A desk reference for political data preprocessing and management

Damon C. Roberts

A desk reference for political data preprocessing and management

Damon C. Roberts

Last Compiled: 2023

Table of contents

List of Figures

Preface

This book was originally inspired by a blog post that I had written and tried to find a more formal outlet for. When writing the blog post, a nagging thought kept hanging in the back of my mind, "What if you write this as a book?" I knew that this was a dangerous thought. At the time I was preparing for the academic job market and was in the throes of writing my dissertation (another book project). What a dumb idea. But, I was passionate about the topic and needed something to do when I needed a break from my dissertation. So, this project was born.

1 Introduction

As I often teach my students in introductory quantitative research methods classes, each and every choice you make for a study has deep implications for the way that you interpret results, what results you come up with, and the conclusions you draw from them. This book is not going to focus on the particular estimator one may choose to model a binary outcome variable nor will it focus on best practices for interpreting those estimands. Rather, this book is meant to address the process that comes before that – how to clean (which is one step of data preprocessing) and manage one's data.

This is an often neglected part of the process. When we enter graduate school, often we learn about best practices for recoding variables, for dealing with missing data, and eventually learn that it is not a good idea to overwrite one's original `.csv` or `.dta` files with our cleaned data. While, I understand why this is the case, it is a concerning approach to quantitative research and for training those to engage in it.

As I am someone who has been tasked to bring, often reluctant, students along to learn the statistical programming language `R` as well as statistical concepts all the way through multiple regression in a 16-week timeframe, I am sympathetic to the tendency to just give students datasets that I've already cleaned and pre-processed. These challenges are even greater when one is tasked with teaching graduate students these things as there are high expectations for each student to master these topics by the end of the term so that they can be successful for the remainder of the program and their academic careers.

Though, I understand this position, it seems that we do not really grow out of these tendencies to learn how to pre-process our data in principled, as opposed to ad-hoc, ways. There are so many examples of replication materials that do not include any documentation (e.g., code) about how the researchers took their raw data and put it in an analyzable format. Their data goes from some messy spreadsheet and viola, it is now a super neat and tidy spreadsheet with variables constructed from the raw variables without any documentation about how that was done.

Where it is becoming harder and harder to find a published paper that has results that we can replicate, the lack of documentation for how the data came to be is disheartening. While many express shock at the fact that we only just now are discovering that university presidents and high-profile scholars who have had 30-year careers have been fabricating data (in a cynical view) or at least are making serious mistakes (in a more optimistic view) when processing their data, I am not surprised at all.

Concurrently, the standards for publishing a quantitative paper are increasing. We are expected to publish more papers that are of better quality. These requirements often force us to move quicker but somehow more accurately. One side effect of demands for quality is that we are often encountering datasets with more and more rows *and* columns in them. This produces more opportunities to make mistakes (to be less accurate) as well as to take longer to pre-process and manage the data since there is so much of it (less efficient).

There are three primary goals of this book. The first goal of this book is to convince others who handle, analyze, draw conclusions from, and make recommendations about policy and political outcomes to take a less ad-hoc and a more principled approach to managing and pre-processing their data. The second goal of the book is to offer recommendations about the ways that we can implement a more principled approach to data pre-processing and management. And the final goal of the book is to act as a useful desk reference for undergraduate students, graduate students, and researchers to the various options we have out there to document our data pre-processing and management – in terms of programming languages and particular libraries within it.

Addressing these three goals should help one be more efficient and more accurate in their data pre-processing and management. If one pre-processes and manages their data with code as opposed to ad-hoc, clicking around and editing a spreadsheet, this should provide some degree of readable documentation about how the analyzed data came to be. As I will discuss in the next chapter, the efficiency comes from choices about the programming languages and the particular libraries that one uses within that language that provide the functions to make this management or pre-processing easier.

1.1 Plan of the book

The following chapter is designed to provide an introduction to the different programming languages and libraries within those languages that are available to us for data pre-processing and management. In the chapter, I will provide directions to get set up with each programming languages, will offer a brief discussion of what libraries and functions are in these programming languages, will discuss some options has to choose from for managing these libraries and functions for each programming language, will discuss some of the most popular libraries used for data management and pre-processing in each of these languages, and will discuss the concept of efficient, readable, and replicable code. Throughout the chapter, I will compare and contrast how effective each coding language and their common data management and pre-processing libraries are for efficient, readable, and replicable code.

The third chapter starts our exploration of the common ways that we can think about managing our data. That is, "how do we keep track of our raw and cleaned data?" It will discuss common bare minimum requirements for data security for human subjects set by Institutional Review Boards in the United States, how a principled approach to data management can help increase one's data security practices, and will make an argument about a workflow that one should consider implementing when working with data throughout a given research project.

The fourth chapter continues the discussion about principled data management. In doing so, the chapter puts chapter three in practice. That is, we see how we can use code to achieve the lofty goals we set in chapter 3. In doing so, it provides coding examples for the data management tasks for each of the programming languages and the common libraries within each of those programming languages.

Now that we are familiar with the theoretical best-practices for data management as well as how to actually implement them, the fifth chapter will move into data pre-processing. This chapter in particular will focus on a type of data pre-processing often referred to as data cleaning or data munging. In doing so, the chapter will discuss common tasks that we have to perform, provide coding examples of how to do so, and will compare the accuracy and efficiency of each programming language and library for these tasks.

The sixth chapter will focus on a more advanced set of data munging steps – variable transformations, standardization and normalization, simple scale creation (e.g., creating an additive scale). Like the fifth chapter, the sixth chapter will provide coding

examples of how to perform these tasks and will compare the accuracy and efficiency of each programming language and their common libraries for these tasks.

The seventh chapter will focus on a special but important type of data that we often need to pre-process: missing data. Each programming language has a different way of internally documenting missing values for a variable. The chapter will discuss these common pitfalls, how to detect whether there is a missing value, and will discuss the need to engage in exploratory data analysis to determine the underlying pattern that creates the missing data and will introduce some common approaches to addressing them. In doing so, the chapter will provide code examples for the common libraries in each programming language. While doing so, it will also discuss the accuracy and the efficiency of these libraries and programming languages for these tasks.

2 Common programming tools, their benefits and drawbacks

Those performing quantitative analyses on data have a wide number of options for tools available to them. I assume that the readers of this book are either reading this book while learning about their first statistical programming language or are already familiar with at least one. When choosing our tools, we often use whatever we are told to learn. Either we are a student that is tasked with learning the statistical programming language `R` or perhaps even `Python`. We may also be an academic researcher who has been doing quantitative analyses for years with `R` and we have done so since graduate school because we were told that our academic discipline was going that way. The tools that we learn and use are still options, however.

Each tool I discuss in the book have an origin in trying to address some sort of need. For example, `Python` is one of the most popular programming languages in the world. Despite this, it is one of the most used languages by data scientists. Data scientists by no means make up the bulk of those who are professional programmers. `Python` is a general-use programming language that is designed to be accessible for those who want to do things like statistical analysis, web-design, and web application and software engineering. That is, it is a general use programming language. `R` and `Julia` are programming languages like `Python` in that they are designed to be highly accessible. They are designed, however, to be useful specifically for statistical analysis and reporting. Their intended use is a bit more narrow – though, you can probably find examples of people using both languages for other things besides statistical analysis.

These examples illustrate an important point at the heart of this book: while there are plenty of options and people will use all of the tools I will discuss in this chapter to do data management and pre-processing, it does not necessarily mean that the tool is a "great" or even "good" option for these tasks. In many cases, a particular tool I will provide an example with in the book is a "good-enough" option that can work. This is because each of these tools are created with an original purpose in

mind. One of the best ways to determine whether or not a tool is appropriate for the task at hand is to ask yourself, "did this tool eventually get this functionality, or was it designed with it?" If the particular tool eventually included that functionality, then it is a tool that is "good-enough" for that task. If the tool was designed with the functionality you need from it, then it is probably a "good" or "great" tool. I will make mention of these tools and what their purposes are by the tools' creators as I introduce them throughout the chapter.

I am a realist though. Many of us are balancing our limited attention, time, and energy to do quantitative analysis. We often have deadlines we are facing and have to go with what is "good-enough." The spirit of the book is not judge or chastise people for the tools that they choose to use, but to give them the opportunity to truly choose their tool rather than to use what they thought they had to use. The hope is that the book will give you the information needed to make the choice between whatever tradeoffs you face with using these tools an easy choice. If you choose to change things up, even if it is reluctantly, then the hope is to provide you something to reference to make that change easier!

2.1 An aside about efficiency, readability, and replicability

At the outset of the book, I kept mentioning the "efficiency, readability, and replicability" as being important features or a programming language. Here, I want to elaborate upon what I mean by these terms.

The first feature is efficiency. When we think of efficiency, we may think of how much effort is being put into a task relative to how much is accomplished by completing the task. That is what we mean here too. While there are some really technical rabbit holes one may fall into among computer scientists when defining the efficiency of one's code and various metrics that we can calculate to quantify efficiency, when I refer to efficiency, all I am referring to is, "how much work is your computer putting in relative to the task?" For example, if it takes 20 seconds for a program we wrote on our computer to evaluate the expression `2+2`, then we would say that is pretty inefficient relative to me just doing it in my head. In that case, we would say that our program is pretty inefficient.

When thinking about efficiency, there are more ways to measure than the time elapsed to calculate it like in my example of evaluating the expression `2+2` – though this is

perhaps one of the most common ways efficiency is measured. We can also think of it as how much memory our computer must commit to using. When purchasing a computer, you may hear someone discuss the `RAM` of the computer. While `RAM` is one type of memory, it is distinct from the Hard Drive on your computer that stores your files and software on it. One way that we can think of the `RAM` is the stuff that you have asked your computer to keep track of at any given point in time. `RAM` is active memory. Let me illustrate this through an analogy. You have probably memorized your phone number. Now say that I ask you to enter your phone number while I am also asking you to keep track of my phone number as I read it out loud to you. This is probably a lot to keep track of. You are not only being asked to keep track of 18 numbers all at once, but also the order in which those numbers appear and to keep track of which one is yours and which one is mine. This task might be hard. If you don't believe me, try it with a friend. The `RAM` simply refers to the amount of stuff you are asking your computer to keep track of and readily accessible within a split second. It is the calendar events, it is the web pages you have open on your browser, it is the contents of the song that you are playing, etc. The files that are currently closed on your computer are stored on your hard drive and you are not necessarily asking your computer to have that information readily accessible in a split second, not until you open that file, at least. While this is an oversimplification, `RAM` is another common metric that people think of when thinking about the efficiency of a program. If the program has **tons** of information stored in it, it will significantly slow down your computer as it is struggling to keep up with all of the things you are asking it to do – just like it would slow you down to enter your phone number while I'm reading out my own to you at the same time.

We want the evaluation of our code for our data management and pre-processing to be both quick and not too laborious for our computer to do. If we use code that is inefficient, it will take longer for us to be able to complete these tasks. Additionally, if we have a computer with more `RAM` and it works on our computer, it may not work on someone elses that has less `RAM`, or at least it will take much longer for them than it would for us. This is an issue I will come back to in a few paragraphs.

Another important feature of a programming language and of our own code is the readability of it. In an ideal world, our code should be understandable even to those that are not familiar with the coding language we are using. One of the primary reasons in a research setting will be discussed shortly, but otherwise it makes it easier for us to detect errors in our code! If our code takes a lot of effort to understand what it is doing, then it will undoubtedly make it more difficult for us to catch a mistake that we have made. We want our code easy to understand so that we can catch our

mistakes as well as others'. Now, we often do not have the luxury of jumping into any coding language and immediately understand what that code is doing, so it is often far-fetched to say that it should be readable to those who have no experience with the language, but we should still make it reasonably easy to get a sense of what is going on for everyone and super clear what is happening to those that are familiar with the language.

Reproducibility is the last key feature that we should consider about code. I have hinted to it a few times. Reproducibility is a challenging but important endeavor for every research project that we work on. Reproducibility refers to the idea that anyone can take our documentation, repeat what we did, and be able to get the same results (or at least come to similar conclusions). How can we achieve reproducibility? Well, this is a topic that is still debated. But in general, we want the steps we take when managing and pre-processing our data to be something that someone else can follow relatively easily. Meaning that the documentation we have is accurate, our code is efficient so almost anyone can do it regardless of their particular computer's hardware, and that it is readable so that as many people as possible can understand the steps we took during the project.

Later in this chapter, I will be evaluating the programming languages and common libraries used for data management and pre-processing on these three key features. It is important to note that these languages and libraries are not deterministically good or bad at achieving these goals. Often the person writing the code is at fault for writing inefficient, unreadable, and unreproducible code. However, there is no one "right" way to do anything. However, there are some programming languages and libraries that do not do much to encourage efficient, readable, and reproducible code. So, my evaluation of these languages and libraries on these features will be limited to their capacity to encourage the programmer to produce code that contains these features.

2.2 What is a library?

Before, I move on, it is important to provide a brief explanation of how many of these programming languages that you are about to learn work. Most of these languages are referred to as object-oriented programming languages (`OOP`). What this means is that we can store the results of some task in something called an `object`. So, say for example that I wan to evaluate the expression, `2+2`. Once I have successfully

determined that the result of the expression is `4`, I may want to store that result for later. In an `OOP`, I can store the result, `4`, in an `object`. Now, say that I won't always be adding `2` and `2`, but I may instead be adding `2` and `3` or `10` and `2`. Instead, I want to quickly be able to evaluate a summative expression and be able to just plug in those numbers when I need to rather than have to keep rewriting `a+b`. This is a perfect candidate for a `function`. A function is just a piece of code that takes a pre-specified set of parameters, such as what integer `a` and `b` represent, and perform some task using those parameters. I can store the function to evaluate the summative expression of `a+b` and just pass in the integers that represent `a` and `b`, such as `a = 2` and `b = 2`. `OOP` allows me to store this function as an object just like I would the result of `2+2`. If you are still not quite sure what this looks like in practice, don't worry, you'll see later in the chapter!

So, a function is some chunk of code that we can name and store as an object so that we do not have to reinvent the wheel over and over. While in the previous example of a function that does `a+b`, this may seem relatively not hard to do, but sometimes we have tasks that are extremely complex and take a lot of code to complete. So, functions represent stored code that take our parameters as an input, does something to that input, and then returns some output. This is illustrated in Figure 2.1 below.

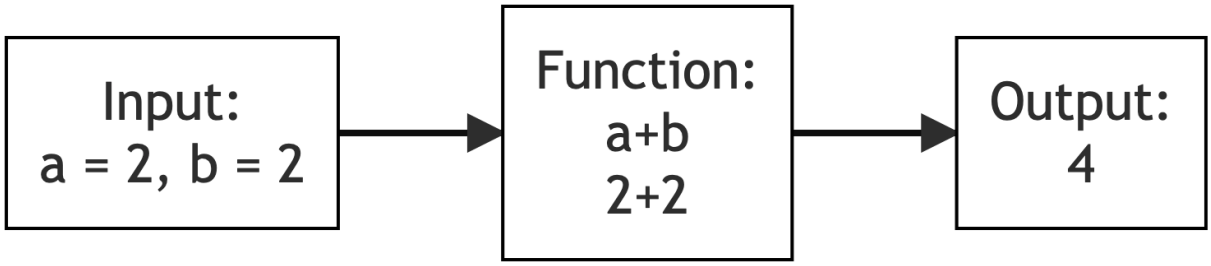

Figure 2.1: Visualizing a function

Since we can save a function as an object thanks to `OOP`, we can store this object in what is called a library. A library refers to a collection of functions which all contain code designed to complete specific tasks. So for example, I can create an object storing the function depicted in Figure 2.1 to evaluate a summative expression as `addition`. I can create another object that stores the function to evaluate a differencing expression as `subtraction`. I can put these two objects in a library, as

depicted in Figure 2.2, and post them to a specialized website for others to download the library so that they can easily download those functions and easily use those functions without having to write their own functions or code.

In essence, libraries are just large files of code that someone else has written that let you use a particular function defined in that code so that you only need to provide some inputs (or arguments is what they are often called), the code does something to those inputs, and then it produces some output.

How is all of this relevant to data management and pre-processing? As I mentioned, most of the languages that we are using here are `OOP`. Most of the time, we do not have to write our own code from scratch to manage or pre-process our data. We will have to write some code, but we are often interacting with functions that someone else has written and put together in a library for us to make our code much shorter. As these functions are written by others and we often are heavily dependent on functions from a library that someone else has written, we should and cannot blindly rely on functions or libraries. We should not always assume that they are efficient, readable, nor enable reproducible code. Just because a programming language enables these features in our code, our experience with this is heavily dependent on the particular libraries that we choose to use within a particular programming language. That is why, in the next section and the book at large, I will be putting a lot of emphasis on the common packages people use within each of these programming languages to pre-process and manage their data.

2.3 Programming language options, setup, and comparisons

To re-iterate, there is no one absolute better programming language for someone to use to pre-process and manage their data. However, this book will focus on emphasizing and advocating for programming languages that are efficient, readable, and reproducible. The options that I include here are not comprehensive. One primary reason for a programming language to be excluded is due to it requiring a license to use. One popular software in the social sciences is `STATA`. While the programming language is not proprietary, you can only use `STATA` as a programming language in their proprietary software, and is therefore not reproducible. As a result, `STATA` is not discussed as an option in this book, though it is popular.

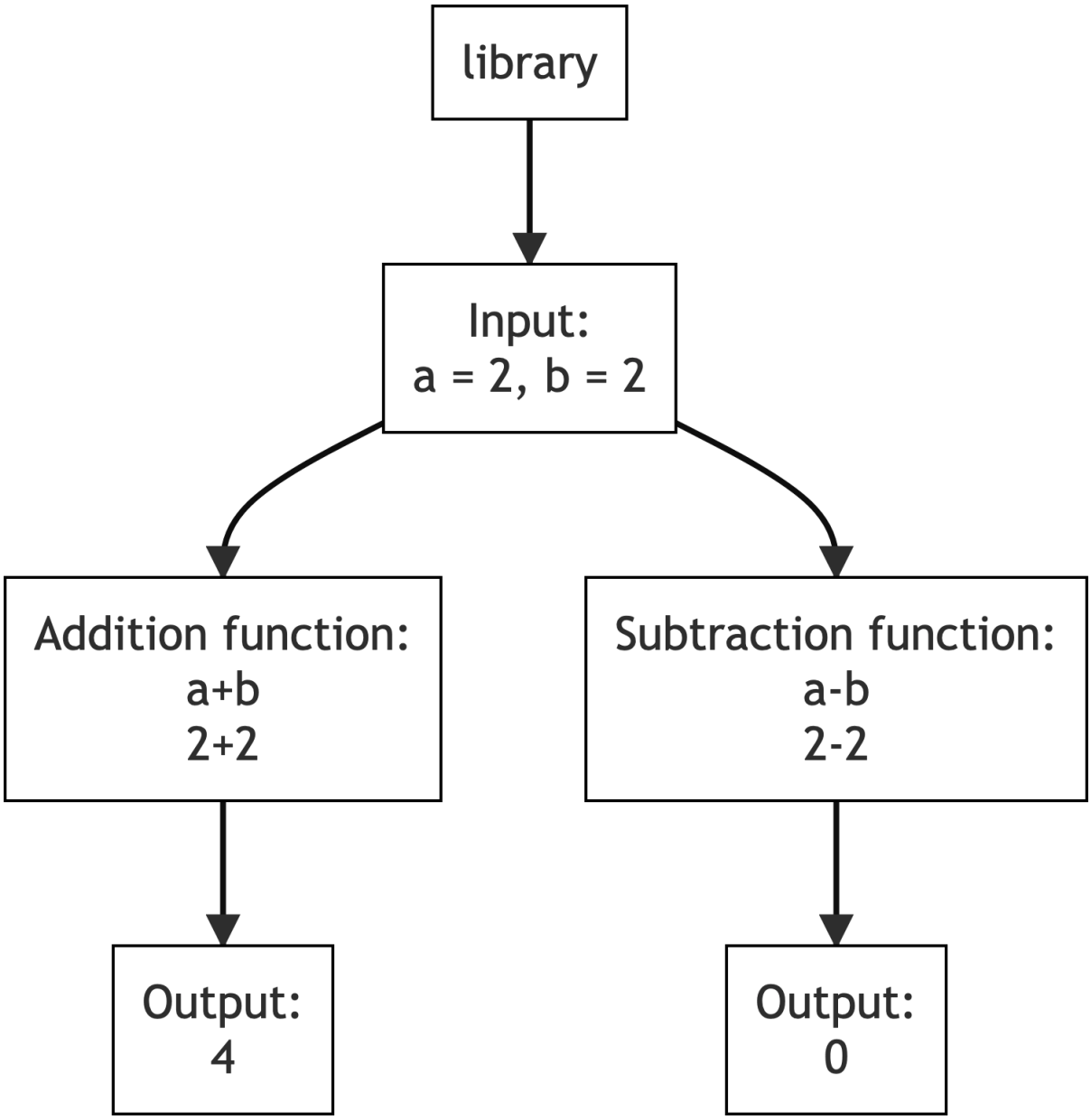

Figure 2.2: Visualizing a library of functions

Before, jumping into the installation and beginning to interact with the particular programming languages, I want to discuss the difference between installing the software for your computer to understand the language and the installation of the Interactive Developer's Environment (`IDE`) that provides nifty features to not just write the files but to interact with the code such as running the code in parts rather than as the whole file, debugging tools, and other features.

These coding languages do not come standard on our computers. We have to install software that links the code already on our computer that it understands (often C++). This is the first thing we will need to download. Often this software will come with extremely basic `IDEs` that we can use to write code, save files that store the code, and run code, as either complete files or as smaller pieces of code such as a single line (interactively). These `IDEs` are often extremely basic and include really limited functionality. So the second think we often download is a more fully-fledged `IDE`.[1] Some programs have their own popular `IDEs` such as `R` with `RStudio`. However, in the spirit of not being beholden to a single coding language for all tasks, I'll provide instructions on how to install a really popular do-it-all `IDE` called Visual Studio Code (`VSCode`). `VSCode` has extensions that enable you to be able to fully interact with all of these coding languages discussed below.

2.3.1 Installing an IDE, `VSCode`

Again, if you want to install RStudio or some other language specific `IDE`, that is totally up to you! Here, in this book I will cover how to install an `IDE` that works for all of the languages in this book.

First, go to https://code.visualstudio.com/download. Then you will want to choose an installation that is appropriate for your operating system. Unless you are wanting to do some custom installation, you should choose the button just under the icon for the type of operating system that you use. You can then follow the standard installation steps you take to install software on your computer. You should not need to change any of the defaults to install `VSCode` on your system.

Once successfully installed, then we can move onto installing the particular coding languages.

[1] Also, for those interested, you can forego `IDEs` altogether and install a text editor such as `neovim`. While these are extremely customizable, they require quite a lot of comfort and sophistication with command line interfaces and require a lot of work to get them to a point where you can be semi-productive with them.

2.3.2 R

R is an extremely popular coding language for statistical analysis among academics, analysts, and for a decent proportion of data scientists. There are a number of reasons for its appeal but chief among them is the robust community to help with the reporting of statistical analyses. Unlike some of the other option discussed below, R has a large number of options for packages that help with generating tables and figures for the results of complicated statistical analyses. This is not to say that the other programming languages do not have any support for data visualization and reporting, but rather that there are more options for popular types of data visualization and reporting in the scientific community. This makes it really popular among academics.

2.3.2.1 Installation

Installation of R is quite straightforward. The first thing that we will want to do is navigate to the CRAN website at https://cran.r-project.org. This website is the "Comprehensive R Archive Network" and is the default place that we not only install R, but we can often find and install many of the libraries (often referred to as packages by R users).

Once on the webpage, we will want to follow the link to "Download R for" the operating system that we are using. So if you are using a Mac, then you will want to follow the link to the "Download R for macOS". Once clicking on the link, it will take you to a page listing different releases of R. You will want to follow the most recent one. At the time of writing this book, that would be `R version 4.3.1`.

> **For macOS users**
>
> If you are installing R on a Mac, you cannot install any package of R. If you are using one of the newer Apple Silicon chips, then you should install the R package made for those using Apple silicon chips and not for Intel. For example, if you are trying to install R version 4.3.1, then you should follow the link `R-4.3.1-arm64.pkg` If you are using a Mac that has an Intel chip, then you should choose `R-4.3.1-x86_64.pkg`.

Once you have clicked on the link, you should follow the normal steps you take to install software on your computer. You should not need to make any customizations to the installation (such as installation destination).

2.3.2.2 Setting up `VSCode` with `R`

If you want to use `R` with `VSCode`, getting them to work together is pretty straight forward! You will want to install an extension that enables `VSCode` to recognize `R` code and to know what to do with it when you run your `R` code. While you can install extensions directly in `VSCode`, we will go to https://marketplace.visualstudio.com/VSCode. Once there, we can search "R". What should be the first option is an extension to work with `R` in `VSCode`. We can install that extension by clicking the "Install button".

Once installed, you may not be quite ready to run `R` in `VSCode` just yet. You should follow the steps in the "Getting Started" section of the page that you downloaded the extension from and view the extension's documentation for any extra requirements they may have for you to start running your `R` code in `VSCode`.

2.3.3 Illustration of `OOP` in `R`

Going back to the discussion above about `OOP`, functions, and libraries in `R`. Here is some simple code to help give you an understanding of how `R` works. To follow along, you can open `VSCocde` and then create a new file called `my_first_r_file.R`. Since you are specifying that the file is an `R` file with the extension `.R`, `VSCode` will now expect to only see `R` code in that file.

First, let me make a comment about comments in R! You should always add comments to your code. This helps with the readability and reproducibility of your code. Comments are incredibly useful in that they allow you to explain what your code is doing and can act as really useful documentation.

To make a comment in `R`, you can simply put the hash or pound symbol at the start of any line in your `R` file. Then you can write freely. There are no requirements for what words you use or how you structure your sentences for lines when you've put a comment in front of them. For example:

```
# This is a comment in R
# Any characters that you place after a "#"
# Will not be evaluated as code.
```

Now, let's evaluate my simple expression from above with R:

```
2+2 # add 2 and 2 together
```

```
[1] 4
```

Once we have written this code, we can highlight the code and click the play button (sideways triangle) at the top. This will run the code. Another option is to use the keyboard shortcut "CMD/CTRL + RETURN/ENTER". This will open our R console at the bottom of our screen. The console is responsible for taking the code passed from our R file and actually doing the evaluation of it.

We should see that the output is 4. Great, that is exactly what we should expect.

Now, let's say that we want to take the result from this expression and use it later such as by doing 2+2+2. Rather than typing out 2+2+2, we can take advantage of the fact that R is an OOP language and store the result of our original expression into an object that I will call "result". The <- is called an assignment operator in R and it is used specifically for assigning the result of some expression to an object. Other programming languages that we encounter will use often just use = as the assignment operator.

Once I have created the "result" object, I will then take the object that I called "result" and will add two to it, then store that result into an object called "result_two". I can then see the value stored in "result_two" by using a function called print. We should expect that the result is 6 if everything worked correctly.

```
result <- 2+2 # add 2 and 2 together, store it into an object called result
result_two <- result + 2 # add 2 to result, store it into an object called result_two
print(result_two) # show me the value of result_two
```

```
[1] 6
```

I'm not limited to adding a bunch of 2's together in R. What if, I want to do what I did above but with other numbers and don't want to continuously repeat myself. Well I can write a function and store that function to an object so that I can just reuse the same code over and over again.

So, I am going to write a function that I am going to call "addition." This function will take two inputs that I am going to call `a` and `b`. These inputs can be any two integers that I want. Once I have provided the basic information about my function to `R`, I'll specify what I want the function to do with those inputs in the main block within the curly brackets. As you can see, I want to add `a` and `b` together and store it in a local object called `temp_result`. I will then `return` the the `result`.

```
# define a function called addition
  #* take the parameters a and b
  #* the arguments for a and b should be an integer
addition <- function(a = 1, b = 1) {
  # take the parameters a and b
  # add them together
  temp_result <- a + b
  # return the temp_result object
  # and forget the value for the object once returned
  return(temp_result)
}
```

So once I have defined this function, I can start using it!

```
# use the addition function
# to add 2 and 2 together
# store the result of the function in an object called result
result <- addition(a = 2, b = 2)
# print the value of result
print(result)
# use the addition function
# this time to add 10 and 10 together
# store the result of the function in an object called result
result <- addition(a = 10, b = 10)
# print the value of result
print(result)
```

There are plenty of subtleties in the code here that I won't go over as they are beyond the scope of the book. There are plenty of excellent discussions out there about `OOP` in `R` if you want to delve more into getting comfortable with writing functions in `R`. But, the main idea is that you can see that I can take the `addition` object as a function to reuse the code within the curly brackets to apply it to different inputs.

As discussed before, there are plenty of functions that we might write that could be useful to share with others. To do this, we can take these functions and put them in a `library` and upload it to `CRAN`. Details about how to do this are also outside of the scope of this book, but thankfully there are plenty of great resources on how to do this too!

`R` comes with a `base-r` library of functions that cover bare minimum needs for the users to work with the language. For example, we have already used multiple functions in the code above. For example, `print` took an input from us – our object `result`, did something with it, and then printed out the value of our object. While the way that the `print` function knew to print out the value of `4` rather than to just print out "result" in our console is a whole other discussion about a concept called pointers that are also beyond the scope of the book, the `print` function had some underlying code that knew that `result` was an object and we wanted to print out the value that `result` represented.

So, how can I access a library that someone else has written? It isn't too bad! There is a function that comes standard with `R`, in the `base-r` library, that allows us to access libraries uploaded to `CRAN`: `install.packages()`. With `install.packages()` all that we have to do is specify the name of the library we want to install, then when we can use those functions after loading them from the library that we've downloaded.[2]

> **How do I know the name of a library or whether a function exists?**
>
> There is no one way that this happens. I often stumble across useful libraries and functions within those libraries in different places of the internet, from talking with others, etc. As you go along, you'll begin to get more comfortable

[2] Functions from the `base-r` library automatically loads at the start of each `R` session. Functions from the libraries that do not come standard, the ones we had to install, require that we load the library at the start of an `R` session. Requiring that we load libraries at the start of each session ensures that we aren't loading a lot of junk libraries that we aren't using for our particular project but may have used for a different project. This keeps our `RAM` clearer and keeps our code more efficient.

finding out these things yourself.

2.3.4 `Python`

`Python` is perhaps one of the most popular coding languages in the world. Unlike `R`, `Python` is used for more than just quantitative analysis. Beyond scientific reporting, it is used heavily in machine learning, artificial intelligence, and is also used relatively frequently in web application design and software engineering.

It is less popular than `R` among academics in the social sciences primarily due to the fewer and less robust packages for reporting scientific quantitative analyses. This is beginning to change, however. As there has been an increased growth of the computational social science community which moves past regression-based quantitative analyses to examine text and images with machine learning approaches, the rich support for these tasks in `Python` have lead to more social scientists to use `Python`. This has also lead to more social scientists taking it upon themselves to work on libraries that are designed to help with the reporting and visualization of scientific results.

`Python` is certainly growing in popularity outside academia but also within academia. It is a language that can certainly be helpful for those who may want to pursue careers outside of academia after completing their education or who would like to use things like machine learning in their research.

Another attractive feature of `Python` is that it often has fewer concerns with efficiency, readability, and reproducibility than `R`, overall. `Python` was originally conceived as a language designed to make software engineering more accessible and easier to perform. Since the bulk of its users are software engineers, data engineers, or data scientists rather than academics, many of the libraries are designed with efficiency, readability, and reproducibility as central tenants of the package's design. This is one reason why many places outside of academia often indicate that they prefer a team using `Python` for their data analysis even though that `R` is much more narrow in its scope relative to `Python`.

So, `Python` can be useful for increasing the efficiency, readability, and reproducibility of your code relative to `R`. It is a language with libraries that let you do more than just quantitative analysis which can be appealing to not learn a tool to complete only a super narrow set of tasks. It is one of the most preferred languages for those doing computational social science due to robust packages for machine learning

and artificial intelligence. It is also a language that is in high-demand for many sectors and job titles outside of academia. However, it is less than ideal for scientific statistical computing and has fewer libraries than `R` for such tasks.

2.3.4.1 Installation

The installation of `Python` is relatively straightforward. You will want to follow this link https://www.python.org/downloads/.

Once on the webpage to download the latest version of `Python` (at the time of writing this was `Python 3.11.4`). You can just click the yellow button to download it. Python should automatically detect the operating system that you are using and will download the proper installation file. If not, there is a link just below the yellow button that lets you choose which operating system you are using.

Once you have downloaded the installation file, you should do a default install and follow the normal installation steps you take for an application. You should not need to make any customizations to the installation (such as installation destination).

2.3.4.2 Setting up `VSCode` with `Python`

If you want to use `Python` with `VSCode`, getting them to work together is probably even easier than it is with `R`. You can go to https://marketplace.visualstudio.com/VSCode. Once there, you can search "Python". The first option should be an extension called "Python" written by Microsoft. We can install that extension by clicking the "Install" button.

Once installed, you may not be quite ready to run `Python` in `VSCode` just yet. You should follow the steps in the "Getting Started" section of the page that you downloaded the extension from and view the extension's documentation for any extra requirements they may have for you to start running your `Python` code in `VSCode`.

2.3.4.3 Illustration of OOP in `Python`

Like `R`, `Python` is also a `OOP` language. Here is some simple code to give you an understanding of how `Python` works. To follow along, you can open `VSCode` and then create a new file called `my_first_python_file.py`. Since you are specifying

that the file is a Python file with the extension .py, VSCode will now expect to only see Python code in that file.

Comments are the same in Python as they are in R. In any language, you should always add comments to your code to increase the readability and reproducibility of it.

The symbol for a comment in Python is the same as in R. You can simply put the hash or pound symbol at the start of any line in your Python file. Then you can write freely. For example:

```
# This is a comment in Python
# Any characters that you place after a "#"
# Will not be evaluated as code
```

Now lets go back to our evaluation of the expression of 2+2 as we did in R:

```
2+2 # add 2 and 2 together
```

Here you'll notice the code looks exactly the same as it did in R. Once we are doing more complicated tasks, the languages will certainly begin to look a lot different.

Like in R when we want to evaluate this Python code, we can highlight the code and click the play button (sideways triangle) at the top of our VSCode window. This will run the highlighted code. Like it did with our R code, we should see a console open at the bottom of our VSCode window. Instead of it being an R console, it will now be a Python window even though the code looked exactly the same between the two. VSCode knew that you were running Python code because the code you were running should have been in a .py file instead of a .R file.

Like we did before, say we want to store the result of our expression 2+2 into an object that we can use later. Since Python is also a OOP language, we can do this. Unlike R, the assignment operator in Python is = instead of <-. So we will take our expression and will assign the result to an object called "result".

Once I have created the "result" object, I can use that object to add 2 to it and store that result in a object called "result_two" like I did in R. To see the value of the "result_two" object I can use a print() function.

```
# add 2 and 2 together, store it in an object called result
result = 2 + 2
# add 2 to result, store it in an object called result_two
result_two = result + 2
# show me the value of result_two
print(result_two)
```

```
6
```

Like I did with `R`, let's say that I want to do addition a few times but I do not want to have to keep writing `a + b` over and over again. So I decide to write a function so I can reuse that code over and over.

I'll write a function that I am going to call "addition." This function will take two inputs that I will call `a` and `b`. Again, these inputs can take any integers that I want. Once I have provided the basic information about my function to `Python`, I'll specify what I want `Python` to do with those inputs in the main block of the function after the colon and when indented by 4 spaces.

```
# define a function called addition
    #* take the parameters a and b
    #* take the arguments for a and b, should be an integer
def addition(a = 1, b = 1):
    # take the parameters a and b
    # add them together
    temp_result = a + b
    # return the temp_result object
    # and forget the value for the object once returned
    return temp_result
```

> **Notice the difference**
>
> Unlike `R`, the main block of code for my function in `Python` is not within curly brackets. Instead, it is indented code. `Python` is extremely sensitive to the indentation in a way that `R` is much more flexible. In `Python` you should use 4 spaces for an indentation – do not use the `tab` key as this can sometimes be an inconsistent number of spaces and may lead `Python` to throw you an error

> about improper indentation.
> How `Python` knows that the code for the function for the function is completed is once it sees a line that does not indent by four spaces. Once it notices that, it will go to the last consecutive line with four spaces and will consider the function's main block of code to end on that line.

Now that I have defined the function, I can use it!

```
# use the addition function
# to add 2 and 2 together
# store the result of the function in an object called result
result = addition(a = 2, b = 2)
# print the value of the result
print(result)
# use the addition function
# this time to add 10 and 10 together
# store the result of the function in an object called result
result = addition(a = 10, b = 10)
# print the value of result
print(result)
```

As `Python` is also an `OOP` language, I am capable of taking the code within the addition function that is stored as an object called "addition" and am able to reapply that code within the function to different inputs.

Like in `R`, I am able to use various functions that come with the standard installation of `Python`. I will want to install libraries containing nifty functions as my code becomes more complex, though. In the next chapter, I will provide concrete examples of how to install and work with some common libraries for data management.

2.3.5 `Julia`

Like `R`, `Julia` is specifically designed for scientific statistical computing. It is a much newer language than `R`, however. `Julia` has started to gain some traction in some areas of machine learning engineering, some circles of data science and in academia. It still remains relatively uncommon, however. The main reason that it remains relatively uncommon is how new it is. It is also much more dedicated to statistical

computing than the other languages which makes it relatively narrow in the tasks that it can complete compared to `Python` and even `R`.

The appeal of `Julia` is that it is an extremely efficient and reproducible language. The language is not as bogged down as languages like `R` and `Python` which reduces a lot of problems. It also has built in support for virtual environments so that one does not have to jump through as many hoops to install, manage, and document the libraries that someone uses for their project.

Julia also has some really interesting features such as allowing you to use scientific and greek symbols as object names. `Python` and `R` does not allow you to do this. While this might be a small feature for some, others it may be super useful. So there is that.

In a time where people are wanting to publish papers faster, are using more data, and are dealing with concerns about how reproducible their results are, `Julia` certainly seems like a language that could eventually become extremely popular in academia. It is already gaining some success in many sectors of industry as a language that reduces the amount of time that analysts are spending to produce reports.

2.3.5.1 Installation

Thankfully, the installation of `Julia` is also quite straightforward. You will want to follow this link: https://julialang.org/downloads/#download_julia.

Once on the webpage to download the latest version of `Julia` (at the time of writing this was ‘Julia 1.9.2). If you are on Windows, you should follow the link to install the Windows 64-bit installer. If you are on a Mac with a non-intel processor, you should follow the link to install the macOS (Apple Silicon) 64-bit (.dmg). If you are on a Mac with a intel processor, then you should follow the link to install the macOS x86 (Intel or Rosetta) 64-bit (.dmg). If you are on linux, you should follow the most appropriate option for your setup.

Once you have downloaded the installation file, you should do a default install and follow the normal installation steps you take for an application. You should not need to make any customizations to the installation (such as installation destination).

2.3.5.2 Setting up VSCode with `Julia`

Like the other languages, working with `Julia` in `VSCode` is not too hard to setup. You can go to https://marketplace.visualstudio.com/VSCode. Once there, you can search "Julia". The first option should be an extension called "Julia". You can install the extension by clicking the "Install button".

Once installed, you may not be quite ready to run `Julia` in `VSCode` just yet. You should follow the steps in the "Getting Started" section of the page that you downloaded the extension form and view the extension's documentation for any extra requirements they may have for you to start running your `Julia` code in `VSCode`.

2.3.5.3 Illustration of `OOP` in `Julia`

Like the other languages so far, `Julia` is also an `OOP` language. To follow along with my examples below, you can open `VSCode` and then create a new file called `my_first_julia_file.jl`. You are specifying the file as being a `Julia` file with the extension `.jl`.

Like the other languages, you can add a comment to your file using the hash or pound symbol. Any characters that follow will not be evaluated by `Julia`. For example

```
# This is a comment in Julia
# Any characters that you place after a "#"
# Will not be evaluated as code
```

Now let's evaluate the expression `2+2` in `Julia`:

```
2+2 # add 2 and 2 together
```

Like in the other languages, the code looks exactly the same. But once we start to do more complicated tasks, we will start to notice the differences in the syntax of the languages.

Like in the other languages, if we want to evaluate this `Julia` code, we can highlight the code and click the play button (sideways triangle) at the top of our `VSCode` window. This will run the highlighted code.

Unlike the other coding languages, instead of opening a console specific to that language, `Julia` will open an instance of the `Julia REPL`. It is not necessary to understand the differences between the two, but how you interact with them will look slightly differently. The reason for these differences is that the goal of `Julia` is for the code to be interactively run as opposed to run as a whole file like is the common expectation with `Python` and to a lesser extent `R`. If you get an error from `VSCode` telling you that the `REPL` is in a different directory than the file, then you may want to go to the top of your `VSCode` window, click on "File", then click on "Open Folder", then choose the folder that you stored your `my_first_julia_file.jl` in. This then should solve the issue and the code should successfully run.

Like we did with the other languages, say that we want to store the result of this into an object. We can do that. Like `Python`, the assignment operator in `Julia` is `=`. I will evaluate the expression `2+2` but store it in an object called "result."

Once I have created the result object, I can use that object to add 2 to it and store that result in an object called "result_two" like I did in the other examples. Unlike the other coding languages, `Julia` should automatically print the value stored in my "result_two" object. If I wanted, I could also use a native `print()` function to print the value for "result_two".

```
# add 2 and 2 together, store it in an object called result
result = 2 + 2
# add 2 to result, store it in an object called result_two
result_two = result + 2
# show me the value of result_two
print(result_two)
```

```
4
```

```
6
```

References