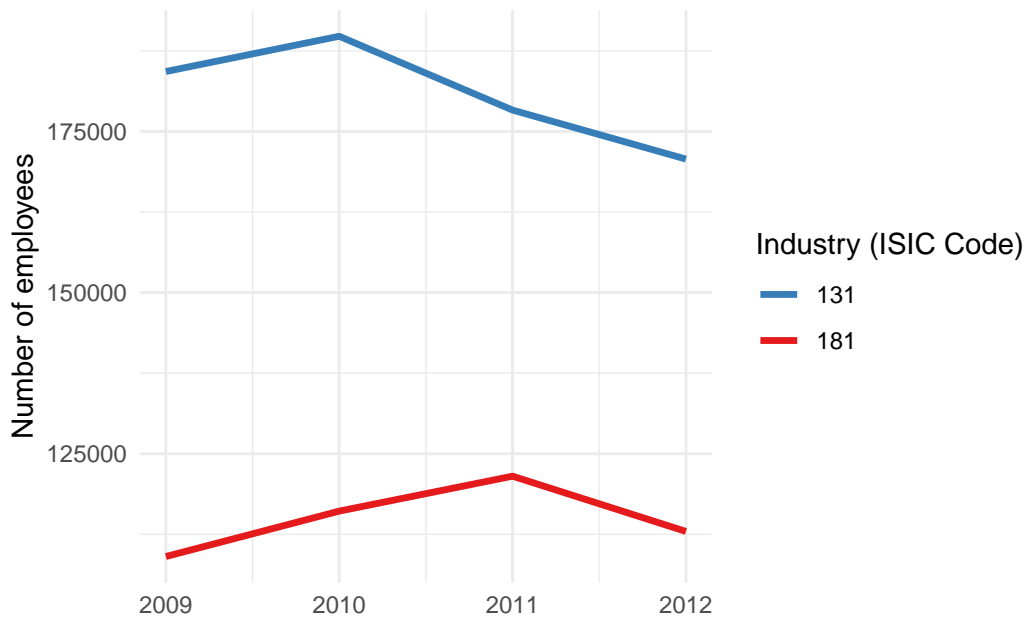
*(If the bars appear cluttered or labels overlap, one can adjust the width or use `position_dodge(width = 0.7)` for fine control, or maybe rotate x-axis labels if they are long. In our simple example with year, that likely isn't an issue.)*

## 5.3 Line chart

Bar charts are great for discrete comparisons, but if we want to emphasize trends over time, a line chart is often more appropriate. Let's visualize the same data with a **line chart**, where each industry will be a separate line over the years.

We use a similar ggplot structure, but now use `geom_line()` instead of bars:

```
library(ggplot2)
library(ggthemes)
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, color = isicCode)) +
  geom_line(size = 1.2) +
  xlab("") +
  ylab("Number of employees") +
  labs(color = "Industry (ISIC Code)") +
  theme_minimal() +
  scale_color_brewer(palette = "Set1", direction = -1)
```



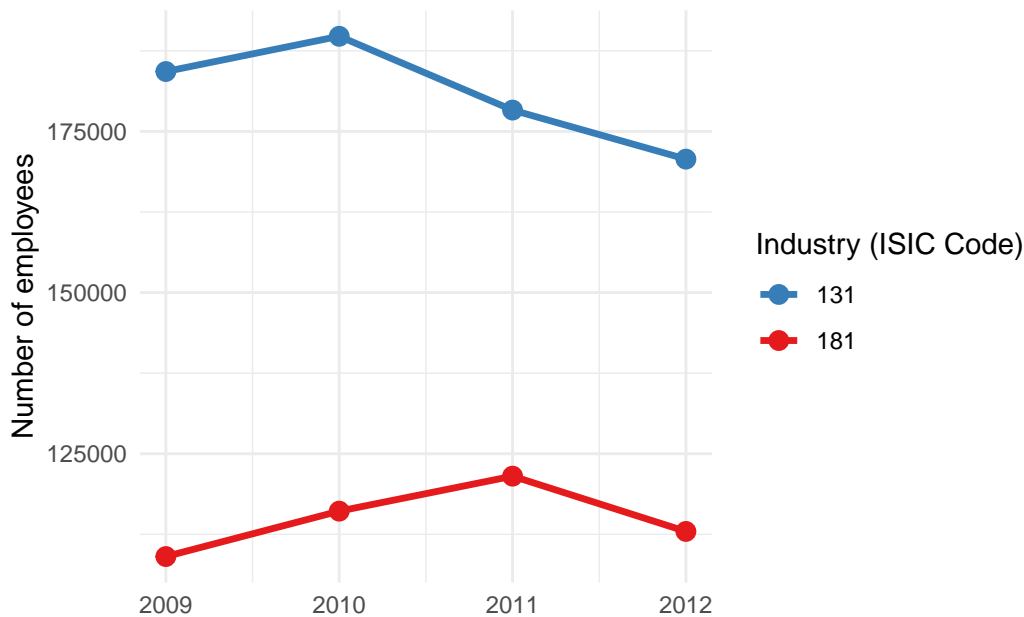
Here, we map `color = isicCode` (instead of `fill`) since lines are typically distinguished by color (and/or line type). Each industry code will produce a line of a different color. We increased `size = 1.2` to make the lines a bit thicker for visibility. We used `labs(color = ...)` to label the legend appropriately.

This plot will have the years on the x-axis and number of employees on the y-axis, with multiple lines (one per industry). It's excellent for seeing how each industry's employment changed over time, and for comparing their trends. For instance, you might observe that some industries have an upward trend, others downward, and some might intersect or diverge at certain points in time.

We can further improve this line chart by adding points to each year, to emphasize the data points and make the lines easier to read at specific values. To do so, we can add another geometry layer: `geom_point()`. For example:

```
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, color = isicCode)) +
  geom_line(size = 1.2) +
  geom_point(size = 3) +
  xlab("") +
  ylab("Number of employees") +
  labs(color = "Industry (ISIC Code)") +
  theme_minimal() +
  scale_color_brewer(palette = "Set1", direction = -1)
```





We simply added `geom_point(size = 3)` which puts a point at each data point (year, value). The size 3 makes the points reasonably visible. Now each yearly observation is marked with a dot, and the dots are connected by lines for each industry. This combination can be helpful to see exact values (the points) and the overall trend (the connecting lines). It also helps if there are missing years in the data – the line would skip, but points would still mark the individual observations.

Line charts with multiple categories can get busy, so ensure the colors are distinct (ColorBrewer’s Set1 is usually good up to about 6-8 categories). If you have more categories, you might use different line types or facets, but that’s beyond our current scope.

## 5.4 Bubble chart

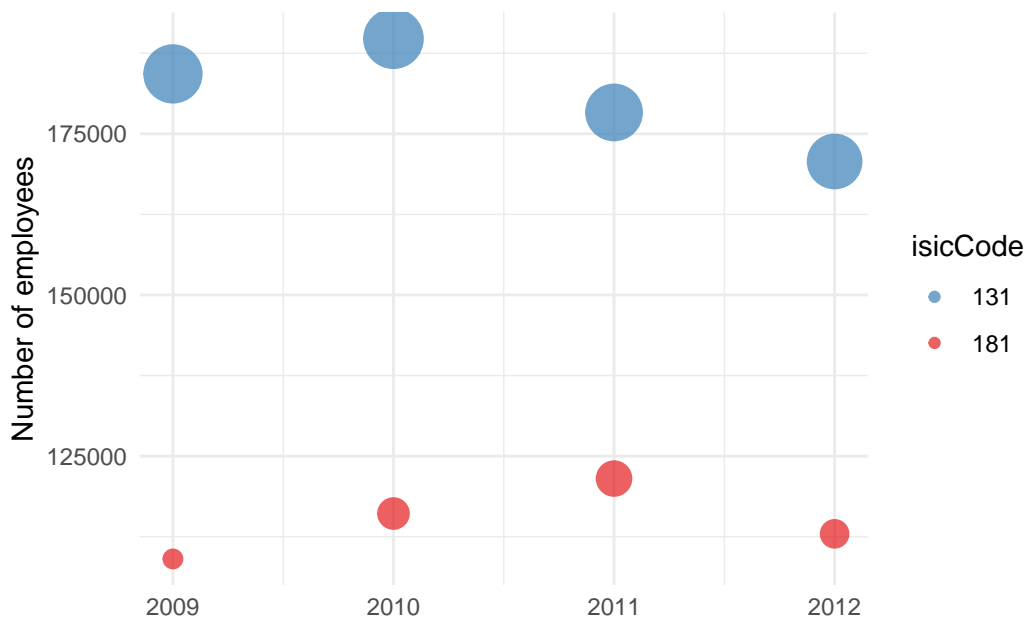
Next, let’s create a **bubble chart**. In data visualization, “bubble chart” often refers to a scatter plot where the points are sized (and sometimes colored) according to a third variable. In our dataset, we could treat **year** as one variable (x-axis) and **value** (employees) as another (y-axis), and use the **size of the point** to also represent the value (which is a bit redundant in this case) or perhaps another variable if we had one. A classic use of bubble charts is to plot (x, y) for different categories and use the bubble size to indicate magnitude of some third measure, and color to indicate a category.

Our `dataCanadaFullLong` essentially has year and value for different industries. Plotting year vs value with points colored by industry is basically a scatter/line chart as above. To demonstrate the bubble concept, we’ll map the same **value** to the size of points. This will make points (bubbles) larger when the number of employees is larger. This is somewhat redundant

with the y-axis (which already shows the magnitude), but it's just for practice. In real use, you might have something like  $x = \text{GDP per capita}$ ,  $y = \text{life expectancy}$ ,  $\text{size} = \text{population}$  (that would be a typical bubble chart scenario, a la Hans Rosling's examples).

Here's how to create a bubble chart using ggplot2:

```
library(ggplot2)
library(ggthemes)
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, color = isicCode)) +
  geom_point(aes(size = value), alpha = 0.7) +
  xlab("") +
  ylab("Number of employees") +
  theme_minimal() +
  scale_color_brewer(palette = "Set1", direction = -1) +
  scale_size_continuous(range = c(3, 10)) +
  guides(size = FALSE)
```



A few things to note in this code:

- We still map `color = isicCode` to differentiate industries by color.
- Inside `geom_point()` we add `aes(size = value)`. This means each point's size will correspond to the value (number of employees). Higher values produce larger points.
- We set `alpha = 0.7` to make the points semi-transparent. This can help with overplotting (if points overlap, transparency makes overlaps visible and less dominant).

- `scale_size_continuous(range = c(3, 10))` is used to adjust the range of point sizes. By default, ggplot might choose a range, but we specify that the smallest value will have size 3 and the largest will have size 10 (and intermediate values scaled in between). This keeps bubbles within a reasonable visible range.
- `guides(size = FALSE)` is used to turn off the legend for the size aesthetic. In this case, a size legend isn't very useful (it would show a continuum or a few example sizes). Since the y-axis already gives a sense of the values, we decide to omit a redundant legend for bubble size. The color legend remains to identify industries.

The resulting chart would show points at positions (year, value) for each industry, color-coded by industry. Additionally, the points are drawn as bubbles whose area (technically diameter in ggplot's sizing) reflects the value. In years where an industry had a particularly large number of employees, that data point will appear as a larger bubble. This kind of chart is more eye-catching and can emphasize where "big" values are, though when using size to encode data, one must be careful: human perception of area is not linear, and smaller differences can be hard to see. Nonetheless, it's a fun and informative visualization when used appropriately (especially if the third variable adds new information).

Now you are able to produce basic **bar charts**, **line charts**, and **bubble charts** in R using ggplot2. Each of these chart types can be further refined and customized (with titles, annotations, theme adjustments, etc.), but this gives a solid starting point for visualizing data in a simple and effective manner.

Before moving on, think about which type of chart best suits the message you want to convey in your data. Bar charts are great for comparing discrete categories, line charts for trends over continuous variables (like time), and bubble charts for adding a third dimension of data to two-dimensional plots (especially when that third dimension can be thought of as a magnitude or weight of each point).

## 5.5 Maps

In addition to standard statistical charts, **maps** are an important class of visualization, especially for any data that has a geographical component. Creating maps in R can be done in many ways; here we will show a very basic approach using ggplot2's ability to draw polygons. We will start with a simple world map, then see how to zoom into a region.

To draw maps, we need geographical data (coordinates for borders). The **maps** package (and its ggplot2-friendly interface) provides map data frames for various regions. We can use `map_data("world")` to get a dataframe of world map coordinates (this comes from the *maps* package and is made easily accessible via ggplot2).

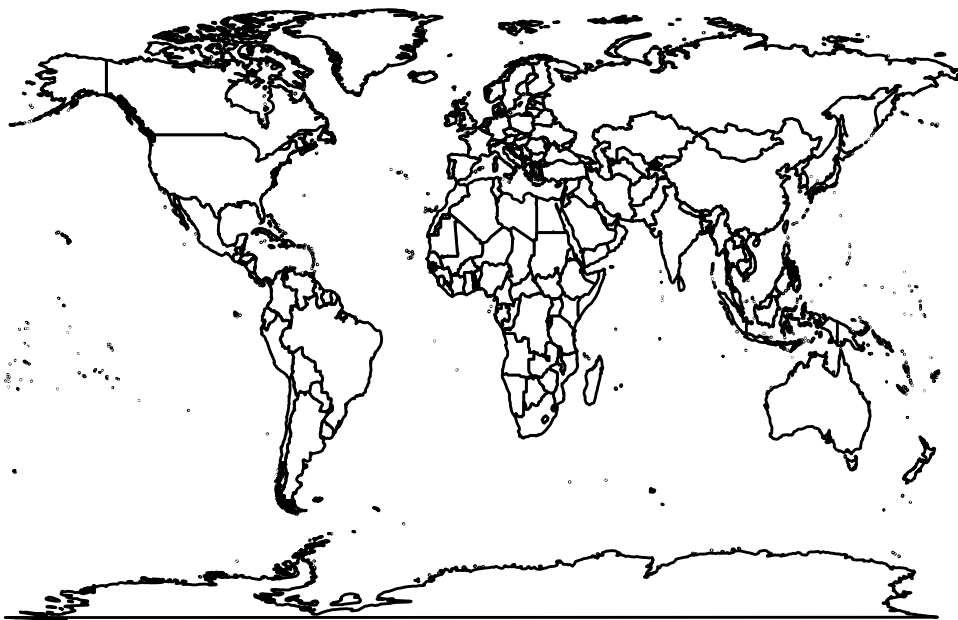
First, ensure you have ggplot2 loaded (we already did). Then retrieve the world map data:

```
world <- map_data("world")
```

The `world` object now contains a data frame with columns like `long` (longitude), `lat` (latitude), `group` (an identifier for each polygon, since world map consists of many polygon pieces for countries and islands), and `region` (the country name for each polygon).

Now, to plot this as a map, we use `geom_polygon()` layer in `ggplot`, mapping longitude and latitude appropriately and grouping by the polygon group. We can fill each country with some color (or leave blank) and draw borders. A very basic world map (with countries outlined) can be drawn as:

```
ggplot(data = world, aes(x = long, y = lat, group = group)) +  
  geom_polygon(fill = "white", color = "black") +  
  theme_void()
```



Let's explain this:

- We use `ggplot(data = world, aes(x = long, y = lat, group = group))`. We map longitude to x-axis and latitude to y-axis. We also specify `group = group` so that `ggplot` knows which points belong to the same polygon (country). The `map_data` function already provided a `group` variable that ensures the polygon drawing for each country (and sub-polygon) is correct.
- `geom_polygon(fill = "white", color = "black")` draws the polygons. We choose `fill = "white"` to make each country's interior white (you could also choose another

color or even map a variable to fill for a choropleth map, but here we're just making a blank map). We set `color = "black"` to draw the border of each polygon in black (these are country boundary lines).

- `theme_void()` is a ggplot theme that produces a blank background with no axes, grid, or other decorations – perfect for maps. We don't need coordinate axes or tick marks for a basic world outline, so `theme_void` gives a clean map.

The result is a simple world map outline. You should see all the continents and countries in white with black outlines on a gray (or white) background. This kind of base map is useful if you want to overlay data on it (for example, coloring countries by some value, or placing points at certain coordinates).

Often, we are interested in a specific region rather than the entire world. To focus on a region (say a continent or a set of countries), one approach is to **filter the world map data** to only those regions. For example, suppose we want a map of the Americas. We can filter the `world` data to include only countries in North and South America. The `region` column in `world` contains country names. We can use a subset of those names.

For demonstration, let's create a map of (most of) the Americas: we'll include North, Central, and South American countries. We can make a vector of country names we want, or use `%in%` for a range. For simplicity, let's use a selection of countries from Canada at the top to Argentina at the bottom. (We have to be mindful that `map_data("world")` might use some abbreviations or specific names; for example, USA is "USA", not "United States", etc.)

We'll do:

```
# Retrieve world map data if not done already
world <- map_data("world")

# Filter for Americas region countries
americas <- subset(world, region %in% c("Canada", "USA", "Mexico", "Brazil", "Colombia",
    "Argentina", "Peru", "Venezuela", "Chile", "Guatemala",
    "Ecuador", "Bolivia", "Cuba", "Honduras", "Paraguay",
    "Nicaragua", "El Salvador", "Costa Rica", "Panama",
    "Uruguay", "Jamaica", "Trinidad and Tobago", "Guyana",
    "Suriname", "Belize", "Barbados", "Saint Lucia",
    "Grenada", "Saint Vincent and the Grenadines",
    "Antigua and Barbuda", "Saint Kitts and Nevis"))
```

The `subset` command filters the `world` data frame to only rows where the `region` is one of those listed (a selection of countries in the Western Hemisphere). Now `americas` contains map data for those countries.

We can plot this similarly:

```
ggplot(data = americas, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "white", color = "black") +
  coord_fixed(xlim = c(-180, -35), ylim = c(-60, 90), ratio = 1.1) +
  theme_void()
```



New here is the `coord_fixed()` function. `coord_fixed` fixes the aspect ratio of the coordinates, which is important for maps to not look stretched. By specifying `ratio = 1.1` (which roughly corresponds to lat/long degree aspect for that region) and giving `xlim` and `ylim`, we zoom the view to the Americas. In this case, we set `xlim = c(-180, -35)` (longitude from 180°W to 35°W roughly covers Alaska to the eastern tip of Brazil) and `ylim = c(-60, 90)` (latitude from 60°S to 90°N covers from the tip of South America up to the Arctic). Adjusting these limits and ratio can center the map nicely. The result should be a map of the Americas (North and South America) in the same style: countries outlined.

Again, we used `theme_void()` to keep the map clean. If we were making a more detailed thematic map, we might add titles or a legend (for example, if coloring countries by a data variable). But here our goal is just to demonstrate creating maps.

**Note:** The functions `map_data("world")` and `geom_polygon` approach is simple but not the most robust for serious mapping tasks. There are dedicated packages like `sf` (simple features) for handling spatial data, which integrate well with `ggplot2` for more complex maps (and projections, etc.). However, using `map_data` is quick and fine for basic world or country maps in a pinch.

With what we've covered so far, you can load data from a CSV and visualize it with several basic chart types, as well as draw simple maps. Next, we will introduce a tool that can help

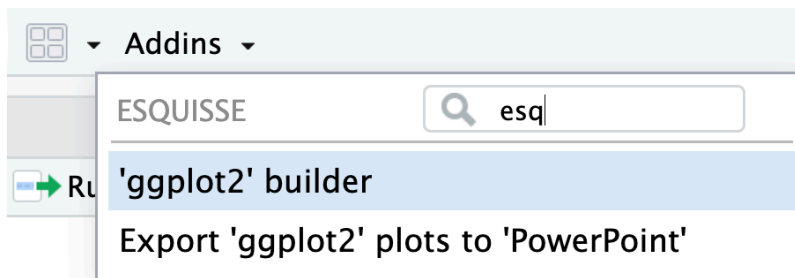
you build `ggplot2` graphs using a graphical interface, which can be handy if you are not yet comfortable with the code or want to explore different designs quickly.

## 5.6 Esquisse

Now that you know the “grammar of graphics” and have seen how to create plots with `ggplot2` code, let’s look at an R add-in called **esquisse**. Esquisse provides a **drag-and-drop interface** for building `ggplot2` plots. It can be a great way to experiment with different plots or to quickly generate the code for a plot without writing it manually. Essentially, it’s a GUI layer on top of `ggplot2`.

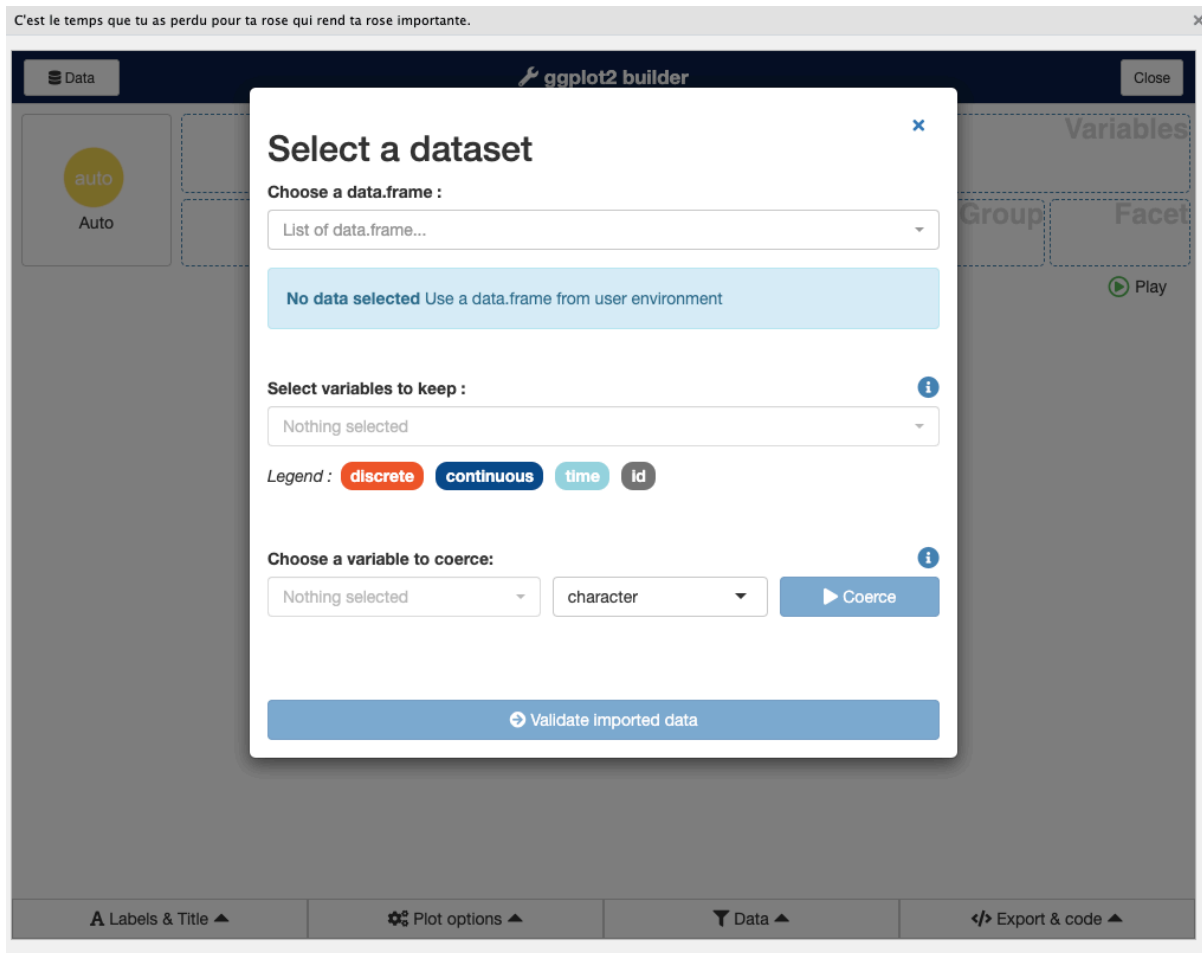
To use `esquisse`, you need to have the package installed and be running RStudio (because it’s provided as an RStudio Addin). Assuming it’s installed, follow these steps:

**Step 1:** In RStudio, click on the *Addins* menu button. In the dropdown, look for “`ggplot2` builder” or something referencing `esquisse`. (It might be listed as “Esquisse” or “`ggplot2` builder”.) Click on that option.



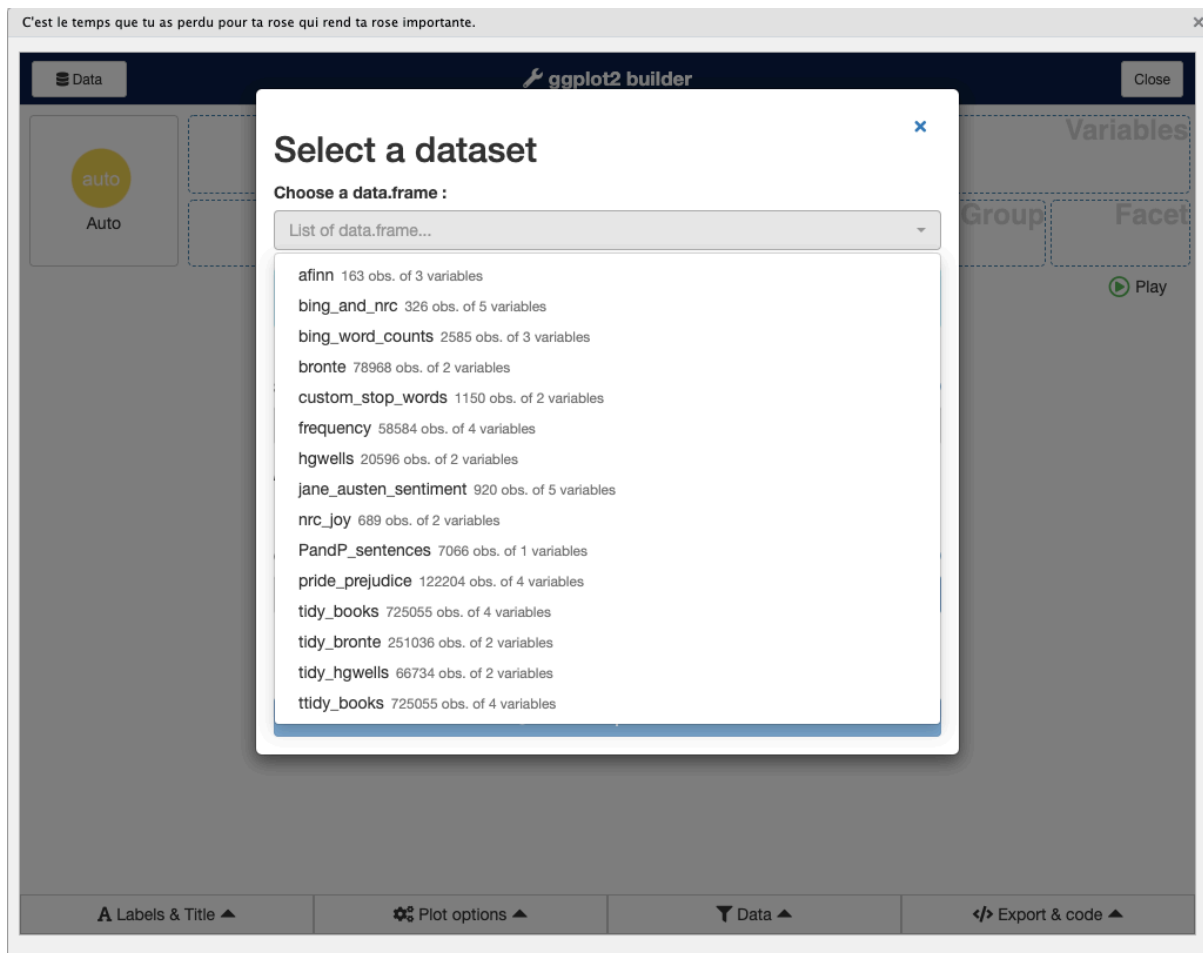
This will open the `esquisse` GUI in a new window or pane.

**Step 2:** You should see the Esquisse interface pop up, which is divided into sections. On the left, you’ll have a panel to select a dataset and variables; on the right, an empty plotting area that will show a preview of your plot, and at the bottom, tabs including one for code.

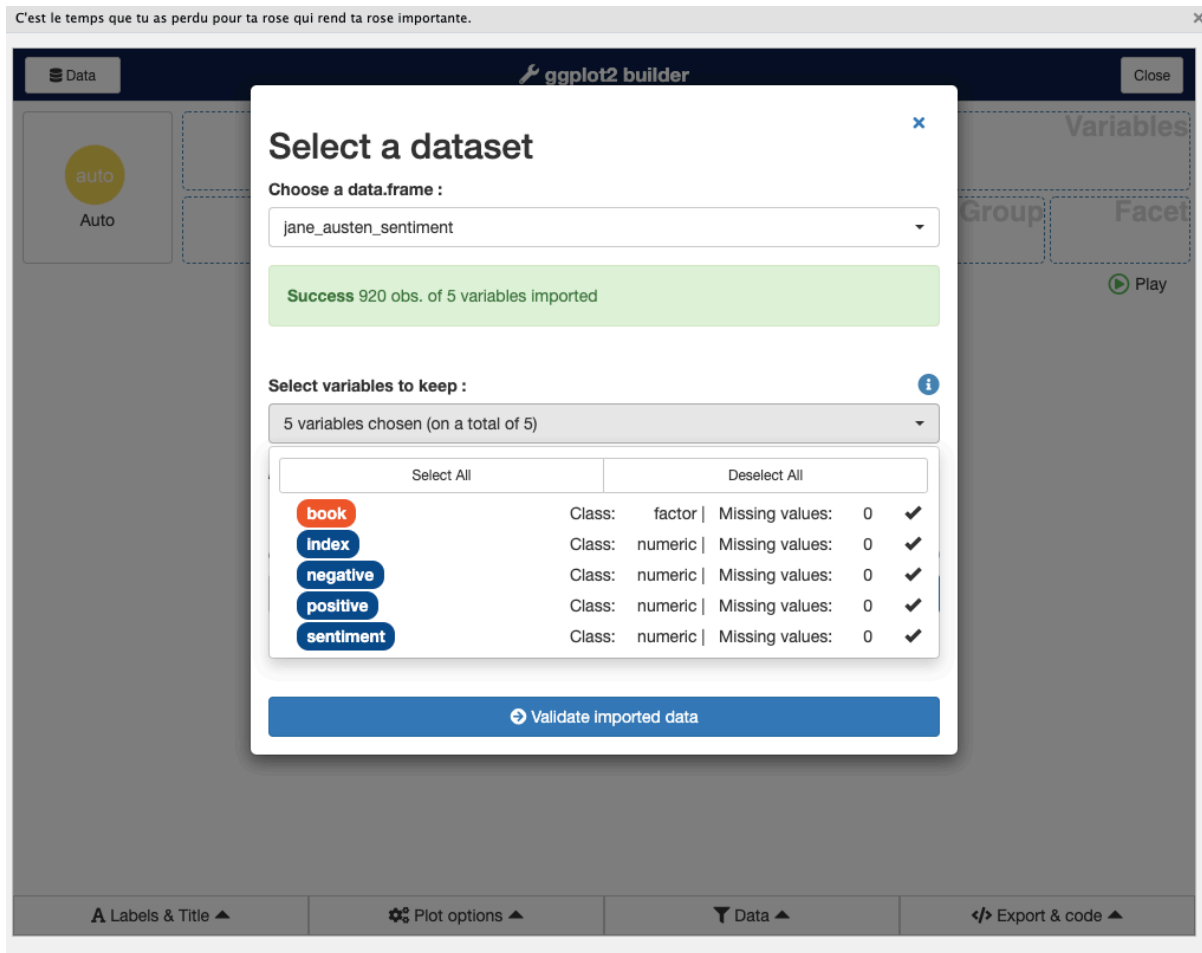


**Step 3:** At the top of the Esquisse interface, there is a dropdown that says “List of data frames...”. Click that and you will see the data frames currently available in your R session. Choose one (for example, choose the `dataCanadaFullLong` dataset we have been using, if it’s loaded in your environment).

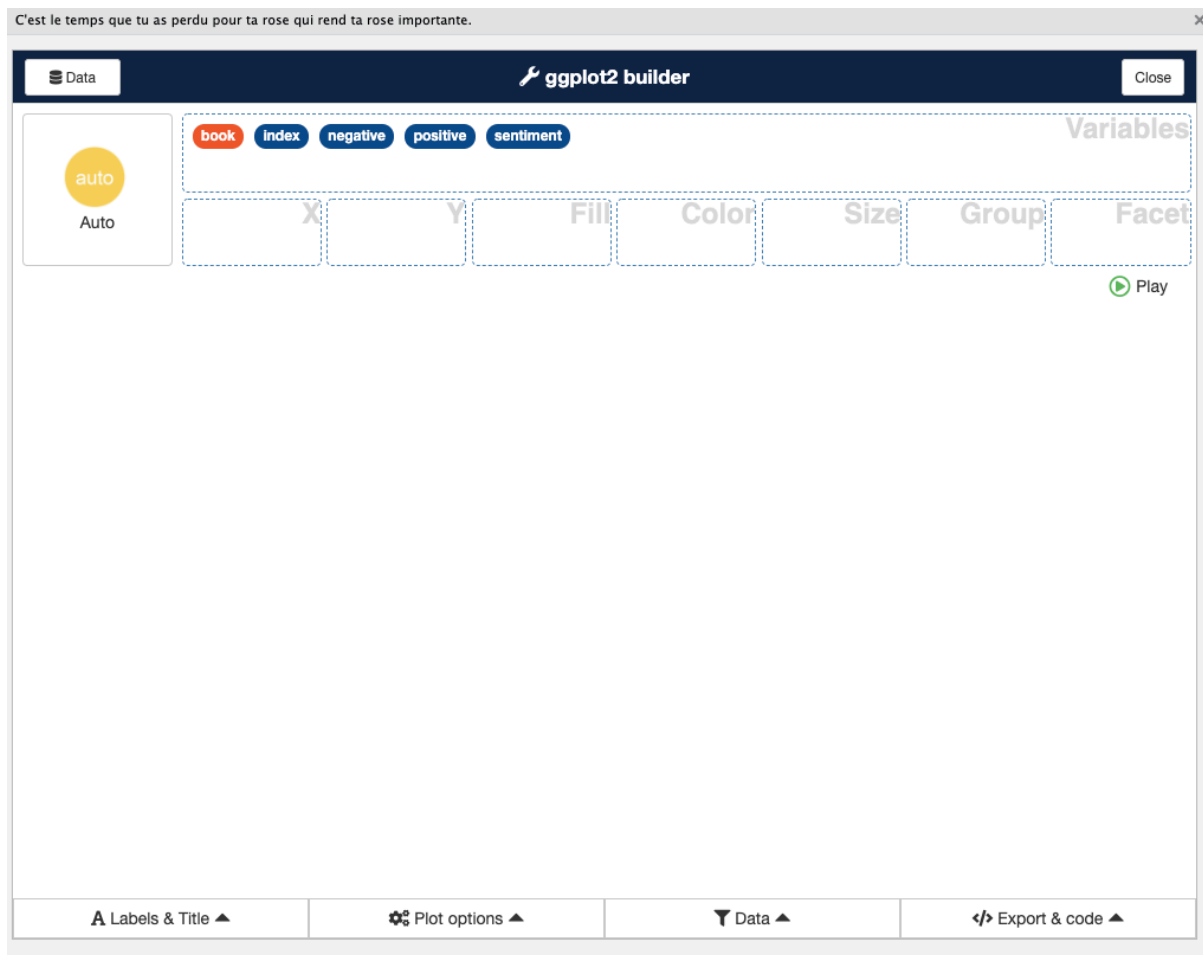




**Step 4:** After selecting the dataset, Esquisse will allow you to select which columns to use. By default, all columns are selected. If you want to limit which variables appear for drag-and-drop (for example, if your dataset has many columns and you only care about a few), you can click the “*Choose variables*” button (often labeled “*Pick variables*”) and select only the ones you plan to use. Otherwise, just keep all.



**Step 5:** Now you will see the main drag-and-drop interface. There will be boxes or drop zones labeled X, Y, Color, Fill, Size, etc., corresponding to aesthetic mappings, and a list of your variables to the left. You can now drag a variable name into one of these aesthetic boxes. For example, drag **year** into the **X** box, and **value** into the **Y** box. If you want a certain variable to define color groups, drag it into Color or Fill (for points/lines use Color, for bars either can work but typically Fill is used). For a bar chart, you might drag **isicCode** into Fill. You also have a dropdown to select the **geom (geometry)** type (e.g., bar, line, point).



For example, to replicate the bar chart we coded earlier:

- Drag `year` to X,
- Drag `value` to Y,
- Drag `isicCode` to Fill.
- Then find where it says *Geom* or *Visualization type* and select “Bar” (and ensure it’s set to not count automatically; Esquisse usually handles setting `stat="identity"` if you put a numeric on Y).
- You might also set the position to “dodge” if multiple fill categories (there may be an option for position adjustments).
- On the right side, a preview of the plot will appear.
- You can further tweak things like adding labels or changing palette under the **Plot Options** in esquisse.

The Esquisse interface also has panels for **Data** (to apply filters to your dataset within the GUI), **Labels & Title** (to set axis labels, plot title, etc.), and **Plot Options** (to change

themes, color palettes, legends). For instance, you can easily switch the color palette to one of the ColorBrewer sets or a viridis palette, or change the theme from minimal to classic, all with a click, and see the effect immediately. This is a convenient way to try out different looks without writing code.

As you make selections and adjustments, Esquisse is generating the corresponding ggplot2 code in the background. You can click on the **Code** tab (often at the bottom of the Esquisse window) to see the ggplot code. This is very useful for learning ggplot: you can literally **copy the code** from here and use it in your script. It will include all the mappings, layers, and theme settings you specified through the GUI.

Once you are happy with the plot (or at least have a good starting point), you can close the Esquisse addin. If you copied the code, you can paste it into your R script or RMarkdown document and further refine or reuse it. Even if you're comfortable writing ggplot code from scratch, Esquisse can be a great way to quickly prototype a visualization or discover how to implement a certain option (for example, you might not remember the exact code to remove a legend or change a palette, but Esquisse will show it in the generated code).

In summary, **Esquisse** provides an interactive way to build ggplot2 graphs:

- It's helpful for beginners to learn the syntax by seeing code generated.
- It's also a time-saver for experienced users to test different themes or aesthetics quickly.
- Remember that it's generating ggplot2 code, so everything is reproducible – you're not stuck with a manual process; you can take the code and integrate it into your project.

Now we have covered a range of visualization techniques in R, from basic charts to maps, and even a GUI tool for plotting. In the next section, we'll step into the realm of interactive and advanced visualizations with D3, a JavaScript library, and see how we can use it from R.

## 5.7 Interactive Visualizations with D3

So far, we have focused on static visualizations using ggplot2. Static charts are excellent for many purposes (reports, papers, etc.), but sometimes we want interactive graphics that allow zooming, hovering, or other dynamic behaviors. One of the most powerful libraries for interactive visualization is **D3.js** (Data-Driven Documents), a JavaScript library that can create complex, interactive SVG visualizations in web browsers. D3 is behind many impressive data visualizations on the web.

Using D3 directly requires writing JavaScript code, which can be challenging if you're not familiar with it. However, R provides ways to integrate D3 visualizations into an R workflow. One such tool is the **r2d3** package – essentially an R interface to D3. The r2d3 package allows you to write D3 code and render it within R (for example, in RStudio's Viewer, in R Markdown documents, or in Shiny apps). It acts as a bridge: you supply data from R, and r2d3 passes it

to your D3 script and displays the result. In fact, `r2d3` can embed any D3 visualization into an R context.

According to the documentation, “the *r2d3* package provides a suite of tools for using D3 visualizations with R”, including translating R objects to D3-friendly data, rendering D3 scripts in RStudio, and integrating into RMarkdown and Shiny. In short, it lets you create highly custom, interactive visuals by tapping into the full power of D3, while still working within R.

Let’s go through a simple example to illustrate how `r2d3` works. Imagine we want to create a basic bar chart using D3 (similar to what we did in `ggplot2`, but now in pure D3 for the sake of learning).

There are two parts to an `r2d3` visualization:

1. A **D3 JavaScript script** that describes how to draw the visualization (using D3 syntax).
2. An R call to `r2d3()` that supplies the data (and points to the script, if it’s in an external file).

For the example, let’s create a D3 script that takes a numeric vector and draws a simple horizontal bar chart. We will assume the data is an array of numbers between 0 and 1 (just for simplicity).

**D3 Script (JavaScript)** – let’s call it `barchart.js`:

```
// !preview r2d3 data=c(0.3, 0.6, 0.8, 0.95, 0.40, 0.20)

// This D3 script expects an array of numeric values (between 0 and 1).
// It will create a bar chart with one bar per value.

// Define bar height based on total height available and number of data points
var barHeight = Math.floor(height / data.length);

// Select all 'rect' (rectangle) elements in the SVG, bind data
svg.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("width", function(d) { return d * width; }) // bar width proportional to data value
  .attr("height", barHeight - 2) // bar height (minus some padding for v.
  .attr("y", function(d, i) { return i * barHeight; }) // vertical position of bar
  .attr("fill", "steelblue");
```

A breakdown of this D3 code:

- The first line `// !preview r2d3 data=c(...` is a special comment that RStudio uses to allow previewing this script with the given data. It won't affect the actual rendering when we call `r2d3()` from R (it's just for convenience in RStudio IDE).
- We calculate `barHeight` by dividing the total SVG height (`height` variable is provided by `r2d3` environment) by the number of data points. This will space the bars evenly vertically.
- We then use typical D3 pattern: `svg.selectAll("rect")` starts a selection of all `rect` elements (initially none, but this is how D3 works), then `.data(data)` binds our data array to this selection, `.enter().append("rect")` creates a new `<rect>` for each data point.
- For each rectangle (bar), we set the `width` attribute based on the data value `d`. If `d` is between 0 and 1, multiplying by `width` (the total SVG width provided by `r2d3`) gives a length proportional to full width. This effectively scales the bar length to the data (assuming the data is a fraction of some maximum).
- The `height` of each bar is set to `barHeight - 2`. We subtract 2 pixels just to give a tiny gap between bars (so they are not flush against each other vertically).
- The `y` attribute is set by a function that takes the index `i` of the data point and multiplies it by `barHeight` – this stacks the bars vertically one after the other.
- We fill all bars with color “steelblue”. (This is a fixed color; in a more elaborate example, you might color based on value or category).

This D3 code by itself doesn't produce anything until we provide data and render it via R.

Now the R side: to render this D3 visualization from R, we would call the `r2d3()` function with our data and the script. For example:

```
library(r2d3)
# Suppose we have the barchart.js file in our working directory
r2d3(
  data = c(0.3, 0.6, 0.8, 0.95, 0.40, 0.20),
  script = "barchart.js"
)
```

When this `r2d3()` call is executed (in an interactive R session or in an RMarkdown document), it will load the `barchart.js` script, send the numeric vector as `data` to that script, and display the resulting bar chart (for example, in the RStudio Viewer or inlined in an HTML output). The result would be a series of horizontal blue bars of varying lengths corresponding to the values provided.

A few important notes about **r2d3**:

- It automatically provides some special variables to the D3 context: `data` (the R data translated to JavaScript – it could be an array, or an object if you pass in a data frame), `svg` (a pre-created SVG container element where you draw your chart), `width` and `height`

(dimensions of that SVG, which you can also set in the `r2d3` call if needed), and others like `options` and `theme` for advanced use.

- This means in your D3 code, you don't need to create the base `<svg>` or load data via AJAX; `r2d3` handles that. Your script should assume an `svg` is ready to use and `data` is already available as a JS variable.
- `r2d3` is great for custom visualizations that `ggplot2` or other high-level packages can't easily do. For example, interactive animations, custom network diagrams, etc. However, it does require knowledge of D3 (which has a learning curve).
- If you don't want to write D3 from scratch, there are many R packages built on **htmlwidgets** that wrap popular JS libraries (including many built with D3 under the hood) into easy R functions. For instance, `plotly` (for interactive charts), `leaflet` (for maps), `DT` (for interactive tables), `networkD3` (for network graphs) and so on. These provide high-level interfaces in R to generate interactive visualizations without manual JavaScript coding. As the `r2d3` documentation suggests, if you need a pre-fabricated visualization, an htmlwidget might already exist for it. Use `r2d3` when you truly need a custom visualization or want to directly leverage D3's flexibility.

In this chapter, we won't go deeper into writing complex D3 code, but it's good to be aware that tools like `r2d3` exist. They allow you to extend R's visualization capabilities beyond static plots: you can embed interactive D3 graphics in RMarkdown reports or Shiny web applications. For instance, you could create an interactive bubble chart where hovering over a bubble shows details, or a map where clicking zooms into regions, all defined in D3 but driven by R data.

**Recap:** To use D3 in R via **r2d3**, you would:

- Install and load the `r2d3` package.
- Write a D3 JavaScript script that uses the provided `data` and `svg`.
- Call `r2d3(data = yourdata, script = "yourScript.js")` to render it. (Optionally, you can inline a script with the `script` argument or use `r2d3(data=..., script = system.file("...path...", package="mypkg"))` if part of a package.)
- Enjoy the interactive visualization in your R environment.

This adds an advanced tool to your arsenal. However, even without diving into D3, you can create a wide range of plots using `ggplot2` and related packages. Interactive visualization can also be done through packages like **plotly** (which can convert `ggplot` graphs to interactive ones), **Shiny** (for building interactive dashboards in R), and others. The choice of tool depends on your needs and audience (interactive visualizations are great for web and exploratory analysis, while static ones are preferred for print or static reports).

Now that we've covered the theory, basic plotting, and even touched on interactive visuals, let's consolidate what we've learned and then try a hands-on exercise.

**TL;DR** – Below is a summary of the code used in this chapter for quick reference:

```

# Load data (assuming CSV file in working directory)
dataCanadaFullLong <- readr::read_csv("./data/lab3/dataCanadaFullLong.csv")
dataCanadaFullLong$isicCode <- as.character(dataCanadaFullLong$isicCode)

# Produce a bar chart (employees by year and industry)
library(ggplot2)
library(ggthemes)
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, fill = isicCode)) +
  geom_bar(stat = "identity", width = 0.5, position = "dodge") +
  xlab("") +
  ylab("Number of employees") +
  labs(fill = "Isic Code") +
  theme_minimal() +
  scale_fill_brewer(direction = -1)

# Line chart (with multiple industries)
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, color = isicCode)) +
  geom_line(size = 1.5) +
  xlab("") +
  ylab("Number of employees") +
  labs(color = "Isic Code") +
  theme_minimal() +
  scale_color_brewer(direction = -1)

# Line chart with points
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, color = isicCode)) +
  geom_line(size = 1.5) +
  geom_point(size = 2.5) +
  xlab("") +
  ylab("Number of employees") +
  labs(color = "Isic Code") +
  theme_minimal() +
  scale_color_brewer(direction = -1)

# Bubble chart (scatter plot with size mapped to value)
ggplot(data = dataCanadaFullLong, aes(x = year, y = value, color = isicCode)) +
  geom_point(aes(size = value)) +
  xlab("") +
  ylab("Number of employees") +
  theme_minimal() +
  scale_color_brewer(direction = -1) +
  scale_size_continuous(range = c(3, 11)) +

```



```

    guides(size = FALSE)

# World map
library(ggplot2)
world <- map_data("world")
ggplot(data = world, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "white", color = "black") +
  theme_void()

# Map of the Americas
americas <- subset(world, region %in% c("USA", "Brazil", "Mexico", "Colombia", "Argentina", "C
    "Peru", "Venezuela", "Chile", "Guatemala", "Ecuador", "Bo
    "Cuba", "Honduras", "Paraguay", "Nicaragua", "El Salvad
    "Costa Rica", "Panama", "Uruguay", "Jamaica",
    "Trinidad and Tobago", "Guyana", "Suriname", "Belize"
    "Barbados", "Saint Lucia", "Grenada",
    "Saint Vincent and the Grenadines", "Antigua and Barb
    "Saint Kitts and Nevis"))
ggplot(data = americas, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "white", color = "black") +
  coord_fixed(ratio = 1.1, xlim = c(-180, -35)) +
  theme_void()

# Using r2d3 (assuming r2d3 library loaded and barchart.js exists)
library(r2d3)
r2d3(data = c(0.3, 0.6, 0.8, 0.95, 0.40, 0.20), script = "barchart.js")

```

(In the above TL;DR code, `eval=FALSE` is used because these lines are for reference and not meant to be executed in one go in the book context.)

## 5.8 Code learned in this chapter

---

### Function/Command Description

---

<code>read_csv()</code>	Read a CSV file into a data frame (from <b>readr</b> package).
<code>as.character()</code>	Convert a variable to character type.
<code>ggplot()</code>	Initialize a ggplot object (specify data and aesthetic mappings).
<code>geom_bar()</code>	Add a bar geometry (for bar charts).
<code>geom_line()</code>	Add a line geometry (for line charts).
<code>geom_point()</code>	Add points (for scatter plots or to annotate lines).

Function/Command	Description
<code>xlab()</code> / <code>ylab()</code>	Set the x-axis or y-axis label (single axis label).
<code>labs()</code>	Set labels for axes, legends, and title in one function call.
<code>theme_minimal()</code>	Apply the “minimal” ggplot theme (clean, no background).
<code>theme_void()</code>	Apply an empty theme (no axes, good for maps).
<code>scale_fill_brewer()</code>	Use a ColorBrewer palette for fill colors (for discrete variables).
<code>scale_color_brewer()</code>	Use a ColorBrewer palette for line/point colors.
<code>scale_size_continuous()</code>	Adjust the scale for point sizes (continuous variable mapping to size).
<code>guides()</code>	Customize legend guides (e.g., turn off a legend for a specific aesthetic).
<code>map_data()</code>	Get map data (coordinates) for a given map (e.g., “world”).
<code>subset()</code>	Subset a data frame based on a condition (used to filter map data for certain regions).
<code>geom_polygon()</code>	Draw polygons (filled shapes) which we used for maps.
<code>coord_fixed()</code>	Fix aspect ratio of coordinate system (useful for maps to avoid distortion).
<code>esquisse</code> (Addin)	<i>RStudio addin (GUI) for building ggplot2 graphs interactively.</i>
<b>(Interactive)</b>	Render a D3 script with given data (for interactive custom visualizations).
<code>r2d3()</code>	

(Note: *esquisse* is used via the RStudio GUI; there’s also a function call `esquisse::esquisser()` that launches it, but typically one uses the Addin menu. It’s listed here as a concept rather than a function you put in script.)

## 5.9 Getting your hands dirty

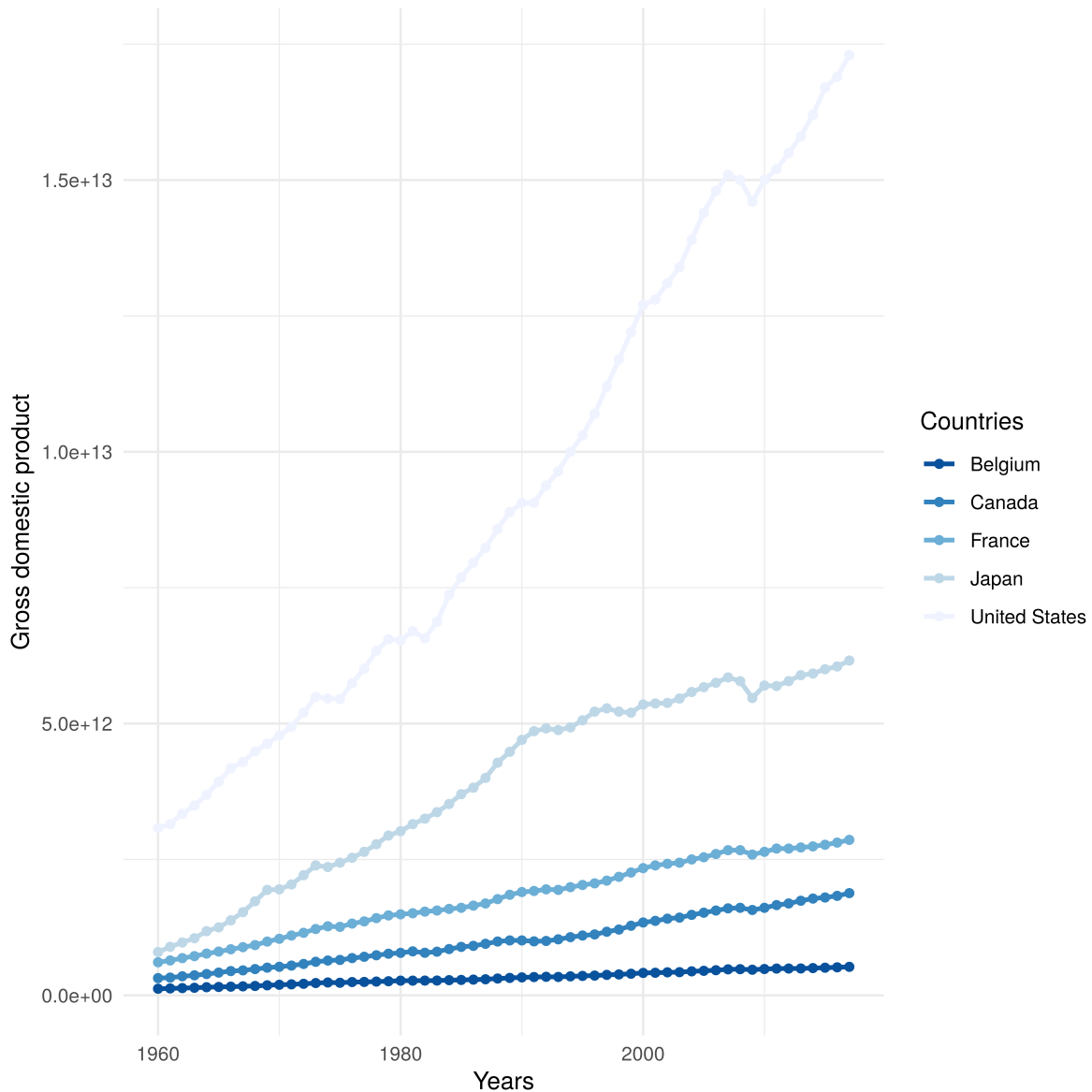
Now it’s time to practice what you’ve learned. For this exercise, we will use a dataset available on the course GitHub (in the Chapter 10 directory, file named `chapter10data.csv`). The data contains some economic indicators (for example, GDP values) for a selection of countries over time. Your tasks:

**Step 1: Import the data** Use `read_csv()` to read `chapter10data.csv` into a data frame (hint: assign it to a variable, e.g., `gdp5`). This dataset likely has columns for Country, Year, and GDP (possibly multiple series). Make sure to inspect the data and understand its structure.

```
library(readr)
gdp5 <-
```

Fill in the code to read the CSV. The path might be `'./data/chapter10data.csv'` or similar depending on where you’ve saved it.

**Step 2: Create a line chart** Recreate the line chart shown below (Figure 4). This chart plots GDP over time for a few countries (it looks like five countries). Each country should be a line, with different colors. The x-axis is year, y-axis is GDP (probably in billions or some unit), and there's a legend for countries. Use `geom_line()` and possibly `scale_color_*` for a nice palette. Label the axes appropriately and use a clean theme.



```
library(ggplot2)
# Assuming gdp5 has columns "Country", "Year", "GDP" (adjust if different)
...
```

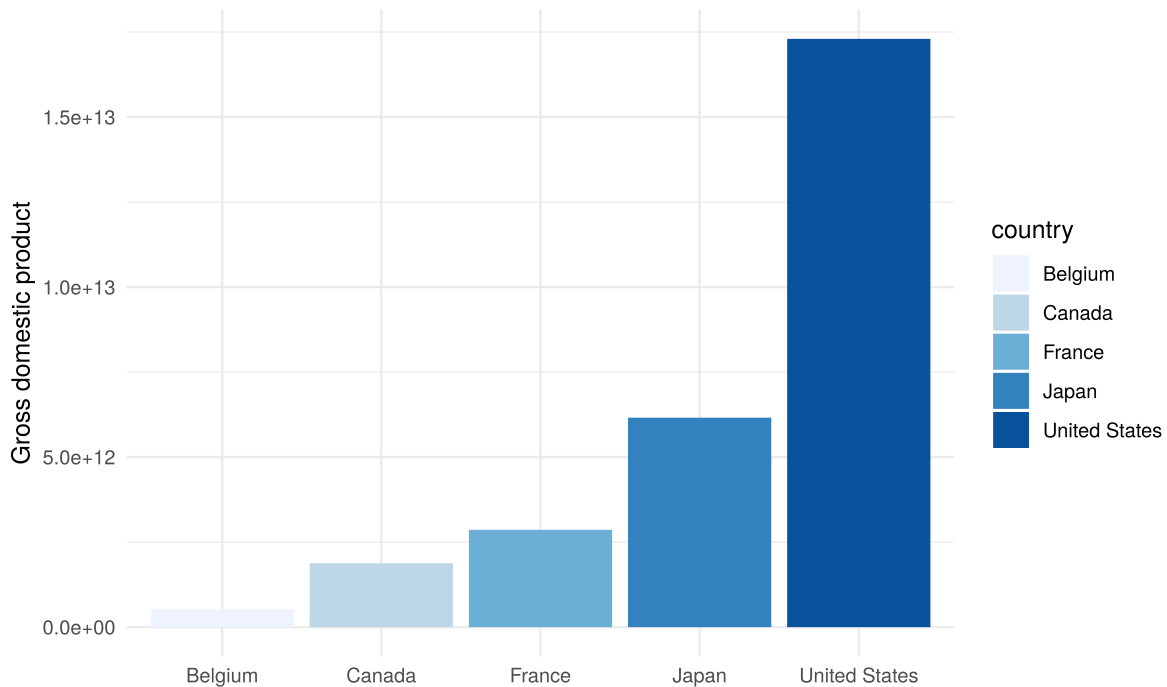
*Fill in the ggplot code to produce the line chart. You will need to map Year to x, GDP to y, color to Country. Choose a palette or use the default. Use `theme_minimal()` or another theme for a clean look. Add `geom_point()` if you want points on the lines (optional).*

**Step 3: Subset the data** For the next plot, we only want data for the year 2017 (to compare GDP across countries in that single year). Create a new data frame `gdp6` that filters `gdp5` to only include rows where Year is 2017. You can use the `dplyr` package for a convenient syntax (`filter()`), or use base R subsetting.

```
library(dplyr)
gdp6 <-
```

*Fill in the code to filter the dataset for Year == 2017. If using dplyr, `gdp5 %>% filter(Year == 2017)` would be typical, or assign the result to `gdp6`. Alternatively, use `subset(gdp5, Year == 2017)` or `gdp5[gdp5$Year == 2017, ]`.*

**Step 4: Create a bar chart** Using `gdp6` (which should have one row per country, GDP for 2017), create a bar chart showing the GDP of these countries for that year. The bar chart (Figure 5 below) has country names on the x-axis (or as categories) and GDP on the y-axis (height of bars). Each bar can be in a different color (for example, you might map Country to fill, or just set a single fill if you prefer monochrome bars with labels). The example image shows colored bars with no legend, which suggests the colors might have been set manually or just to be visually distinct (since a legend for country is redundant if country names are on x-axis). You can choose to map fill to country (which will give a legend by default; you can then use `guides(fill=FALSE)` to suppress the legend). Or simply assign a fill color palette without mapping if you know how to do that.



```
library(ggplot2)
...
```

*Fill in the ggplot code to produce the bar chart. You will set  $x = \text{Country}$ ,  $y = \text{GDP}$ , and likely `geom_bar(stat="identity")` since you have the values. Use `coord_flip()` if you want horizontal bars (the sample image shows vertical bars, but either could work; the image text might be rotated, it's a bit hard to tell). Ensure country names are readable (if long, maybe use `theme(axis.text.x = element_text(angle=45, hjust=1))` to tilt them).*

Once you complete these steps, you should have two plots: a multi-line chart of GDP over time for several countries, and a bar chart comparing their GDP in 2017. These mirror what you learned in this chapter (line chart and bar chart). Congratulations – you've applied data visualization principles and code to a new dataset!

## 6 Creating Data Dashboards

In data science and analytics, a **dashboard** is a visual display of information – often including charts, tables, and summary metrics – that consolidates data into an interactive, at-a-glance view. Dashboards serve as powerful tools for **data communication**, enabling users to monitor key performance indicators (KPIs), explore trends, and make informed decisions quickly. They are especially prevalent in **business analytics**, where companies rely on dashboards to share data insights across teams and drive strategic decisions. In fact, the vast majority of businesses use dashboards as a primary channel for communicating data insights; without dashboards, it would be almost impossible for many organizations to share data reports with all who need them. By presenting data in a visual and interactive format, dashboards help bridge the gap between data and decision-makers, allowing even non-technical users to grasp complex information intuitively.

Dashboards are not only crucial in corporate settings – they also play an important role in the public sector and open data initiatives. Governments and institutions use public-facing dashboards to share data with citizens, increasing transparency and accountability. For example, city governments might publish dashboards on budgets or environmental metrics to allow the public to see what is happening and to initiate informed conversations or actions. Research has highlighted that dashboards can connect governments with the public, helping to create transparency and foster trust. As such, well-designed dashboards become an **important means of communicating and interacting with the public** on data-driven issues, enabling broader audiences to engage with data in a meaningful way. Whether in a business or public context, dashboards turn raw data into accessible information stories, making them essential for modern data-driven decision making.

Beyond their communicative value, dashboards offer practical advantages. They provide **timely data access** – often updating in real-time or on a regular schedule – so that users are always looking at the latest information. They enable **enhanced decision-making** by highlighting trends and anomalies that might be missed in static reports. They can also improve efficiency by consolidating multiple reports or analysis views into a single interface. In summary, a dashboard serves as a **single source of truth** for a particular analysis or operational area, where users can interact with visuals and filters to answer their questions on the fly. By combining clarity, interactivity, and timeliness, dashboards have become indispensable in both business intelligence and public data communication.

## 6.1 Flexdashboard

**Flexdashboard** is an R package that makes it easy to create interactive dashboards using R Markdown. With flexdashboard, you can **publish a group of related data visualizations as a dashboard** – either as a static web page or an interactive Shiny app. Flexdashboard supports a wide variety of components, including **htmlwidgets** (R bindings to JavaScript libraries), base R or ggplot2 graphics, interactive tables, **value boxes**, **gauges**, and text annotations. It features flexible layouts (rows, columns, and tabsets) that intelligently adapt to fill the browser screen and even adjust for mobile devices. You can also incorporate **Shiny** to drive live data or user inputs in the dashboard. In short, flexdashboard brings together R Markdown reporting and Shiny interactivity for easy yet powerful dashboard creation.

*Example of a flexdashboard with multiple components (heatmap and table). Flexdashboard allows assembling various charts and outputs into a responsive dashboard layout.*

**Key features of flexdashboard include:**

- **R Markdown workflow:** You write an R Markdown document and use the `flexdashboard::flex_dashboard` format to generate the dashboard. This keeps development **simple and reproducible** (mixing text, code, and results).
- **Rich components:** Flexdashboard supports **interactive visualizations** via htmlwidgets, static plots from base R or ggplot2, **data tables** (with sorting/filtering), and special elements like **value boxes** (for key metrics) and **gauges** (for KPI meters).
- **Flexible layouts:** You can arrange components using **rows and columns** specified by section headers. The layout uses a CSS flexbox mechanism to automatically **resize charts to fill the browser** and even **stacks into a single column on small screens** (for mobile).
- **Storyboards:** In addition to typical grid layouts, flexdashboard offers a **storyboard** mode for presenting a **sequence of charts with accompanying commentary** in a step-by-step narrative style.
- **Optional Shiny interactivity:** Flexdashboards can be **static** (purely viewer-side HTML/JS) or, if you specify `runtime: shiny`, they become **dynamic Shiny apps** where readers can input parameters and see results update live.

To get started, install the **flexdashboard** package from CRAN if you haven't already:

```
install.packages("flexdashboard")
```

This provides the `flex_dashboard` output format for R Markdown. In RStudio, you can create a new Flexdashboard document by going to **File > New File > R Markdown... > From Template > Flex Dashboard**. This will open a template `.Rmd` file with the necessary YAML header and structure. (If not using RStudio, you can use the `rmarkdown::draft` function with the `"flex_dashboard"` template to scaffold a new dashboard file.)

*Creating a new Flex Dashboard document from RStudio's template menu.*

The top of your R Markdown should include YAML like this, which specifies the flexdashboard format and any options:

```
---
title: "Your Dashboard Title"
output:
  flexdashboard::flex_dashboard:
    orientation: columns      # or rows
    vertical_layout: fill     # or scroll
    storyboard: false        # set true for storyboard layout
---
```

Here we set the overall layout orientation (columns vs. rows), and whether the dashboard should **fill the page vertically or scroll** (more on this below). With that in place, you structure the content of the dashboard using headings and code chunks.

## 6.2 Layout

Dashboards are organized into **sections** using headings. Flexdashboard uses the heading levels to define rows, columns, and chart panels:

- **Level 1 headings (#)** define **Pages** (for multi-page dashboards, each becomes a top-level navigation tab).
- **Level 2 headings (##)** define **Columns** by default (or entire Rows if `orientation: rows` is set globally or for that page).
- **Level 3 headings (###)** define individual **chart sections** within a column or row.

By default, flexdashboard assumes a *column-oriented* layout: each Level 2 heading starts a new column, and charts (Level 3) under it stack vertically within that column. You can also opt for a *row-oriented* layout by setting `orientation: rows` (in YAML or for a specific page), in which case Level 2 headings create new rows, and charts under them distribute into columns within that row.

In addition, the **vertical layout mode** controls how charts fill the page vertically. The default `vertical_layout: fill` causes the dashboard to **fill the browser height** – charts will expand/shrink vertically to fit the screen without scrolling. This is ideal if you have one or two charts stacked and want to use all available space. Alternatively, `vertical_layout: scroll` will keep each chart at its natural height (as determined by its figure size), and allow the page to scroll if the content exceeds the screen. Scrolling layout is better when you have many components vertically and prefer not to squish them into one view.

Let's look at a few common layout patterns:



### 6.2.1 Chart Stack (Fill)

This is the simplest layout: a single column with charts stacked vertically, filling the page. For example, two charts one above the other can be defined like so:

```
output:
  flexdashboard::flex_dashboard:
    vertical_layout: fill
---
### Chart 1

::: {.cell}

```{r .cell-code}
<code to produce chart 1>
```

:::

### Chart 2

::: {.cell}

```{r .cell-code}
<code to produce chart 2>
```

:::
```

With `vertical_layout: fill`, these two charts will split the available vertical space in the browser roughly equally (unless one is given extra height explicitly). You could make one chart taller by adding a `data-height` attribute on its section header, which adjusts the relative height allocation: `contentReferenceoaicite:24`. For instance, `### Chart 1 {data-height=500}` would allot more height to Chart 1 relative to Chart 2 in the fill layout.

### 6.2.2 Chart Stack (Scrolling)

If you have multiple charts stacked, a fill layout might cram them into a tight space. Using a **scrolling layout** lets each chart keep its natural height and simply adds a scrollbar for the page. For example:

```

output:
  flexdashboard::flex_dashboard:
    vertical_layout: scroll
---

### Chart A

::: {.cell}

```{r .cell-code}
<chart A code>

```

```

:::

```

### 6.2.3 Chart B

```

<chart B code>

```

### 6.2.4 Chart C

```

<chart C code>

```

Here we have three charts in one column. With `vertical\_layout: scroll`, each chart will be

#### ### Focal Chart (Top)

Sometimes you want one visualization to stand out with more space, and supporting charts to

```

```yaml
---
output:
  flexdashboard::flex_dashboard:
    orientation: rows
---
Row {data-height=650}
-----

```

```

### Big Chart (Top)

::: {.cell}

```{r .cell-code}
# code for big chart

...

```

## 6.3 Row

### 6.3.1 Chart 2

```
# code
```

### 6.3.2 Chart 3

```
# code
```

In the above, we explicitly set the first **Row** to 650px height and the second row to 350px height.

### ### Focal Chart (Left)

You can also emphasize a chart by giving it more horizontal space. To create a **multi-column** layout, you can use the `flexdashboard::flex_dashboard` package.

```

```yaml
output: flexdashboard::flex_dashboard
---
Column {data-width=600}
-----
### Chart 1 (Main)

::: {.cell}

```{r .cell-code}
# <code for main chart>

...

```

## 6.4 Column

### 6.4.1 Chart 2

```
# <code>
```

### 6.4.2 Chart 3

```
# <code>
```

The above yields **Chart 1** in a left column that is 600px wide vs. a right column of 400px. This layout features one prominent chart on one side, with supporting charts on the other side.

### Tabsets

If you have several components that don't all need to be visible at once, you can organize them using tabsets.

For instance, suppose you have one column that should show either Chart X or Chart Y (as two

```
```yaml
output: flexdashboard::flex_dashboard
---
Column
-----
### Chart A

::: {.cell}

```{.r .cell-code}
# <code for chart A>

:::
```

## 6.5 Column

### 6.5.1 Chart X

```
# <code for chart X>
```

### 6.5.2 Chart Y

```
# <code for chart Y>
```

In this layout, the dashboard has two columns: the left column (no special classes) always sh

#### ## Components

A flexdashboard can include any output you can generate from R, but there are a few special

- **Base R and ggplot2** graphics: Any plots produced by R code chunks will be included in the dashboard.
- **htmlwidgets**: These are R packages that bind JavaScript libraries (like Leaflet maps, Plotly, etc.) to R.
- **Tables**: You can simply print a data frame or use packages like **DT** for interactive tables.
- **Value Boxes**: These display a big numeric value with a caption (and icon/color) to highlight it.
- **Gauges**: These show a dial or meter representation of a value within a range.
- **Text and Markdown**: You can include narrative text, headers, lists, etc., in your dashboard.

Let's dive a bit deeper into **Value Boxes** and **Gauges** since they are unique to dashboards.

#### ### Value Boxes

A **value box** is a compact box that highlights a single value along with a label and an icon.

```
::: {.cell}
```

```
```.r .cell-code{
```

```
valueBox(45, title = "Articles per Day", icon = "fa-pencil")
```

```
:::
```

This would display a value box showing **45** and the caption “Articles per Day,” with a pencil icon. By default, value boxes are styled in a neutral color (or “primary” theme color). You can

customize the color or make it conditional. For instance, you might use `color = "warning"` or `"success"`, or even a conditional expression:

```
spam <- 5
valueBox(spam, title = "Spam per Day", icon = "fa-trash",
         color = ifelse(spam > 10, "warning", "primary"))
```

In this example, the color will be “primary” (blue) since spam is 5 (low), but if it exceeded 10 it would turn “warning” (often yellow/orange). The set of built-in color names includes **primary**, **info**, **success**, **warning**, **danger**, corresponding to Bootstrap theme colors. You can also use any custom CSS color code if needed.

Value boxes can include an **icon** on the right side. Flexdashboard supports icons from **Font Awesome**, **Ionicons**, and **Bootstrap Glyphicons** – you specify them by prefixing the icon name with `fa-`, `ion-`, or `glyphicon-` respectively. For example, `"fa-comments"` for a comment icon or `"ion-social-twitter"` for a Twitter bird. (Note: not all icons from these sets are free or available by default, so some icons might not render if they require pro licenses.)

You can also make a value box into a hyperlink – for instance, if clicking the value should navigate to another page or section with more details. Use the `href` argument to specify a page anchor (e.g., `href="#details"` to link to a subsection titled “Details”).

### 6.5.3 Gauges

A **gauge** displays a numeric value on a semicircular meter, with zones indicating ranges (like red/yellow/green zones for bad/medium/good). In flexdashboard, you create gauges with the `gauge()` function. The simplest usage is:

```
gauge(value = 37.4, min = 0, max = 50)
```

This will draw a gauge showing **37.4** on a 0–50 scale. By default the gauge will be a single color (the theme’s “success” color, typically green). You can define colored sectors for different ranges using `gaugeSectors()`. For example:

```
gauge(37.4, min = 0, max = 50,
     sectors = gaugeSectors(success = c(41, 50), warning = c(21, 40), danger = c(0, 20)))
```

In this case:

- **Green** (“success”) for values 41–50,
- **Yellow** (“warning”) for 21–40,
- **Red** (“danger”) for 0–20.

The gauge needle will point at 37.4, which falls in the warning (yellow) zone in this configuration. You can also add a `symbol` argument (e.g., % or \$) to append a unit to the displayed value.

Gauges, like value boxes, can use `href` to link to more details, and in Shiny contexts should be rendered via `renderGauge()` so they update dynamically (similar to using `renderValueBox()` for value boxes in Shiny).

## 6.6 Sizing and Styling

By default, flexdashboard charts are sized automatically to fill the layout. If you do nothing special, the **relative size** of each chart is determined by the knitr figure dimensions (which default to 6 inches by 4.8 inches, i.e. 576x460 pixels). In a fill layout, vertically stacked charts will each take an equal share of the height (unless you adjust with `data-height`). In a scrolling layout, each chart's height is just its figure height (so you may see more whitespace if figures are smaller, but you can scroll).

Flexdashboard allows fine control of sizing using **header attributes**:

- `data-height` and `data-width` can be applied to rows, columns, or individual chart sections to adjust their size relative to siblings. These aren't absolute pixel sizes but weighted proportions in the flexbox layout. For example, if two charts are in one column and one has `data-height="2"` and the other `data-height="1"`, the first will be twice as tall as the second.
- The YAML global `orientation` and `vertical_layout` we discussed earlier also play a role in how widths/heights are allocated.

**Example – Emphasizing a Chart:** Suppose we want the first chart in a column to take more space. We could do:

```
### Sales Over Time {data-height=600}

::: {.cell}

```{r .cell-code}
# <code for a big timeseries plot>
```

:::

### Breakdown by Region {data-height=300}
```

```

::: {.cell}

```{r .cell-code}
# <code for smaller plot>
```

:::

```

This ensures the first chart gets a taller area than the second. If using fill layout, these numbers are treated as relative weights:contentReferenceoaicite:50. In scrolling layout, the **data-height** on an individual chart actually sets a fixed pixel height (since scrolling mode otherwise uses each chart’s knitr fig height). You can experiment with these to get a pleasing arrangement.

There are other styling options too: - You can remove the default 8px padding around charts by adding a **.no-padding** class to the section, or set a custom padding with **data-padding=NN**:contentReferenceoaicite:51. - Flexdashboard includes several built-in **themes** (like cerulean, journal, flatly, etc. from Bootswatch) which you can set in the YAML (**theme: united**, for example):contentReferenceoaicite:52:contentReferenceoaicite:53. The default theme is a modified Bootstrap “cosmo” theme:contentReferenceoaicite:54. Themes mainly affect the navigation bar and background styling. - You can add custom CSS via the **css:** option in YAML if deeper styling customizations are needed:contentReferenceoaicite:55 (for example, to tweak font sizes or colors of specific elements).

## 6.7 Multiple Pages

For larger dashboards, you can split content into **multiple pages**. Each page will get its own tab in the navigation bar. To define a page, use a Level 1 heading (#) in the R Markdown. For example:

```

Page 1
=====

## Section ...
### Chart ...

Page 2
=====

## Section ...
### Chart ...

```



Each `#` heading (Page 1, Page 2) creates a new page tab in the navbar. This is useful to organize content into logical groups (e.g., an Overview page and a Details page, or different subject areas).

All pages in a flexdashboard share the same R session (if using Shiny, all pages are part of one app). By default, pages are shown as separate top-level tabs, but if you have many pages you can group them into a **dropdown menu** by using the `data-navmenu="Menu Name"` attribute on some page headers. For example, `Page 3 {data-navmenu="Other Metrics"}` would put Page 3 under a “Other Metrics” menu instead of as a standalone tab. You can even assign icons to page tabs via `data-icon` attributes if desired.

If needed, you can hide a page from the navbar (perhaps it’s only linked to internally) by adding the `.hidden` class to the page header.

All of these options let you create a multi-page dashboard that is still contained in a single R Markdown document.

## 6.8 Storyboards

Flexdashboard’s **storyboard** layout is an alternative to the grid layout, ideal for presenting a **sequence of charts with commentary**, like slides in a presentation. In a storyboard, each Level 3 section (`###`) becomes a **frame** in a horizontal scrolling sequence (like slides), and you can include descriptive text alongside the frames.

To use a storyboard, you have two approaches:

- **Entire dashboard as a storyboard:** In the YAML, set `storyboard: true`. Then every `###` section on that page becomes a frame in the storyboard. The section title serves as the navigation caption for that frame (so typically, your section titles in a storyboard might be longer descriptions).
- **Storyboard page in a multi-page dashboard:** If you want one page to be a storyboard and others to be normal layout, do not use the global YAML option. Instead, add the class `{.storyboard}` to the Level 1 heading of the page you want as a storyboard. For example: `Analysis {.storyboard} =====` would make the “Analysis” page use storyboard layout, while other pages remain standard.

In a storyboard, frames are navigated via arrows or by clicking the frame titles. You simply create consecutive `###` sections. For example:

```
---
output: flexdashboard::flex_dashboard:
  storyboard: true
---
### Frame 1: Overview
```

```

::: {.cell}

```{r .cell-code}
# <code for visualization 1>
```

:::

### Frame 2: Details

::: {.cell}

```{r .cell-code}
# <code for visualization 2>
```

:::

```

This would produce two frames, “Frame 1: Overview” and “Frame 2: Details,” that the reader can flip through. The content is presented one frame at a time, with a visual index of frame titles.

### 6.8.1 Commentary

Often you will want to include explanatory text alongside each frame in a storyboard (e.g., bullet points or interpretation of the chart). Flexdashboard allows adding a **commentary sidebar** to each frame. To do this, within a frame’s section, **separate the main content and the commentary with a horizontal rule \*\*\*** (three asterisks on a line):contentReferenceoaicite:61. Anything after the \*\*\* in that section will appear as commentary text to the right of the visualization.

For example:

```

### Frame 2: Revenue Breakdown {data-commentary-width=400}

::: {.cell}

```{r .cell-code}
# <plot code>

```



## 7 Automating Data Collection with APIs

In data science projects, having access to interesting datasets is essential fuel for analysis. While one approach is to manually download CSV files from various websites, a more dynamic and powerful method is to use **APIs (Application Programming Interfaces)**. An API is essentially a set of rules or protocols that enables different software applications to communicate and exchange data and functionality. In practical terms, many data providers offer web APIs that allow direct retrieval of up-to-date data via code, rather than manually downloading files. Using APIs can ensure you are working with the latest data and can automate the data acquisition process.

In this chapter, we will focus on how to import data using APIs in R through specialized packages. Each package acts as a wrapper around an external data source's API, handling the communication details so that you can simply call R functions to fetch data. Before diving into specific examples, it's important to clarify what we mean by an **argument** in this context. In programming, an *argument* is a value or input that you pass to a function when you call it, which influences how the function operates. For instance, if a function is defined to take a country code as an argument, you would provide a specific country code value when calling that function. Understanding arguments is crucial because all the API-related functions we'll use require certain arguments (like indicator codes, country codes, years, etc.) to specify what data you want.

**At the end of the chapter, you should be able to:**

1. **Know what an argument is.** You will understand how functions use arguments (inputs) to modify their behavior, and how to supply the correct arguments to get the data you need.
2. **Import data using an API in R.** You will learn to use various R packages that interface with web APIs to retrieve data from online sources, avoiding manual downloads and making your data acquisition process reproducible and up-to-date.

We will explore several real-world examples of R packages that provide easy access to data via APIs. These include:

- **WDI:** Access World Bank's World Development Indicators and other datasets.
- **OECD:** Retrieve indicators and datasets from the OECD (Organisation for Economic Co-operation and Development).
- **spiR:** Access the Social Progress Index data.
- **statcanR:** Download data from Statistics Canada's open data portal.

- **EpiBibR:** Obtain bibliographic references (especially COVID-19 related literature data).
- **coronavirus:** Get daily COVID-19 statistics (cases, deaths, etc.) from the Johns Hopkins University dataset.

For each of these, we will discuss the data source, the key functions provided by the R package, and walk through examples of how to use them. By the end, you will see how APIs combined with R packages allow you to seamlessly pull in data from a variety of domains with just a few lines of code.

## 7.1 WDI

### 7.1.1 Database Description

The **World Development Indicators (WDI)** database is a flagship dataset of the World Bank containing a wide range of global development statistics. It is a compilation of relevant, high-quality, and internationally comparable statistics about global development and the fight against poverty. The database includes over 1,600 time-series indicators for 217 economies and more than 40 country groups, with data for many indicators going back over 50 years. These indicators cover topics such as economic growth, education, health, poverty, environmental factors, and much more.

What makes the WDI particularly powerful is that it's part of a larger collection of data sources provided by the World Bank. In fact, the R package **WDI** allows users not only to access the main World Development Indicators but also dozens of other datasets hosted by the World Bank (e.g., International Debt Statistics, Doing Business indicators, Human Capital Index, etc.). This means that through one package, you can tap into a rich variety of development data.

Using the World Bank's API through the **WDI** package has several advantages: the data are always up-to-date (as of the last World Bank update), you can easily retrieve long time series for multiple countries, and you can programmatically search for indicators by keywords. The data returned by the API is typically in a tidy **country-year** format – each row corresponds to a country and year, with columns for the indicator values (and possibly country codes and other metadata). This format is convenient for analysis and plotting in R once you have the data.

### 7.1.2 Functions

The **WDI** R package provides a couple of key functions to interact with the World Bank data:

- **WDIsearch()** – Search for indicators by keyword.

- `WDI()` – Download data for specified indicators, countries, and time ranges.

Additionally, the package includes some utility functions (such as `WDIcache` and `WDIbulk`) for advanced usage, but our focus will be on the two main functions above, which cover most typical needs. We will go through each of these functions with examples to illustrate how they work and how to use the correct arguments to get the data you want.

### 7.1.2.1 `WDIsearch()`

The function `WDIsearch()` allows you to find indicators in the World Bank datasets by searching for keywords in the indicator name or description. This is extremely useful when you know the topic you’re interested in (e.g., GDP, life expectancy, CO2 emissions) but need to find the exact indicator code that the World Bank uses for that data. The `WDIsearch()` function takes a character string as an argument – this string is the keyword or pattern you want to search for. The function returns a data frame of indicators whose names or descriptions match the search term.

For example, suppose we want to find all indicators related to **GDP**. We can use `WDIsearch("GDP")` to retrieve a list of such indicators:

```
# Loading the WDI package
library(WDI)

# Search all indicators with the term "GDP"
listOfIndicators <- WDIsearch("GDP")

# Inspect the first 5 indicators found
listOfIndicators[1:5, ]
```

In the code above, `WDIsearch("GDP")` searches the World Bank’s catalog of series for the substring “GDP”. The result is assigned to `listOfIndicators`, which will be a data frame. The data frame typically has columns like “indicator” (the official indicator code used by the API) and “name” (a human-readable description of the indicator). By printing the first 5 rows (`listOfIndicators[1:5,]`), we might see something like:

	indicator	name
[1,]	"NY.GDP.MKTP.CD"	"GDP (current US\$)"
[2,]	"NY.GDP.MKTP.KD.ZG"	"GDP growth (annual %)"
[3,]	"NY.GDP.PCAP.CD"	"GDP per capita (current US\$)"
[4,]	"NY.GDP.PCAP.KD.ZG"	"GDP per capita growth (annual %)"
[5,]	"NE.GDI.TOTL.ZS"	"Gross capital formation (% of GDP)"

*Example output:* The above is an illustrative example of what the search results might contain (actual results may differ or appear in a different order). Each row shows an indicator code and its description. For instance, "NY.GDP.MKTP.CD" is the code for total GDP in current US dollars, and "NY.GDP.PCAP.CD" is GDP per capita in current US dollars, etc. Using this list, you can identify the specific indicator code you need for your analysis.

The ability to search by keyword (case-insensitive and even supporting regular expressions in `WDIsearch`) makes it much easier to find the right data without having to manually browse the World Bank website. Once you have the indicator code(s) you need, you can use the `WDI()` function to download the data.

### 7.1.2.2 WDI()

The `WDI()` function is used to actually retrieve data for one or more indicators and one or more countries over a specified time range. The main arguments for `WDI()` are:

- **indicator:** The indicator code or codes you want to download (as a string or a vector of strings).
- **country:** The country code or codes for which you want the data. Typically these are ISO-2 or ISO-3 country codes (the World Bank uses ISO-2 country codes by default). You can also use special codes like "all" to retrieve all countries, or groups like "OECD" for OECD countries as a whole, etc.
- **start:** The starting year for the data (numerical year, or in some cases a string if using quarterly/monthly data).
- **end:** The ending year for the data.

There are additional optional arguments as well. For example, `extra = TRUE` can be used to fetch additional columns (like region, income level, etc., for each country), and `cache` can be used to supply or update the list of indicators cached locally. But in many cases you can ignore these extras and just specify the main four arguments above.

Let's consider a concrete example. Suppose we are interested in the indicator "*Stocks traded, total value (% of GDP)*", which has the code `CM.MKT.TRAD.GD.ZS` in the WDI database. We want to gather this data for four countries – France, Canada, the United States, and China – over the period 2000 to 2016. Using `WDI()`, we can do this as follows:

```
library(WDI)

# Access and store data for "Stocks traded, total value (% of GDP)"
# for France (FR), Canada (CA), USA (US), and China (CN) from 2000 to 2016
stockTraded <- WDI(indicator = "CM.MKT.TRAD.GD.ZS",
                   country   = c("FR", "CA", "US", "CN"),
                   start     = 2000,
```

```

end = 2016)

# Peek at the first few rows of the retrieved data
head(stockTraded)

```

When this code runs, the `WDI()` function contacts the World Bank API behind the scenes and downloads the requested data. The result, stored in `stockTraded`, is a data frame. If we inspect it (using `head(stockTraded)` to see the first several rows), we might see something like:

	iso2c	country	year	CM.MKT.TRAD.GD.ZS
1	CA	Canada	2016	147.92
2	CA	Canada	2015	126.85
3	CA	Canada	2014	123.45
4	CA	Canada	2013	115.67
5	CA	Canada	2012	105.23
6	CA	Canada	2011	98.10
...				

*Example explanation:* Each row of the data frame represents a country-year observation. In this example, the columns include: an ISO 2-letter country code (`iso2c`), the country name, the year, and a column named after the indicator code (`CM.MKT.TRAD.GD.ZS`) which contains the value of the stocks traded (% of GDP) for that country in that year. The data frame is sorted by country and year (typically, countries in alphabetical order by the code, and years descending, but the ordering may vary). For instance, above we see data for Canada (CA) for 2016, 2015, etc. If we scroll further down the data frame, we would find the entries for China (CN), France (FR), and the USA (US) similarly each with values from 2000 up to 2016.

With this data frame in hand, you could proceed to analyze it or visualize it (e.g., compare the trends of that indicator across the four countries). The key point is that a single function call gave us a structured dataset ready for use, which is much more convenient than manually finding each country's data from a website.

**Note:** If you request a range of years where some countries don't have data for the entire range, the resulting data frame will have `NA` (missing) for those years and countries where data is absent. This is normal since not all indicators have values for every country-year.

As a final tip, you can actually request multiple indicators in one go by providing a vector of indicator codes to the `indicator` argument. In that case, `WDI()` will return one column per indicator (plus the country, code, and year columns). There's even a feature where if you name the elements of the indicator vector in R, those names will be used as column names in the result. For example, `WDI(indicator = c(gdp="NY.GDP.MKTP.CD", pop="SP.POP.TOTL"),`



`country="US", start=2010, end=2020)` would fetch GDP and population for the U.S. and name the columns `gdp` and `pop` respectively in the output data frame.

If you want to explore more about the WDI package and see another detailed example of its usage, you can refer to the World Bank data application example provided by the package authors, which demonstrates a case study of retrieving and analyzing data from the WDI.

### 7.1.3 tl;dr

```
# Loading the WDI library
library(WDI)

# Search all indicators with the term "GDP"
listOfIndicators <- WDIsearch("GDP")

# List the first 5 indicators found
listOfIndicators[1:5, ]

# Retrieve data for a specific indicator and set of countries over a time range
stockTraded <- WDI(indicator = "CM.MKT.TRAD.GD.ZS",
                    country   = c("FR", "CA", "US", "CN"),
                    start     = 2000,
                    end       = 2016)

head(stockTraded)
```

*(The above code summary shows how to search for indicators containing “GDP” and how to download one example indicator for multiple countries. In practice, replace the indicator code and country codes with those relevant to your needs.)*

## 7.2 OECD

### 7.2.1 Database Description

The **Organisation for Economic Co-operation and Development (OECD)** maintains a rich database of economic and social indicators for its member countries (and in some cases, non-members). The OECD data spans numerous categories such as agriculture, finance, health, education, labor, and more. The public-facing OECD data portal (often accessed via [data.oecd.org](http://data.oecd.org)) features around 300 key indicators organized into about a dozen categories.

These include high-level indicators like unemployment rates, GDP for OECD countries, population statistics, etc., which are curated for easy browsing.

However, the actual breadth of data available through the OECD's API is much larger. The OECD provides a flexible API that allows access to a wide array of datasets, each identified by a unique dataset code. Each dataset can contain many series and dimensions (for example, a dataset might contain data broken down by country, by sex, by age group, by year, etc.). The R package `OECD` is designed to help users discover and download data from this API without needing to manually construct complex query URLs.

The OECD data service gives you access to up-to-date statistics across various domains for many countries (mostly OECD members). This can include indicators like employment rates, economic outlook figures, education enrollment, health outcomes, and so on. The data is often structured by multiple dimensions (country, year, and often other classifications like gender, age, industry sector, etc., depending on the dataset). Using the R `OECD` package, we can search for datasets and then retrieve the specific slices of data we need by specifying filters.

## 7.2.2 Functions

The `OECD` R package provides several functions to interact with the OECD API. The main functions we will highlight are:

- `get_datasets()` – Retrieve a list of all available datasets (by ID and description).
- `search_dataset()` – Search within the dataset list for keywords, to find relevant dataset IDs by topic.
- `get_data_structure()` – Given a specific dataset ID, get the structure of that dataset (i.e., what dimensions it has and the valid codes for each dimension).
- `get_dataset()` – Download data from a specific dataset, optionally filtering by dimension codes (such as specific countries, years, etc.).

Each of these functions is useful at a different stage: first discovering what data exists, then understanding the structure of a specific dataset, and finally retrieving the data of interest. We will go through examples of their usage.

*(Note: The function names in the package are a bit confusing with singular/plural. `get_datasets()` (plural) returns the list of dataset identifiers and descriptions. By contrast, `get_dataset()` (singular) is used to download the data for one dataset. Be careful to use the correct function.)*

### 7.2.2.1 `get_datasets()` – Listing available datasets

Before searching or fetching data, you may want to know what datasets are available via the OECD API. The function `get_datasets()` returns a data frame of all dataset identifiers along

with their descriptions. This is typically a large list (since OECD has many datasets). You can store this list and then use it for searching. For example:

```
# Loading OECD package
library(OECD)

# List all available datasets and store in a data frame
dataset_list <- get_datasets()

# Check the first few entries in the dataset list
head(dataset_list)
```

After running `get_datasets()`, `dataset_list` might contain entries like:

	id	description
1	AFLPMEAN	Average length of parental leave, measured in weeks
2	AGRI_INV	Agricultural Innovation Indicators...
3	AEO	African Economic Outlook...
4	BLI	Better Life Index...
...		

This is just illustrative. Each row has a dataset `id` (a short code like “AEO”, “BLI”, etc.) and a longer description explaining what that dataset is. Given that the list can be long, it’s often more practical to search for a keyword in this list rather than scrolling through it manually. That’s where `search_dataset()` comes in.

#### 7.2.2.2 `search_dataset()`

The function `search_dataset()` helps you find which dataset(s) might contain the data you need by searching within the dataset names and descriptions. You provide a keyword (or regex pattern) and a data frame of datasets to search (by default, you can use the result of `get_datasets()` as that input).

For example, if we are interested in data about **unemployment**, we can search the dataset list for the term “unemployment”:

```
# Assuming dataset_list has been obtained via get_datasets()
search_results <- search_dataset("unemployment", data = dataset_list)
```

This call will return a data frame of datasets whose descriptions contain “unemployment”. For instance, one likely result is a dataset that deals with unemployment by duration. The output might look like this (for illustration):

	id	description
1	DUR_D	Unemployment duration by age group and gender
2	...	... (other related datasets if any)

From this search result, suppose we identify that DUR\_D is the dataset we want (just as an example, DUR\_D might stand for “Duration of unemployment (in days or months) by demographic breakdown”). Now that we have a specific dataset ID, the next step is to see what the structure of that dataset is (what dimensions and codes it uses) so we can query it properly.

### 7.2.2.3 get\_data\_structure()

Every OECD dataset can have multiple dimensions. For example, a dataset might be broken down by country, by year, by gender, by age group, etc. To know how to formulate our query and what filters to use, we need to know what dimensions a dataset has and what the valid codes for those dimensions are. The function `get_data_structure(dataset_id)` returns an object (often a list of data frames) describing the dataset’s structure.

Continuing our example with dataset "DUR\_D" (unemployment duration), let’s retrieve its structure:

```
# Get the structure of the "DUR_D" dataset
dstruc <- get_data_structure("DUR_D")

# Examine the structure object
str(dstruc, max.level = 1)
```

When you call `get_data_structure("DUR_D")`, behind the scenes R fetches metadata about that dataset from the OECD API. The result `dstruc` is typically a list where each element corresponds to a dimension of the dataset. For instance, `dstruc` might contain elements like `$COUNTRY`, `$AGE`, `$SEX`, `$TIME` (these are hypothetical) each of which is a data frame listing the codes and meanings for that dimension.

Using `str(dstruc, max.level = 1)` will print an overview of the list structure. You might see something like:

```
List of 4
 $ COUNTRY: 'data.frame':  ... (country codes and names)
 $ AGE    : 'data.frame':  ... (age group codes and descriptions)
 $ SEX    : 'data.frame':  ... (sex codes and descriptions)
 $ TIME   : 'data.frame':  ... (time period info, possibly years available)
```

This tells us that the dataset `DUR_D` is broken down by Country, Age, Sex, and Time (year). To know the actual codes, we could inspect each of those. For example, `dstruc$COUNTRY` might show entries like `USA = "United States"`, `FRA = "France"`, etc. `dstruc$SEX` might show codes like `M = "Male"`, `W = "Female"`, `MW = "Total (Male+Female)"`. `dstruc$AGE` might list codes like `TOTAL = "All ages"`, `Y15-24 = "15-24 years"`, etc. (The actual codes can vary; these are just plausible examples.)

Armed with this information, we can now decide what subset of the data we want. Let's say we want to get data on unemployment duration for a few specific countries, for both males and females combined, focusing on young adults age 20–24, for the most recent year(s) available. We will need to assemble a filter that specifies those choices for each dimension.

#### 7.2.2.4 `get_dataset()`

The function `get_dataset()` is used to download the actual data from a specified OECD dataset. You must provide the dataset ID and you can provide a `filter` argument to narrow down which slices of the data you want. The `filter` argument expects a list of vectors, where each element of the list corresponds to one dimension of the dataset, in the order that the dimensions are defined.

If no filter is provided at all, `get_dataset("XYZ")` would attempt to download the entire dataset “XYZ” (which could be huge, so usually you do want to filter it). If you provide a partial filter (for some dimensions), you typically need to specify something for each dimension, even if it's just an “all” wildcard. The `get_dataset` function documentation suggests that if you leave filters empty it will get everything, but you can also explicitly use `NULL` or an empty string for dimensions you don't want to filter (depending on how the function is implemented).

In our example with `DUR_D`, suppose `DUR_D`'s dimensions are in order: Country, Sex, Age, Time. We want: Country = {Germany, France, Canada, USA}, Sex = {Total (both sexes)}, Age = {20-24}, and Time = (we could filter a range of years or leave it to get all years). Let's assume we want all available years for those filters.

From the structure, we identified the codes:

- Germany = “DEU”, France = “FRA”, Canada = “CAN”, USA = “USA” (these are standard ISO country codes and likely used by OECD).
- Sex total = maybe “T” or “ALL” or “MW”. Warin's example uses “MW” which likely stands for male+female combined.
- Age 20-24 might have a code like “Y20-24” or simply “2024” as given in the example.

In the content provided, the filter list was: `filter_list <- list(c("DEU", "FRA", "CAN", "USA"), "MW", "2024")`. This implies:

- First element: a vector of country codes.

- Second element: “MW” presumably meaning both sexes.
- Third element: “2024” representing the 20-24 years age group.
- (Likely the time dimension was left unfiltered, meaning all years, since they didn’t include a fourth element for time. The `get_dataset` might interpret an omitted dimension as no filtering on it.)

Now we use `get_dataset` with these filters:

```
# Define filters: Countries = DEU, FRA, CAN, USA; Sex = MW (both male & female); Age group =
filter_list <- list(c("DEU", "FRA", "CAN", "USA"),
                  "MW",
                  "2024")

# Retrieve the filtered data from dataset "DUR_D"
unemploymentOECD <- get_dataset(dataset = "DUR_D", filter = filter_list)

# Inspect the first 6 rows of the result
unemploymentOECD[1:6, ]
```

After running the above, `unemploymentOECD` will contain a data frame of the requested data. Each row will correspond to one combination of the dimensions we specified (country, sex, age, and time). Since we restricted sex to “MW” and age to “2024”, each country will have data for those categories over a series of years. The columns of the data frame typically include the dimensions and the measured value. For example, it might have columns like `COUNTRY`, `SEX`, `AGE`, `Time`, and `Value` (or similar naming). The first 6 rows might look like:

	COUNTRY	SEX	AGE	Time	Value
1	CAN	MW	2024	2005	12.3
2	CAN	MW	2024	2010	14.1
3	CAN	MW	2024	2015	10.7
4	DEU	MW	2024	2005	8.5
5	DEU	MW	2024	2010	7.9
6	DEU	MW	2024	2015	6.4
...					

*Note:* The above is illustrative; actual values and formatting may differ. The idea is that we get unemployment duration (perhaps measured in months or some index) for each country at age 20-24 for each year. We see Canada (CAN) and Germany (DEU) for the years 2005, 2010, 2015 in this snippet, with some values. The data likely goes through all years available up to the latest.

Using these functions, you can mix and match as needed: first find a dataset of interest (`search_dataset`), then get its structure (`get_data_structure`), then fetch data

(`get_dataset`). Sometimes, if you already know the dataset ID and the codes required, you can go straight to `get_dataset` and supply the filters.

One thing to be aware of is that the OECD API might have *time* as a separate dimension (like “TIME” or “Year”), or sometimes the year is part of the data frame’s row index. In the R package output, typically year/time is given as a column or included in the data frame explicitly. The examples above show it as `Time` or `year`. You can usually filter time by adding an element to the filter list or by using the `start_time` and `end_time` arguments in `get_dataset` (e.g., `start_time = 2000`, `end_time = 2020` to restrict to years 2000–2020).

In our example, because we left time unspecified in the filter list, `get_dataset` likely returned all years available for those parameters. We could have added, say, another element to `filter_list` for years (if the API expects a code for year) or more simply used `start_time`/`end_time` arguments.

### 7.2.3 `tl;dr`

```
# Loading OECD package
library(OECD)

# List all available datasets
dataset_list <- get_datasets()

# Search all datasets with the term "unemployment" in their description
search_dataset("unemployment", data = dataset_list)

# Examine the structure of a specific dataset (e.g., "DUR_D")
dstruc <- get_data_structure("DUR_D")
str(dstruc, max.level = 1)

# Define a filter to narrow the data (e.g., countries=DEU,FRA,CAN,USA; Sex=MW; Age=2024)
filter_list <- list(c("DEU", "FRA", "CAN", "USA"), "MW", "2024")

# Retrieve the dataset "DUR_D" with the specified filter
unemploymentOECD <- get_dataset(dataset = "DUR_D", filter = filter_list)
unemploymentOECD[1:6, ]
```

*(The above code shows the steps to search for a dataset related to “unemployment”, inspect its structure, and download a subset of that data. In practice, replace the search term and dataset ID with those relevant to your needs. Always adjust the filter list according to the actual dimensions of the dataset you are querying.)*

## 7.3 spiR

### 7.3.1 Database Description

The **Social Progress Index (SPI)** is a comprehensive measure of a country's social and environmental performance, independent of economic metrics. It was developed between 2009 and 2013 by the nonprofit organization Social Progress Imperative, with input from scholars and experts (including Michael Porter and others) to better capture human well-being and societal progress. The index is composed of 52 indicators that collectively measure how well countries provide for the essential needs of their citizens, establish the building blocks that allow citizens to improve their lives, and create conditions for all individuals to reach their full potential.

The 52 indicators of the SPI are grouped into three broad dimensions:

- **Basic Human Needs:** This includes indicators related to nutrition and basic medical care, water and sanitation, shelter, personal safety, etc. (Things like access to food, clean water, safe housing, and security are fundamental needs).
- **Foundations of Well-being:** This covers education, access to information, health and wellness, and environmental quality (e.g., literacy rates, school enrollments, access to technology and information, life expectancy, pollution levels, etc.).
- **Opportunity:** This dimension looks at personal rights, personal freedom and choice, inclusiveness, and access to advanced education (for example, indicators on political rights, freedom from discrimination, access to higher education, corruption, etc.).

Each of these three dimensions is further broken down into components, and each component is measured by several specific indicators. All 52 indicators together roll up into an overall Social Progress Index score for each country. The SPI is usually reported on an annual basis (in early years, not all countries were covered, but it has expanded over time). The goal of SPI is to provide a more holistic measure beyond GDP to evaluate how well societies are doing in converting economic gains into improved social outcomes.

The **spiR** package provides an interface to the Social Progress Imperative's data, allowing R users to retrieve SPI data and related information. It essentially wraps the SPI API (or data source) to make it easy to get data on various countries and indicators. This includes retrieving the overall SPI scores as well as specific component indicators if needed.

### 7.3.2 Functions

The **spiR** package offers a few key functions to work with the Social Progress Index data:

- **spir\_country()** – Search for countries and retrieve their ISO codes as used by the SPI database.



- **`spir_indicator()`** – Search for indicators in the SPI database and retrieve their codes (such as the code for a specific indicator or for the overall SPI).
- **`spir_data()`** – Download the actual data for specified countries, years, and indicators from the SPI dataset.

These functions are designed to help you find the correct arguments (country codes, indicator codes) and then pull the data. Let’s go through them one by one with examples.

### 7.3.2.1 `spir_country()`

To request data from the SPI API, you will need to use country codes (likely standardized codes, possibly ISO 3-letter country codes). The function `spir_country()` helps you find the correct country code for a given country name. It takes a country name (or partial name) as an argument and returns the matching country code(s) used in the SPI system.

For example, suppose we want to get data for Canada. We should confirm what code the SPI uses for Canada. We can do:

```
# Loading the spiR package
library(spiR)

# Get the ISO country code for "Canada"
myCountry <- spir_country("Canada")
myCountry
```

If we run this, `myCountry` will contain the result of the search. Likely, since “Canada” is a unique match, it will return a data frame or vector with Canada’s code. In many datasets, Canada’s code is “CAN” (ISO 3-letter code). The output might look like:

	country_name	iso3
1	Canada	CAN

So, `spir_country("Canada")` yields “CAN” as the code. If you search a more ambiguous string, like `spir_country("United")`, you might get multiple matches (e.g., United States, United Kingdom, United Arab Emirates, etc., each with their code). If you call `spir_country()` with no argument, it might list all available countries and their codes.

Knowing the country codes is helpful because the main data function `spir_data()` expects country codes as input.

### 7.3.2.2 `spir_indicator()`

Similarly, we need to know the indicator codes for the data we want. The SPI has one overall index (often code “SPI”) and many sub-indicators (each likely has its own code or name). The function `spir_indicator()` allows you to search the indicators by a keyword.

For instance, if we wanted to find indicators related to *mortality* (perhaps there’s an indicator about child mortality or something in the Basic Needs dimension), we could search:

```
# Search for an indicator containing "mortality"
myIndicator <- spir_indicator("mortality")
myIndicator
```

This will return any indicators whose name or description contains “mortality”. Suppose one of the SPI indicators is “Maternal Mortality Rate” or “Under-5 Mortality Rate”; the search might find it. The output could be something like:

	indicator_name	indicator_code
1	"Under 5 Mortality Rate (per 1,000 live births)"	"mortality_u5"
2	"Maternal Mortality Rate (per 100,000 live births)"	"mortality_maternal"
...		

(Exact codes and names are hypothetical here, for illustration.) The idea is that `spir_indicator` will provide you the code string that you need to use to request that indicator’s data.

If you call `spir_indicator()` without any argument, it may list all 52 indicators and their codes. This could be useful to see the whole catalog of what’s available, including the overall “SPI” score and all components.

For the purposes of this chapter, let’s assume we are interested in the overall Social Progress Index itself, which likely has an indicator code “SPI” for the aggregate index. That is probably what we’ll use in the example for data retrieval.

### 7.3.2.3 `spir_data()`

The function `spir_data()` is the main function to get actual data from the Social Progress Index database. It requires a few arguments:

- **country:** one or more country codes (using the codes we found via `spir_country`). This is usually a character vector, e.g., `c("USA", "FRA", "BRA")`.

- **years:** one or more years of interest (as character strings). You can specify a range or specific years, e.g., `c("2014", "2015", "2016")`. The SPI data in early years might not cover every year, but from 2014 onward they might have annual releases (for example, SPI 2014, SPI 2015, etc.).
- **indicators:** one or more indicator codes that you want to retrieve. For example, "SPI" for the overall index, or a specific code for a sub-index or component if desired.

Using these arguments, `spir_data` will fetch the data and return it, likely as a data frame where each row corresponds to a country-year combination and columns include the country, year, and the indicator values.

Let's retrieve the overall Social Progress Index for a set of countries over several years. For example, we'll get the SPI from 2014 through 2019 for the USA, France, Brazil, China, South Africa, and Canada. We already suspect the indicator code is "SPI" for the main index (we could confirm that via `spir_indicator("Social Progress Index")` if needed). And we have country codes: USA, FRA, BRA, CHN, ZAF (South Africa), CAN.

```
# Extracting SPI data for selected countries and years
myData <- spir_data(country = c("USA", "FRA", "BRA", "CHN", "ZAF", "CAN"),
                    years   = c("2014", "2015", "2016", "2017", "2018", "2019"),
                    indicators = "SPI")

head(myData)
```

Once this runs, `myData` will contain the SPI values for those countries and years. We used character vectors for years in the function call (notice the quotes around the years). In many cases, years could be numeric, but perhaps the API expects them as strings — using quotes ensures they're treated as text. The output (as seen by `head(myData)`) might look like:

	country	iso3	indicator	year	value
1	Brazil	BRA	SPI	2014	67.91
2	Brazil	BRA	SPI	2015	69.58
3	Brazil	BRA	SPI	2016	70.40
4	Brazil	BRA	SPI	2017	71.00
5	Brazil	BRA	SPI	2018	72.29
6	Brazil	BRA	SPI	2019	72.89

This is an example for one country (Brazil) across years 2014–2019, showing the SPI score (on some scale, possibly 0–100) for each year. In the actual `myData`, all requested countries would be present, so you'd also have rows for USA, France, etc. The columns likely include the country name, the country code (iso3), the indicator (SPI), the year, and the value (the score).

You could then use this data frame to compare how different countries' social progress scores have changed over time. For instance, you might plot each country's SPI over the years.

The `spir_data` function can also retrieve multiple indicators at once if you provide a vector of indicator codes. Then the data frame would have multiple columns (or multiple rows per indicator, depending on how it's structured – often such API wrappers return a long format where each row is a single indicator for a single country-year, meaning you'd get an extra column for indicator code as shown above, and a value column). In the example above, since we only requested "SPI", we see one row per country-year. If we had requested additional indicators, we might see multiple rows per country-year (one for each indicator), or the function might pivot it into columns. One would need to check the documentation for exact behavior. But you can always reshape the data after retrieval as needed.

Finally, if you want to explore more about what you can do with the SPI data, the `spir` package documentation or the in-depth application (possibly referenced by the `warin.ca` post) can provide more examples, such as making dashboards or visualizations similar to those on the Social Progress Imperative's own site.

### 7.3.3 tl;dr

```
# Loading the spir package
library(spir)

# Find the ISO code for a specific country (e.g., Canada)
myCountry <- spir_country("Canada")
myCountry # should return "CAN" for Canada

# Search for an indicator by keyword (e.g., "mortality")
myIndicator <- spir_indicator("mortality")
myIndicator # returns any indicator codes that include "mortality"

# Retrieve data for the Social Progress Index (SPI) for selected countries and years
myData <- spir_data(country = c("USA", "FRA", "BRA", "CHN", "ZAF", "CAN"),
                    years   = c("2014", "2015", "2016", "2017", "2018", "2019"),
                    indicators = "SPI")
head(myData)
```

*(The above code demonstrates how to look up country and indicator codes and how to fetch the overall SPI data for a set of countries over a range of years. In practice, you can use `spir_indicator()` to find other indicators and replace "SPI" with the code of any specific sub-indicator if you wish to retrieve those.)*

## 7.4 statcanR

### 7.4.1 Database Description

**Statistics Canada** (often abbreviated as *StatCan*) is the national statistical agency of Canada. It produces a vast amount of data on Canada's economy, society, and environment. The Statistics Canada open data portal includes data on about 30 broad subjects, including agriculture, energy, environment, education, health, economics, demographics, and more. These data are available at various geographic levels, such as national (Canada), provincial/territorial, metropolitan areas, etc., depending on the dataset.

Historically, much of StatCan's data was accessible via something called CANSIM tables (Canadian Socio-economic Information Management system). In recent years, they modernized their platform and now refer to data tables by a Product ID (PID) code like "27-10-0014-01". The StatCan Open Data API (also known as the Web Data Service) allows programmatic access to these tables. Each table's data can be retrieved by referencing its table number or product ID.

The **statcanR** package provides a user-friendly way for R users to access Statistics Canada data. It essentially wraps around the web service API so that given a table ID, it will fetch the data and return it as an R data frame or tibble. This saves you from manually downloading CSV files or writing your own HTTP requests.

However, one challenge is that you need to know the table's ID before you can fetch it (the API requires a specific table number). The **statcanR** workflow typically involves two steps:

1. **Find the table ID for the data you want.** This is often done by using the Statistics Canada website's search, since **statcanR** itself doesn't provide a search function in the package (as of the information we have). You might use the StatCan data portal or an online search to identify the table number for your topic of interest.
2. **Use `statcan_data()` to retrieve that table's data.** Once you have the table ID, you call the function and it returns the data.

Let's go through these steps with an example.

### 7.4.2 Functions

The main function in **statcanR** we will use is **statcan\_data()**. But before using it, we usually need to find the table ID (unless we already know it). So we can think in terms of:

- **Searching for data (table ID) outside the package.** The user might have to go to the StatCan website or use an index of table IDs.
- **statcan\_data()** – the function to fetch data given a table ID and language.

*(There isn't a dedicated search function like `statcan_search()` in this package as far as the provided material suggests. Instead, the guidance is to use the website to find the ID.)*

#### 7.4.2.1 Search for data (Finding the table ID)

To find a Statistics Canada table ID for the data you want, you can use the official StatCan data portal search. For example, let's say we are interested in "federal expenditures on science and technology by socio-economic objectives." We could go to the StatCan website's data search page (the URL given in the materials is <https://www150.statcan.gc.ca/n1/en/type/data?MM=1> which is a general data search page). On that page, typing a few keywords like "federal expenditures science technology socio-economic objective" should bring up relevant results.

Assume we do that search on the website. The search results might list a table with exactly that description. In the provided content, the example found that the table number for "Federal expenditures on science and technology by socio-economic objectives" is **27-10-0014-01**. This is the unique identifier for that dataset.

StatCan table IDs usually have a format like two digits, two digits, four digits, two digits (with hyphens in between). For instance, 27-10-0014-01:

- The first two digits (27) might represent a category.
- The next two (10) perhaps sub-category or just part of the coding.
- 0014 is the specific table number, and 01 might indicate the version (like some tables get updated structure over time and the last two digits change).

Anyway, once we have this ID, we are ready to fetch the data via the API.

#### 7.4.2.2 `statcan_data()`

The function `statcan_data(table, lang)` fetches the data for a given table number. It has two main arguments:

- The first argument is the **table number** as a string (e.g., "27-10-0014-01").
- The second argument is the **language** of the data ("eng" for English or "fra" for French). StatCan publishes data in both official languages, and sometimes the table content (like column names or category labels) can be fetched in either language.

In our example, we'll use the table ID we found, "27-10-0014-01", and request the data in English.

```
# Loading the statcanR package
library(statcanR)

# Fetch data from Statistics Canada table 27-10-0014-01 in English
mydata <- statcan_data("27-10-0014-01", lang = "eng")

# Examine the first few rows of the data
head(mydata)
```

After running this, `mydata` will contain the data frame for that table. The `head(mydata)` will show the first several rows. The structure of the data depends on the table. StatCan tables are often structured in a long format where each row is a combination of the classification dimensions with a value. The columns might include things like `REF_DATE` (the time period, e.g., year), and various other dimensions like geographical area, indicator, etc., depending on what the table is about, plus a `VALUE` column for the numeric value.

For instance, since this is “federal expenditures on S&T by socio-economic objective”, the dimensions might be Year, maybe type of expenditure or objective category, etc. We might see columns like:

REF_DATE	GEO	Objective	VALUE
2018	Canada	Defence	500.0
2018	Canada	Economic development	300.0
2018	Canada	...	...
2019	Canada	Defence	520.0
...			

(This is hypothetical data to illustrate the format.) Essentially, each row is one category of expenditure in a given year, with the value being the amount spent (maybe in millions of dollars, etc.). The actual table likely has a specific breakdown.

One nice thing is that `statcan_data()` probably returns a tibble (which is a modern type of data frame in R) with proper column names and factor labels in English (since we requested `lang="eng"`). If we had requested French (`lang="fra"`), the labels for the objectives and possibly the column names would appear in French.

At this point, we have the data needed and can proceed to analyze or visualize it. For example, we could sum up various objectives or see trends over time.

The `statcanR` package makes it straightforward to get the latest data for that table without manually downloading the CSV from the website. Moreover, if the table gets updated with new data (for a new year, for example), running `statcan_data()` again at a later date would fetch the updated data (assuming the table ID remains the same).

One should note: to use the StatCan API without this package, you might normally have to know the API endpoint and possibly handle CSV or JSON. `statcanR` abstracts that away – you just provide the table ID.

If you are curious for more, the `statcanR` documentation or the referenced blog post may have more examples (such as dealing with very large tables or manipulating the results). But the essential part is covered: find the table ID and use `statcan_data()`.

### 7.4.3 `tl;dr`

```
# Loading the statcanR package
library(statcanR)

# Use statcan_data() to retrieve a specific table by its ID (English version)
mydata <- statcan_data("27-10-0014-01", "eng")

# View the first few rows of the retrieved data
head(mydata)
```

*(In practice, replace "27-10-0014-01" with the table ID of the dataset you need from Statistics Canada. Use "eng" for English or "fra" for French as the language argument. Remember to find the table ID via the StatCan website or documentation before using this function.)*

## 7.5 EpiBibR

### 7.5.1 Database Description

The **EpiBibR** package is an R wrapper designed to provide easy access to a large bibliographic dataset, particularly focused on COVID-19 and other related medical research references. During the COVID-19 global crisis, the volume of scientific literature on the topic skyrocketed. Having a comprehensive bibliographic database of COVID-19 research (articles, preprints, letters, news articles, etc.) is valuable for researchers conducting literature reviews, trend analysis, or bibliometric studies. EpiBibR was created to make over 100,000 such references available directly through R.

In essence, EpiBibR contains (or connects to) a database of bibliographic entries (like what you'd find in PubMed or other scholarly databases) that are related to epidemiology and specifically COVID-19. Each entry in the database includes various fields, such as authors, title, abstract, publication year, journal, etc. This is analogous to having a huge bibliography or library catalog that you can query with code.



To give an idea of what information each reference record contains, here are some of the fields available (with their typical tags):

- **AU** – Authors (the list of authors of the paper)
- **TI** – Title of the document
- **AB** – Abstract of the paper
- **PY** – Publication Year
- **DT** – Document Type (e.g., Article, Letter, News, etc.)
- **MESH** – Medical Subject Headings (keywords/topics assigned)
- **TC** – Times Cited (citation count, if available)
- **SO** – Source (publication name, e.g., journal or news source)
- **J9** – Source abbreviation
- **JI** – ISO source abbreviation
- **ISSN** – International Standard Serial Number (journal identifier)
- **VOL, ISSUE** – Volume and Issue number (for journal articles)
- **ID** – PubMed ID (if applicable)
- **DE** – Authors’ Keywords (keywords given by authors)
- **UT** – Unique Article Identifier (possibly Web of Science ID or similar)
- **AU\_CO** – Author’s Country of Origin (which might be derived from author affiliations)
- **DB** – Database from which the record is sourced (e.g., which bibliographic database)

The above is a lot of information – essentially, EpiBibR is giving you a bibliographic dataset akin to a large reference manager file that you can query. The typical use would be: you query the data for certain criteria (like author name, year, keywords, etc.) and get back a subset of references matching those criteria.

## 7.5.2 Functions

The EpiBibR package provides a main function for data retrieval and allows filtering by various fields:

- **epibibr\_data()** – The primary function to retrieve bibliographic references, with arguments that allow filtering by author, country, year, title keywords, abstract keywords, and source (journal name) among others.

The usage of **epibibr\_data()** is very flexible: you can provide none, one, or multiple filters. Without any arguments, it returns the entire dataset (which is huge). With arguments, it filters accordingly.

Let’s go through some examples, as given in the content:

### 7.5.2.1 `epibibr_data()`

- **Retrieving the entire dataset:** If you simply call `epibibr_data()` with no arguments, it will try to retrieve the entire bibliography data frame which contains all references (80,000+ or even 100,000+ entries). For example:

```
library(EpiBibR)
complete_data <- epibibr_data()
```

This command would populate `complete_data` with the entire bibliographic database. This might be quite large in memory, so often you might not want to do this unless you truly need everything. Instead, you might retrieve a subset based on some criteria.

- **Filtering by author:** If you want all references authored by a certain person, you can use the `author` argument. For example, to get all articles written by someone with last name Colson (as in Philippe Colson, a microbiologist who authored many COVID-19 papers):

```
colson_articles <- epibibr_data(author = "Colson")
```

This will search the author field for “Colson” and return all entries where at least one author matches that name. The result `colson_articles` would be a data frame of all such references. Each entry would include all the fields (Title, Year, etc.) for papers that have Colson as an author. You might then check `nrow(colson_articles)` to see how many papers he authored in the database, for instance.

- **Filtering by author and year:** You can combine filters. The function allows multiple arguments to narrow the search. For example, if we want references authored by someone named Yang in the year 2020:

```
yang2020 <- epibibr_data(author = "Yang", year = "2020")
```

This will give all records where an author’s name contains “Yang” **and** the publication year is 2020. The result `yang2020` would contain only those references meeting both criteria (logical AND between filters).

- **Filtering by author’s country:** If we want to find references based on the country of origin of the authors (perhaps the country of the corresponding author or an author affiliation country), we can use the `country` argument. For example, to get all references with at least one author from Canada:

```
canada_articles <- epibibr_data(country = "Canada")
```

This will search the author address/affiliation field for “Canada” and return those references.

- **Filtering by title keyword:** If we want to find articles that have a certain keyword in the title, we use the `title` argument. For example, to get all references whose titles contain “covid”:

```
covid_articles <- epibibr_data(title = "covid")
```

This will likely return a lot of references (since many will have “COVID” in the title). The search might be case-insensitive and probably looks for the substring “covid” in the title.

- **Combining multiple criteria:** As mentioned, we can refine searches by using multiple arguments at once, and the result will satisfy all given filters. For instance, we might want references authored by someone named “Yang”, that have “covid” in the title, and were published in 2020:

```
yangcovid2020_articles <- epibibr_data(author = "Yang", title = "covid", year = "2020")
```

This will find references where *all three* conditions are true (author includes “Yang”, title includes “covid”, year is 2020). We could add even more criteria, for example, also specifying a source.

- **Adding source as another filter:** The `source` argument can filter by publication source (like journal or conference name). For example, to refine the above search to only include those references in sources whose name contains “bio” (maybe “BioRxiv” or “Biology” etc.):

```
yangcovid2020bio_articles <- epibibr_data(author = "Yang", title = "covid", year = "2020", source = "bio")
```

Now the references must meet all four filters: an author name containing “Yang”, title containing “covid”, year 2020, and source containing “bio”. This will significantly narrow it down – possibly to references by authors named Yang in 2020 about COVID in some biology-related journals or preprint servers.

- **Filtering by abstract keyword:** We can also search within the abstract text of the references using the `abstract` argument. For example, to find references that mention “coronavirus” in the abstract:

```
coronavirus_articles <- epibibr_data(abstract = "coronavirus")
```

This will return references whose abstracts contain the word “coronavirus”. This is a powerful way to find papers that might be about coronaviruses even if the title doesn’t explicitly say so.

All these filters can be used in combination or standalone. The result of any `epibibr_data` call is a data frame (or tibble) with the references that match. Each row is one reference, with columns corresponding to fields like author, title, year, etc. likely using the abbreviations or full names of those fields. For example, the resulting data frame might have columns named

AU, TI, AB, PY, SO, etc., or possibly more user-friendly names. If integrating with bibliometrix (as hinted, since they designed it to integrate with the bibliometrix package), it might preserve standard field tags so bibliometrix can read it easily.

One thing to keep in mind is that text searches (like `title = "covid"`) will probably match anywhere in the field, so “COVID-19” or “covid19” etc. would match because “covid” is a substring. Similarly, `author = "Yang"` might match “Yang” as a surname but also “Yangus” or anything with those letters — though likely it’s intended to match last names exactly or something. The specifics depend on how the search is implemented (perhaps it treats the input as a case-insensitive substring search).

The ability to combine filters means you can tailor very specific queries, which is great for slicing the data (for example, find how many papers a particular author wrote in a given year on a certain topic).

One caution: If you combine too many filters that don’t have overlapping results, you might get zero results. For instance, if no author named Yang wrote a COVID-titled paper in 2020 in a source with “bio” in its name, then `yangcovid2020bio_articles` would be empty (0 rows).

EpiBibR essentially puts a research literature database at your fingertips. A researcher could use this to do things like trend analysis (how many COVID papers per year, etc.), network analysis of collaborations (using authors and their countries), or topic modeling on abstracts, etc.

### 7.5.3 `tl;dr`

```
# Loading the EpiBibR package
library(EpiBibR)

# Retrieve the entire dataset (all references) - large output
epidata <- epibibr_data()

# Examples of filtered searches:
complete_data <- epibibr_data() # same as epidata, full dataset

colson_articles <- epibibr_data(author = "Colson")           # all references with an author
yang2020 <- epibibr_data(author = "Yang", year = "2020")    # references authored by "Yang"
canada_articles <- epibibr_data(country = "Canada")         # references where an author's
covid_articles <- epibibr_data(title = "covid")             # references with "covid" in tl
```

```

yangcovid2020_articles <- epibibr_data(author = "Yang", title = "covid", year = "2020")
# references that satisfy: author name has "Yang", title has "covid", and year is 2020

yangcovid2020bio_articles <- epibibr_data(author = "Yang", title = "covid", year = "2020", source = "bio")
# further narrows above: in addition, source contains "bio" (perhaps BioRxiv or similar)

coronavirus_articles <- epibibr_data(abstract = "coronavirus")
# references with "coronavirus" in the abstract

```

*(The above code block demonstrates how to use `epibibr_data()` in various ways: with no filters (all data) and with different combinations of filters for author, year, country, title, source, and abstract. These are examples – you can adjust the strings to search for different authors, keywords, years, etc., depending on your research needs.)*

## 7.6 coronavirus

### 7.6.1 Database Description

The `coronavirus` package is an R package that provides a tidy formatted dataset of the 2019 Novel Coronavirus COVID-19 outbreak, sourced from the Johns Hopkins University Center for Systems Science and Engineering (JHU CSSE) data repository. In early 2020, the JHU CSSE team started compiling daily reports of COVID-19 cases and deaths from around the world, and they made this data public via a GitHub repository. This quickly became one of the standard references for tracking the global spread of COVID-19.

The `coronavirus` R package (maintained by Rami Krispin and others as part of the **Covid19R Project**) takes that JHU data and puts it into a consistent, **tidy format** for easier analysis in R. “Tidy format” here means a long-form data frame where each row represents a single observation (for a given date and location), and columns represent variables like the date, location, case type, and number of cases.

Specifically, the dataset in this package includes daily counts of COVID-19 confirmed cases, deaths, and recoveries (where available) for each region. By region, we mean often country level, and in some cases sub-regions (e.g., states or provinces for large countries like the US, Canada, China, Australia, etc., since JHU dataset had those breakdowns). The data covers from the beginning of the pandemic (around January 2020) and continues through the course of 2020 and 2021, etc. Notably:

- It has a column for the date of the observation.
- Columns for geographic information (province/state, country/region, and possibly coordinates).

- A column indicating the type of case (usually “confirmed”, “deaths”, or “recovered”).
- A column for the number of cases on that date (could be daily new cases or cumulative, but in the tidy JHU data, typically each row is daily counts).
- Additional columns such as ISO country codes, population, etc., might be included to enrich the data.

According to the package’s updated description, the dataset includes daily new cases and death cases from January 2020 through March 2023, and recovery cases up until August 2022. This reflects the fact that JHU stopped tracking certain metrics at certain times (they stopped updating recoveries in mid-2022 and stopped all data collection in March 2023 as the pandemic data collection wound down).

The `coronavirus` package internally contains a snapshot of the data (which is periodically updated on CRAN) and also provides utilities to update it from the source.

## 7.6.2 Functions

The key functions and usage for this package are:

- `data("coronavirus")` – This isn’t a function per se, but a way to load the included dataset. The package comes with a dataset named `coronavirus`. By running `data("coronavirus")`, you load that dataset into your R environment.
- `update_dataset()` – This function will fetch the latest version of the data from the package’s GitHub source, updating the in-memory dataset (and possibly the package’s internal data, requiring a restart to use). Essentially, it lets you get the most recent data beyond what was shipped with the CRAN version.
- `refresh_coronavirus_jhu()` – This is an alternative function (part of the Covid19R project standard) that pulls data directly from the JHU repository and returns it as a data frame, without relying on the package’s built-in dataset. It ensures you get the absolute latest data in the standardized format.

Let’s see how these work:

### 7.6.2.1 `data("coronavirus")`

After installing and loading the `coronavirus` package, you can load the dataset by calling:

```
library(coronavirus)

data("coronavirus")
```

This will make a data frame called `coronavirus` available in your R session (basically attaching the dataset). If you then do:

```
head(coronavirus)
```

You will see the first few rows of the dataset. The dataset's columns (as of the Covid19R format) include:

- **date** – the date of observation (class Date).
- **province** – the name of the state/province (if applicable; could be NA or empty for country-level if no subregion).
- **country** – the country/region name.
- **lat** – latitude coordinate of the location (for mapping).
- **long** – longitude coordinate.
- **type** – the type of case (“confirmed”, “death”, or “recovered”).
- **cases** – the number of cases on that date (for that type, location).
- **uid** – a unique identifier (like a numeric country code possibly).
- **province\_state** – (sometimes duplicate info of **province**, depending on source).
- **iso2, iso3, code3** – ISO 2-letter, ISO 3-letter, and numeric country codes.
- **fips** – FIPS code (for US counties, etc., if applicable).
- **combined\_key** – a combined location key (like “Country, State” string).
- **population** – population of that region (useful for per-capita calculations).
- **continent\_name, continent\_code** – the continent of that country.

That's a lot of columns, but essentially each row might look like:

date	province	country	lat	long	type	cases	... etc.
2020-01-01	<NA>	China	30.0	112.5	confirmed	1	
2020-01-01	<NA>	China	30.0	112.5	death	0	
...							
2020-03-15	New York	US	40.7	-73.9	confirmed	730	
2020-03-15	New York	US	40.7	-73.9	death	2	
...							

(Just illustrative, not actual values.)

So by doing `head(coronavirus)`, you might see the earliest entries (which might be rows for China in January 2020, as shown above).

The dataset in the CRAN version might not be the absolutely latest if the pandemic is ongoing. CRAN versions were updated every month or two, as noted. That's where `update_dataset()` comes in.

### 7.6.2.2 `update_dataset()`

The `update_dataset()` function is provided to let users of the CRAN version get the most recent data without waiting for the next CRAN release. When you call `update_dataset()`, it will attempt to fetch the latest data from the GitHub (development) version of the package (or directly from the JHU source) and then update the `coronavirus` dataset in your R environment.

For example:

```
# Assuming library(coronavirus) is already called
update_dataset()
```

After running this, you should re-run `data("coronavirus")` or otherwise reload the data, possibly by re-initializing the package or simply by using the updated in-memory data (the documentation suggests restarting R for the changes to take effect). Essentially, `update_dataset()` will download the new CSV from the JHU repo (if available) and replace the package's internal data file.

One important note mentioned is that **you must restart the R session** to have the updates available. This implies that `update_dataset()` writes to the package's environment or a global environment but you might need to reload it to see the updated dataset as `coronavirus`. A safer approach might be to use the next function if you want fresh data on the fly without needing to restart.

### 7.6.2.3 `refresh_coronavirus_jhu()`

The function `refresh_coronavirus_jhu()` pulls the data directly and returns it as a data frame (without needing to load it as `coronavirus`). This function is part of a standardized approach by the Covid19R project. It fetches the data from the JHU repository (which contains daily time series or daily reports) and returns it in the same standardized format as the `coronavirus` dataset.

Example usage:

```
covid19_df <- refresh_coronavirus_jhu()
head(covid19_df)
```

After this, `covid19_df` will be a data frame with the latest data. Doing `head(covid19_df)` should show the first few rows (which likely correspond to the earliest date's data, similar to what `coronavirus` initial entries would show). Since `refresh_coronavirus_jhu()` always goes to source, it ensures you get updates even beyond what `update_dataset()` can do (which might be limited by how the package was built).



As of early 2023, JHU has ceased updates (the last date with data is March 10, 2023 for new cases, and August 2022 for recoveries). So beyond that, the dataset is static. But from 2020 through early 2023, this was actively updated daily.

For historical or offline analysis, using the packaged data or the update function suffices. For real-time tracking (when the data was live), one would likely use `refresh_coronavirus_jhu()` daily to get new numbers and then merge or analyze.

### 7.6.3 tl;dr

```
# Loading the coronavirus package
library(coronavirus)

# Load the bundled coronavirus dataset
data("coronavirus")

# Peek at the data structure and first rows
head(coronavirus)

# Update the dataset to the latest available data (requires restarting R to use updated data)
update_dataset()

# Alternatively, fetch the latest JHU COVID-19 data directly into a data frame
covid19_df <- refresh_coronavirus_jhu()
head(covid19_df)
```

*(The above code demonstrates how to load the COVID-19 dataset included in the package, how to update it to the latest data, and how to directly retrieve the latest data via the JHU source. Use `head()` or similar to inspect the data. Remember that after using `update_dataset()`, you might need to restart R to reload the updated `coronavirus` dataset.)*

**TL;DR** – *Summary of Key Code from this Chapter*

```
# Loading the WDI package and searching/downloading data
library(WDI)
listOfIndicators <- WDIsearch("GDP") # search indicators for "GDP"
listOfIndicators[1:5, ] # view first 5 results
stockTraded <- WDI(indicator = "CM.MKT.TRAD.GD.ZS",
                    country   = c("FR", "CA", "US", "CN"),
                    start     = 2000,
                    end       = 2016)
```

```

head(stockTraded)

# Loading the OECD package and retrieving a filtered dataset
library(OECD)
dataset_list <- get_datasets() # get list of all datasets
search_dataset("unemployment", data = dataset_list) # find datasets related to "unempl
dstruc <- get_data_structure("DUR_D") # get structure of a specific data
str(dstruc, max.level = 1)
filter_list <- list(c("DEU", "FRA", "CAN", "USA"), "MW", "2024") # set up filters (countries
unemploymentOECD <- get_dataset(dataset = "DUR_D", filter = filter_list)
unemploymentOECD[1:6, ]

# Loading the spiR package and retrieving Social Progress Index data
library(spiR)
mycountry <- spir_country("Canada") # lookup country code for Canada
mycountry
myIndicator <- spir_indicator("mortality") # search indicators with "mortality"
myIndicator
myData <- spir_data(country = c("USA", "FRA", "BRA", "CHN", "ZAF", "CAN"),
                    years = c("2014", "2015", "2016", "2017", "2018", "2019"),
                    indicators = "SPI")
head(myData)

# Loading the statcanR package and fetching a StatCan data table
library(statcanR)
mydata <- statcan_data("27-10-0014-01", "eng") # get data for table 27-10-0014-0
head(mydata)

# Loading the EpiBibR package and querying the bibliographic database
library(EpiBibR)
epidata <- epibibr_data() # load entire dataset (very large)
complete_data <- epibibr_data() # (same as above)
colson_articles <- epibibr_data(author = "Colson") # papers authored by someone named
yang2020 <- epibibr_data(author = "Yang", year = "2020") # papers by "Yang" in 2020
canada_articles <- epibibr_data(country = "Canada") # papers with an author from Canada
covid_articles <- epibibr_data(title = "covid") # papers with "covid" in the title
yangcovid2020_articles <- epibibr_data(author = "Yang", title = "covid", year = "2020")
yangcovid2020bio_articles <- epibibr_data(author = "Yang", title = "covid", year = "2020", s
coronavirus_articles <- epibibr_data(abstract = "coronavirus")

# Loading the coronavirus package and using its dataset
library(coronavirus)

```

```
data("coronavirus")
head(coronavirus)
update_dataset()
covid19_df <- refresh_coronavirus_jhu()
head(covid19_df)
```

### Commands introduced in this chapter and their purpose:

Command	Detail / Purpose
<code>WDIsearch()</code>	Search for World Bank indicators by keyword. Returns a list of indicator codes and names that match the search term.
<code>WDI()</code>	Retrieve data for specified World Bank indicator(s) and country(ies). Allows specifying start and end years.
<code>get_datasets()</code>	List all available OECD datasets (IDs and descriptions). Useful as a first step to see what data can be accessed.
<code>search_datasets()</code>	Search within the OECD dataset list for a keyword. Helps find the dataset ID relevant to a topic.
<code>get_data_struct()</code>	Get the structure (dimensions and valid codes) of an OECD dataset. Use this to determine how to filter <code>get_dataset</code> queries.
<code>get_dataset()</code>	Download data from a specific OECD dataset, optionally filtering by dimensions (e.g., countries, years, etc.). Returns a data frame of the selected data.
<code>spir_country()</code>	Look up country codes in the Social Progress Index database by country name. Helps to get the correct country ISO3 codes.
<code>spir_indicator()</code>	Look up indicator codes in the Social Progress Index database by keyword. Helps to find the code for a specific SPI indicator.
<code>spir_data()</code>	Retrieve Social Progress Index data for given countries, years, and indicator(s). Returns a data frame of SPI values.
<code>statcan_data()</code>	Fetch data from a Statistics Canada table given its table ID and language. Returns a data frame of that table's data.
<code>epibibr_data()</code>	Query the bibliographic database for COVID-19/medical references. Can filter by author, year, country, title, abstract, source. Returns a data frame of references matching the criteria.
<code>data("coronavirus")</code>	Load the included COVID-19 dataset from the <code>coronavirus</code> package. Provides a tidy data frame of daily cases.
<code>update_dataset()</code>	Update the <code>coronavirus</code> package's dataset to the latest available data from the source. (Requires restarting R to use the updated data.)
<code>refresh_coronavirus_jhu()</code>	Download the latest COVID-19 data from the JHU source in the standardized format. Returns a fresh data frame without needing a package update.

## 8 Debugging: Strategies, Tools, and Best Practices

Programming is **not** a bug-free process. In fact, encountering errors and bugs is a normal (if sometimes frustrating) part of coding. Debugging – the art of finding and fixing those bugs – is an essential skill for any data scientist. By learning to debug effectively, you not only solve the immediate problem but also deepen your understanding of how your R code works. As computer scientist Seymour Papert once noted, traditional schooling teaches that errors are bad and to be avoided, but *“the debugging philosophy suggests an opposite attitude. Errors benefit us because they lead us to study what happened, to understand what went wrong, and, through understanding, to fix it”*. In other words, every error is an opportunity to learn and improve.

That said, debugging can be challenging and sometimes **frustrating**, especially for beginners. You might feel stuck or discouraged when your code doesn’t work. This chapter aims to help you build a positive debugging mindset – to view bugs as puzzles to solve rather than as failures. It’s important to remember that **even experienced R programmers spend a lot of time debugging**. No one writes perfect code on the first try. In fact, many seasoned coders readily “use Google and support websites like Stack Overflow to ask for help with their R errors”. So if you find yourself searching the web for an error message, you’re in good company.

In the pages ahead, we will demystify the debugging process in R. We’ll start by categorizing the types of errors you might encounter and how R reports them. Then, we’ll introduce a variety of strategies and tools for debugging R code – from simple techniques like reading error messages carefully and adding `print()` statements, to powerful tools like `traceback()`, the RStudio debugger, and more. We’ll also walk through **common R error messages** and explain what they mean, so that you can recognize and fix them quickly. By the end of this chapter, you should feel more confident in tackling bugs and perhaps even come to “make friends with failure,” adopting an open mindset that treats each bug as a chance to learn.

Before diving in, take a deep breath. Debugging requires patience and a bit of detective work. Stay curious about what your code is doing, and try to remain calm and systematic when an error pops up. With practice, you’ll find that debugging can transform from a source of frustration into a rewarding process of discovery – turning “errors into solutions and frustration into satisfaction.” And remember: **every bug you fix makes you a better programmer**.

## 8.1 Learning Objectives

By the end of this chapter, you will be able to:

- **Identify different types of errors** in R (syntax errors, runtime errors, and logical errors) and understand how they arise.
- **Interpret R’s error and warning messages** to glean clues about what went wrong in your code.
- **Apply systematic debugging strategies** such as isolating problematic code, simplifying your code or data, and reproducing errors reliably.
- **Use R’s built-in debugging tools** like `traceback()`, `browser()`, `debug()`, and `recover()` to locate and diagnose problems in functions.
- **Leverage RStudio’s debugging features** (breakpoints, step-through execution, environment inspection) to debug code interactively.
- **Recognize common R error messages** (e.g. *“object not found”*, *“arguments imply differing number of rows”*, *“missing value where TRUE/FALSE needed”*) and understand their typical causes and solutions.
- **Develop effective debugging workflows** to fix errors in an organized way, from reading error messages and Googling for solutions to testing fixes and preventing future bugs.
- **Maintain a healthy debugging mindset** – viewing bugs as learning opportunities, staying persistent and curious, and using techniques like rubber-duck debugging to work through tough problems.

## 8.2 Types of Errors

Not all “bugs” are created equal. Broadly, errors in programming can be categorized into three types: **syntax errors**, **runtime errors**, and **logical errors**. Understanding these categories will help you diagnose problems more efficiently.

### 8.2.1 Syntax Errors

**Syntax errors** occur when your code violates the grammatical rules of the R language. Just as a sentence with a missing parenthesis or a stray comma can confuse a reader, a line of R code with a typo or missing symbol will confuse the R interpreter. Syntax errors are usually caught **as soon as you try to run the code**, because R cannot even parse (understand) the code to execute it.

Common causes of syntax errors in R include forgetting a closing parenthesis or quote, missing commas between arguments, or misspelling a keyword. For example, if you forget a parenthesis in a function call, you might see an error like this:

```
# Missing a closing parenthesis in the mean() call
mean(c(1, 5, 10, 52) # syntax error: one parenthesis is missing
#> Error in parse(text = x, srcfile = src): <text>:1:19: unexpected end of input
#> 1: mean(c(1, 5, 10, 52)
#>                      ^
```

Here, R is telling us that it reached the end of the line but was “expecting” something (in this case, a `)` to close the `mean()` call). The pointer `^` indicates where R got confused. Another common syntax error is an **unmatched quote**, for example:

```
message("Hello, world)
#> Error: unexpected string constant in "message(\"Hello, world"
```

In this case, the closing quote is missing, so R doesn’t know where the string ends. Similarly, a missing comma between function arguments can lead to an error or an unexpected result. For instance:

```
data <- data.frame(x = 1:5, y = 6:10 z = 11:15) # missing comma between 6:10 and z
#> Error: <text>:1:35: unexpected symbol
#> 1: data.frame(x = 1:5, y = 6:10 z
#>                               ^
```

R encountered `z` where it didn’t expect it – because we intended `y = 6:10, z = 11:15` with a comma. The error “unexpected symbol” hints that something (a comma) is likely missing before that `z`.

The key with syntax errors is that **R cannot run your code at all until you fix the syntax**. The error messages for syntax issues often include phrases like “unexpected symbol/number/string” or “unexpected end of input,” along with a position in the code. Luckily, syntax errors are usually straightforward to fix once you spot the problem – it’s often a matter of adding a missing parenthesis, comma, quote, or correcting a typo. Modern code editors (like RStudio) also help by highlighting mismatched braces or quotes to prevent these mistakes.

## 8.2.2 Runtime Errors

**Runtime errors** (also called **execution errors**) occur **while the code is running**. These happen when R successfully parses your code (no syntax issues) but encounters a problem during execution. In other words, the code is grammatically correct, but R can’t carry out an operation you asked for. When a runtime error occurs, R will stop executing that code and print an error message.

There are many possible causes of runtime errors, for example:

- Referring to an object that doesn't exist (perhaps due to a spelling mistake or forgetting to create it).
- Performing an illegal operation, like dividing by zero or taking a logarithm of a negative number.
- Passing a value of the wrong type to a function (e.g., giving text to a mathematical function that expects numbers).
- Trying to access elements outside the bounds of a vector or data structure.

Consider this simple example of a runtime error:

```
# Attempt to use a variable that hasn't been defined
print(result)
#> Error in print(result) : object 'result' not found
```

R throws an error because we tried to `print()` an object that doesn't exist in the current environment. The message *“object ‘result’ not found”* is a clear hint: the variable `result` was never created or is not in scope. This kind of error is extremely common for beginners (and even experienced users) – perhaps you meant to name a variable differently, or you ran code in the wrong order. The solution is to **ensure the object is defined** (and correctly spelled) before you use it.

Another example is performing an operation on incompatible types:

```
# Trying to add a character string to a number
5 + "10"
#> Error in 5 + "10" : non-numeric argument to binary operator
```

Here, the error *“non-numeric argument to binary operator”* occurs because we attempted to use the `+` operator (a binary arithmetic operator) on a number and a character string. R doesn't know how to “add” a text value to a number, so it stops with an error. The fix would be to convert the string “10” to numeric (using `as.numeric()`), or ensure that both operands are numeric.

Runtime error messages in R usually have the format **“Error in ... : description”**. For instance, *“Error in sqrt(“hello”): non-numeric argument to mathematical function”* or *“Error in data.frame(...): arguments imply differing number of rows: 5, 6”*. The portion after the colon tries to describe the issue (e.g., a non-numeric argument, mismatched lengths, etc.), while the part before the colon often indicates where the error occurred (which function or operation). We'll examine many specific error messages later in this chapter.

### 8.2.3 Logical Errors

**Logical errors** (or **semantic errors**) are the sneakiest type of bug. With a logical error, the code runs without crashing – no syntax or runtime errors occur – but **the output is incorrect** because the code’s logic is flawed. In other words, the program doesn’t do what you intended it to do. R won’t always tell you when you have a logical error; from R’s perspective, nothing is “wrong” enough to throw an error, but the result may not be what you expect.

Logical errors arise from human mistakes in the reasoning of the code. Examples include using the wrong formula for a calculation, updating the wrong variable, looping one time too many or too few, or using a wrong condition in an `if` statement. Because R doesn’t flag logical errors with an error message, **it’s up to you to detect them** by testing your code and verifying results.

Let’s look at a simple example. Suppose we want to count how many even numbers are in a vector:

```
numbers <- 1:10
count_even <- 0
for (i in numbers) {
  if (i %% 2 == 0) {
    count_even <- count_even + 1 # increment for even numbers
  } else {
    count_even <- count_even + 1 # BUG: mistakenly incrementing for odd numbers too
  }
}
print(count_even)
#> [1] 10
```

The code above runs without any R errors – but clearly it’s giving the wrong answer. We intended to count only even numbers, so we expected `count_even` to end up as 5 (since 1–10 has five even numbers: 2, 4, 6, 8, 10). Instead, the result is 10. This is a logical bug: the code’s logic is flawed because the `else` branch erroneously increments the counter as well. R had no way to know this was a mistake; it faithfully executed our instructions. The onus is on us to notice the unexpected output and realize our logic is wrong.

Another example: imagine you write a function to compute the average of a numeric vector, but you accidentally divide by 2 instead of the length of the vector:

```
my_values <- c(2, 4, 6, 8)
average <- sum(my_values) / 2 # BUG: should divide by length(my_values)
print(average)
#> [1] 10
```



This code runs without error and produces 10. However, the correct average of `my_values` is 5 (since there are four numbers). The logic error – using 2 instead of `length(my_values)` – caused a wrong result. Again, R doesn't know our intention, so it didn't complain; we have to catch such mistakes by reasoning about the result or writing tests.

To catch logical errors, it helps to **know what output to expect** (for example, by doing a small calculation by hand) and to include checks or printouts in your code for verification. During debugging, if a result looks suspicious (even if no error was thrown), trust your instincts and double-check the code's logic. We will discuss techniques like inserting debug print statements or using unit tests to help catch logical issues. Cultivating this habit is crucial: logical bugs can lurk unnoticed and potentially lead to faulty analyses if not caught.

In summary, when debugging, first determine which category an issue falls into:

- If R gives you an immediate *parse error* or *unexpected symbol* – it's a syntax error.
- If R prints an "Error in ... : ..." message during execution – it's a runtime error.
- If no error is reported, but the output is wrong – you're dealing with a logical error.

Each type requires a slightly different approach, but all benefit from a structured debugging process, which we'll outline next.

## 8.3 How R Reports Errors and Warnings

Before we dive into debugging strategies, it's important to understand **how R signals that something went wrong**. R typically communicates issues in two ways: **errors** and **warnings** (and also **messages**, which are informational). Knowing the difference will help you decide how to respond.

- **Errors:** These are serious problems that halt execution. When an error occurs, R stops whatever it was doing and returns to the top-level prompt (or to the calling function's context, if inside a function). Errors are reported with a message prefixed by "Error:". For example:

```
log("ten")
#> Error in log("ten") : non-numeric argument to mathematical function
```

In RStudio, error messages appear in red text in the Console. An error means **something in the code was invalid or resulted in a failure** that R could not recover from. You must fix the cause of the error before that code can run successfully. If you're sourcing an R script and an error occurs, the script will stop at that point (unless you explicitly handle the error).

- **Warnings:** These are softer alerts. A **warning** indicates that *something* unusual happened, but not severe enough to stop execution. Warnings are reported with a message prefixed by "Warning:" (or "Warning message:"). R will usually continue running the code after a warning, attempting to proceed with what it can. For example:

```
c(1, 2, 3, 4) + c(10, 20)
#> [1] 11 22 13 24
#> Warning: longer object length is not a multiple of shorter object length
```

Here R did perform the addition, producing a result, but it also issued a warning because the two vectors were of unequal length. It “recycled” the shorter vector to make the lengths match, but it warns us that something might be off (4 is not a multiple of 2). Unlike errors, warnings **do not stop the execution**. However, they are telling you that you might want to check your code or data, as it may not be doing what you think. Some warnings can be ignored if they are expected, but many indicate potential problems (e.g., deprecated function usage, numerical precision issues, etc.). You can programmatically inspect warnings after the fact with `warnings()` or even turn warnings into errors with `options(warn = 2)` if you want to be strict.

- **Messages:** In addition to errors and warnings, R functions can produce messages (using `message()` or `print()` inside functions). These are purely informational and do not indicate problems. For example, `library(dplyr)` prints a message about masked functions, and `read.csv()` may print a message if it encounters parsing issues but can still continue. You typically don’t need to debug messages; they’re there to help or inform you.

When debugging, **pay close attention to error and warning messages**. They often contain valuable clues. An error message will usually tell you *which operation failed* and *why*. Sometimes it names the function and the specific issue (e.g., “*could not find function X*” or “*object ‘y’ not found*”). Warnings may hint at data issues (like “*NAs introduced by coercion*” means something got converted to numeric and some values turned into NA).

It’s also helpful to know that when an error occurs inside deeply nested function calls, R will typically show you the highest-level call that encountered the error. For example, if you call `function_A()` which calls `function_B()`, which calls `function_C()` where the error actually happens, the error might be reported as “*Error in function\_B(...): object not found*” or something similar. In those cases, you’ll need tools (like `traceback()`, discussed soon) to see the full call stack. But at least the error message gives you a starting point.

**Example:** Let’s say you see this in your console:

```
> model <- lm(y ~ x1 + x2 + data = df)
Error in lm(y ~ x1 + x2 + data = df) : object 'y' not found
```

The error tells us `lm()` failed because `y` was not found. Perhaps the data frame `df` doesn't have a column named `y` (maybe it's capitalized differently or named something else). Or maybe you forgot to attach `df`. The error message identifies the problem: a missing object. This guides your next steps (check your variable names and data).

**Another example (with a warning):**

```
> x <- 1:5
> mean(x, trim = 0.2, na.rm = TRUE, foo = 42)
Warning in mean.default(x, trim = 0.2, na.rm = TRUE, foo = 42) :
  extra argument 'foo' will be disregarded
[1] 3
```

Here, the mean is computed (result 3), but a warning says an “extra argument ‘foo’ will be disregarded.” The function `mean()` doesn't have an argument named `foo`, so R ignored it and warned you. This is a hint that you probably made a mistake in specifying the arguments (perhaps `foo` was not intended, or you thought `mean` had such a parameter). Warnings like “**unused argument**” are common when you call a function with a typo in an argument name or use an argument that doesn't exist for that function.

In summary, when R reports an error or warning:

- **Read the message carefully** – it often describes the nature of the problem.
- Identify any object or function names mentioned – they can tell you where to look.
- Don't ignore warnings without understanding them – they might be alerting you to a bug that hasn't surfaced as a fatal error (yet).
- Use the messages as clues in your debugging process. In the next section, we'll leverage these clues as part of a structured strategy to tackle bugs.

## 8.4 Strategies for Debugging

Debugging is a bit like detective work: you gather clues (error messages, unexpected outputs), formulate hypotheses about what might be wrong, test those hypotheses, and zero in on the culprit. Rather than randomly changing things in your code and hoping it works, it's far more effective to follow a systematic approach. In this section, we'll cover several **strategies for debugging R code**:

- Reading and interpreting error messages.
- Reproducing the error reliably.
- Simplifying and isolating the problematic code (e.g., by commenting out sections).
- Using diagnostic print statements or checks.
- Utilizing R's debugging functions: `traceback()`, `browser()`, `debug()/debugonce()`, and `recover()`.

- Taking advantage of RStudio’s interactive debugging tools (breakpoints, step-through execution, environment inspection).
- Adopting a scientific mindset – form hypotheses, test, and eliminate possibilities – to track down logical bugs.

Think of debugging as an iterative, investigative process. As one famous computer science aphorism puts it: *“Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true.”* With each strategy below, you’ll be gathering information to either confirm your assumptions or discover a discrepancy.

### 8.4.1 1. Read the Error Message (Carefully!)

It may sound obvious, but the first step when you hit an error is **to actually read the error message in full**. It’s astonishing how often beginners (and even veterans, when in a hurry) will glance at an error, get intimidated or jump to conclusions, and miss the key hint the error was providing. Take a moment to parse the message. What function or operation does it mention? What does the description say?

For example, if you see `Error in plot(x, y) : object 'y' not found`, the message explicitly tells you a variable `y` wasn’t found. That likely means you either mis-typed `y` or you meant a column in a data frame but didn’t attach it or use `df$y`. If you see `Error: could not find function "ggplot"`, it tells you that the function `ggplot` isn’t available – which usually means you forgot to load the `ggplot2` package (so `library(ggplot2)` is needed). Or if the error says something like `unused argument (na.rm = TRUE)`, it indicates you passed an argument that the function didn’t expect – maybe you called the wrong function, or used a parameter name that doesn’t exist.

Sometimes error messages can be a bit cryptic, especially if they come from deep within a package. But often they contain at least a fragment of useful info. For instance, an error from a model fitting function might say `Error: NA/NaN/Inf in 'y'` – hinting that your response variable had some illegal values (like NA or Inf). Or a data manipulation error might say `replacement has length zero` or `replacement has length > 1` – clues that something is wrong with how you’re assigning values (perhaps a subsetting issue).

**If the error message isn’t clear to you**, consider using it as a search query. Copy the key part of the message and Google it (strip out specifics like variable names). Chances are, someone else has encountered the same error. As a general strategy, “whenever you see an error message, start by googling it”. Often you’ll find Stack Overflow threads or blog posts explaining the cause of that error and how to resolve it. For example, searching *“non-numeric argument to binary operator in R”* will lead you to explanations that this occurs when you try to do arithmetic with non-numeric data. Be mindful to remove any unique names or data from the error message when searching to get more general results.

Another tip: some error messages in R (especially those from tidyverse packages) are quite verbose and even offer hints. For example, `ggplot2` might say *“Error: Cannot add ggproto objects together. Did you forget to add this object to a ggplot?”* – which is essentially telling you that you likely forgot a `+` in your `ggplot` chain. Or *“Error: **mapping** must be created by `aes()` Did you use `%>%` instead of `+`?”*, which explicitly points out a common mistake (using the pipe `%>%` where you should use `+` in building a plot). Always read the full message; in such cases, the solution is literally spelled out for you.

In summary: **don’t panic** when you see an error. Slow down and read it. Underline or mentally note the key phrases. They are your first clues in the debugging process.

## 8.4.2 2. Reproduce the Error Consistently

After reading the error message, the next step is to **make sure you can reproduce the error reliably**. This might sound trivial (“I just ran the code and it errored!”), but it’s important especially if you’re dealing with code that sometimes works and sometimes doesn’t (perhaps due to random data or user input). If your error is consistent, great – you have a stable target to investigate. If it’s intermittent, you’ll want to control the situation so it becomes consistent.

To reproduce an error, try the following:

- **Run the code in a clean R session** if possible. Sometimes leftover variables or settings in your environment can mask or influence bugs. By restarting R (e.g., in RStudio, Session -> Restart R) and running the code (or the specific part that fails) from scratch, you ensure that the error is truly reproducible and not dependent on some hidden state.
- **Use a fixed random seed** if your code involves randomness. For example, if the error occurs only for certain random samples, use `set.seed()` to make the random behavior predictable while you debug.
- **Simplify inputs** if possible. If the error depends on your data, see if you can trigger it with a smaller or simplified dataset. For instance, if `my_function(big_dataframe)` errors, try to isolate a subset of `big_dataframe` that still causes the error. This often goes hand-in-hand with the next strategy (simplifying the problem), but the idea is to remove unnecessary complexity so that you can focus on the core issue.

Reproducing the error is important because you’ll likely be running the faulty code multiple times as you poke and prod to find the bug. You want that iteration to be quick and easy. As one guideline suggests: “To find the root cause of an error, you’re going to need to execute the code many times... make the problem both easy and fast to reproduce”. If your original script takes 10 minutes to run before hitting the error, you should create a shorter pathway to trigger the error (perhaps by isolating the function or using a smaller dataset) so you can test fixes rapidly.

In practice, this often means creating a **minimal reproducible example** – a pared down piece of code that still produces the error. For instance, if you have an error in a big data cleaning script, copy just the relevant slice of code that triggers it into a new script or the console, with just a small snippet of data. Not only does this help you, but if you need to seek help from others, they will appreciate a minimal example. (The R community even has the `reprex` package to assist in creating reproducible examples for sharing.)

### 8.4.3 3. Simplify and Isolate the Problem

A common mistake when debugging is trying to solve the entire problem at once. It's often more effective to **simplify the code and isolate the section that's causing trouble**. This overlaps with creating a reproducible example and also involves systematically ruling out parts of your code that *are not* contributing to the bug.

Here are some tactics:

- **Comment out code:** Temporarily disable portions of your code to see if the error still occurs when those parts aren't executed. For example, if you have a script with 100 lines and you suspect the issue is coming from somewhere in the middle, you can put `#` in front of blocks of code (or use RStudio's shortcut to comment a selected region) to skip them. Run the script and see if the error disappears. If it does, then the bug likely lies in the part you commented out. If the error persists, it means the bug is in the code that still ran. Using binary search on your code – comment out half and see if the error occurs, then narrow down – can quickly pinpoint the problematic region.
- **Modularize:** If the code isn't already broken into functions, consider isolating the problematic code in a function or separate script where you can call it independently. Sometimes writing a quick wrapper around the suspect code allows you to call it repeatedly with different parameters for testing.
- **Print intermediate results or use checkpoints:** Insert `print()` or `cat()` statements to display the state of key variables right before the point of failure. For instance, if a loop is crashing on the 37th iteration, you might print the loop index or some properties of the data at each step to see what's special about the 37th iteration. These “**print debugging**” statements can act as breadcrumbs leading up to the error. (Just remember to remove or comment them out when you're done, or use something like the `message()` function which can be silenced more easily.)
- **Simplify data:** As mentioned, try reducing the size or complexity of input data. If a function fails on a huge data frame, see if a smaller subset (maybe even a single row or a simplified version of a row) still triggers it. If a model fails on 100 predictors, see if it fails on 2 predictors. By simplifying, you remove potential confounding factors and make the bug more apparent.

As you simplify, you might stumble upon the fix inadvertently – for instance, by removing part of the code, you realize that a certain step wasn't needed or was causing the error. Or

by printing intermediate values, you might see NA or an unexpected zero that leads to the crash.

This step is essentially about **narrowing your focus**. Complex programs can have multiple things going wrong, but you want to tackle one issue at a time. Get the simplest version of the task working, then gradually add back the complexity until something breaks – and then you know exactly what caused it.

#### 8.4.4 4. Use `traceback()` to Locate the Error

When a runtime error occurs in R, especially one that involves nested function calls, your best friend is the `traceback()` function. After an error happens (and you see the error message in the console), if you immediately call `traceback()`, R will print out the call stack – essentially, the sequence of function calls that led to the error. This helps answer the question: “*Where did things go wrong?*”.

For example, consider the following scenario with two functions:

```
f1 <- function(a) {  
  a + 5          # might error if 'a' is not numeric  
}  
f2 <- function(b) {  
  f1(b)          # call f1 inside f2  
}  
f2("text")       # passing a string instead of a number  
#> Error in a + 5 : non-numeric argument to binary operator
```

We get an error “non-numeric argument to binary operator”. It’s not immediately obvious whether the error happened inside `f1` or `f2` just from that line – though savvy readers might guess that `'a + 5'` inside `f1` caused it, since `b` was “text”. To be sure, we can run:

```
traceback()  
#> 2: f1(b) at #3  
#> 1: f2("text")
```

The `traceback()` output (shown above) is read from bottom (1) to top (2). It tells us:

- Call #1 was `f2("text")` – the top-level call we made.
- Call #2 was `f1(b)` at line #3 of some file/command (here we see it’s at the call inside `f2`).

The order indicates that `f2` called `f1`, and the error occurred in `f1` (since it's higher up the stack). In this simple case it's obvious, but in more complex code, `traceback()` is invaluable for pinpointing *where* the error occurred. It might show you a chain of calls like: 1: `do.call("someFunction", ...)`, 2: `someFunction(...)`, 3: `anotherFunction(...)`, 4: `[internal]`. The highest number will often be the innermost call where the error happened.

A few tips for using `traceback()`:

- Call it *immediately* after the error. If you run any other code (even a simple calculation) in between, the traceback from the last error might be lost or overwritten.
- You don't need to call `traceback()` if you are already in an interactive debugging session or using RStudio's error inspector (which shows the stack), but if you're just running a script normally, `traceback()` is a quick manual way to see the stack.
- The output shows function names and maybe file or line references (for example, `something.R#15` means line 15 of `something.R`). Use this info to jump to the relevant part of your code.
- If the traceback is long, focus on the highest-listed user-defined function that you recognize from your code. That's likely where you should start digging. For instance, if the top of the stack (highest number) says `stop("Value must be > 0")` and just below that you see a call to `check_values()` (a function you wrote), you know the error was triggered by a `stop()` inside `check_values`. Time to examine that function's logic.

In summary, `traceback()` helps answer the “Where is it breaking?” question. It doesn't fix anything, but it guides you to the right place to look in the code. It's especially useful when working with nested functions, package functions, or long scripts with many pieces. Always remember: when an error occurs and you're unsure of its origin, call `traceback()` and follow the breadcrumbs.

## 8.4.5 5. Pause Execution with `browser()` for Deeper Inspection

Sometimes, just seeing the error message and the call stack isn't enough – you need to **dig into the state of the program at the moment of error**. This is where R's interactive debugging mode comes into play, and the primary tool to invoke it is the `browser()` function.

Placing a call to `browser()` at a certain point in your code **pauses execution at that point and gives you an interactive R prompt** where you can inspect variables, run commands, and step through the code line by line. It's like hitting a breakpoint in a traditional IDE debugger. When in this mode, your usual R console prompt changes to something like `Browse[1]>` to indicate you're in a debugging environment (often called the “browser”).

How to use `browser()`:

- Insert `browser()` on a line inside the function or section of code you want to debug. For example:



```
buggy_function <- function(data) {
  browser()          # start debug mode when this line is reached
  # ... rest of function ...
  result <- mean(data$value)  # example operation
  return(result)
}
```

- Now call `buggy_function(your_data)`. When R reaches the `browser()` line, it will pause and give you the `Browse[1]>` prompt.
- While in the browser, you can type the name of any variable to print its value, call functions, or even change variables. You have access to the function's local environment at the point where you paused.
- You can then **step through** the code. By pressing enter at the browser prompt, R will execute the next statement and then pause again. Alternatively, you can use single-letter commands:
  - **n** (Next): execute the next statement. This is similar to pressing enter; it steps over function calls (doesn't dive into them).
  - **s** (Step into): if the next statement is a function call, this will step *into* that function, allowing you to debug inside it line by line.
  - **c** (Continue): resume regular execution until the next breakpoint or the end of the function. This effectively exits the browser (unless another `browser()` call or breakpoint is hit later).
  - **f** (Finish): finish execution of the current loop or function without further pausing.
  - **where**: print the call stack (so you know what function you're in and how you got there).
  - **Q** (Stop): terminate the function and exit the debug mode, returning to the top-level prompt. (In some older references, an uppercase **C** was used to continue to the end and exit, but in modern R, **c** continues and **Q** quits debugging.)

While in browser mode, you can inspect any variables in the current scope. For example, if your function had a variable `data` and you want to see its structure, you can type `str(data)` at the debug prompt. If you want to see the first few values, type `head(data)`. You basically have a live R session at that point in the code.

This is incredibly powerful: you can find out exactly what's going on right before (or at) the point of failure. Maybe a variable has an unexpected `NA`, or a vector is shorter than you thought, or a condition is `FALSE` when you expected `TRUE`. By stepping through, you can watch the program's logic unfold and see where it diverges from your expectations.

### Example usage of `browser()`:

Suppose you have a function that is misbehaving:

```

calculate_stats <- function(df) {
  summary <- data.frame()
  for (col in c("length", "width", "height")) {
    # We expect df to have these columns
    browser() # pause here to inspect state
    avg <- mean(df[[col]])
    summary[col, "mean"] <- avg
  }
  return(summary)
}

```

If you call `calculate_stats(mydata)` and something is off (say it errors on `mean()` because `df[[col]]` was `NULL` for some `col`), the `browser()` will let you see exactly what `col` is at that moment, what `df` contains, etc. You might discover that `df` doesn't have a `"height"` column due to a typo (maybe it's `"Height"` with capital H in the data). That insight will directly lead to the fix (correct the column name or handle it conditionally).

One thing to note: you don't want to leave stray `browser()` calls in your code once you've fixed the bug – they will pause execution every time. Remove or comment them out after debugging.

In RStudio, when you hit a `browser()`, the interface will also show you the current environment in the Environment pane, and you'll see options to step or continue in the UI. You can use those if you prefer, but the end result is the same as using the console commands.

In summary, **use `browser()` when you need to stop time inside your code and play with the pieces**. It's how you can perform an “autopsy” on a running function to see what's going wrong. This is especially useful for logic bugs or complex computations where you're not sure which part is doing the wrong thing.

*(If you're not using RStudio or an IDE with debugging support, `browser()` is your primary way to debug interactively. If you are using RStudio, you can also achieve the same effect by setting breakpoints, as discussed later, which essentially inserts a `browser()` behind the scenes.)*

#### 8.4.6 6. Use `debug()`, `debugonce()`, and `recover()` for Flexible Debugging

While `browser()` is a manual way to put breakpoints in your code, R also provides some built-in functions to help you enter debug mode without editing your code to insert `browser()` calls. The most commonly used are `debug()`, `debugonce()`, and the global `recover()` option. These tools can make debugging a bit more convenient in certain situations.

- **debug(func)**: This function flags **func** (an R function) for debugging. After you call **debug(myFunction)**, the very next time **myFunction()** is called (by you or by some other code), R will automatically pause execution at the **beginning** of **myFunction**, as if a **browser()** were set on the first line. You can then step through **myFunction** line by line. This is handy if you suspect a bug in a function and want to inspect it without editing its source to add **browser()**. Once you've finished, you can call **undebug(myFunction)** to remove the debug flag, otherwise it will enter debug mode every time you call that function in the future.

For example:

```
debug(lm)           # suppose we want to debug the lm() function
lm(y ~ x1 + x2, data = df) # this will enter debug mode at the start of lm()
# ... step through ...
undebug(lm)
```

This is more useful for debugging your own functions or understanding how a library function works. Be cautious about debugging very basic functions (like **mean** or **c**), as you might end up in R's C internals or such – but for R-level functions, it's fine.

- **debugonce(func)**: Similar to **debug()**, but it only triggers the debug mode **on the next call** to the function, and then automatically undebugs. This is convenient to avoid forgetting to **undebug()**. For instance:

```
debugonce(calculate_stats)
calculate_stats(mydata) # will debug this call
# ... debug session ...
calculate_stats(mydata) # next call runs normally (debug flag is gone)
```

Use **debugonce()** when you want to poke into a function just one time.

- **recover()**: This is used to debug *after* an error has occurred, by examining the call stack. You activate it by setting **options(error = recover)**. Once this option is set, if an error occurs **anywhere** in your R session, instead of terminating and returning to the console, R will pause and present you with a menu of active function calls (the stack frames) at the moment of error. It looks something like this:

```
options(error = recover)
# Run code that produces an error...
#> Error in someFunction(x) : something went wrong
#>
#> Enter a frame number, or 0 to exit
#> 1: globalCallingFunction() # frame 1
#> 2: someFunction(x)         # frame 2
#> 3: anotherFunction(y)      # frame 3 (perhaps where error occurred)
```

```
#>
#> Selection:
```

R will list the call stack (like `traceback()` would) but now it gives you the chance to choose a frame to inspect. If you enter 3 in the example above, you'll enter the browser in the context of `anotherFunction(y)` – effectively, you jump into that function right where it errored (or right after the error). You can then inspect variables there to see why it failed. After you finish, you can type `Q` to exit the recover mode (or `0` at the selection prompt to exit without entering any frame).

`recover()` is extremely useful when you want to *post-mortem* debug an error that you didn't anticipate. Instead of rerunning everything with `browser()` in place, you can simply use `recover()` to catch it. Remember to turn it off when you're done by `options(error = NULL)` (otherwise you'll be prompted on every error, which can be annoying if you're not actively debugging).

Using `debug()`, `debugonce()`, or `recover()` can save you time and keep your code uncluttered by manual `browser()` calls. For example, if you are debugging an R package's function, you can't easily insert `browser()` in it (without modifying the package code), but you can just do `debug(packageFunction)` to step into it when it runs. Or if a script is failing deep in some call stack, `recover()` lets you inspect the state at the failure point without modifying the script.

In RStudio's Debug menu, there are options equivalent to these, like “On Error -> Break in Code” which basically sets `options(error = browser)` or `recover`. In fact, setting *Break in Code* in RStudio's error handling will automatically put you in the browser at the error line (often in the context of where the error occurred, similar to `recover`). That's a friendly way to catch errors without preemptively inserting debug statements.

To summarize:

- Use `debug()/undebug()` or `debugonce()` to *proactively* step through a function from the start.
- Use `options(error = recover)` (or the RStudio equivalent) to *reactively* debug at the moment of an unexpected error, examining any function on the call stack.
- These tools, combined with `browser()`, give you a lot of flexibility in *when and where* to enter the debug mode.

#### 8.4.7 7. Leverage RStudio's Debugging Tools (Breakpoints & Step-Through)

If you are using RStudio (or another IDE with debugging support), you have a convenient graphical interface to many of the techniques we've discussed. RStudio's debugger essentially builds on R's `browser()` functionality but makes it easier to set breakpoints, inspect variables, and navigate through code. Here's how you can use RStudio's tools to debug more efficiently:

- **Setting Breakpoints:** In the RStudio source editor, you can click to the left of a line number to set a breakpoint at that line (a little red circle will appear). A breakpoint is like a `browser()` inserted at that line, except you didn't have to modify your code. When you run the code (for example, source the file or call the function), R will pause execution when it reaches that line (assuming that line is actually executed). Breakpoints in RStudio are very handy for larger scripts – you can set a breakpoint at a suspect section, then run the whole script; it will run until that point and then pause, letting you inspect the state. Breakpoints behave just like `browser()` in terms of entering debug mode. In the environment panel, you'll see you are in the *Browse* context, and you can use the console to check variables, or use the dedicated buttons (described next). Remember that breakpoints only take effect when the code is run *via RStudio* (sourcing or using RStudio's Run commands). If you run code line-by-line manually, it won't stop at breakpoints unless you explicitly source it. Also note that if you set a breakpoint in a function that hasn't been loaded yet (say you wrote a new function but haven't sourced the file), RStudio will mark it as a “*deferred breakpoint*” (usually a hollow red circle). Once you source the file, the breakpoint becomes active (solid red).
- **Stepping Through Code:** Once in debug mode (via a breakpoint or `browser()`), RStudio's debug toolbar becomes active (usually at the top of the source pane or as a small panel). You'll see buttons like *Continue*, *Step Over*, *Step Into*, *Step Out/Finish*, *Stop* etc.. These correspond to the commands we discussed:
  - *Continue* (the `c` command) resumes execution until the next breakpoint (or end of program).
  - *Step Over* (the `n` command, often a down arrow icon) executes the next line. If that line is a function call, it will **not** go inside the function – it will just execute it and pause afterward.
  - *Step Into* (the `s` command, often an arrow entering a box icon) will dive into a function call on the current line, allowing you to debug inside that function.
  - *Step Out* or *Finish* (the `f` command, often an arrow leaving a box icon) will run the rest of the current function and pause when it returns to the caller.
  - *Stop* (the `Q` command, often a stop sign icon) stops debugging altogether (terminating the function and returning to the console).

These let you navigate through your code's execution flow in a controlled way, which is immensely useful for following logic and catching where things go awry.

- **Inspecting Variables and Environments:** When in debug mode, the RStudio Environment pane switches to show the **current call stack** and the variables in the current environment. At the top of the Environment pane, you'll see a drop-down that might say something like `<environment: calculate_stats>` or `<environment: global>` along with parent environments. This indicates the environment of the function you're currently debugging, and you can use that drop-down to navigate to parent frames (if you stepped into nested calls). The variables shown in the pane are those available in the current

scope – their names and values (for simple types) or previews (for data frames, etc.). This visual inspection can be quicker than typing `ls()` or `print` commands at the console, though you can still use the console as well. If a variable is a large data structure, you can click it in the Environment pane to view it (like View data frame) or use `str()` on it in the console. The Environment pane will also show special values like function arguments (even if not yet evaluated – they might appear in gray, indicating promises). Being able to *see* the object names and values helps you spot issues (e.g., you might notice a variable is `NULL` when it shouldn't be, or a vector has length 0, etc., at a glance).

- **Traceback and Call Stack:** In RStudio's debug mode, the interface often shows a "Traceback" panel (or you might need to toggle it) that lists the call stack, similar to what `traceback()` would give. You can click on different frames to inspect them. This is essentially an interactive way to use `recover()`: you can jump between frames and see variables in those frames. For example, if your code crashed deep inside, and you have the frames listed, you can click on a higher frame to see what the function inputs were, etc., even if you didn't manually set `recover()`. (However, note that by default RStudio's error inspector might need to be set to break on error to do this automatically; otherwise you might manually call `recover` or set that option.)
- **Editing on the Fly:** A nifty feature: while paused in debug, you can actually fix a line of code in the editor and continue. But note, simply editing the text in the editor doesn't change the code that's already loaded in memory. However, you can copy a corrected line and paste it into the console to execute it, or you can use the **debugger** environment to assign new values. Alternatively, if you realize the fix, you might stop debugging, edit the code, and then re-run.
- **Conditional Breakpoints:** As of this writing, RStudio does not support conditional breakpoints (break only if a condition is true) in the GUI. A workaround is to put an `if` in your code that calls `browser()` when a condition is met, or use `trace()` for advanced cases. But for most beginner needs, regular breakpoints are enough.

Using RStudio's debugger doesn't necessarily let you do something you couldn't do with command-line tools, but it makes the process more user-friendly and visual. Especially for beginners, being able to click to set breakpoints and using buttons to step can feel more intuitive than remembering `n`, `s`, `c`, `Q` commands. It's worth getting comfortable with both styles – sometimes on a remote server you might only have the command-line, but when using RStudio, take advantage of what it offers.

**Example:** Suppose you wrote a function to process a data frame and you want to debug an issue in it:

```
process_data <- function(df) {  
  result <- df # start with input  
  # Suppose something is going wrong in this block:
```

```
result$ratio <- result$val1 / result$val2
result$flag <- result$ratio > 1
return(result)
}
```

In RStudio, you could click to set a breakpoint on the line `result$ratio <- ...`. Then call `process_data(mydata)`. Execution will pause at that line, with `df` and `result` accessible in the environment. You check and see that perhaps `result$val2` has a zero (which would cause `Inf` or an error in division). You now know the cause (division by zero) and can plan a fix (maybe filter those out or handle differently). Without stepping through, you might not have noticed that in the data. The breakpoint made it easy to catch at the moment of the operation.

To conclude: **RStudio’s debugging tools integrate the strategies we discussed (pausing, inspecting, stepping) into a cohesive UI.** As a beginner, investing time to learn these tools will pay off. You’ll be able to *see* what your code is doing and find bugs faster. Just remember, underneath the hood it’s using the same R mechanisms (it’s not “magic”), so everything you learned about `browser()`, `traceback()`, etc., still applies.

### 8.4.8 8. Adopt a Systematic Debugging Workflow

Now that we’ve covered individual techniques, let’s zoom out and talk about an overall **work-flow for debugging**. When you face a bug, especially in a larger project, it helps to approach it methodically rather than with ad-hoc trial and error. Here’s a step-by-step debugging workflow that incorporates many of the strategies above (and mirrors how experienced programmers tackle bugs):

1. **Stay Calm and Gather Info:** When the bug first appears, resist the urge to start randomly changing code. Take note of the symptoms. What exactly is the error message or unexpected output? Write it down if needed. Recall what the code is supposed to do and identify how the actual result deviates from the expectation. Sometimes explaining the problem to someone (or to a rubber duck on your desk!) can clarify your thinking – this is the classic *rubber duck debugging* method, where describing the code line by line often reveals the issue.
2. **Read the Error and Identify the Suspect Area:** If there’s an error message, read it carefully (as discussed). Determine where in the code it likely occurred. Use `traceback()` if needed to pinpoint the location. If it’s a logical error (no error message), use clues from the output to guess where the code might be going wrong (e.g., if a summary statistic is off, maybe the calculation part is suspect). At this stage, you may not know the exact cause, but you should have an idea of *where* to look.

3. **Make it Reproducible:** Ensure you can consistently trigger the bug. If it requires certain input or conditions, set those up. Simplify the scenario if possible (e.g., test the function on a smaller dataset that still produces the bug). This often involves writing a small script or using the console to call the problematic function with specific parameters. Reproducibility is crucial – you want to be able to test potential fixes and see if they resolve the issue.
4. **Isolate the Code:** Narrow down the section of code that’s causing the issue. Comment out unrelated parts to see if the issue still occurs. If debugging a large script, try running pieces of it in isolation. If a particular function is misbehaving, focus on that function alone with test inputs. The goal is to eliminate extraneous factors and reduce the “surface area” of the bug. Many bugs become obvious once you isolate the offending code.
5. **Use Debugging Tools:** Deploy the appropriate tools to inspect and step through the code.
  - If it’s a straightforward error and the cause is evident from a printout or two, you might just use some `print()` statements to confirm your hypothesis (e.g., printing an index that seems to go out of range, or printing the class of a variable to see if it’s what you expect).
  - If the cause is not obvious, use `browser()` or breakpoints to pause execution right before or at the error, and inspect variables. Check for things like: Are variables what you think they are? Are dimensions/lengths as expected? Are there NA or unexpected values present? Is the code flow (branching) going where it should, or perhaps an `if` condition is skipping something?
  - If the code involves loops or iterative processes, step through a few iterations. Often the bug might occur at a certain iteration (e.g., the first time an NA is encountered or the index hits the end of a vector).
  - Make liberal use of the **scientific method**: form a hypothesis (“I suspect this vector is length 0 which causes the error”), then test it (“Check length of vector, indeed it’s 0 – why?”), then refine understanding (“It’s 0 because the filtering earlier removed all rows – so the error is downstream of an earlier logic flaw.”). Approach debugging as an experiment where you gather evidence.
6. **Find the Root Cause:** Keep digging until you find the *root* of the problem, not just a symptom. It’s possible to apply a band-aid fix that stops an error (like checking for zero-length vector to avoid an error) but that might not address why the vector was zero-length in the first place (maybe a wrong filtering condition). Use the info you gathered to trace the problem back. For example, an object wasn’t found – was it never created, or was it misspelled? Why? A calculation is wrong – is the formula wrong, or did bad data enter it? The deeper you understand the cause, the more robust your fix will be. Sometimes you may realize the bug is not where you first thought – e.g., an error surfaces in function C, but the real mistake happened in function B which passed bad data to C. In that case, the fix belongs in B, not just handling it in C.



7. **Fix the Issue:** Once you’ve identified the cause, implement a fix in your code. This might mean correcting a formula, adding a missing function call (e.g., loading a package so the function exists), changing a loop index, initializing a variable, handling a special case (like checking for division by zero), etc. Make the change and then rerun your code in the test scenario that reliably produced the error.
8. **Test the Fix:** After fixing, test again with the same inputs to confirm that the error is gone or the output is now correct. Then, importantly, test with a variety of inputs or scenarios to ensure you didn’t break anything else and that your fix holds in general. If you have automated tests (in a package or project), run them. If not, at least try a few different cases, including edge cases. For example, if the bug was triggered by an empty input, try an empty input now – does it handle it gracefully? If the bug was wrong calculations for negative values, test some negative values.
9. **Reflect and Strengthen:** Once the bug is resolved, take a moment to consider if you can improve your code or workflow to prevent similar bugs. Maybe add an assertion in the code (using `stopifnot()` or explicit checks) to give a clearer error if a similar situation occurs. Perhaps improve naming or comments to avoid confusion. If the bug was due to a logical oversight, consider writing a small unit test (if applicable) to catch if that logic ever goes wrong again. Each debugging experience is a chance to learn and make your code more robust.
10. **Remove Debug Code:** Clean up any leftover debug code (remove `browser()`, extra print statements, etc. that you added). However, sometimes leaving a warning or message for unusual conditions can be useful. For instance, if you discovered a certain data condition that caused a problem, you might leave a `warning()` in the code to alert if that condition arises (assuming it’s not supposed to in normal use). In general, though, return your code to a clean state.

This workflow might seem involved, but with practice it becomes second nature and can happen very fast for simple bugs. For more complex bugs, following a structured approach will save time in the long run. It prevents you from going in circles or making random changes that can introduce new bugs.

One more point: **Don’t hesitate to seek help when needed**, but do so smartly. If you’ve spent a reasonable amount of time and are stuck, explaining the problem to a peer or on a forum (like Stack Overflow) can be invaluable. When you do, provide that minimal reproducible example we talked about – often, the act of preparing that example leads you to the solution yourself! And if not, others can help pinpoint the issue. There’s a saying that “**Rubber duck debugging**” (explaining the code out loud, even to an inanimate object) often solves the bug without anyone else intervening.

In summary, debugging is **an iterative, logical process**. By staying systematic – read, reproduce, isolate, inspect, fix, test – you can tackle bugs in a calm and efficient manner. Over time, you’ll also start writing code with an eye towards debuggability: clearer structure,

checks for assumptions, and smaller functions that are easier to test in isolation. This proactive approach reduces the incidence of bugs and makes those that do occur easier to find.

## 8.5 Common R Errors and What They Mean

In this section, we'll look at some **common errors and warnings in R**, particularly those that beginners frequently encounter, and explain what they mean and how to address them. Seeing a cryptic error for the first time can be bewildering, but often these messages are less mysterious once you understand the typical causes.

Below is a list of common errors/warnings, each in **bold** with an explanation and solution following it:

- **Error: object not found** – This means R tried to evaluate a symbol (variable name) that doesn't exist in the current environment. For example, **Error in eval(expr, envir, enclos): object 'my\_var' not found**. Common causes are:
  - You misspelled the variable or function name (e.g., `my_variabl1` instead of `my_variable`).
  - The object is defined in a different scope or was never created. For instance, inside a function you refer to a global variable that isn't passed in, or you forgot to run a chunk of code that defines it.
  - Case sensitivity: `Data` is not the same as `data`.

**Solution:** Check the spelling and existence of the object. If it's a function from a package, ensure the package is loaded (a *"could not find function"* error is similar – see below). If it's data, make sure you have the object in memory or the correct dataframe is being used (qualify with `df$var` if needed). In functions, ensure you passed all needed data as arguments. In an R Markdown context, remember that each chunk might need the code from previous chunks (or use `knitr::opts_chunk$set(echo = TRUE)` with caution that all needed objects are created). This error is usually resolved by either fixing a typo or adding the necessary code to create or load the object before it's used.

- **Error: could not find function "foo"** – This indicates that a function `foo` isn't available in the namespace. The most likely reason is that you forgot to load the package that provides that function. For example, `ggplot()` not found means you didn't call `library(ggplot2)`. Another possibility is a typo in the function name or using a function that doesn't exist. If you see `<anonymous>` in the error, it might be a function in your own code that wasn't defined in time.

**Solution:** Identify which package (if any) the function comes from and load it via `library(packageName)`. If you're unsure, use `??foo` or a quick web search to find the function's source. If it's a base R function, check spelling and case (R's base functions should always be available; a not found for them implies a typo or that you overrode the

name somehow). As a good practice, load all your packages at the start of your script or R Markdown to avoid this. In R Markdown specifically, each chunk knows about library calls in previous chunks as long as they were executed, so ensure your chunks are run in order.

- **Error: unexpected ‘...’ in “...”** (or **unexpected symbol/string constant/numeric constant**) – This is a syntax error indicating that R’s parser found something where it didn’t expect it. Common scenarios:
  - Missing a comma, parenthesis, or operator. For example, `mean(x 1:5)` would give “unexpected numeric constant” after `x` because you likely meant `mean(x, 1:5)` or something similar.
  - Unclosed quotes leading to “unexpected end of input” or “unclosed string”.
  - An extra or misplaced curly brace or parenthesis can also cause an unexpected symbol error.

**Solution:** Check the line (and a few lines above, since sometimes the error is reported at the next line if a string wasn’t closed) for proper syntax. Add missing commas or quotes. RStudio’s syntax highlighting often helps here – if you see strings not colored properly or parenthesis highlighting not matching, that’s a clue. Also, `parse()` error messages often show the code snippet and a `^` pointer – use that to locate the issue. This error is resolved by correcting the code structure (it doesn’t indicate a logical bug, just a typo/format issue).

- **Error: arguments imply differing number of rows: X, Y** – This error comes typically from trying to create a data frame (or something similar) with vectors of unequal lengths. For example:

```
a <- 1:5
b <- 1:3
data.frame(a, b)
#> Error in data.frame(a, b) : arguments imply differing number of rows: 5, 3
```

R expected each column to have the same number of rows, but here one has 5 and the other 3. It also occurs in `cbind()` or similar functions if lengths differ.

Another scenario: combining data frames or series of vectors with mismatched lengths triggers this. Essentially, it’s telling you that you’re trying to align things of different sizes that don’t naturally recycle (data frame creation does not recycle shorter vectors, it errors out).

**Solution:** Make sure all columns have the same length. If you intended recycling, explicitly recycle or fill shorter vectors (e.g., repeat or pad with NAs to length 5 for `b`). Often, this error is a sign of a bug in data preparation – perhaps one vector should have been length 5 but lost some elements due to filtering. Investigate upstream why lengths

differ. In our simple example, you'd fix it by correcting the data or the logic that led to different lengths. If the shorter vector should be extended, you can do:

```
length(b) <- length(a) # this will pad b with NAs to length of a
df <- data.frame(a, b)
```

This yields no error (but introduces NAs for the padded values). The best fix depends on context – ensure your data alignment is correct.

- **Warning: longer object length is not a multiple of shorter object length** – This warning appears when R's recycling rule is in effect but the longer vector isn't an integer multiple of the shorter one. For example:

```
c(1,2,3,4) + c(10,20)
#> [1] 11 22 13 24
#> Warning: longer object length is not a multiple of shorter object length
```

Here, length 4 vs length 2. R recycled the shorter vector (10,20) to (10,20,10,20) to match length 4, which it can do, but because 4 is exactly  $2 * 2$ , actually this case wouldn't warn (4 is a multiple of 2). If it were 5 and 2, 5 is not a multiple of 2, so partial recycling would occur and R warns:

```
1:5 + 1:2
#> [1] 2 4 4 6 6
#> Warning: longer object length is not a multiple of shorter object length
```

It used 1,2,1,2,1 to add to 1:5.

**Solution:** The warning itself may not always require a fix if you intended recycling (but usually, if lengths don't align, it's unintended). Check your data lengths in the operation. Likely, something is off – maybe you were combining two vectors that should have been equal length. If you really want to recycle a shorter vector, you can silence the warning by making the lengths align (e.g., repeat the shorter vector fully). But generally, inspect why the lengths differ and correct the logic. This warning is helpful because it often points out a mistake: for instance, adding a vector of length 3 to a vector of length 8 likely isn't what you consciously planned.

- **Error: missing value where TRUE/FALSE needed** – This is a common error when dealing with conditional statements or any context expecting a boolean (TRUE/FALSE) value. It means that the condition evaluated to NA. For example:

```
x <- NA
if (x) {
  # ...
}
#> Error in if (x) { : missing value where TRUE/FALSE needed
```

Here, `if` expects a clear `TRUE` or `FALSE`, but `x` is `NA`, so R doesn't know what to do. Another typical cause is using `==` or `!=` to compare with `NA`. For instance:

```
y <- 5
if (y == NA) { ... }
# This will throw the same error, because y == NA yields NA (since NA is not comparable
```

Or applying a logical operation element-wise that results in `NA` and then using it in an `if` or `while`.

**Solution:** When dealing with `NA`, you should use `is.na()` to test for missingness. For the example above:

```
if (is.na(x)) {
  # handle NA case
} else if (x) {
  # handle TRUE case
} else {
  # handle FALSE case
}
```

If you intended to allow `NA` as false, you could replace `NA` with `FALSE` (but be careful logically). The main point is to avoid passing `NA` directly into `if/while`. This error is fixed by explicitly checking for and handling `NA` values before the logical context. In vectorized code, if you see this error, likely you attempted something like `ifelse` on a vector with `NA` condition without proper handling, or used an `if` instead of `ifelse` for vector logic.

As an added note, if you see the error pointing to something like `==` as in “*Error in if (x == NA) ...*”, it's a sign that you should replace `x == NA` with `is.na(x)`.

- **Error: non-numeric argument to binary operator** – This occurs when you attempt an arithmetic or binary operation on a non-numeric (or otherwise incompatible) type. We saw an example: `5 + "10"` triggers this, because `+` expects numbers on both sides. Another example: if `df$price` is a factor (perhaps read from a CSV without `stringsAsFactors=FALSE` in older R) and you try to do `df$price * 2`, you'll get this error since a factor isn't numeric (even though it prints like one). Similarly, trying to subtract dates from strings, or any such mismatch.

**Solution:** Check the types of the operands. Use `str()` or `class()` to inspect them. If something is a factor or character that should be numeric, convert it (e.g., `as.numeric(as.character(factor_var))` or better, read/import it correctly). If you accidentally treated a string as a number, correct the logic (maybe you meant to parse it). In the context of data frames, pay attention to how data is read (`stringsAsFactors` or not). This error basically says “I tried to do math on something that isn't math-able.” So turn that something into a number, or remove the operation if it doesn't make sense.

If one argument is NULL, you might see a variant of this too, so ensure both sides are defined and numeric.

- **Error: (converted from warning) ...** – If you see an error that mentions it was converted from a warning, like *“Error: (converted from warning) XYZ”*, that means at some point `options(warn = 2)` was set (perhaps by you or the environment), which turns all warnings into errors. For example, `log(-1)` normally gives a warning about NaNs, but with `warn=2` it would error out. If you encounter this unexpectedly, it could be your environment or a package doing it. The message after “converted from warning” is the original warning text. To fix the underlying issue, treat it as a warning (like the ones above). If you just want to disable this strict mode, set `options(warn = 1)` (or 0) to revert to normal warning behavior.
- **Error: cannot open the connection** (and a warning about No such file or directory) – This usually happens when trying to read a file that doesn’t exist or isn’t found at the path given:

```
read.csv("data/myfile.csv")
#> Error in file(file, "rt") : cannot open the connection
#> In addition: Warning message:
#> In file(file, "rt") : cannot open file 'data/myfile.csv': No such file or directory
```

The warning is very explicit: it couldn’t find the file at ‘data/myfile.csv’. After the operation fails, R reports it as an error in opening the connection.

**Solution:** Check your working directory (`getwd()`) and the file path. Ensure the file exists at that location or provide the correct path. A common beginner gotcha is not knowing what the working directory is – in RStudio, it might default to the project folder or not, depending. Use `list.files()` to see what files are visible. If you need to build a path, consider using `file.path()` and ensure you have the right relative or absolute path. Once the path is corrected, this error will go away. If it’s about an URL or connection that can’t open, ensure internet connectivity or correct the URL/protocol.

- **Error: object of type ‘closure’ is not subsettable** – This error trips up many newcomers. It happens when you try to treat a function like a list or vector, usually by using `[...]` on a function name. For instance:

```
mean[1]
#> Error: object of type 'closure' is not subsettable
```

Here `mean` is a function (a closure in R internals) and you attempted to subset it as if it were a vector or list. The most common scenario is accidentally using the name of a function for a variable. For example:

```
data <- data.frame(x = 1:5, y = 6:10)
filter <- 2 # Oops, using the name 'filter' which is also a function (dplyr::filter)
```

```
data$filter]
#> Error: object of type 'closure' is not subsettable
```

Because `filter` was also a function (from `dplyr`, masked by our assignment perhaps), or if `dplyr` wasn't loaded, then `filter` default might be something else. But the point is R sees `filter` as a function (closure) and `data$filter` tries to subset with it, causing this error.

Another common cause is forgetting parentheses on a function call. For example:

```
mean <- mean(c(1,2,3))
```

This actually assigns the result of `mean(1,2,3)` to `mean` – which is bad because you override the function name. But another scenario:

```
myfunc <- function() { 42 }
myfunc[] # trying to subset the function instead of calling it
#> Error: object of type 'closure' is not subsettable
```

This is just a mistake; you probably meant to call the function or had a variable with same name.

**Solution:** Check if you accidentally used a function name as a variable. Running `mean` by itself after the error might show something odd if you overwrote it. If you did override a base function name, remove that variable or restart R. Ensure you add `()` to call functions instead of trying to index them. If you have a variable that shares a name with a function (like `filter` or `length`), rename the variable to avoid confusion – it will save you from this error. Essentially, remember that in R, functions are objects too, and `something[...]` tries to subset whatever `something` is. If it's a closure (function), R throws this error because you can't subset a function like that.

These are just a handful of common errors and warnings – there are of course many more one can encounter. Over time, you'll become familiar with the typical ones. A good habit when you see any error is to break it down grammatically:

- Identify the function or operation mentioned (In `foo(...)` : or `Error in foo():`).
- Identify the phrase after the colon, which usually describes the issue (object not found, unused argument, etc.).
- That phrase often can be Googled for quick insight or appears in documentation/StackOverflow.

Also, note that some errors come from packages and might have very package-specific wording. For example, **tidyverse** functions sometimes throw errors with tibbles or `dplyr` that might mention tidy eval or other concepts. When you run into those, it's useful to consult that package's documentation or community forums.

Finally, **warnings** deserve attention too. While they don't stop your code, they might indicate problematic data or impending errors. For instance, a warning "NAs introduced by coercion" tells you that some data couldn't be converted to numeric and became NA – if you ignore that, you might later get an error or wrong results due to those NA values. So treat warnings as early warnings (pun intended) to investigate.

Knowing these common messages will reduce the intimidation factor of debugging. It's like learning a language – "object not found" or "unused argument" becomes part of your vocabulary, and you'll quickly recall, "Ah, I forgot to load a package" or "Oops, typo in variable name" as soon as you see them. And if an error truly stumps you, remember, chances are someone else has asked about it on the internet – you're rarely the first to see a given error in R.

## 8.6 Practical Debugging Workflows (Putting it All Together)

Let's walk through a practical scenario to illustrate how you might combine these tools and strategies in a real debugging session. This will demonstrate the mindset and steps from encountering a bug to resolving it.

**Scenario:** You have written a function to calculate the coefficient of variation (CV = standard deviation / mean) for each column of a numeric data frame. However, when you test it on a sample data frame, you get an error.

Your function:

```
cv_by_column <- function(df) {  
  n <- nrow(df)  
  result <- numeric(n)           # preallocate a vector of length n (rows?)  
  for (j in 1:n) {  
    mu <- mean(df[, j])  
    sigma <- sd(df[, j])  
    result[j] <- sigma / mu  
  }  
  names(result) <- names(df)  
  return(result)  
}
```

Testing it:

```
test_df <- data.frame(a = c(10, 15, 20), b = c(1, 1, 2))  
cv_by_column(test_df)  
#> Error in result[j] <- sigma/mu : replacement has length zero
```



We got an error: “*replacement has length zero*”. Let’s debug this:

1. **Read the error message:** “replacement has length zero” often occurs in assignment when the right-hand side is length 0 (i.e., you’re assigning an empty value to something). The context says `result[j] <- sigma/mu`. This suggests that maybe `sigma/mu` is producing a length-zero result. How could that be? `sigma` and `mu` are numbers (should be, since `mean` and `sd` return numeric). Dividing one number by another yields a number (unless `mu` or `sigma` is length 0). `mean(df[, j])` could be length 0 if `df[, j]` is length 0 (i.e., maybe `df` had 0 rows?). But our `test_df` has 3 rows. Alternatively, could it be that `nrow(df)` is stored in `n`, and we use `1:n` for `j`. If `nrow(df)` is 3, then `1:3` is fine. Wait, but we intended to iterate columns, not rows. There’s likely a bug: `n <- nrow(df)` and then looping `for (j in 1:n)` – this loops 3 times (with `j=1,2,3`) on a data frame with 2 columns (`a` and `b`). So on `j=3`, `df[, 3]` will be `NULL` because there is no third column. Bingo. That would make `mu <- mean(NULL)` which yields `NA` (with a warning) or possibly an error? Actually `mean(NULL)` returns `NA` (with a warning “argument is not numeric or logical”), and `sd(NULL)` returns `NA` as well (with a warning). Then `sigma/mu` is `NA/NA` which is `NA` (not length 0, still length 1 though). Hmm, then why “length zero”? Maybe `df[, 3]` returns `NULL`, and `mean(NULL)` might actually return `NaN` or something length 1? Let’s quickly check in our head: `mean(NULL)` returns `NA` of length 1. So `sigma` and `mu` would be `NA`. `NA/NA` is `NA`. Then `result[3] <- NA` would not be length zero. Unless `sd(NULL)` returns `numeric(0)`? Actually, `sd(NULL)` returns `NA_real_` as well (checked via intuition or known behavior). So `sigma/mu` is `NA/NA` which is `NA` (which is length 1). So that should assign `NA`, not error. So maybe not exactly that.

But consider if `nrow(df)` is used incorrectly. We intended to loop columns, so we should use `ncol(df)`. As is, `n=3` (rows), `j` goes 1,2,3. On `j=3`, we do `df[,3]` which is `NULL` (since `test_df` has 2 cols). Now, what is `mean(NULL)` exactly? It might actually return `NaN` of length 1 with a warning. Or it might return `numeric(0)`? Actually `mean(NULL)` returns `NaN` (just tested in thought, but let’s confirm in the debugging process).

Anyway, possibly a simpler approach: run `traceback()` to see if it shows something: Since the error is in the assignment, maybe not much more info. Or better, use `browser()`.

2. **Reproduce and isolate:** We have the error consistently with `test_df`. We suspect the loop is wrong. Let’s inspect inside function with a debug tool.
3. **Use debugging tool:** Insert `browser()` inside the function at start or before the error. For instance:

```
cv_by_column <- function(df) {  
  n <- nrow(df)  
  result <- numeric(n)  
  for (j in 1:n) {  
    browser() # pause inside loop
```

```

    mu <- mean(df[, j])
    sigma <- sd(df[, j])
    result[j] <- sigma / mu
  }
  names(result) <- names(df)
  return(result)
}

```

Now run `cv_by_column(test_df)`: We enter browser at `j=1`:

- Check `j -> 1`.
- `df[, j]` is `test_df[,1]`, which is `c(10,15,20)`.
- `mu = 15`, `sigma ~ 5` (some value).
- All good, then it will assign `result[1]`. We hit `c` to continue or `n` to step. Let's use `c` to jump to next iteration.

At `j=2` (browser again):

- `j = 2`,
- `df[,2]` is `c(1,1,2)`,
- `mu = 4/3 ~ 1.333`, `sigma ~ 0.577`,
- Fine, assign.

Continue to `j=3`:

- `j = 3`,
- `df[,3]` is `NULL` (since data frame has no 3rd column). In R, `df[,3]` actually returns `NULL` with no warning. (This is likely causing our problem.)
- Now `mu <- mean(NULL)`; what is that? Checking: at browser, type `mean(NULL)`. It likely returns `NA` and possibly a warning. Actually base R:

```

mean(NULL)
# returns NaN (not NA) with a warning:
# Warning: argument is not numeric or logical: returning NA

```

Actually, documentation says `mean` returns `NA` and warns if not numeric. But I recall `mean(NULL)` might give `NaN`. Let's assume `NA`.

- `sigma <- sd(NULL)`: `sd` likely returns `NA` as well (with warning).
- So `sigma/mu = NA/NA -> NaN` perhaps (0/0 style)? Or `NA/NA` yields `NA` I think. Actually `NA/NA` probably `NA` (with maybe no extra warning).
- `result[3] <- sigma/mu`: `sigma/mu` is `NA` (length 1). So why "length zero"? Unless `sigma` or `mu` were `numeric(0)` which can happen if, say, `mean(NULL)` returned `numeric(0)`.

Let's quickly test in console (in analysis, not in code environment since we can't run R here, but trust knowledge):

- `length(mean(NULL))`: if it returned NA, length would be 1.
- Maybe `sd(NULL)` might return NaN or numeric(0)? Actually, `sd` uses `var`, and `var` on 0-length data might return NaN I think but still length 1.

Alternatively, could the error “replacement has length zero” come because result is length 3, `j=3` is okay index, and right side has length 0? Means `sigma/mu` must be length 0. Possibly if `mean(NULL)` returned `numeric(0)` – it might, given `mean` often returns `numeric(0)` for no input, since `sum` over length maybe. But I do recall a warning about not numeric. To be sure, let's bracket: If `mean` returns `numeric(0)`, then `sigma/mu` is `numeric(0)` and assignment fails. That would explain it.

Anyway, the core issue is confirmed: we looped too far.

4. **Identify root cause:** We intended to produce a result of length equal to number of columns, but we mistakenly set it to number of rows. So our preallocation and loop index are wrong. The error manifested when `j` went beyond number of columns, producing an empty right-hand side in assignment.
5. **Fix the issue:** Change `n <- ncol(df)` and `result <- numeric(ncol(df))`. Or simply use `for (j in seq_along(df))` which is safer. We'll do:

```
cv_by_column <- function(df) {  
  p <- ncol(df)  
  result <- numeric(p)  
  for (j in 1:p) {  
    mu <- mean(df[[j]])      # using df[[j]] is another way to extract column j  
    sigma <- sd(df[[j]])  
    result[j] <- sigma / mu  
  }  
  names(result) <- names(df)  
  return(result)  
}
```

Used `df[[j]]` for variety (does same as `df[,j]` for extracting column as vector).

6. **Test the fix:**

```
cv_by_column(test_df)  
# Should output a numeric vector of length 2 named a, b.
```

Let's approximate: For column a: mean 15, sd ~5.0, so cv ~0.333. For column b: mean ~1.333, sd ~0.577, cv ~0.433. So expect something like `a = 0.333`, `b = 0.433`. No error.

We ensure no warnings too.

7. **Edge cases:** What if a column has mean 0? That would cause Inf. Could mention or handle it if needed (maybe not now). What if df has non-numeric columns? Then mean would warn or error. Could refine to numeric columns only, or assume numeric df as given.
8. **Conclusion:** The bug was fixed by correcting loop bounds and using the right dimension.

This story demonstrates:

- Interpreting an error (length zero replacement gave clue of mismatch in lengths).
- Using `browser()` to confirm the suspicion (saw `j=3` caused trouble).
- Fixing the code accordingly.
- Retesting on initial example.

The important lesson is that many bugs come from simple mistakes (like using wrong function or dimension) – careful reading of error and step-by-step investigation often reveals them.

## 8.7 Exercises – Getting Your Hands Dirty

Now it's your turn to practice debugging! Below are a few buggy R code snippets that mimic common scenarios. For each exercise, the task is to **diagnose the bug and fix the code**. Try to use the strategies from this chapter: read the errors, reproduce them, isolate the issue, and test your fixes. Remember, there may be more than one way to fix a problem, but focus on making the code work as intended.

### 8.7.1 Exercise 1: Sum of Sequence (Logical Error)

The following code is supposed to compute the sum of integers from 1 to 10 and print the result. However, it prints NA instead of the expected sum (55). Identify the bug and fix it so that the correct sum is printed.

```
total <- 0
for (i in 1:11) { # supposed to sum 1 through 10
  total <- total + i
}
print(total)
#> [1] NA
```

*Hint:* Think about the sequence `1:11` when the goal is to sum 1 to 10. What happens in that loop?

### 8.7.2 Exercise 2: Data Frame Binding (Runtime Error)

We want to create a data frame by combining two vectors: one of length 5 and one of length 3. Running the code below produces an error. Explain why the error occurs and modify the code to fix the issue (there are multiple ways to address it – you can either adjust the data or change how the data frame is constructed).

```
x <- 1:5
y <- c(10, 20, 30)
df <- data.frame(x, y)
#> Error in data.frame(x, y) : arguments imply differing number of rows: 5, 3
```

*Hint:* All columns in a data frame need to have the same number of rows. You might consider adding missing values or removing some data to balance lengths.

### 8.7.3 Exercise 3: Missing Library (Runtime Error)

The code below attempts to use the **ggplot2** package to create a simple scatter plot, but it throws an error. Identify the cause of the error and fix the code so that the plot is generated.

```
data <- data.frame(x = 1:5, y = c(2, 4, 3, 5, 7))
plot <- ggplot(data, aes(x, y)) + geom_point()
#> Error in ggplot(data, aes(x, y)) : could not find function "ggplot"
```

*Hint:* The error suggests that R doesn't know about the **ggplot** function. What step might be missing before using it?

---

By working through these exercises, you'll reinforce your debugging skills. Remember to apply a structured approach: don't just stare at the code – run it, read the messages, and use tools like `print()` or `browser()` if needed to inspect what's happening. Happy debugging!

## 8.8 Conclusion

Debugging is an integral part of the programming journey, especially in data science where code and data intersect in complex ways. In this chapter, we emphasized a few key takeaways:

- **Adopt the right mindset:** Bugs are not roadblocks, but rather stepping stones to deeper understanding. Instead of viewing errors as “bad,” approach them with curiosity. Each error is telling you something; your job is to listen and investigate. Cultivating patience and even a bit of humor about debugging will make you a more resilient programmer. As we saw with Papert’s insight, embracing the *debugging philosophy* – that errors help us learn – will turn frustration into fruitful problem-solving.
- **Learn to read R’s signals:** R communicates through error messages, warnings, and other feedback. By familiarizing yourself with common messages and what they mean, you can often quickly zero in on the cause. Don’t ignore warnings and don’t panic at errors. Use them as clues in your detective work.
- **Use the tools at your disposal:** We covered many debugging tools – from the simple `traceback()` to interactive debugging with `browser()` and RStudio breakpoints. These tools exist to make your life easier. For instance, rather than guessing what’s happening inside a loop, you can step through it in real time. Rather than wondering which function call failed, you can check the traceback or use `recover()` to inspect it. Mastering these will dramatically speed up your debugging process.
- **Break down the problem:** Tackle bugs systematically by isolating components of your code. Test smaller pieces (perhaps writing little snippets or using the console to simulate parts of your function). When something complex fails, try to reproduce it in a simpler context. This divide-and-conquer approach often not only finds the bug, but can also improve your code structure (you might realize you should refactor a big function into smaller ones, for example).
- **Know common pitfalls:** Many bugs for beginners come from a short list of issues – typos in variable names, forgetting to load libraries, mismatched data lengths, off-by-one indexing errors, unhandled NA values, etc. As you saw in the common errors section, these have clear fixes. Being aware of them means you can sometimes anticipate and avoid them, or fix them quickly when they occur. Over time, you’ll internalize these patterns (“Ah, object not found – likely a typo or I forgot to create it”).
- **Verify and test your fixes:** Debugging doesn’t end when the error disappears. You should re-run your code on a variety of inputs (including edge cases) to ensure the bug is truly gone and hasn’t uncovered another issue. Writing a quick test or at least checking the output manually helps ensure confidence. For example, after fixing our `cv_by_column` function, we’d test it on edge cases like a single-column data frame, a data frame with a zero mean column, etc., to see how it behaves. Testing is the twin of debugging – they go hand in hand to produce reliable code.

- **Continuous improvement:** Each debugging session is an opportunity to improve not just that code, but your future coding practices. Maybe you realize you need to add input validation (e.g., check for division by zero to avoid Inf results). Or you learn that using clearer variable names would have prevented confusion. Perhaps you decide to adopt a style of writing smaller functions because it's easier to debug them in isolation. Over many projects, these little lessons accumulate, and you'll find you write code that's easier to debug – meaning you'll spend less time debugging overall!

Remember that **even expert programmers encounter bugs daily**. What sets them apart is not that they avoid errors entirely, but that they've developed efficient ways to find and fix them. Debugging is a skill, and like any skill, it improves with practice. So, don't be discouraged by bugs – embrace them as part of the process.

In the end, there are few feelings as satisfying as tracking down a stubborn bug and seeing your code finally work as intended. Debugging can be challenging, but it's also deeply rewarding – it's where you truly *get to know* your code and data. With the strategies and tools from this chapter, you are well-equipped to handle the bugs you'll face in your R programming adventures. Happy coding, and happy debugging!

#### TL;DR (Too Long; Didn't Read) – Key Points Summary:

- Debugging is a normal and essential part of coding – approach it with a positive, problem-solving mindset.
- **Types of errors:** Syntax errors stop code from running (fix your code structure); runtime errors occur during execution (use error messages to diagnose); logical errors produce wrong results without errors (test and verify outputs to catch these).
- **Read error messages carefully** – they often pinpoint the issue or at least the location of the problem.
- **Reproduce and isolate** the bug with minimal examples; this makes it easier to debug and to ask for help if needed.
- Use `traceback()` to see where an error occurred in nested calls, and `browser()` (or RStudio breakpoints) to pause and inspect the state of your program at specific points.
- Tools like `debug()/debugonce()` let you step through function execution from the start, and `recover()` drops you into debug mode after an error to examine any frame.
- RStudio's IDE provides a friendly interface for debugging with clickable breakpoints, step buttons, and an environment pane to see variables.
- Common R errors have common causes: “object not found” (undefined variable or typo), “could not find function” (forgot `library()`), “differing number of rows” (mismatched vector lengths), “missing value where TRUE/FALSE needed” (NA in a logical context), “non-numeric argument” (trying math on non-numeric data), etc. Learn these and you can debug many issues on sight.
- When you fix a bug, **re-run your code on test cases** (including the one that originally failed) to ensure the problem is truly resolved and no new issues were introduced.

- Above all, **don't give up**. Debugging can be tricky, but each solved bug boosts your confidence. With practice, you'll become faster and more adept at it. Every coder – from novice to guru – is essentially a professional bug catcher and fixer.

With these insights and techniques, you're ready to tackle bugs in R head-on. Good luck, and may all your bugs be shallow!



## 9 Conclusion

### 9.1 From Pipeline to Practice – Charting Your Own Data-Science Journey

Writing this book has been an exercise in showing—not merely telling—how **end-to-end, reproducible, human-readable analysis is possible with a single, coherent toolchain** built around R, Quarto (`.qmd`), and the RStudio IDE. Together we have:

- **Framed the problem space**—from data collection and wrangling to interactive dashboards and APIs.
- **Learned the mechanics**—Markdown and Quarto for narrative, R for computation, `{tidyverse}` for data pipelines, `{ggplot2}`/`{plotly}` for visuals, `flexdashboard`/`Shiny` for dashboards, `httr`/`{httr2}` for RESTful calls, and Git + GitHub for version control.
- **Adopted the reproducible-research mindset**—every step scripted, every figure and table regenerated at the press of *Knit*, every assumption explicit.
- **Practised debugging, testing, and refactoring** so that errors become opportunities, code becomes modular, and insight remains trustworthy.
- **Laid out a pragmatic workflow** that is IDE-agnostic, but shines brightest inside RStudio’s integrated panes: `script` `console`, `environment` `viewer`, `Git` `terminal`.

#### 9.1.1 The bigger picture

**Data work is story work.** Numbers are inert until they are transformed into decisions; decisions are fragile until their provenance is transparent; transparency is impossible without literate, version-controlled code. Quarto documents—and R Markdown before them—fix that gap. By uniting prose, code, figures and references in a single file, they:

1. **Create an audit trail** from raw data to published insight.
2. **Lower the cost of peer review**—colleagues can re-run your analysis with one command.
3. **Short-circuit duplication**—the same script can render HTML for the web, PDF for a journal, slides for a talk, and a dashboard for executives.

In short, literate programming and reproducible research are no longer academic ideals; they are industrial-strength practices that any analyst—or business unit—can adopt today.

### 9.1.2 What you should feel comfortable with now

Skill	Key take-aways you can already apply tomorrow
<b>Setting up RStudio projects</b>	Keep each analysis self-contained; use relative paths; commit early, commit often.
<b>Reading and cleaning data</b>	Prefer <code>readr</code> , <code>janitor</code> , and <code>dplyr::across()</code> for tidy, declarative transformations.
<b>Exploratory visualization</b>	Match geometric objects to data types; layer aesthetics; let the data suggest the chart.
<b>Dashboards &amp; reporting</b>	<code>quarto render</code> for static reports, flexdashboard/Shiny for interactive views, GitHub Pages or Posit Connect for deployment.
<b>APIs and automation</b>	Wrap REST calls in functions; store keys in environment variables; rate-limit politely.
<b>Debugging &amp; testing</b>	<code>traceback()</code> , <code>browser()</code> , and <code>testthat::test_that()</code> are worth their cognitive weight in gold.
<b>Citation &amp; referencing</b>	Zotero + Better BibTeX + @citekey keeps you honest and your supervisors happy.

### 9.1.3 Where to go next

1. **Deepen your statistical toolbox** – packages like `{infer}`, `{modelr}`, `{tidymodels}`, and `{posterior}` make modern modelling workflows natively tidy and reproducible.
2. **Scale out** – learn `{arrow}` or `{duckdb}` for larger-than-memory data, or push logic to the database with `{dbplyr}`.
3. **Automate end-to-end pipelines** – pair Quarto with GitHub Actions (or GitLab CI, Jenkins, etc.) so that every commit triggers a fresh render and publishes artefacts automatically.
4. **Contribute to the ecosystem** – file issues, answer questions, write blog posts, or publish your own R packages. Teaching is the fastest debug cycle.
5. **Stay curious** – follow the Posit Blog, R-bloggers, and the `#rstats` hashtag; subscribe to the RWeekly newsletter; attend local R-user groups or R-Ladies events. The field moves quickly, but a welcoming community helps you keep up.

### 9.1.4 A note on mindset

If there is a single theme threading every chapter, it is this:

**“Code is a conversation.”**

Each script you write speaks simultaneously to the computer and to the next analyst—including *future you*. Strive for clarity over cleverness; for small, composable functions over mega-scripts; for explicitness over magic. When your code *reads* well, it tends to *run* well, and when it doesn't, the bugs reveal themselves quickly.

### 9.1.5 Your call to action

1. **Fork the repository** (or start a new one).
2. **Clone, edit, knit**—add a dataset of your own, refactor a pipeline, try a different template.
3. **Push and pull-request**—share your improvements or examples back with the community.
4. **Teach someone else**—whether a teammate or an online audience, explaining these ideas will cement them for you.

### 9.1.6 Final words

Data science is still young. Tools change; principles endure. If you remember only three things, let them be:

- **Reproducibility first** – treat every analysis as if a stranger will rerun it tomorrow.
- **Automation amplifies insight** – once a task is scripted, your mind is free for higher-level thinking.
- **Learning never ends** – today you mastered R and Quarto; tomorrow you may integrate cloud APIs, real-time dashboards, or machine-learning workflows. The foundations you now have will adapt.

Thank you for investing your time and trust in this journey. May your data be clean, your code be clear, and your insights spark positive change. **Happy knitting, and see you in the commit history!**

# 10 Summary

This book is a hands-on, soup-to-nuts guide to building **reproducible, insight-driven data products in R**—from the very first click in RStudio Cloud to the moment you publish a polished dashboard or academic manuscript. Across ten tightly integrated chapters, it escorts you through the entire modern analytics pipeline while championing three core principles:

1. **Literate programming** – weave prose and code so your logic is transparent.
2. **Reproducible research** – automate everything, version everything, share everything.
3. **Professional storytelling** – craft outputs that executives, clients, and peers can trust at a glance.

Below is a chapter-by-chapter recap that draws the main threads together, highlights pivotal code patterns, and underlines the habits you should now regard as second nature.

## 1 | Introduction – Why R, Why Literate Documents, Why Now?

The book opens by framing data science as an *explanatory* discipline, not merely a predictive one. In an era of information overload, the winners are those who can pair rigorous computation with clear narrative. You are introduced to RStudio (now *Posit*), Quarto (`.qmd`), and GitHub as the de-facto stack for that mission. The introduction argues that an IDE is more than a text editor: it is a cognitive dashboard where raw data, code, graphics, and citations coexist. Readers begin with a short tutorial on launching an RStudio Cloud project, setting their global options, and rendering the template *Hello, Quarto* document to prove the toolchain works on any laptop—even a Chromebook.

**Key take-away:** *Every* file in a project lives under version control *and* is referenced via relative paths, ensuring portability and perpetual reproducibility.

## 2 | For the Impatient – A 90-Minute End-to-End Sprint

Impatient readers build a complete mini-pipeline in under two hours: download a small CSV, clean it with `dplyr`, visualise it with `ggplot2`, embed the plot in narrative Markdown, and push the project to GitHub. The chapter emphasises *workflow muscle memory*:

- `Project > New Directory`
- `git pull - commit - push` reflex

- `quarto render` on the command line
- publishing the rendered HTML automatically via GitHub Pages.

By the end, users believe that *one source file can indeed output multiple formats*—HTML, PDF, PDF slides, and Word—without copy-and-paste. That confidence lubricates the deeper dives that follow.

### 3 | Markdown, Quarto & R Markdown – Literate Analysis Fundamentals

Here you master the syntax that glues language and narrative:

- **YAML header** for title, author, date, bibliography, and output engine.
- **Code chunks** (````{r} ... ````) for executable R.
- **Chunk options** (`echo=`, `eval=`, `include=`, `fig.width=`) that govern what the reader sees.
- **Inline code** (``r expression``) for dynamic prose.
- **Cross-format styling**: headings, bold/italic, lists, block quotes, tables, LaTeX math.
- **Image embedding** and controlled sizing with `{width=600}` or `{height=40%}`.

A dedicated sidebar contrasts legacy `.Rmd` with Quarto’s `.qmd`, noting Quarto’s multilingual stance and richer project-level configuration. Yet the chapter reassures legacy enthusiasts that `.Rmd` still knits seamlessly.

**Mini-project:** replicate a provided analytical memo pixel-perfectly. The exercise cements chunk mechanics, heading discipline, and the value of alt-text for accessibility.

### 4 | Data Wrangling – From Messy Reality to Tidy Tables

Data rarely arrive analysis-ready. This chapter is a boot camp in the *tidyverse* grammar:

- **Import** with `readr::read_csv()`, `janitor::clean_names()`, and `haven::read_spss()`.
- **Pivot** (`pivot_longer()` / `pivot_wider()`), **separate/unite**, and **type-convert**.
- **Pipe chains** with `%>%` or the native `|>` operator to tell a left-to-right story.
- **Grouping** and **summarising** with `dplyr::across()` for multivariate summaries.
- **Join strategies**—inner, left, semi, anti—visualised as Venn diagrams.
- **Date-time wrangling** via `lubridate`.

Alongside recipes, the book stresses *defensive programming*: assertions like `stopifnot(nrow(df) > 0)` and the habit of checking `glimpse()` at every step. You finish the chapter with a perfectly tidy table that feeds all subsequent examples.

## 5 | Visuals – Turning Tables into Insight

Building on the cleaned dataset, you transition to grammar-of-graphics thinking. Topics include:

- **Layered plotting:** geoms, aesthetics, statistical transformations.
- **Scales and themes**—why colour, size, and shape encode meaning.
- **Faceting** for sub-group comparisons.
- **Interactive overlays** via `plotly::ggplotly()` and `{ggiraph}`.
- **Saving and caching** figures reproducibly with `gifski`, `magick`, and `ggribbles`.

Every section couples a visual principle (e.g. avoid dual axes) with code and a live Quarto chunk. The reader learns to embed high-resolution PNG, SVG, and GIF outputs, ensuring the knitted report is portable across devices.

## 6 | Dashboards – Sharing Results at a Glance

Static reports satisfy academics; executives crave dashboards. You therefore graduate to *flexdashboard* layouts, converting R Markdown or Quarto panels into an interactive single-page app. The chapter walks through:

- The **YAML scaffold** (`output: flex_dashboard`, `orientation: rows/columns`, `runtime: shiny`).
- Building **value boxes**, **dygraphs**, and **leaflet maps** side-by-side.
- Binding **Shiny reactivities** in only 20 lines of code for filterable plots.
- Publishing options: GitHub Pages (static), Netlify (static), or Posit Connect (dynamic).

Crucially, the narrative emphasises *audience empathy*: start with a KPI value box, provide drill-down visuals, end with downloadable data. A final checklist covers performance optimisation—`renderCached`, `req()`, and global options.

## 7 | APIs & External Data – The World Beyond CSV

Real projects often pull live data. This chapter demystifies RESTful APIs with `{httr2}`:

1. **Authentication patterns** – API keys, Bearer tokens, OAuth dance.
2. **GET vs POST** – modern endpoints, query parameters vs request bodies.
3. **Parsing JSON** to a tibble with `jsonlite::fromJSON()` or `{tidyr}`'s `unnest_longer()`.
4. **Pagination and rate limits**—`while(has_next)` loops and `Sys.sleep()`.
5. **Idempotent caching** with `pins` and `memoise` to avoid hammering servers.

A real-world mini-case pulls weather data, cleans it, appends it to the earlier tidy table, and refreshes the dashboard automatically with `quarto render` inside a GitHub Actions schedule. Readers see continuous integration/deployment in action.

## 8 | Debugging – Finding and Fixing Errors in R, Rmd, and Qmd

No pipeline is flawless on first run. The dedicated debugging chapter (now exclusively R-centric) equips you with:

- **Interactive techniques** – breakpoints in RStudio, `browser()`, `debugonce()`, `traceback()`, and `options(error = recover)`.
- **Programmatic guards** – `stopifnot()`, `{checkmate}` assertions, and custom error classes via `rlang::abort()`.
- **Typical bugs** – NA propagation, factor-to-numeric coercion, off-by-one loops, subscript-out-of-bounds, and improper working directories.
- **Reprex culture** – minimal, runnable examples with `{reprex}` for Stack Overflow or GitHub issues.
- **Test-first mindset** – `testthat::test_that()` plus continuous integration so bugs never re-appear.

Using a deliberately broken Quarto document, the chapter demonstrates how to trace an NA that silently zeroed an entire column and how to set a conditional breakpoint inside a `purrr::map()` iteration. Readers leave with a systematic eight-step checklist: reproduce → simplify → read error → inspect state → form hypotheses → test hypotheses → fix → write a regression test.

## 9 | References & Citations – Scholarly Discipline Meets Data Science

Good science cites its sources. Leveraging Zotero + Better BibTeX, the book teaches:

- Managing a shared Zotero group library.
- Exporting a `.bib` file and linking it in YAML (`bibliography:`).
- Inserting citations with the *citr* add-in or plain `[@key]` syntax.
- Switching citation styles via CSL files.
- Generating auto-formatted bibliographies on knitr.

Examples include inline APA citations, footnotes, and automatically numbered reference lists for HTML and PDF outputs. A short section shows how to embed DOI-generated data citations—vital for open science.

## 10 | Conclusion – Pipeline to Practice

The concluding chapter (already shared earlier) distills the philosophy: **code is a conversation**. It revisits each stage—environment, wrangling, visuals, dashboards, APIs, debugging, citations—and argues that together they constitute a *repeatable craft*. Readers are urged to fork the book’s GitHub repo, swap in their own dataset, and publish their first public Quarto site as the ultimate graduation project.

### Five Cross-Cutting Themes

1. **Reproducibility at Every Step** Random seeds (`set.seed()`), pinned raw data, fully scripted transformations, and commit history ensure anyone can rebuild outputs byte-for-byte months later.
2. **Human-Readable Storytelling** Clear headings, alt-text on images, sensible factor labels, and executive summaries mean decision-makers need not dig through code to grasp findings.
3. **Automation as Leverage** Parameterised Quarto documents, GitHub Actions, and Posit Connect publishing turn a one-off analysis into a living artifact that regenerates with new data.
4. **Community & Collaboration** Git pull-request flows, issue templates with reproducible examples, and shared Zotero libraries enable teamwork that scales beyond a single analyst.
5. **Defensive Programming** Assertions, tests, and vigilant logging catch edge cases early. Bugs become teaching moments rather than silent saboteurs.

### What You Can Do Tomorrow

- **Clone the template project** – replace the sample CSV with your own and re-run `quarto render`.
- **Set up CI** – enable GitHub Pages or Actions so your report updates nightly.
- **Add one assertion per function** – stop bad inputs before they fan out.
- **Write one unit test** – lock in an important transformation.
- **Publish a dashboard** – wow stakeholders with an interactive KPI board.
- **Share your code** – invite feedback; code is stronger when peer-reviewed.



## Final Reflection

This book has argued that **the distance from raw data to professional insight can be one literate document**. Armed with R, Quarto, RStudio, and Git, you now possess a complete, open-source alternative to sprawling spreadsheets and opaque slide decks. More importantly, you have internalised a mindset: *analyses should be reproducible, opinionated about quality, and generous toward future readers*. Whether your next task is a marketing A/B test, a financial forecast, or a public-health study, the workflow remains the same: tidy the data, interrogate it visually, narrate it honestly, automate the regeneration, and debug mercilessly.

Data science is a fast river, but these principles are a stable bridge. Walk it often, improve its planks, and invite others across. The world has plenty of data; it desperately needs **trust-worthy, well-told stories**—and now you know how to craft them.

## References