R 4 Epidemiology

2024-07-23

Table of contents

Welcome

Welcome to R for Epidemiology!

This electronic textbook was originally created to accompany the Introduction to R Programming for Epidemiologic Research course at the University of Texas Health Science Center School of Public Health. However, we hope it will be useful to anyone who is interested in R, epidemiology, or human health and well-being.

Acknowledgements

This book is currently a work in progress (and probably always will be); however, there are already many people who have played an important role (some unknowingly) in helping develop it thus far. First, we'd like to offer our gratitude to all past, current, and future members of the R Core Team for maintaining this *amazing*, *free* software. We'd also like to express our gratitude to everyone at Posit. You are also developing and *giving away* some amazing software. In particular, we'd like to acknowledge Garrett Grolemund and Hadley Wickham. Both have had a huge impact on how we use and teach R. We'd also like to thank our students for all the feedback they've given us while taking our courses. In particular, we want to thank Jared Wiegand and Yiqun Wang for their many edits and suggestions.

This electronic textbook was created and published using R, RStudio, the Quarto, and GitHub.

Introduction

Goals

We're going to start the introduction by writing down some basic goals that underlie the construction and content of this book. We're writing this for you, the reader, but also to hold ourselves accountable as we write. So, feel free to read if you are interested or skip ahead if you aren't.

The goals of this book are:

1. **To teach you how to use R and RStudio as tools for applied epidemiology**.[1] Our goal is not to teach you to be a computer scientist or an advanced R programmer. Therefore, some readers who are experienced programmers may catch some technical inaccuracies regarding what we consider to be the fine points of what R is doing "under the hood."

2. **To make this writing as accessible and practically useful as possible without stripping out all of the complexity that makes doing epidemiology in real life a challenge**. In other words, We're going to try to give you all the tools you need to *do* epidemiology in "real world" conditions (as opposed to ideal conditions) without providing a whole bunch of extraneous (often theoretical) stuff that detracts from *doing*. Having said that, we will strive to add links to the other (often theoretical) stuff for readers who are interested.

3. **To teach you to accomplish common *tasks***, rather than teach you to use functions or families of functions. In many R courses and texts, there is a focus on learning all the things a function, or set of related functions, can do. It's then up to you, the reader, to sift through all of these capabilities and decided which, if any, of the things that *can* be done will accomplish the tasks that you are *actually trying* to accomplish. Instead, we will strive to start with the end in mind. What is the task we are actually trying to accomplish? What are some functions/methods we could use to accomplish that task? What are the strengths and limitations of each?

[1] In this case, "tools for applied epidemiology" means (1) understanding epidemiologic concepts; and (2) completing and interpreting epidemiologic analyses.

4. **To start each concept by showing you the end result** and then deconstruct how we arrived at that result, where possible. We find that it is easier for many people to understand new concepts when learning them as a component of a final product.
5. **To learn concepts with data** instead of (or alongside) mathematical formulas and text descriptions, where possible. We find that it is easier for many people to understand new concepts by seeing them in action.

Text conventions used in this book

- We will hyperlink many keywords or phrases to their glossary entry.
- Additionally, we may use **bold** face for a word or phrase that we want to call attention to, but it is not necessarily a keyword or phrase that we want to define in the glossary.
- `Highlighted inline code` is used to emphasize small sections of R code and program elements such as variable or function names.

Other reading

If you are interested in R4Epi, you may also be interested in:

- Hands-on Programming with R by Garrett Grolemund. This book is designed to provide a friendly introduction to the R language.
- R for Data Science by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund. This book is designed to teach readers how to do data science with R.
- Statistical Inference via Data Science: A ModernDive into R and the Tidyverse. This book is designed to be a gentle introduction to the practice of analyzing data and answering questions using data the way data scientists, statisticians, data journalists, and other researchers would.
- Reproducable Research with R and RStudio by Christopher Gandrud. This book gives you tools for data gathering, analysis, and presentation of results so that you can create dynamic and highly reproducible research.
- Advanced R by Hadley Wickham. This book is designed primarily for R users who want to improve their programming skills and understanding of the language.

Contributing

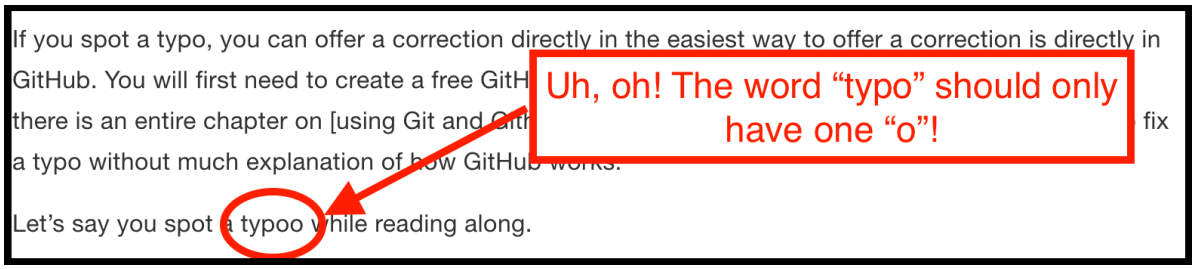
Over the years, we have learned so much from our students and colleagues, and we anticipate that there is much more we can learn from you – our readers. Therefore, we welcome and appreciate all constructive contributions to R4Epi!

Typos

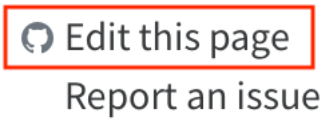
The easiest way for you to contribute is to help us clean up the little typos and grammatical errors that inevitably sneak into the text.

If you spot a typo, you can offer a correction directly in GitHub. You will first need to create a free GitHub account: sign-up at github.com. Later in the book, we will cover using GitHub in greater depth (See Chapter 12). Here, we're just going to walk you through how to fix a typo without much explanation of how GitHub works.

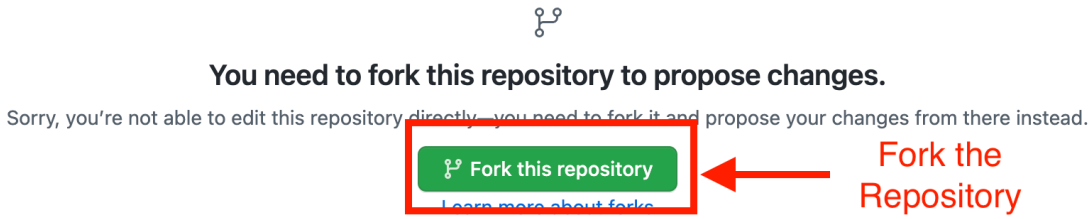
Let's say you spot a typo while reading along.



Next, click the edit button in the toolbar as shown in the screenshot below.



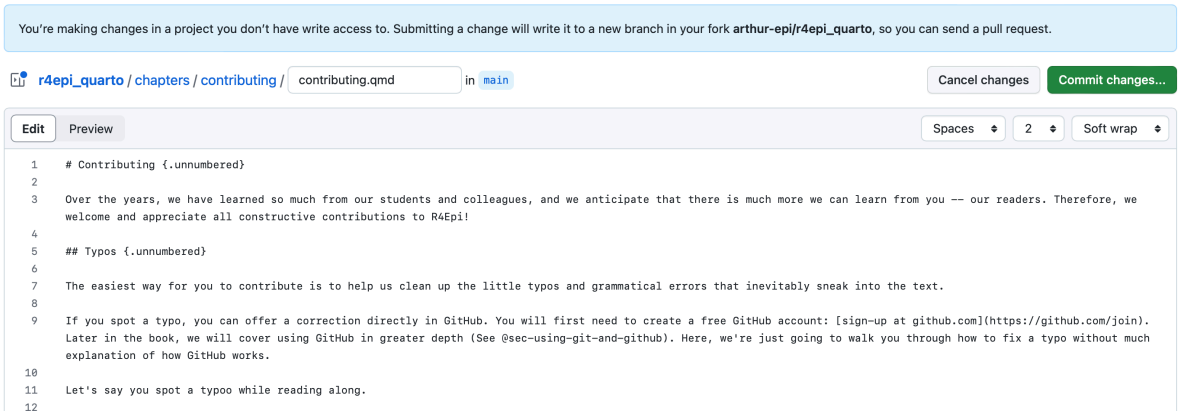
The first time you click the icon, you will be taken to the R4Epi repository on GitHub and asked to fork it. For our purposes, you can think of a GitHub repository as being similar to a shared folder on Dropbox or Google Drive.



"Forking the repository" basically just means "make a copy of the repository" on your GitHub account. In other words, copy all of the files that make up the R4Epi textbook to your GitHub account. Then, you can fix the typos you found in your *copy* of the files that make up the book instead of directly editing the *actual* files that make up the book. This is a safeguard to prevent people from accidentally making changes that shouldn't be made.

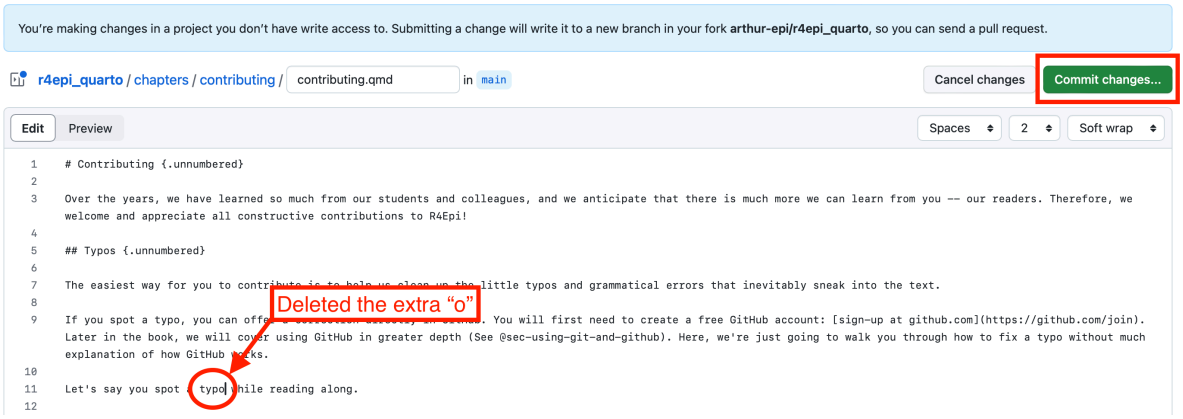
> **Note**
>
> Forking the R4Epi repository does not cost any money or add any files to your computer.

After you fork the repository, you will see a text editor on your screen.

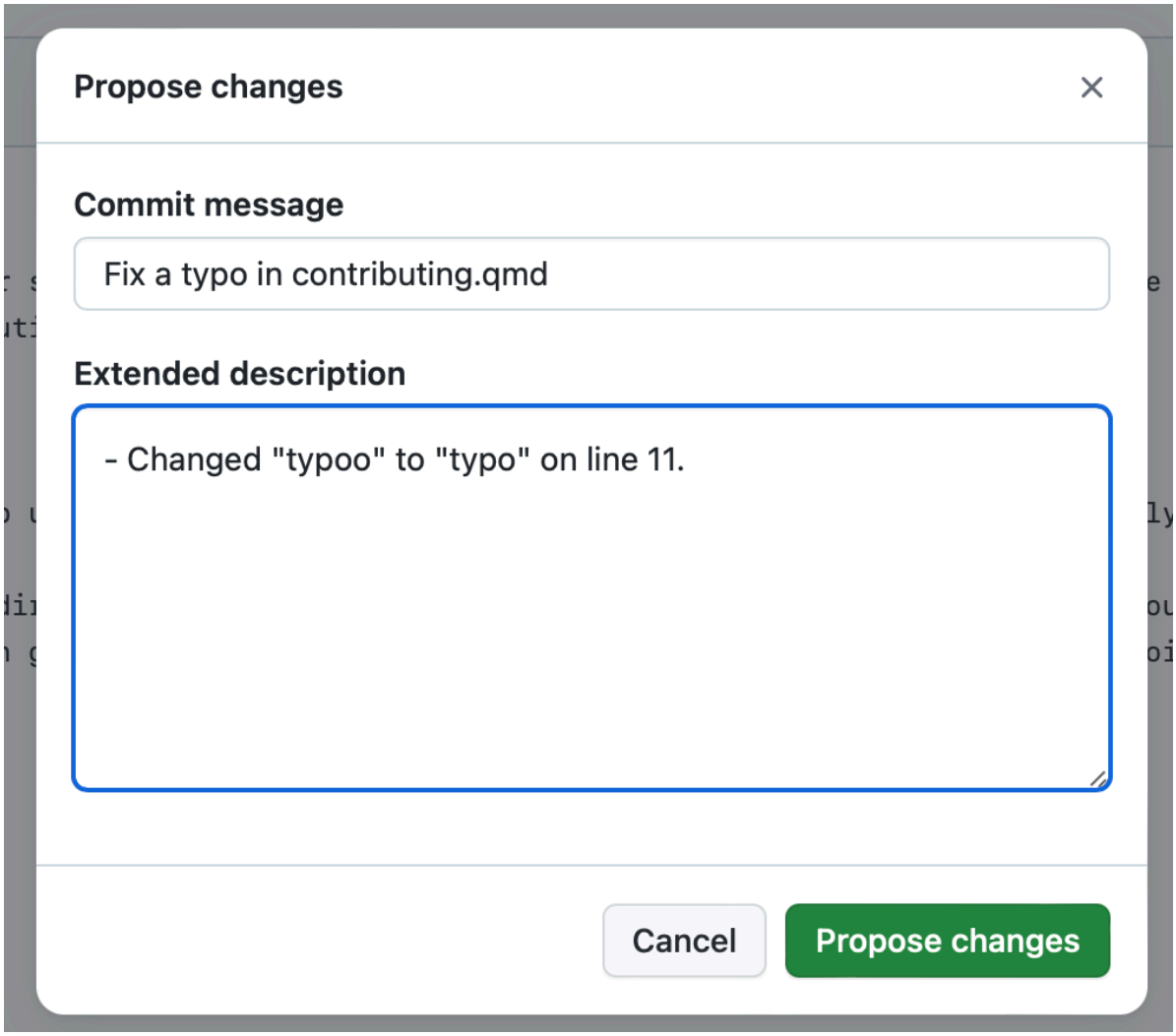


The text editor will display the contents of the file used to make the chapter you were looking at when you clicked the `edit` button. In this example, it was a file named `contributing.qmd`. The `.qmd` file extension means that the file is a Quarto file. We will learn more about Quarto files in **?@sec-quarto-files**, but for now just know that Quarto files can be used to create web pages and other documents that contain a mix of R code, text, and images.

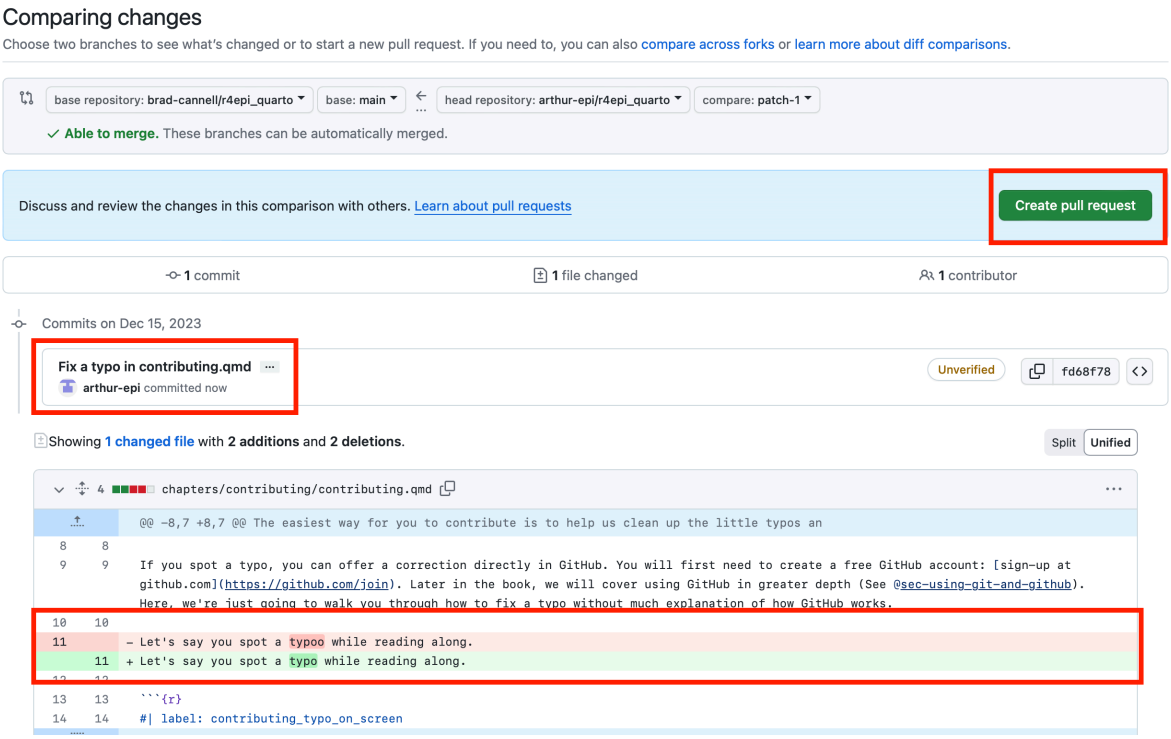
Next, scroll down through the text until you find the typo and fix it. In this case, line 11 contains the word "typoo". To fix it, you just need to click in the editor window and begin typing. In this case, you would click next to the word "typoo" and delete the second "o".



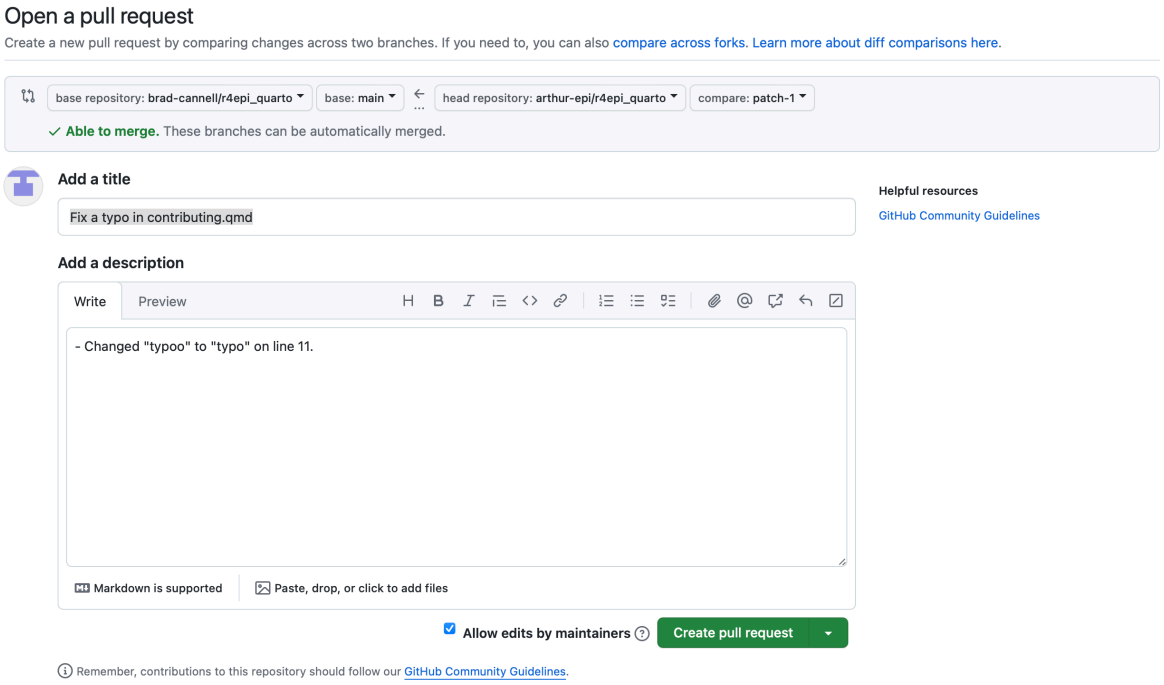
Now, the only thing left to do is propose your typo fix to the authors. To do so, click the green `Commit changes...` button on the right side of the screen above the text editor (surrounded with a red box in the screenshot above). When you click it, a new `Propose changes` box will appear on your screen. Type a brief (i.e., 72 characters or less) summary of the change you made in the `Commit message` box. There is also an `Extended description` box where you can add a more detailed description of what you did. In the screenshot below, shows an example commit message and extended description that will make it easy for the author to quickly figure out exactly what changes are being proposed.



Next, click the `Propose changes` button. That will take you to another screen where you will be able to create a pull request. This screen is kind of busy, but try not to let it overwhelm you.

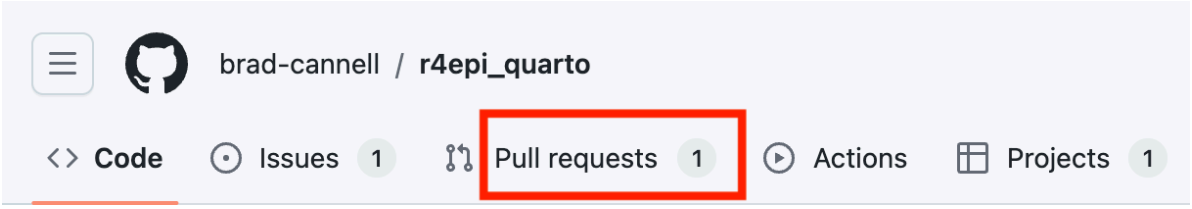


For now, we will focus on the three different sections of the screen that are highlighted with a red outline. We will start at the bottom and work our way up. The red box that is closest to the bottom of the screenshot shows us that the change that made was on line 11. The word "typoo" (highlighted in red) was replaced with the word "typo" (highlighted in green). The red box in the middle of the screenshot shows us the brief description that was written for our proposed change – "Fix a typo in contributing.qmd". Finally, the red box closest to the top of the screenshot is surrounding the `Create pull request` button. You will click it to move on with your pull request.

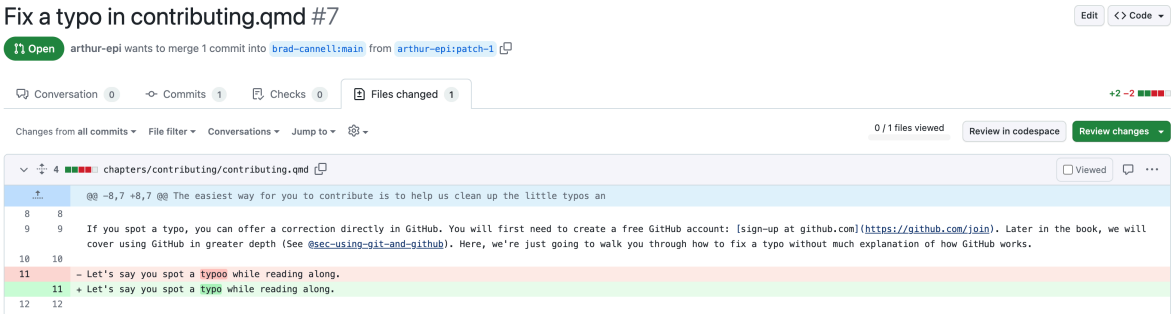


After doing so, you will get one final chance to amend the description of your proposed changes. If you are happy with the commit message and description, then click the `Create pull request` button one more time. At this point, your job is done! It is now up to the authors to review the changes you've proposed and "pull" them into the file in their repository.

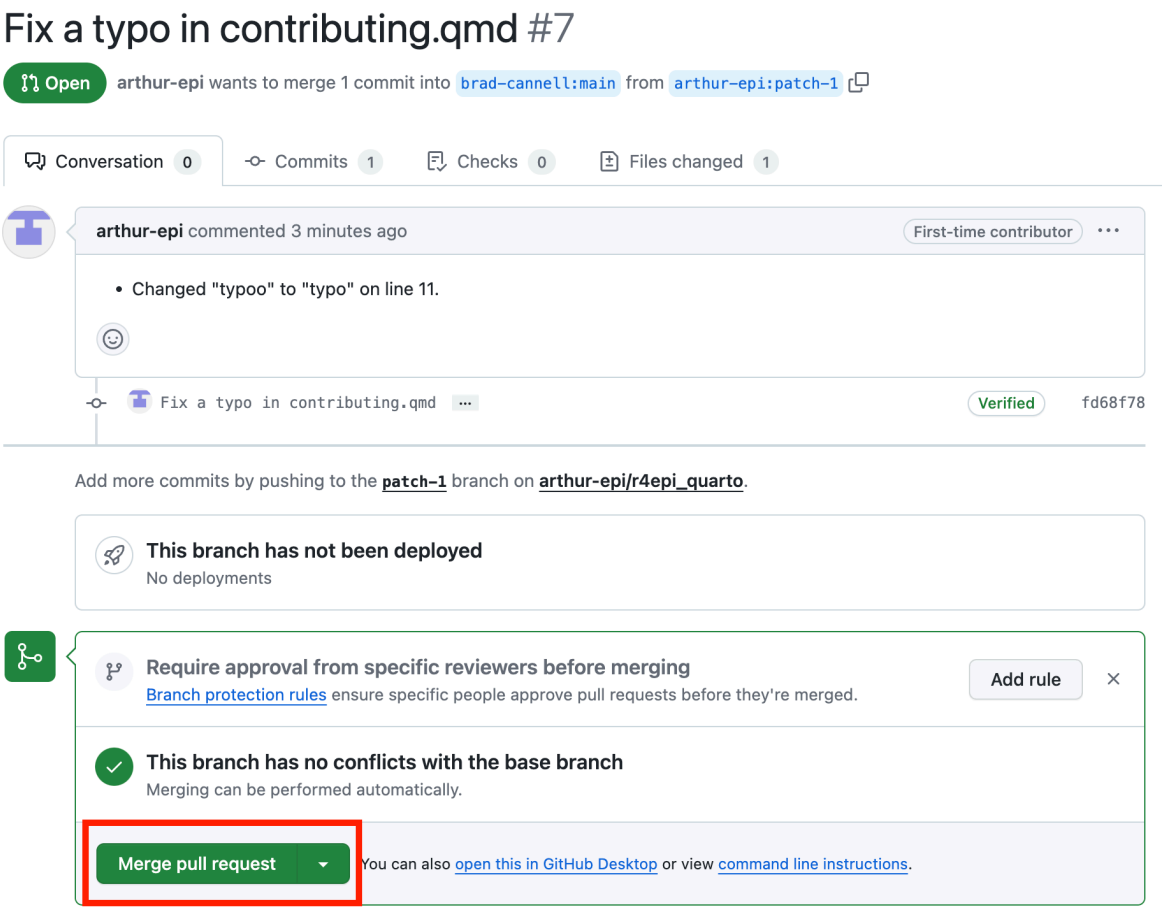
In case you are curious, here is what the process looks like on the authors' end. First, when we open the R4Epi repository page on GitHub, we will see that there is a new pull request.



When we open the pull request, we can see the proposed changes to the file.

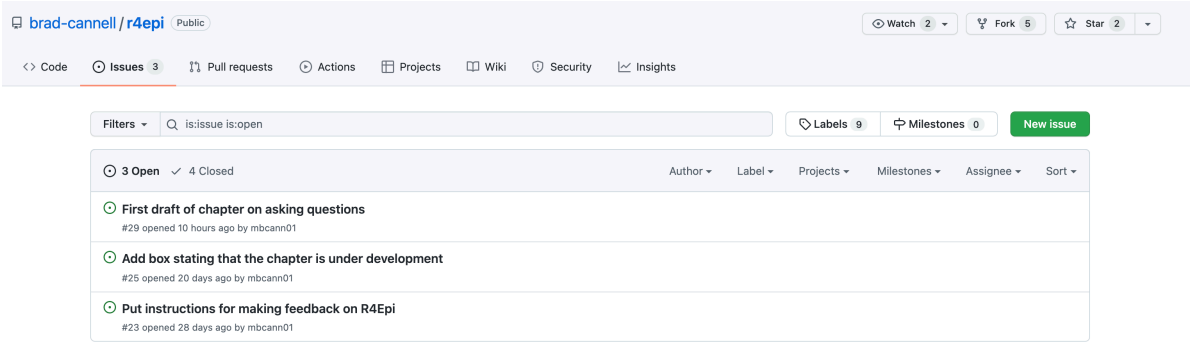


Then, all we have to do is click the `Merge pull request button` and the fixed file is "pulled in" to replace the file with the typo.



Issues

There may be times when you see a problem that you don't know how to fix, but you still want to make the authors aware of. In that case, you can create an issue in the R4Epi repository. To do so, navigate to the issue tracker using this link: https://github.com/brad-cannell/r4epi/issues.



Once there, you can check to see if someone has already raised the issue you are concerned about. If not, you can click the green "New issue" button to raise it yourself.

Please note that R4Epi uses a Contributor Code of Conduct. By contributing to this book, you agree to abide by its terms.

License Information

About the Authors

Brad Cannell

Michael (Brad) Cannell, PhD, MPH
Associate Professor
Elder Mistreatment Lead, UTHealth Institute of Aging
Director, Research Informatics Core, Cizik Nursing Research Institute
UTHealth Houston
McGovern Medical School
Joan and Stanford Alexander Division of Geriatric & Palliative Medicine
www.bradcannell.com

Dr. Cannell received his PhD in Epidemiology, and Graduate Certificate in Gerontology, in 2013 from the University of Florida. He received his MPH with a concentration in Epidemiology from the University of Louisville in 2009, and his BA in Political Science and Marketing from the University of North Texas in 2005. During his doctoral studies, he was a Graduate Research Assistant for the Florida Office on Disability and Health, an affiliated scholar with the Claude D. Pepper Older Americans Independence Center, and a student-inducted member of the Delta Omega Honorary Society in Public Health. In 2016, Dr. Cannell received a Graduate Certificate in Predictive Analytics from the University of Maryland University College, and a Certificate in Big Data and Social Analytics from the Massachusetts Institute of Technology.

He previously held professional staff positions in the Louisville Metro Health Department and the Northern Kentucky Independent District Health Department. He spent three years as a project epidemiologist for the Florida Office on Disability and Health at the University of Florida. He also served as an Environmental Science Officer in the United States Army Reserves from 2009 to 2013.

Dr. Cannell's research is broadly focused on healthy aging and health-related quality of life. Specifically, he has published research focusing on preservation of physical and cognitive function, living and aging with disability, and understanding and preventing elder mistreatment. Additionally, he has a strong background and training in epidemiologic methods and predictive analytics. He has been principal or co-investigator on multiple trials and observational studies in community and healthcare settings. He is currently the principal investigator on multiple data-driven federally funded projects that utilize technological solutions to public health issues in novel ways.

Contact
Connect with Dr. Cannell and follow his work.

Melvin Livingston

Melvin (Doug) Livingston, PhD
Research Associate Professor
Department of Behavioral, Social, and Health Education Sciences
Emory University Woodruff Health Sciences Center
Rollins School of Public Health
Dr. Livingston's Faculty Profile

Dr. Livingston is a methodologist with expertise in the the application of quasi-experimental design principals to the evaluation for both community interventions and state policies. He has particular expertise in time series modeling, mixed effects modeling, econometric methods, and power analysis. As part of his work involving community trials, he has been the statistician on the long term follow-up study of a school based cluster randomized trial in low-income communities with a focus on explaining the etiology of risky alcohol, drug, and sexual behaviors. Additionally, he was the statistician for a longitudinal study examining the etiology of alcohol use among racially diverse and economically disadvantaged urban youth, and co-investigator for a NIAAA- and NIDA-funded trial to prevent alcohol use and alcohol-related problems among youth living in high-risk, low-income communities within the Cherokee Nation. Prevention work at the community level led him to an interest in the impact of state and federal socioeconomic policies on health outcomes. He is a Co-Investigator of a 50-state, 30-year study of effects of state-level economic and education policies on a diverse set of public health outcomes, explicitly examining differential effects across disadvantaged subgroups of the population.

His current research interests center around the application of quasi-experimental design and econometric methods to the evaluation of the health effects of state and federal policy.

Contact
Connect with Dr. Livingston and follow his work.

Part I

Getting Started

1 Installing R and RStudio

Before we can do any programming with R, we first have to download it to our computer. Fortunately, R is free, easy to install, and runs on all major operating systems (i.e., Mac and Windows). However, R is even easier to use as when we combine it with another program called RStudio. Fortunately, RStudio is also free and will also run on all major operating systems.

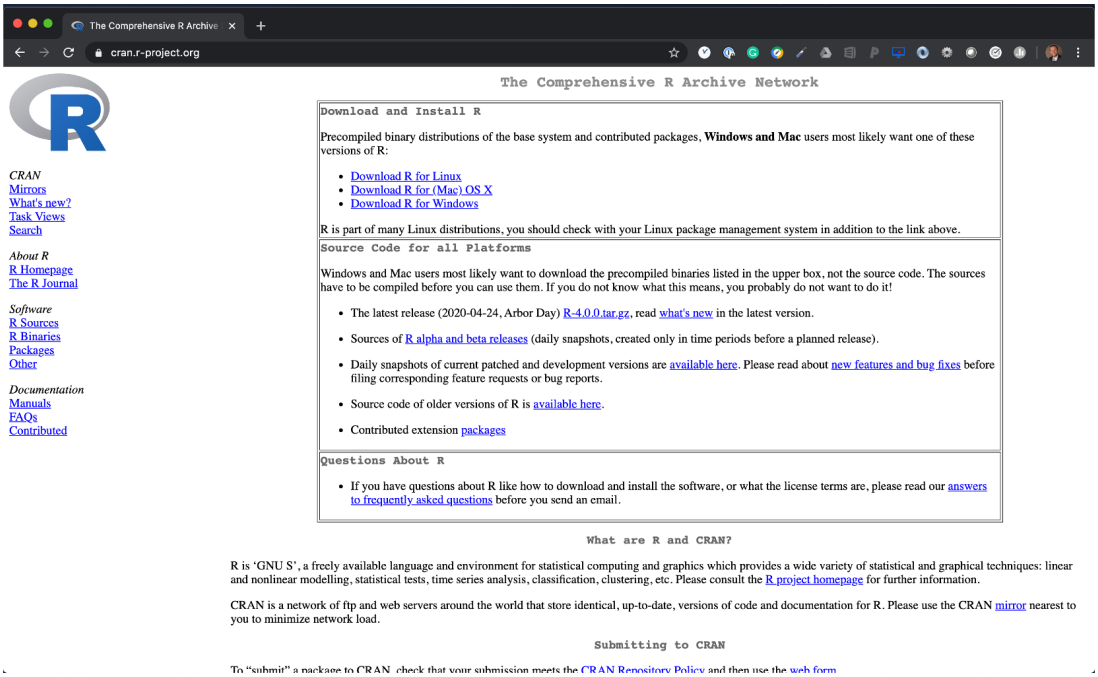
At this point, you may be wondering what R is, what RStudio is, and how they are related. We will answer those questions in the near future. However, in the interest of keeping things brief and simple, We're not going to get into them right now. Instead, all you have to worry about is getting the R programming language and the RStudio IDE (IDE is short for integrated development environment) downloaded and installed on your computer. The steps involved are slightly different depending on whether you are using a Mac or a PC (i.e., Windows). Therefore, please feel free to use the table of contents on the right-hand side of the screen to navigate directly to the instructions that you need for your computer.

> **Note**
>
> In this chapter, we cover how to download and install R and RStudio on both Mac and PC. However, the screenshots in all following chapters will be from a Mac. The good news is that RStudio operates almost identically on Mac and PC.

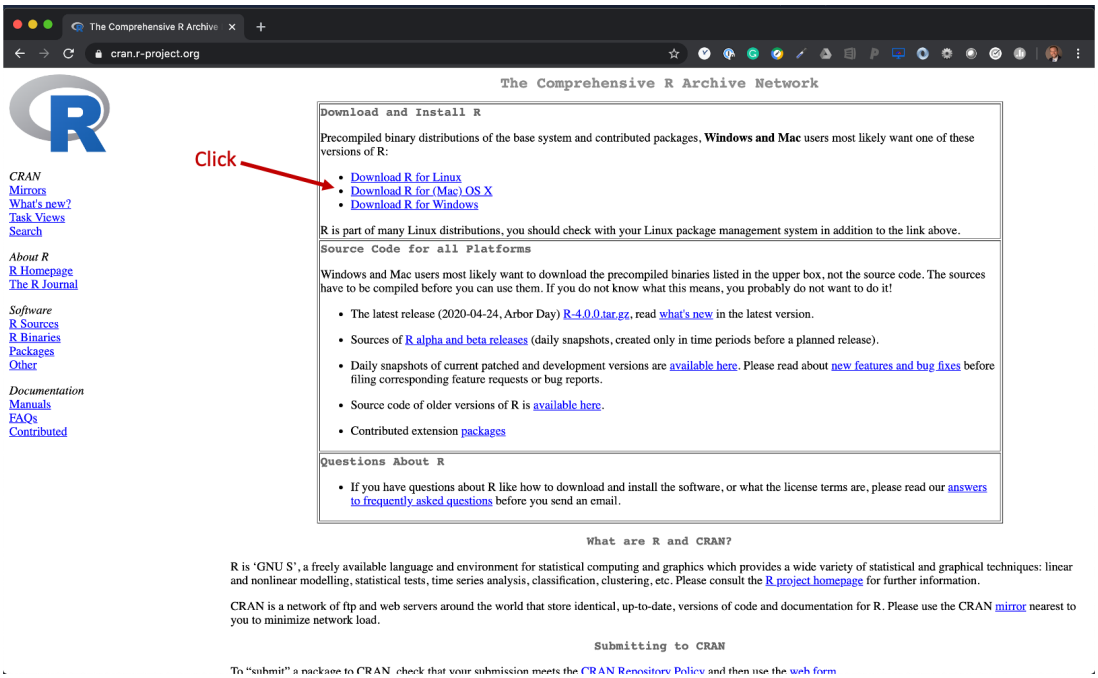
Step 1: Regardless of which operating system you are using, please make sure your computer is on, properly functioning, connected to the internet, and has enough space on your hard drive to save R and RStudio.

1.1 Download and install on a Mac

Step 2: Navigate to the Comprehensive R Archive Network (CRAN), which is located at https://cran.r-project.org/.

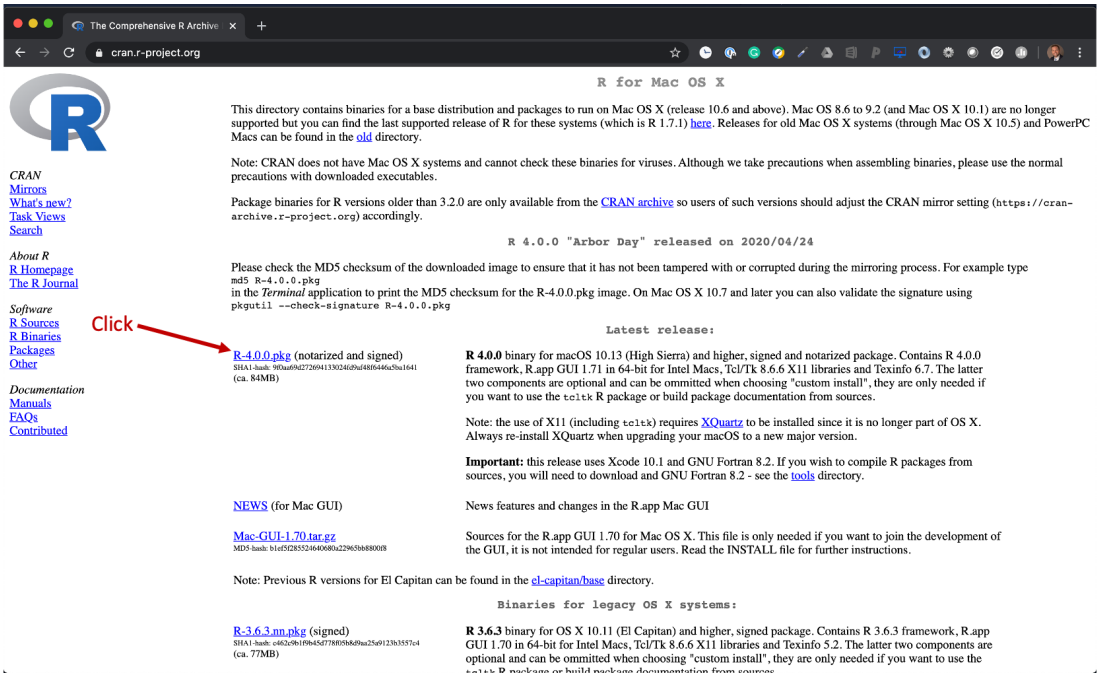


Step 3: Click on Download R for macOS.

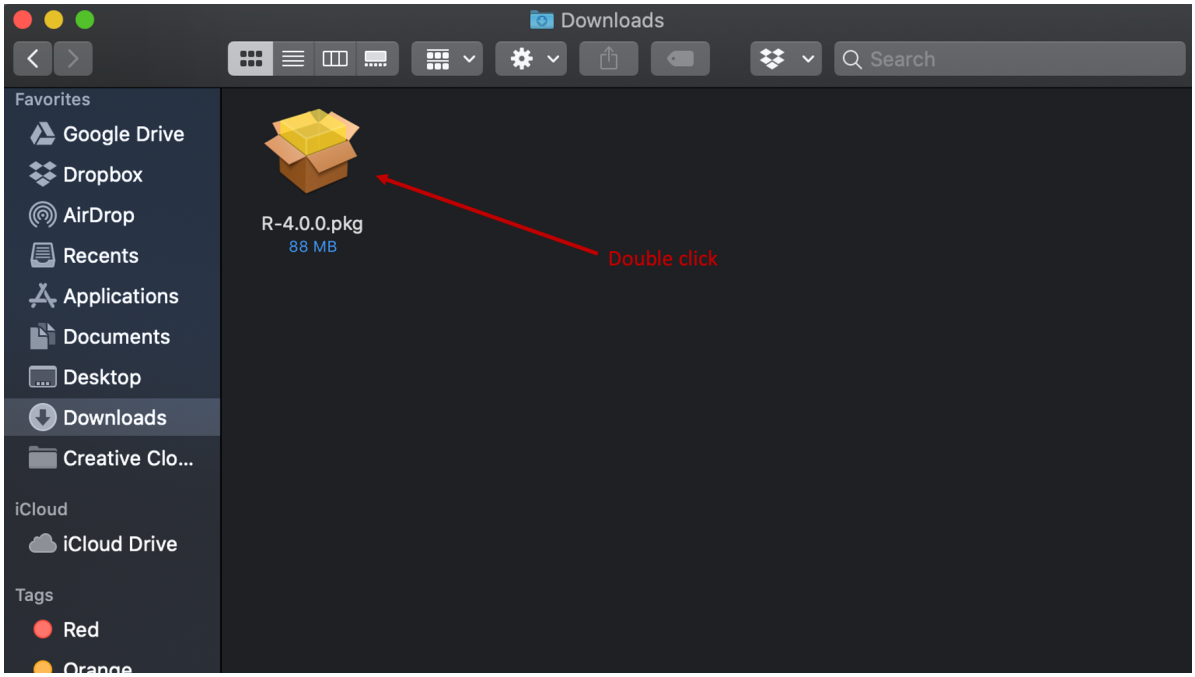


Step 4: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version

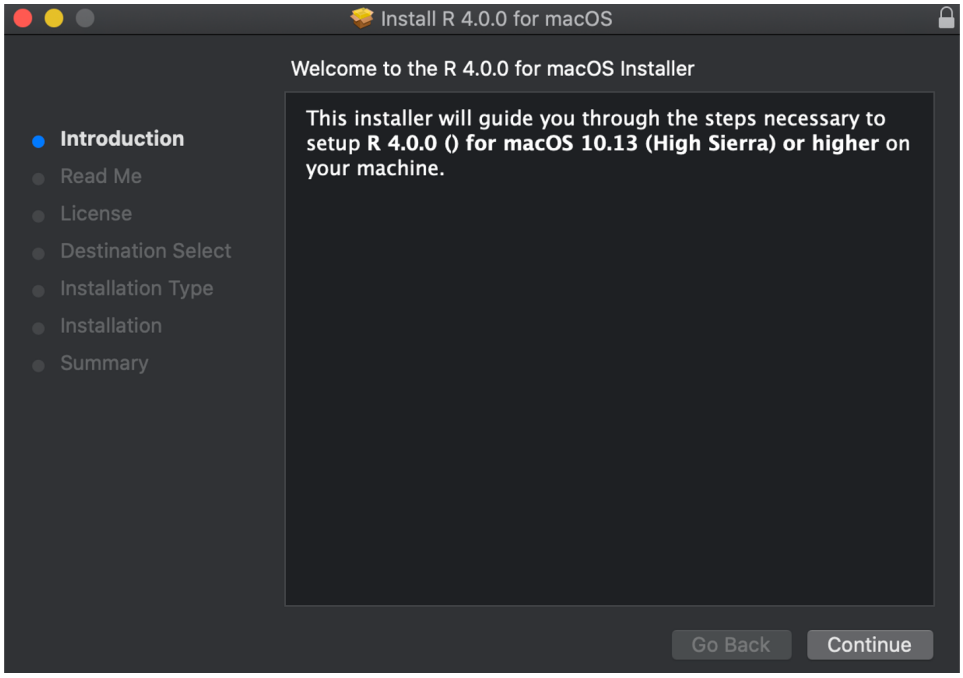
should be roughly in the same place – the middle of the screen under "Latest release:". After clicking the link, R should start to download to your computer automatically.



Step 5: Locate the package file you just downloaded and double click it. Unless you've changed your download settings, this file will probably be in your "downloads" folder. That is the default location for most web browsers. After you locate the file, just double click it.

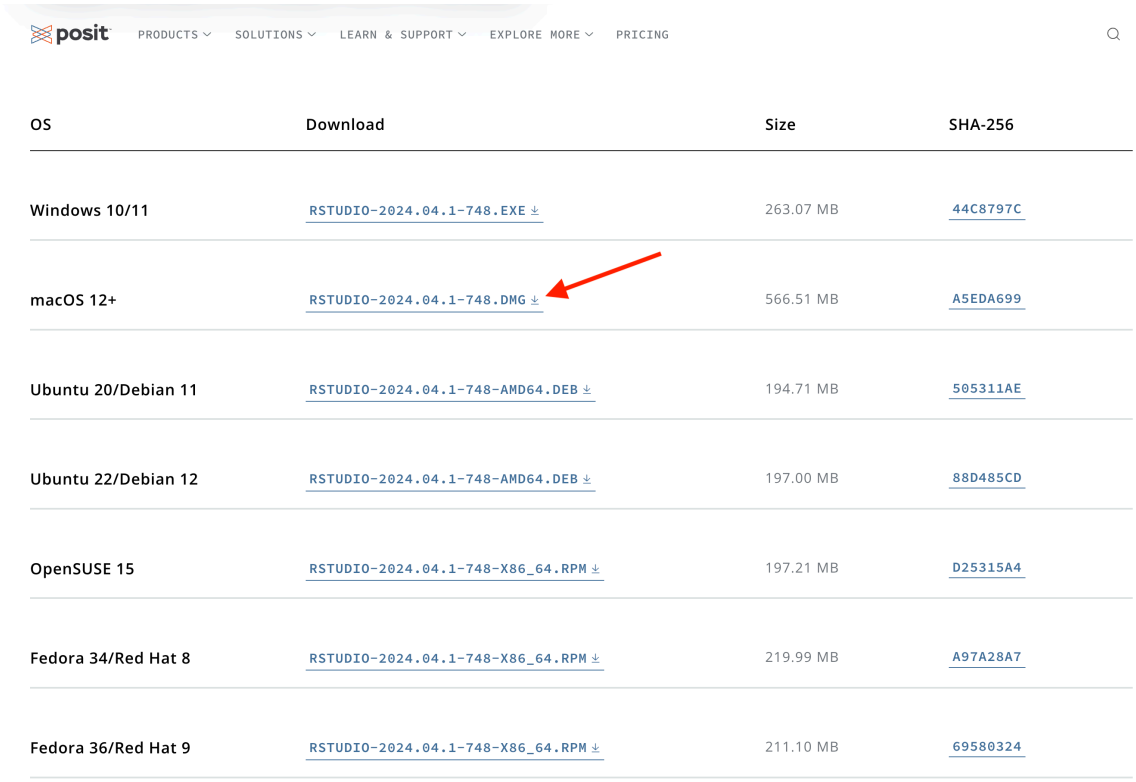


Step 6: A dialogue box will open and ask you to make some decisions about how and where you want to install R on your computer. We typically just click "continue" at every step without changing any of the default options.

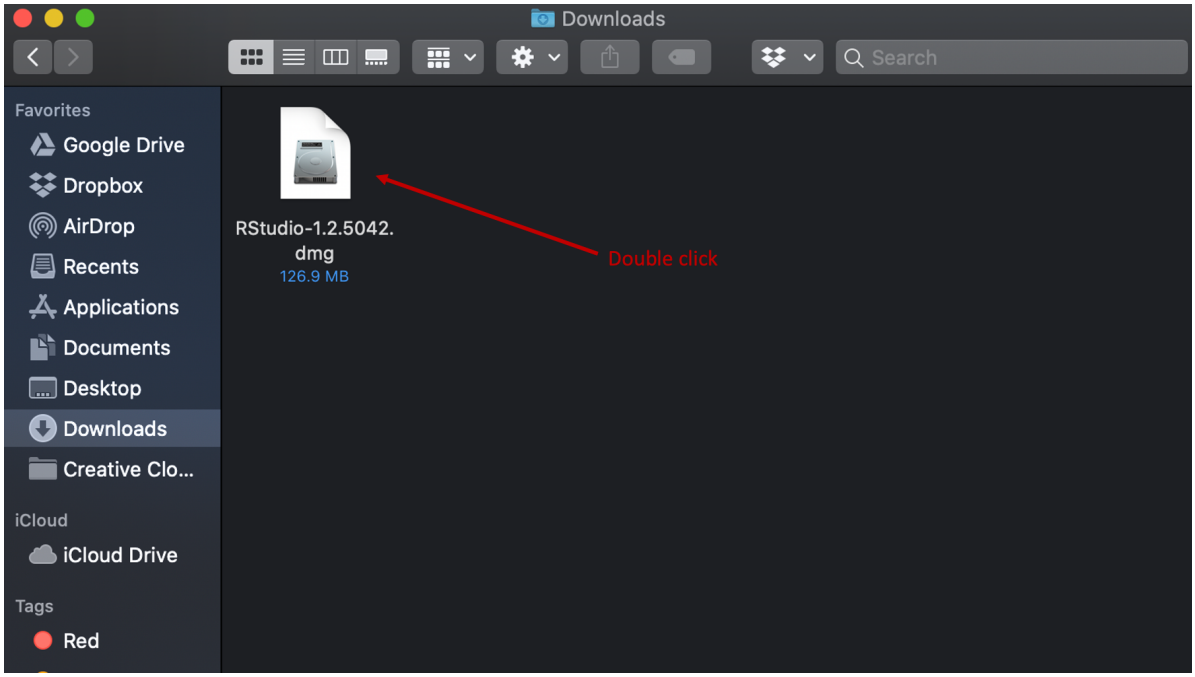


If R installed properly, you should now see it in your applications folder.

Step 7: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at https://posit.co/download/rstudio-desktop/. On that page, click the button to download the latest version of RStudio for your computer. Note that the website may look different that what you see in the screenshot below because websites change over time.



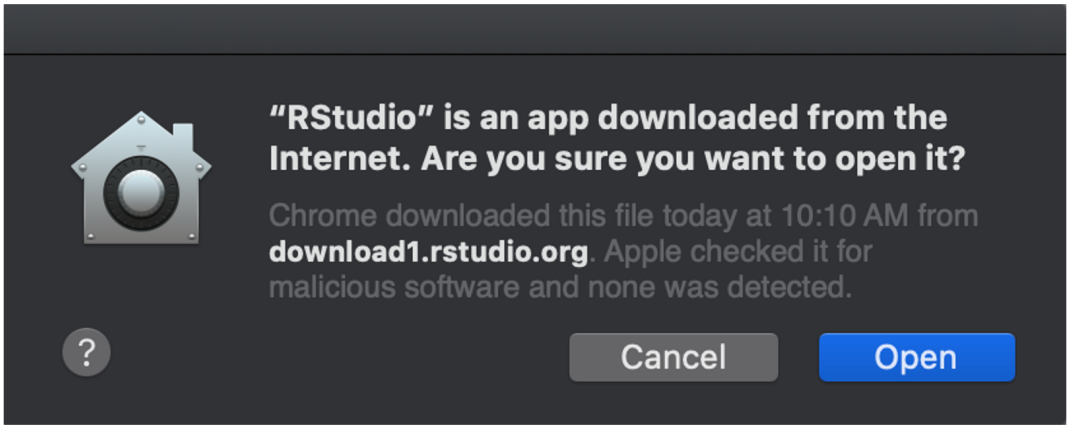
Step 8: Again, locate the DMG file you just downloaded and double click it. Unless you've changed your download settings, this file should be in the same location as the R package file you already downloaded.



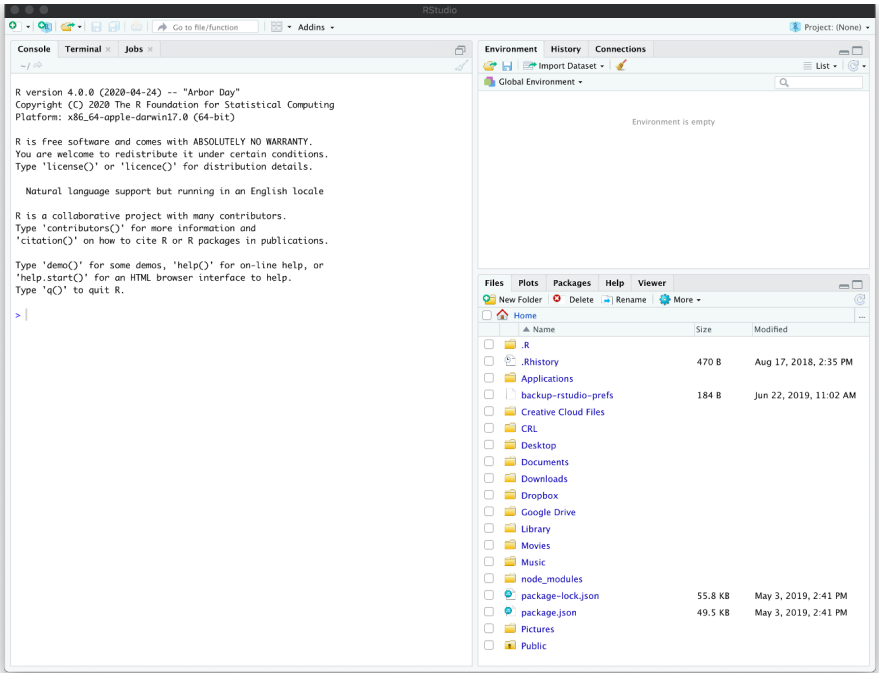
Step 9: A new finder window should automatically pop up that looks like the one you see below. Click on the RStudio icon and drag it into the Applications folder.

You should now see RStudio in your Applications folder. Double click the icon to open RStudio.

If this warning pops up, just click Open.

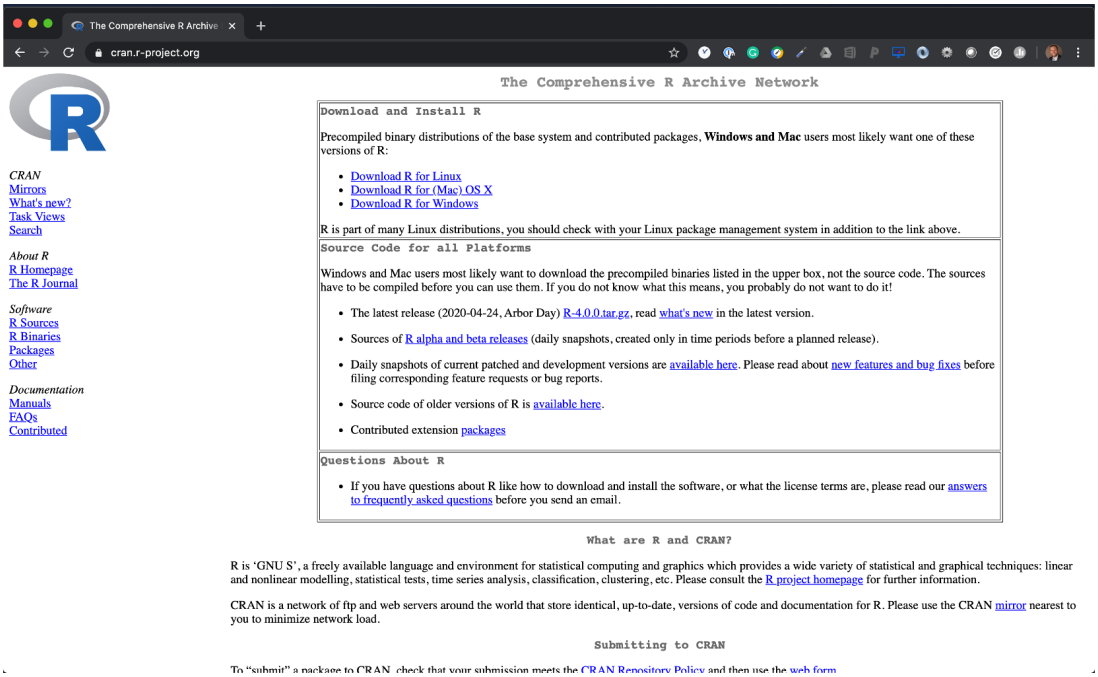


The RStudio IDE should open and look something like the window you see here. If so, you are good to go!

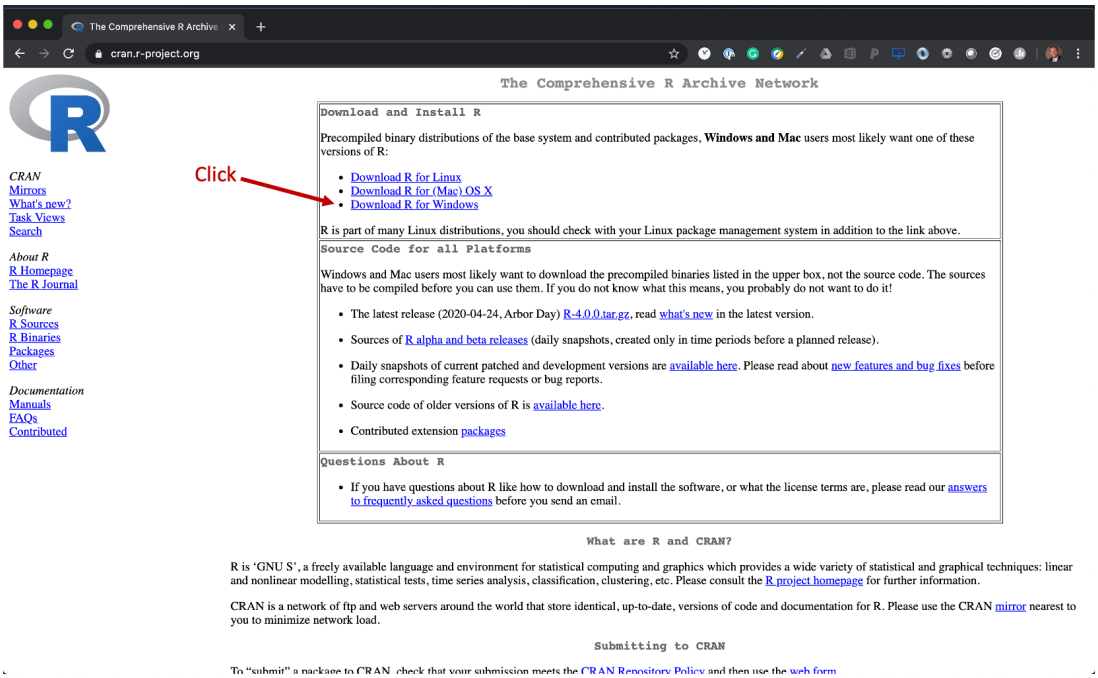


1.2 Download and install on a PC

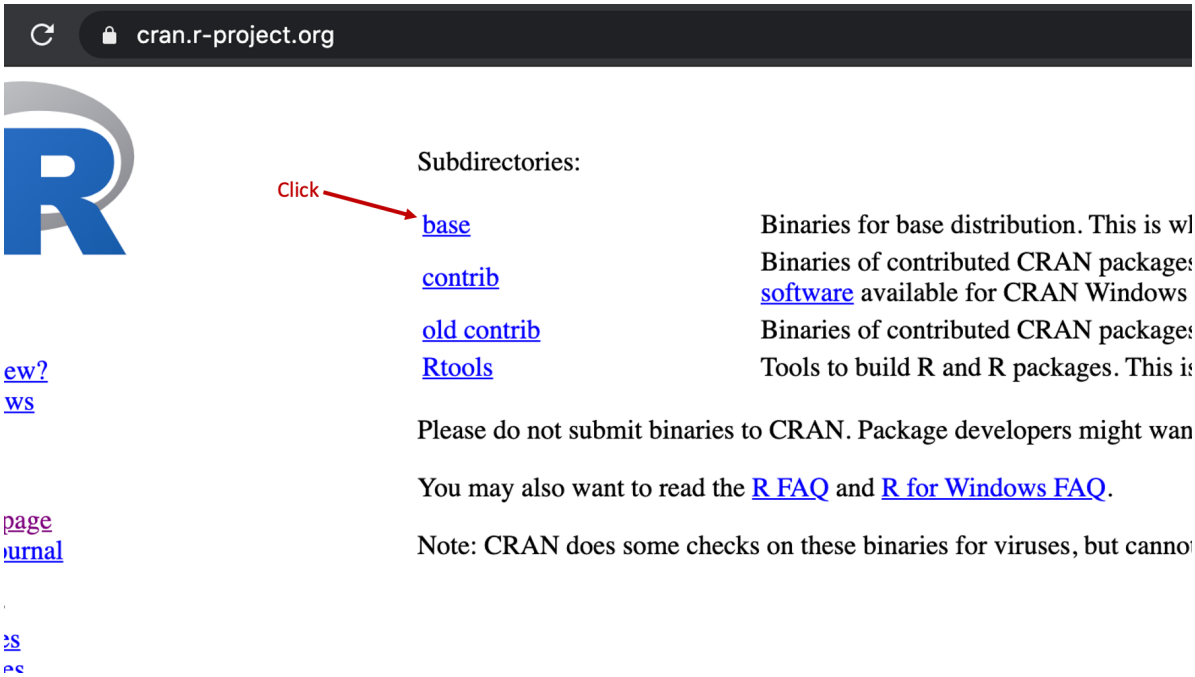
Step 2: Navigate to the Comprehensive R Archive Network (CRAN), which is located at https://cran.r-project.org/.



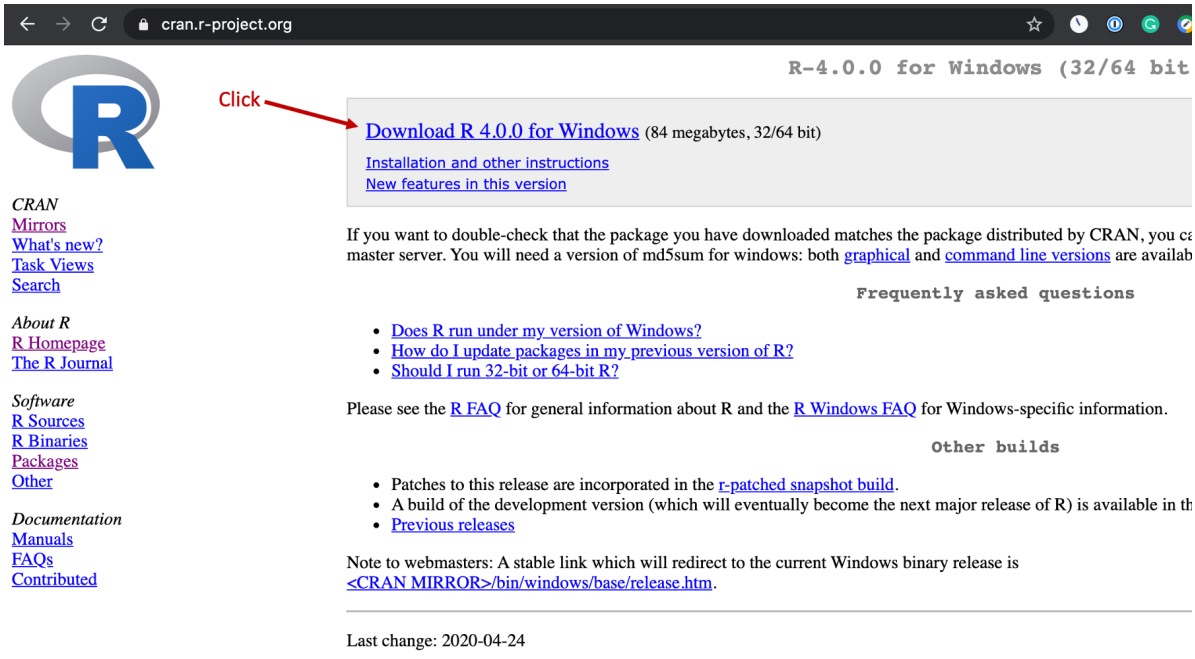
Step 3: Click on Download R for Windows.



Step 4: Click on the base link.

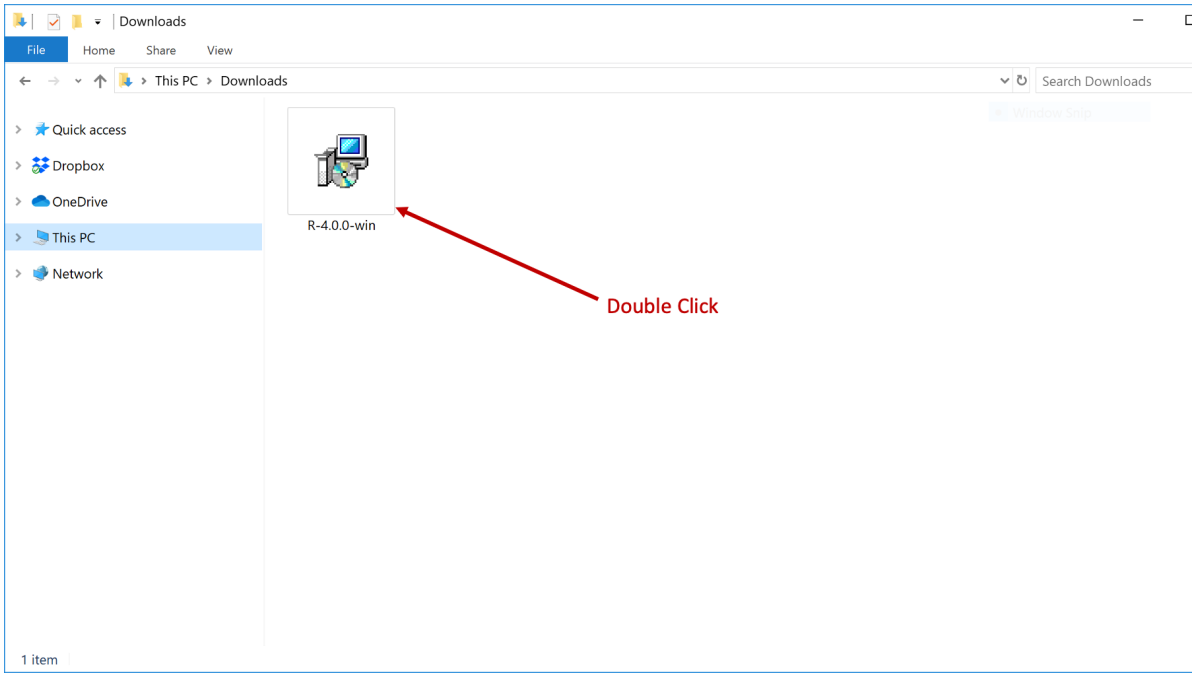


Step 5: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version should be roughly the same. After clicking, R should start to download to your computer.

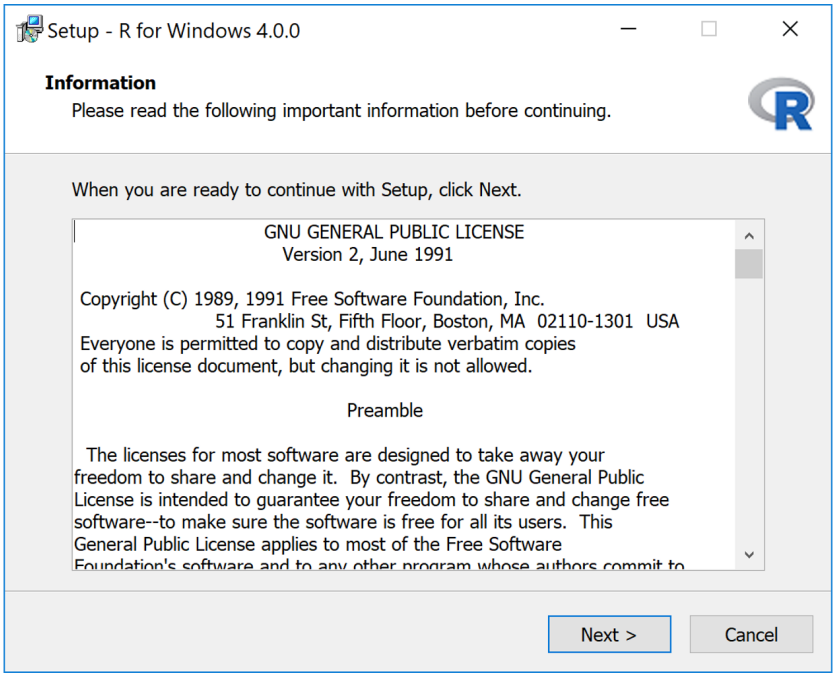


Step 6: Locate the installation file you just downloaded and double click it. Unless you've

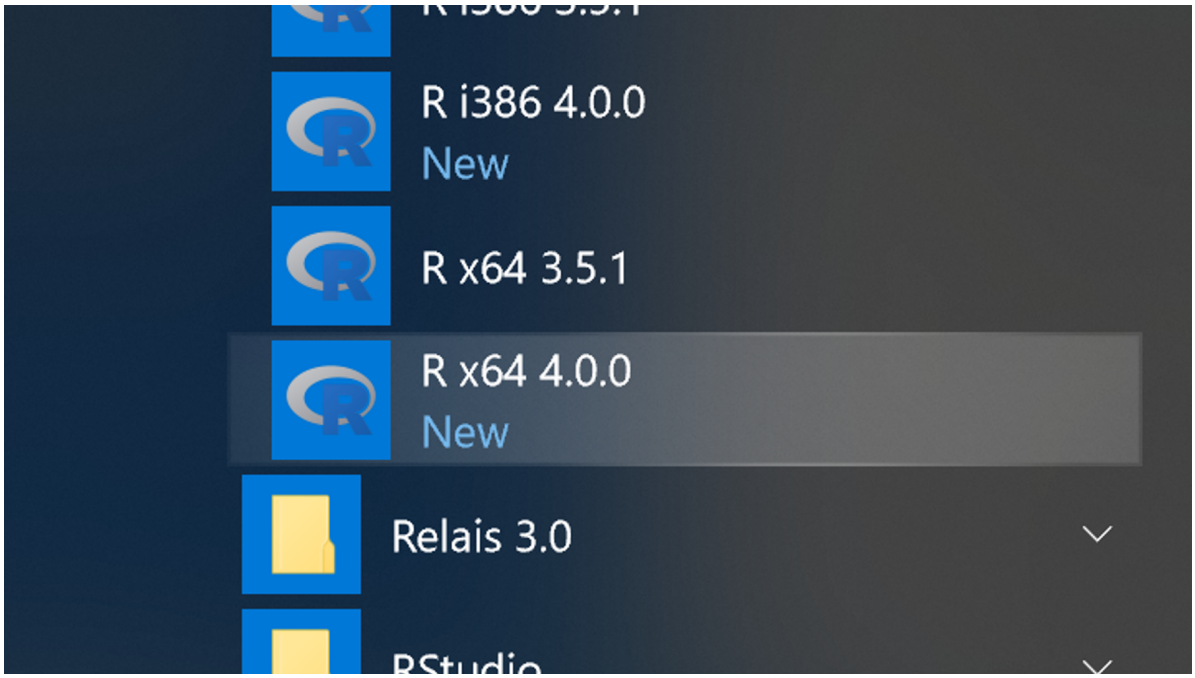
changed your download settings, this file will probably be in your downloads folder. That is the default location for most web browsers.



Step 7: A dialogue box will open that asks you to make some decisions about how and where you want to install R on your computer. We typically just click "Next" at every step without changing any of the default options.

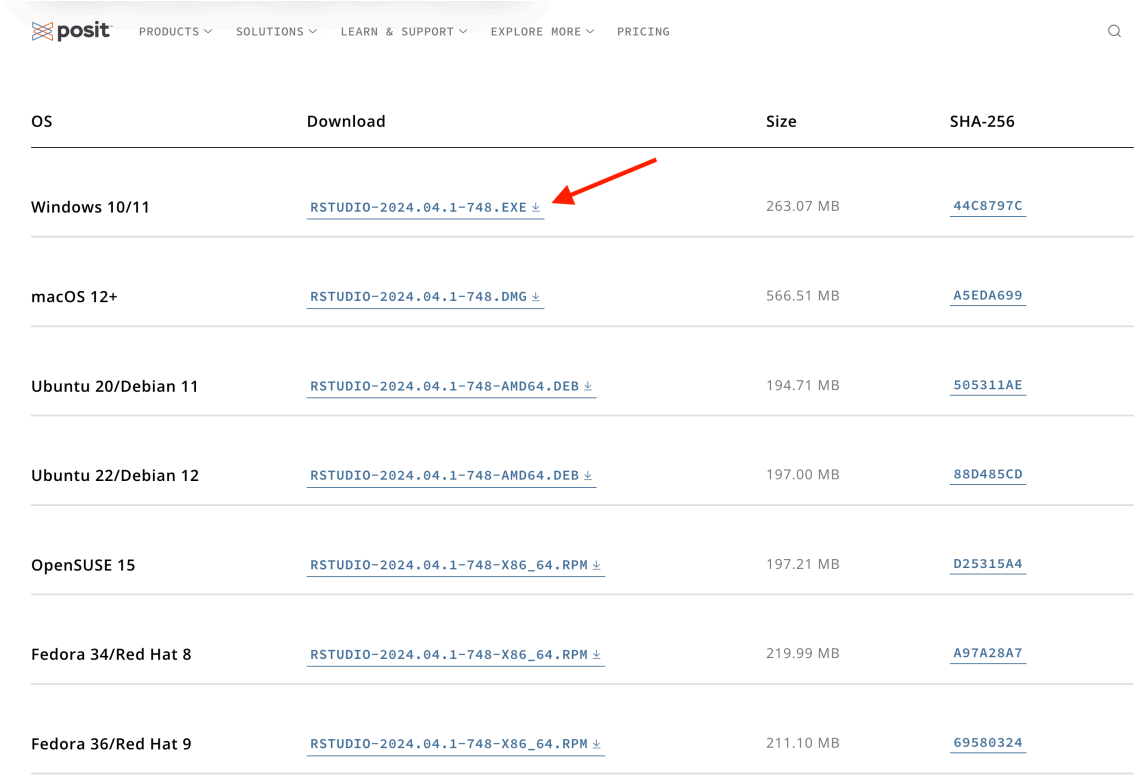


If R installed properly, you should now see it in the Windows start menu.

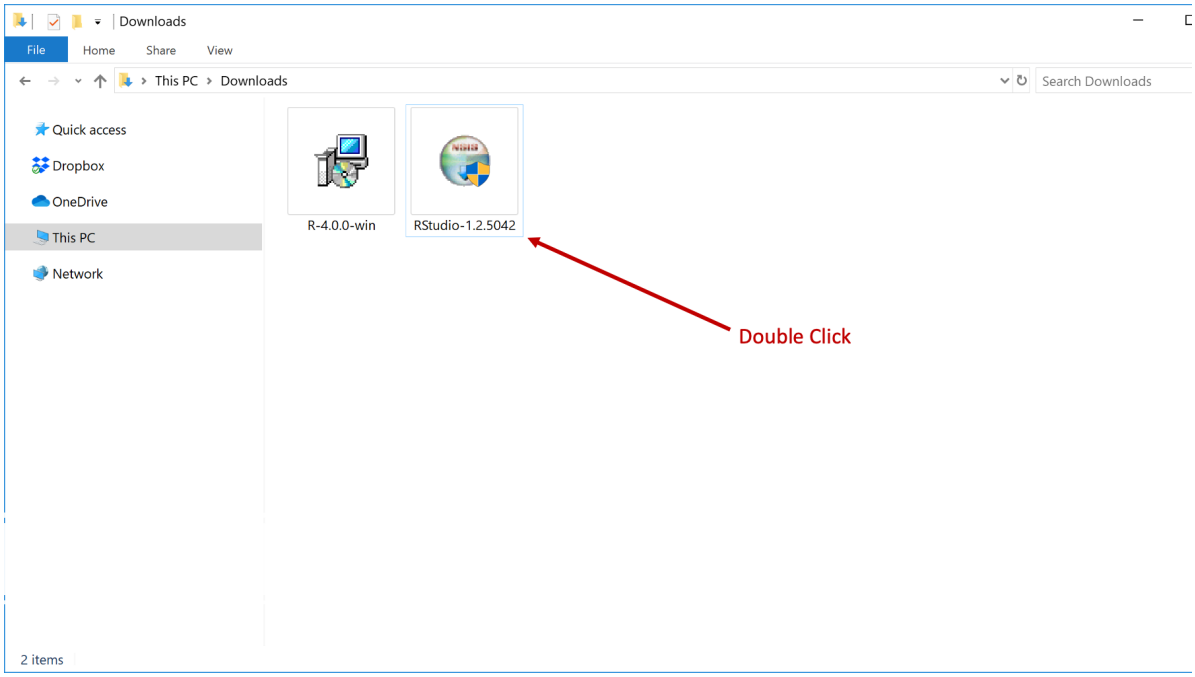


Step 8: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at https://posit.co/download/rstudio-desktop/. On that page, click the button to download the latest version of RStudio for your computer. Note that

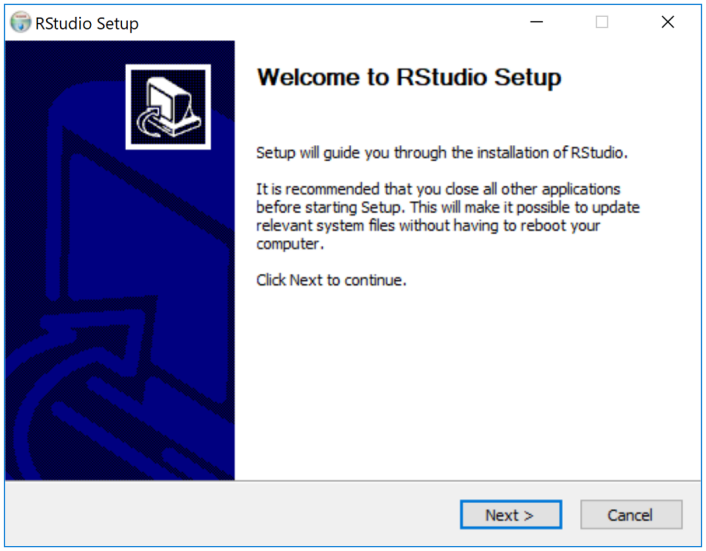
the website may look different that what you see in the screenshot below because websites change over time.



Step 9: Again, locate the installation file you just downloaded and double click it. Unless you've changed your download settings, this file should be in the same location as the R installation file you already downloaded.

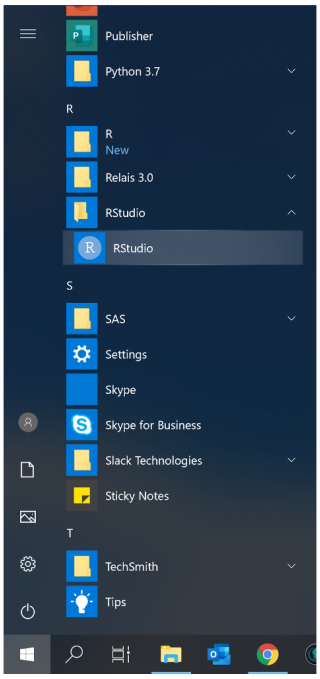


Step 10: Another dialogue box will open and ask you to make some decisions about how and where you want to install RStudio on your computer. We typically just click "Next" at every step without changing any of the default options.

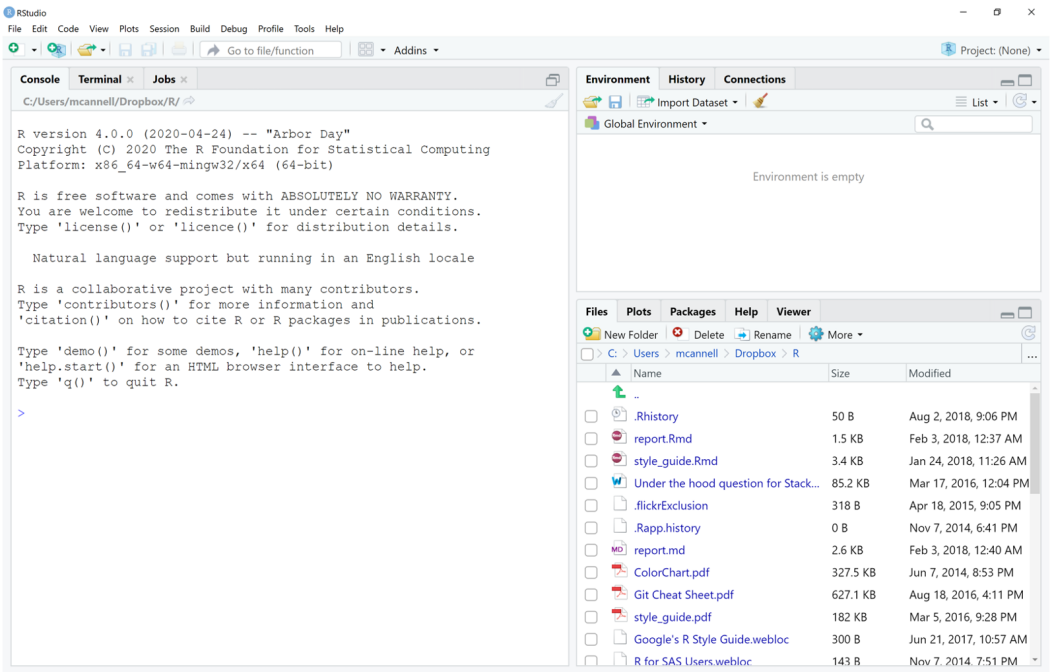


When RStudio is finished installing, you should see RStudio in the Windows start menu. Click

the icon to open RStudio.



The RStudio IDE should open and look something like the window you see here. If so, you are good to go!



2 What is R?

At this point in the book, you should have installed R and RStudio on your computer, but you may be thinking to yourself, "I don't even know what R is." Well, in this chapter you'll find out. We'll start with an overview of the R language, and then briefly touch on its capabilities and uses. You'll also see a complete R program and some complete documents generated by R programs. In this book you'll learn how to create similar programs and documents, and by the end of the book you'll be able to write your own R programs and present your results in the form of an issue brief written for general audiences who may or may not have public health expertise. But, before we discuss R let's discuss something even more basic – data. Here's a question for you: What is data?

2.1 What is data?

Data is information about objects (e.g., people, places, schools) and observable phenomenon (e.g., weather, temperatures, and disease symptoms) that is recorded and stored somehow as a collection of symbols, numbers, and letters. So, data is just information that has been "written" down.

Here we have a table, which is a common way of organizing data. In R, we will typically refer to these tables as **data frames**.

ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

Each box in a data frame is called a **cell**.

ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

Moving from left to right across the data frame are **columns**. Columns are also sometimes referred to as **variables**. In this book, we will often use the terms columns and variables interchangeably. Each column in a data frame has one, and only one, type. For now, know

that the type tells us what kind of data is contained in a column and what we can *do* with that data. You may have already noticed that 3 of the columns in the table we've been looking at contain numbers and 1 of the columns contains words. These columns will have different types in R and we can do different things with them based on their type. For example, we could ask R to tell us what the average value of the numbers in the height column are, but it wouldn't make sense to ask R to tell us the average value of the words in the Gender column. We will talk more about many of the different column types exist in R later in this book.

ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

The information contained in the first cell of each column is called the **column name** (or variable) name.

R gives us a lot of flexibility in terms of what we can name our columns, but there are a few rules.

1. Column names can contain letters, numbers and the dot (.) or underscore (_) characters.

2. Additionally, they can begin with a letter or a dot – as long as the dot is not followed by a number. So, a name like ".2cats" is not allowed.

3. Finally, R has some reserved words that you are not allowed to use for column names. These include: "if", "else", "repeat", "while", "function", "for", "in", "next", and "break".

ID	Gender	Height	Weight
	Male	71	190
	Male	69	176
003	Female	64	130
004	Female	65	154

1. Numbers and the dot (.) or underscore (_) characters
2. Begins with a letter or a dot as long as the dot is not followed by a number
3. No reserved words

Moving from top to bottom across the table are **rows**, which are sometimes referred to as records.

ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

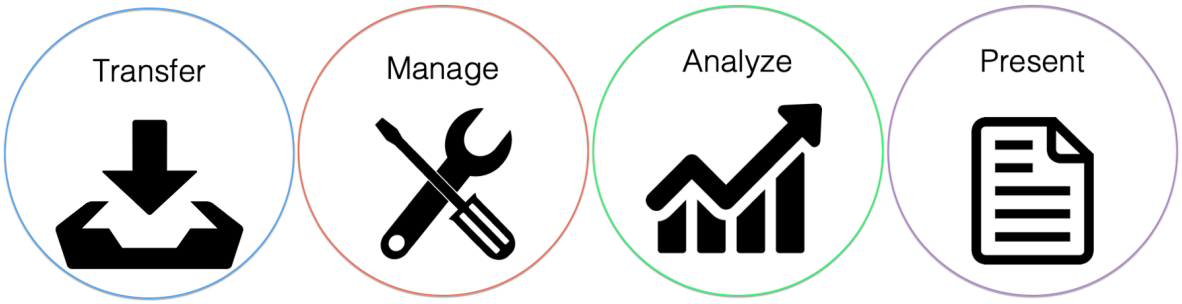
Finally, the contents of each cell are called **values**.

ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

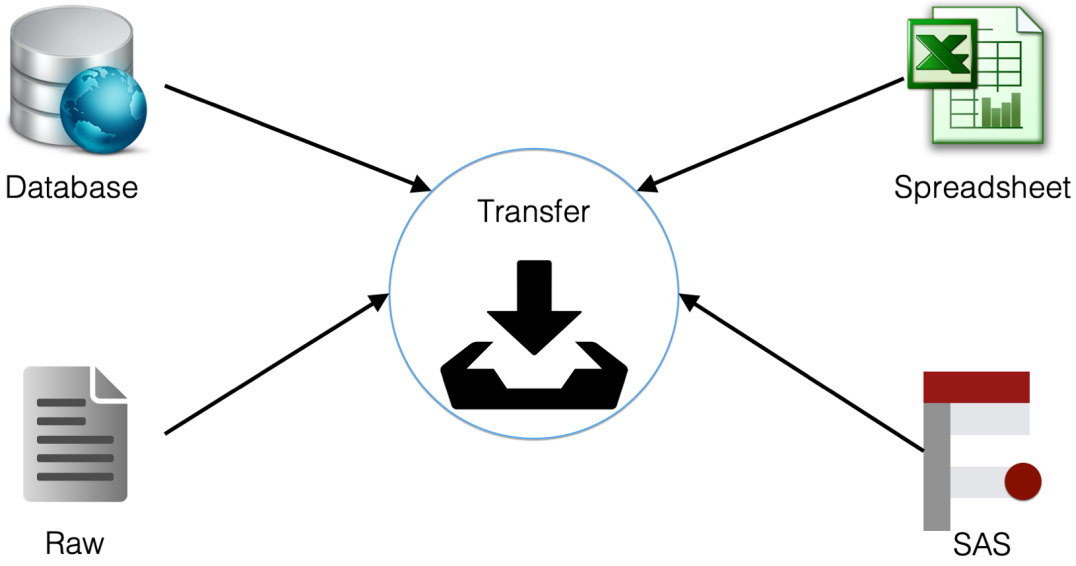
You should now be up to speed on some basic terminology used by R, as well as other analytic, database, and spreadsheet programs. These terms will be used repeatedly throughout the course.

2.2 What is R?

So, what is R? Well, R is an **open source** statistical programming language that was created in the 1990's specifically for data analysis. We will talk more about what open source means later, but for now, just think of R as an easy (relatively) way to ask your computer to do math and statistics for you. More specifically, by the end of this book you will be able to independently use R to transfer data, manage data, analyze data, and present the results of your analysis. Let's quickly take a closer look at each of these.



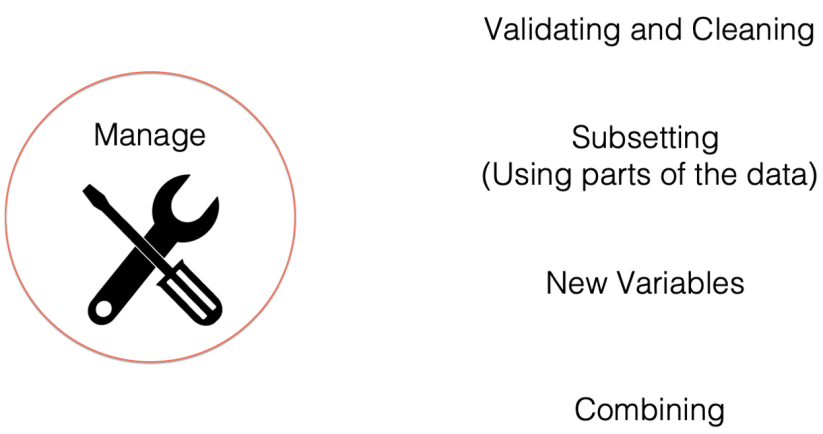
2.2.1 Transferring data



So, what do we mean by "transfer data"? Well, individuals and organizations store their data

using different computer programs that use different file types. Some common examples that you may come across in epidemiology are database files, spreadsheets, raw data files, and SAS data sets. No matter how the data is stored, you can't do anything with it until you can get it into R, in a form that R can use, and in a location that you can reach. In other words, transferring your data. Therefore, among our first tasks in this course will be to transfer data.

2.2.2 Managing data



This isn't very specific, but managing data is all the things you may have to do to your data to get it ready for analysis. You may also hear people refer to this process as data wrangling or data munging. Some specific examples of data management tasks include:

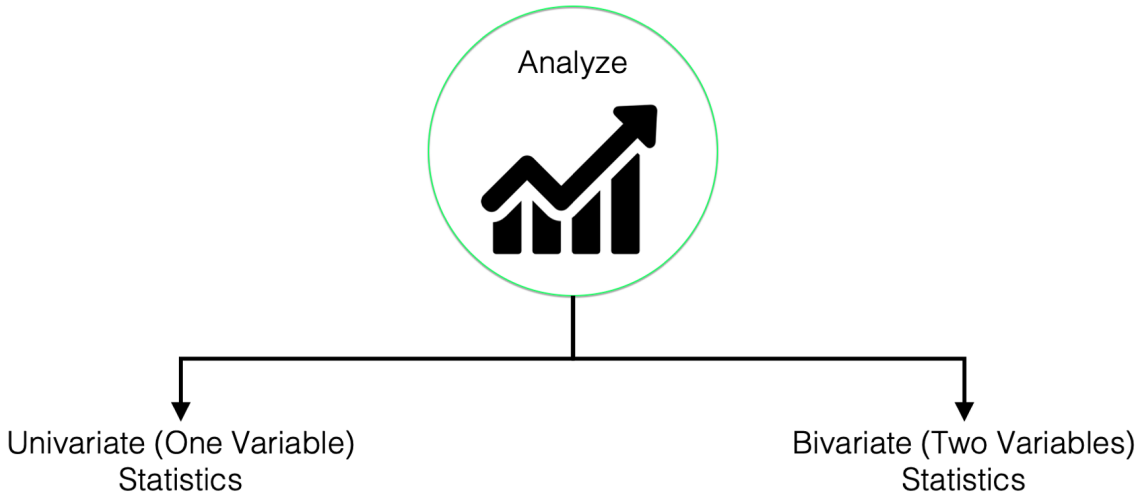
- Validating and cleaning data. In other words, dealing with potential errors in the data.
- Subsetting data. For example, using only some of the columns or some of the rows.
- Creating new variables. For example, creating a BMI variable in a data frame that was sent to you with height and weight columns.
- Combining data frames. For example, combining sociodemographic data about study participants with data collected in the field during an intervention.

You may sometimes hear people refer to the 80/20 rule in reference to data management. This "rule" says that in a typical data analysis project, roughly 80% of your time will be spent on data management and only 20% will be spent on the analysis itself. We can't provide you with any empirical evidence (i.e., data) to back this claim up. But, as people who have been involved in many projects that involve the collection and analysis of data, we can tell you anecdotally that this "rule" is probably pretty close to being accurate in most cases.

Additionally, it's been our experience that most students of epidemiology are required to take one or more classes that emphasize methods for analyzing data; however, almost none of them have taken a course that emphasizes data management!

Therefore, because data management is such a large component of most projects that involve the collection and analysis of data, and because most readers will have already been exposed to data analysis to a much greater extent than data management, this course will heavily emphasize the latter.

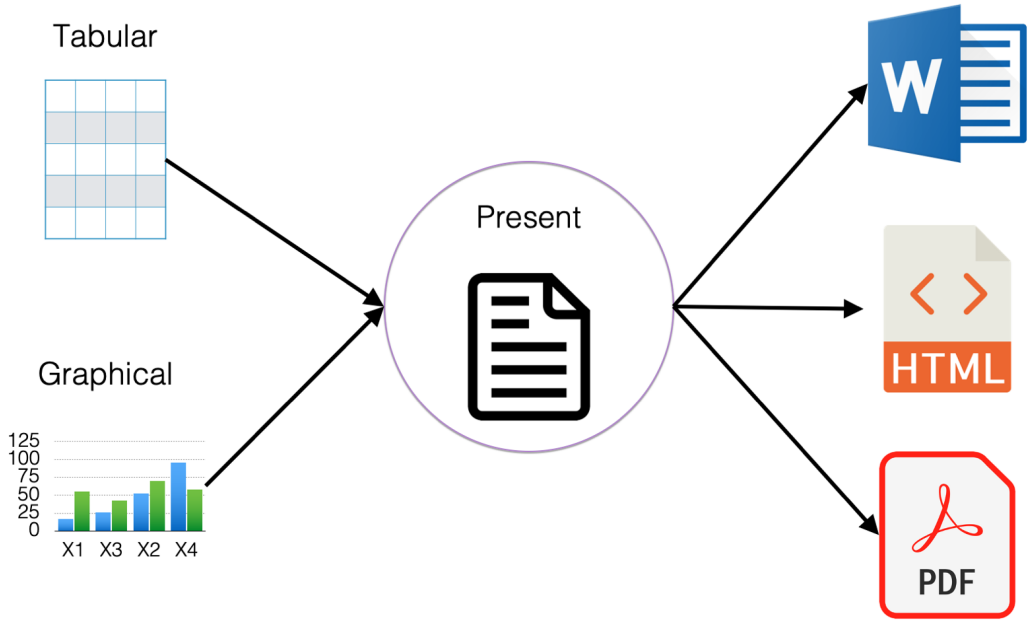
2.2.3 Analyzing data



As just discussed, this is probably the capability you most closely associate with R, and there is no doubt that R is a powerful tool for analyzing data. However, in this book we won't go beyond using R to calculate basic descriptive statistics. For our purposes, descriptive statistics include:

- Measures of central tendency. For example, mean, median, and mode.
- Measures of dispersion. For example, variance and standard error.
- Measures for describing categorical variables. For example, counts and percentages.
- Describing data using graphs and charts. With R, we can describe our data using beautiful and informative graphs.

2.2.4 Presenting data



And finally, the ultimate goal is typically to present your findings in some form or another. For example, a report, a website, or a journal article. With R you can present your results in many different formats with relative ease. In fact, this is one of our favorite things about R and RStudio. In this class you will learn how to take your text, tabular, or graphical results and then publish them in many different formats including Microsoft Word, html files that can be viewed in web browsers, and pdf documents. Let's take a look at some examples.

1. **Microsoft Word documents**. Click here to view an example report created for one of our research projects in Microsoft Word.

2. **PDF documents**. Click here to view a data dictionary we created in PDF format.

3. **HTML files**. Hypertext Markup Language (HTML) files are what you are looking at whenever you view a webpage. You can use R to create HTML files that others can view in their web browser. You can email them these files to view in their web browser, or you can make them available for others to view online just like any other website. Click here to view an example dashboard we created for one of our research projects.

4. **Web applications**. You can even use R to create full-fledged web applications. View the RStudio website to see some examples.

3 Navigating the RStudio Interface

If you followed along with the previous chapters, you have R and RStudio installed on your computer and you have some idea of what R and RStudio are. At this point, it can be common for people to open RStudio and get totally overwhelmed. *"What am I looking at?" "What do I click first?" "Where do I even start?"* Don't worry if these, or similar, thoughts have crossed your mind. You are in good company and we will start to clear some of them up in this chapter.

When we load RStudio, we should see a screen that looks very similar to Figure 3.1 below. There, we see three **panes**, and each pane has multiple tabs.

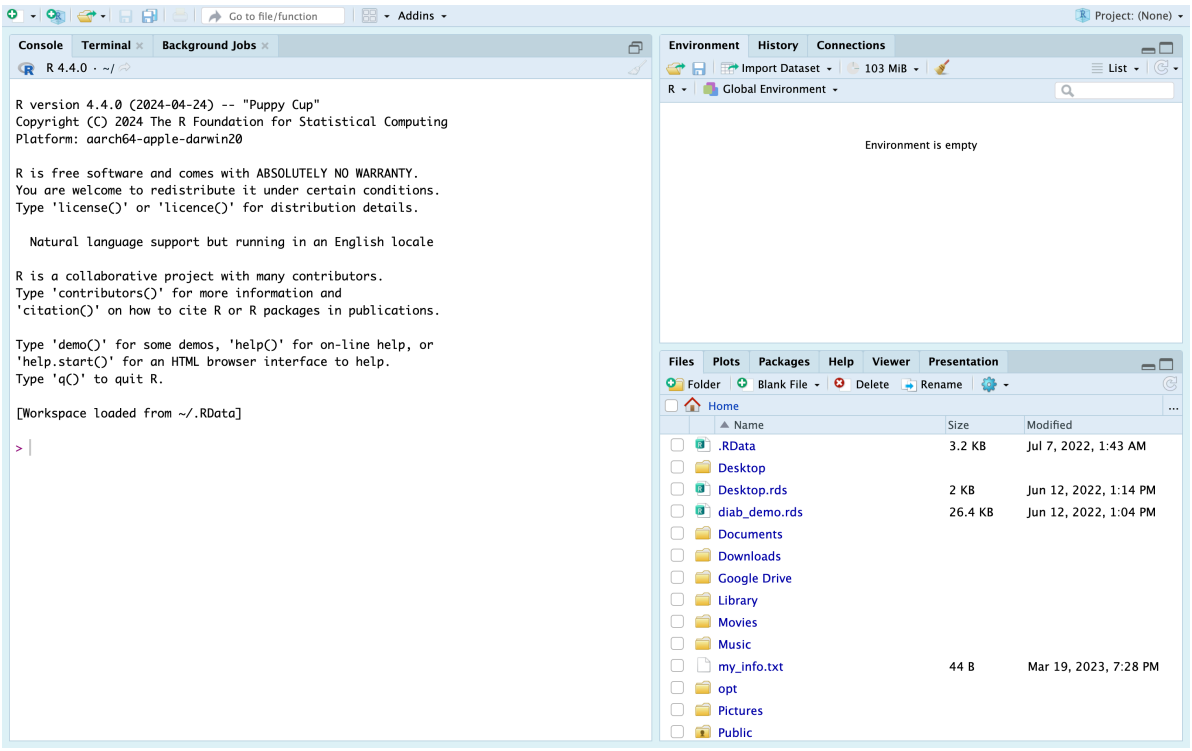


Figure 3.1: The default RStudio user interface.

3.1 The console pane

The first pane we are going to talk about is the **console/terminal/background jobs** pane.

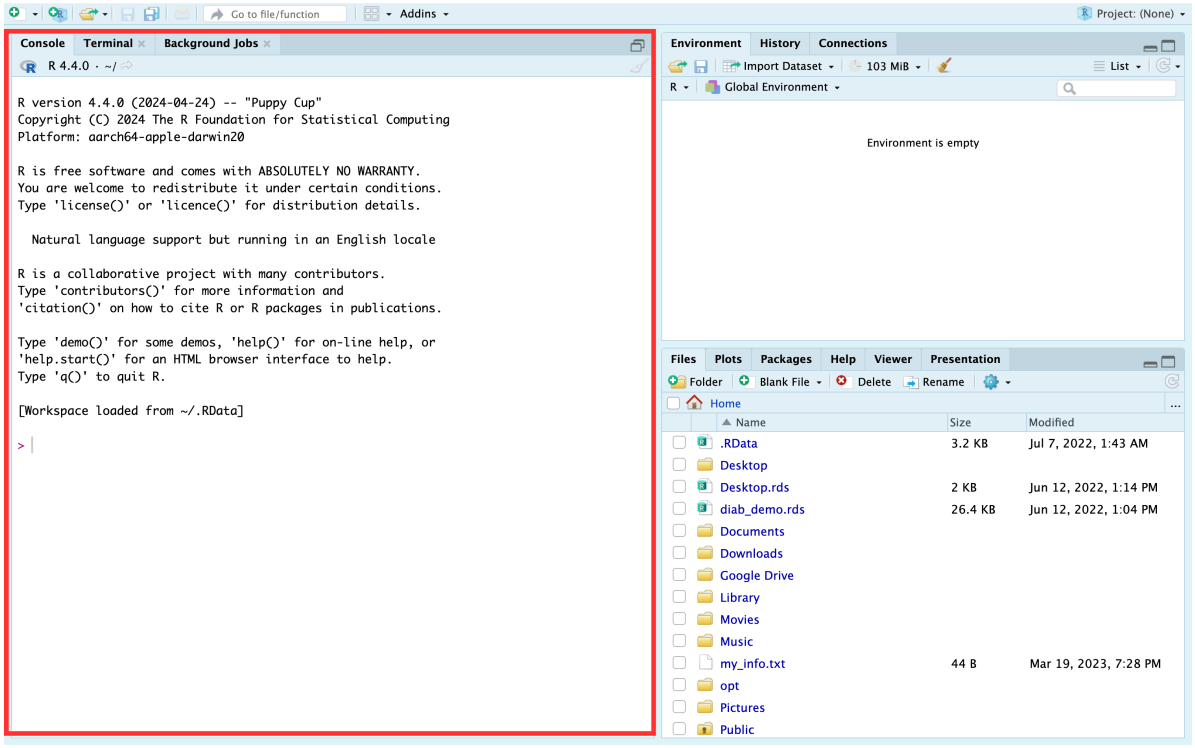


Figure 3.2: The R Console.

It's called the "console/terminal/background jobs" pane because it has three tabs we can click on by default: "console", "terminal", and "background jobs". However, we will refer to this pane as the "console pane" and will mostly ignore the terminal and background jobs tabs for now. We aren't ignoring them because they aren't useful; instead, we are ignoring them because using them isn't essential for anything we will discuss in this chapter, and we want to keep things as simple as possible for now.

The console is the most basic way to interact with R. We can type a command to R into the console prompt (the prompt looks like ">") and R will respond to what we type. For example, below we typed "1 + 1," pressed the return/enter key, and the R console returned the sum of the numbers 1 and 1.

The number 1 we see in brackets before the 2 (i.e., [1]) is telling us that this line of results starts with the first result. That fact is obvious here because there is only one result. So, let's look at a result that spans multiple lines to make this idea clearer.

In Figure 3.4 we see examples of a couple of new concepts that are worth discussing.

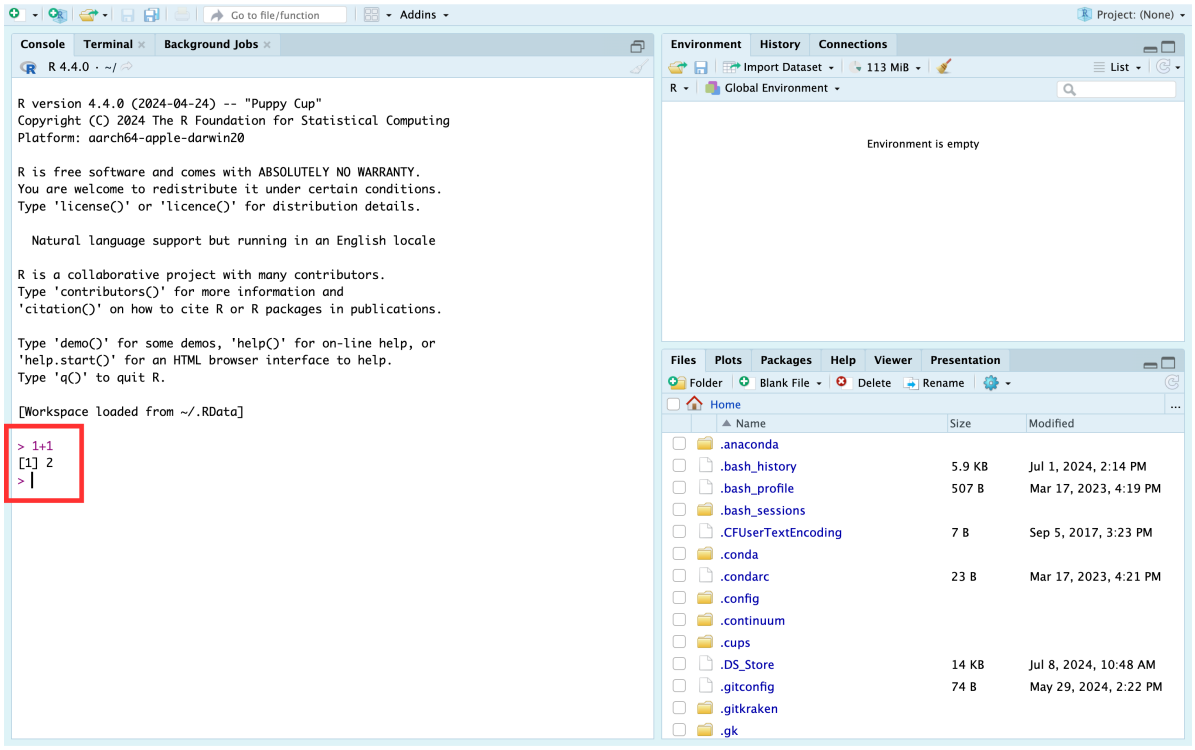


Figure 3.3: Doing some addition in the R console.

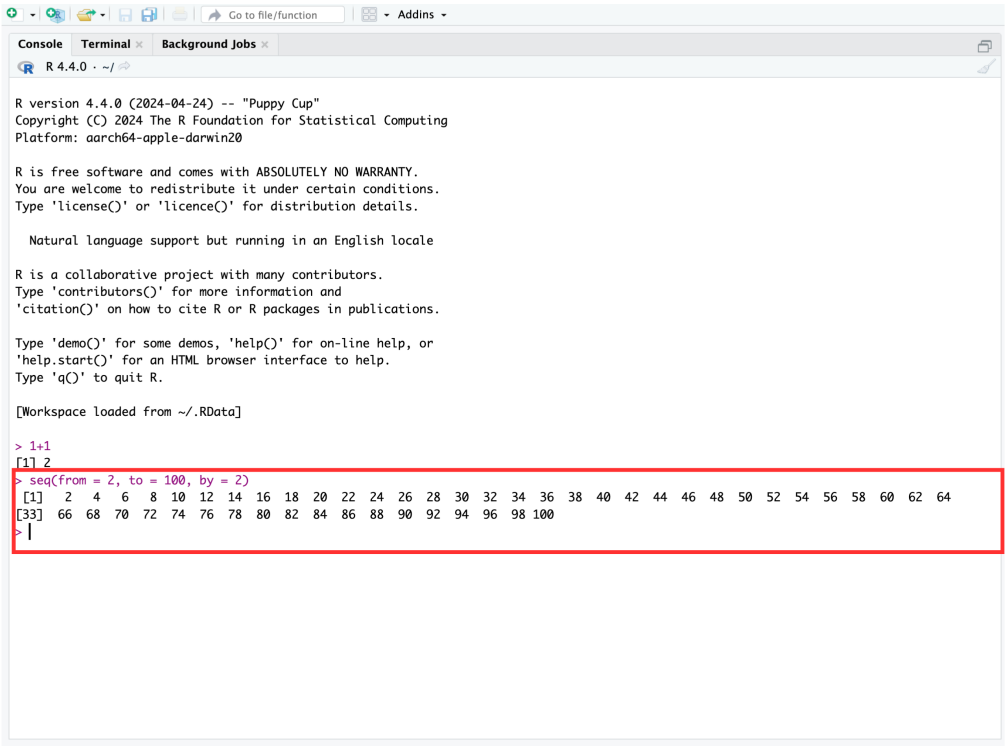


Figure 3.4: Demonstrating a function that returns multiple results.

First, as promised, we have more than one line of results (or output). The first line of results starts with a 1 in brackets (i.e., [1]), which indicates that this line of results starts with the first result. In this case, the first result is the number 2. The second line of results starts with a 29 in brackets (i.e., [29]), which indicates that this line of results starts with the twenty-ninth result. In this case, the twenty-ninth result is the number 58. If we count the numbers in the first line, there should be 28 – results 1 through 28. We also want to make it clear that "1" and "29" are *NOT* results themselves. They are just helping us count the number of results per line.

The second new thing that you may have noticed in Figure 3.4 is our use of a **function**. Functions are a **BIG DEAL** in R. So much so that R is called a *functional language*. We don't really need to know all the details of what that means; however, we should know that, in general, everything we *do* in R we will *do* with a function. By contrast, everything we *create* in R will be an *object*. If we wanted to make an analogy between the R language and the English language, we could think of functions as verbs – they *do* things – and objects as nouns – they *are* things. This distinction likely seems abstract and confusing at the moment, but we will make it more concrete soon.

Most functions in R begin with the function name followed by parentheses. For example, `seq()`, `sum()`, and `mean()`.

Question: What is the name of the function we used in the example above?

Answer: We used the `seq()` function – short for sequence - in the example above.

You may notice that there are three pairs of words, equal symbols, and numbers that are separated by commas inside the `seq()` function. They are, `from = 2`, `to = 100`, and `by = 2`. The words `from`, `to`, and `by` are all arguments to the `seq()` function. We will learn more about functions and arguments later. For now, just know that arguments *give functions the information they need to give us the result we want.*

In this case, the `seq()` function returns a sequence of numbers. But first, we had to give it information about where that sequence should start, where it should end, and how many steps should be in the middle. Above, the sequence began with the value we passed to the `from` argument (i.e., 2), it ended with the value we passed to the `to` argument (i.e., 100), and it increased at each step by the number we passed to the `by` argument (i.e., 2). So, 2, 4, 6, 8 … 100.

Whether you realize it or not, we've covered some important programming terms while discussing the `seq()` function above. Before we move on to discussing RStudio's other panes, let's quickly review and reinforce a few of terms we will use repeatedly in this book.

- Arguments: Arguments always live *inside* the parentheses of R functions and receive information the function needs to generate the result we want.

- Pass: In programming lingo, we *pass* a value to a function argument. For example, in the function call `seq(from = 2, to = 100, by = 2)` we could say that we *passed* a value of 2 to the `from` argument, we *passed* a value of 100 to the `to` argument, and we *passed* a value of 2 to the `by` argument.
- Return: Instead of saying, "the `seq()` function *gives us* a sequence of numbers..." we say, "the `seq()` function *returns* a sequence of numbers..." In programming lingo, functions *return* one or more results.

> **Note**
>
> **Side Note:** The `seq()` function isn't particularly important or noteworthy. We essentially chose it at random to illustrate some key points. However, arguments, passing values, and return values are extremely important concepts and we will return to them many times.

3.2 The environment pane

The second pane we are going to talk about is the environment/history/connections pane in Figure 3.5. However, we will mostly refer to it as the environment pane and we will mostly ignore the history and connections tab. We aren't ignoring them because they aren't useful; rather, we are ignoring them because using them isn't essential for anything we will discuss anytime soon, and we want to keep things as simple as possible.

The Environment pane shows you all the **objects** that R can currently use for data management or analysis. In this picture, Figure 3.5 our environment is empty. Let's create an object and add it to our environment.

Here we see that we created a new object called `x`, which now appears in our **Global Environment**. Figure 3.6 This gives us another great opportunity to discuss some new concepts.

First, we created the `x` object in the console by *assigning* the value 2 to the letter x. We did this by typing "x" followed by a less than symbol (<), a dash symbol (-), and the number 2. R is kind of unique in this way. we have never seen another programming language (although I'm sure they are out there) that uses `<-` to assign values to variables. By the way, `<-` is called the assignment operator (or assignment arrow), and "assign" here means "make x contain 2" or "put 2 inside x."

In many other languages you would write that as `x = 2`. But, for whatever reason, in R it is `<-`. Unfortunately, `<-` is more awkward to type than `=`. Fortunately, RStudio gives us a keyboard shortcut to make it easier. To type the assignment operator in RStudio, just hold down Option + - (dash key) on a Mac or Alt + - (dash key) on a PC and RStudio will insert `<-` complete with spaces on either side of the arrow. This may still seem awkward at first, but you will get used to it.

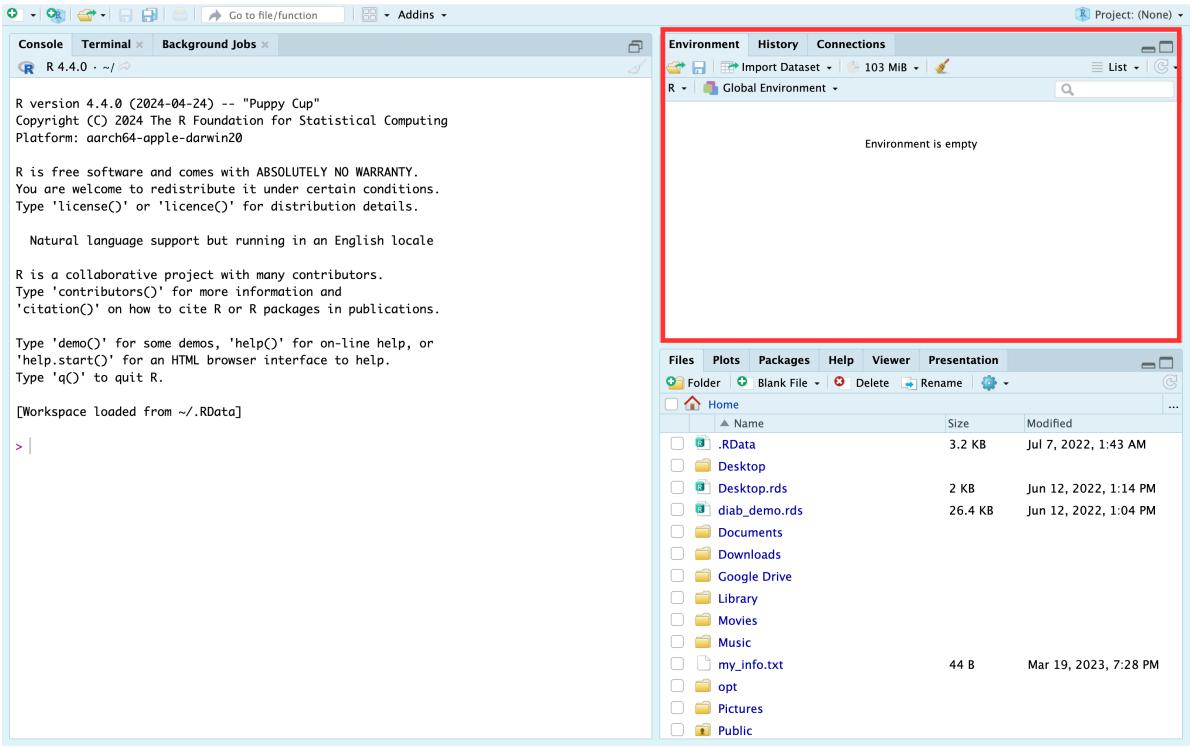


Figure 3.5: The environment pane

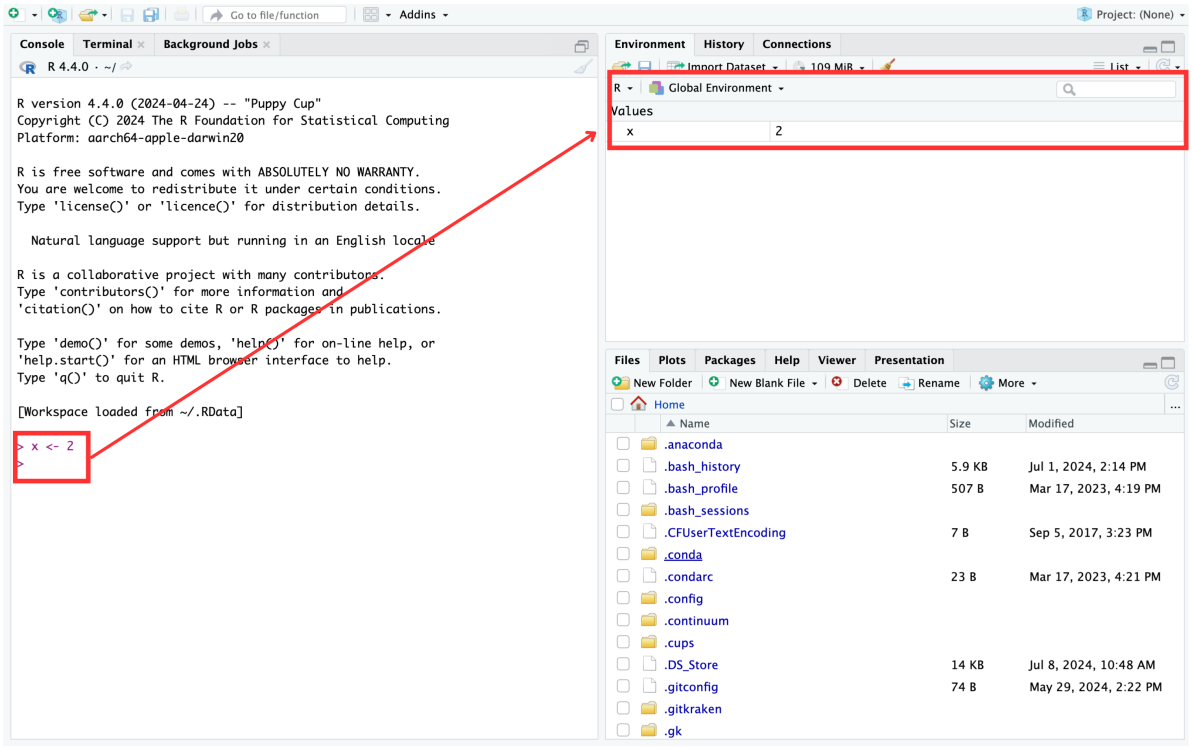


Figure 3.6: The vector x in the global environment.

> **Note**
>
> **Side Note:** A note about using the letter "x": By convention, the letter "x" is a widely used variable name. You will see it used a lot in example documents and online. However, there is nothing special about the letter x. We could have just as easily used any other letter (`a <- 2`), word (`variable <- 2`), or descriptive name (`my_favorite_number <- 2`) that is allowed by R.

Second, you can see that our Global Environment now includes the object `x`, which has a value of 2. In this case, we would say that `x` is a **numeric vector** of length 1 (i.e., it has one value stored in it). We will talk more about vectors and vector types soon. For now, just notice that objects that you can manipulate or analyze in R will appear in your Global Environment.

> **Warning**
>
> **Warning:** R is a **case sensitive** language. That means that uppercase x (X) and lowercase x (x) are different things to R. So, if you assign 2 to lower case x (`x <- 2`). And then later ask R to tell what number you stored in uppercase X, you will get an error (`Error: object 'X' not found`).

3.3 The files pane

Next, let's talk about the Files/Plots/Packages/Help/Viewer pane (that's a mouthful). Figure 3.7

Again, some of these tabs are more applicable for us than others. For us, the **files** tab and the **help** tab will probably be the most useful. You can think of the files tab as a mini Finder window (for Mac) or a mini File Explorer window (for PC). The help tab is also extremely useful once you get acclimated to it.

For example, in the screenshot above Figure 3.8 we typed the `seq` into the search bar. The help pane then shows us a page of documentation for the `seq()` function. The documentation includes a brief description of what the function does, outlines all the arguments the `seq()` function recognizes, and, if you scroll down, gives examples of using the `seq()` function. Admittedly, this help documentation can seem a little like reading Greek (assuming you don't speak Greek) at first. But, you will get more comfortable using it with practice. We hated the help documentation when we were learning R. Now, we use it *all the time*.

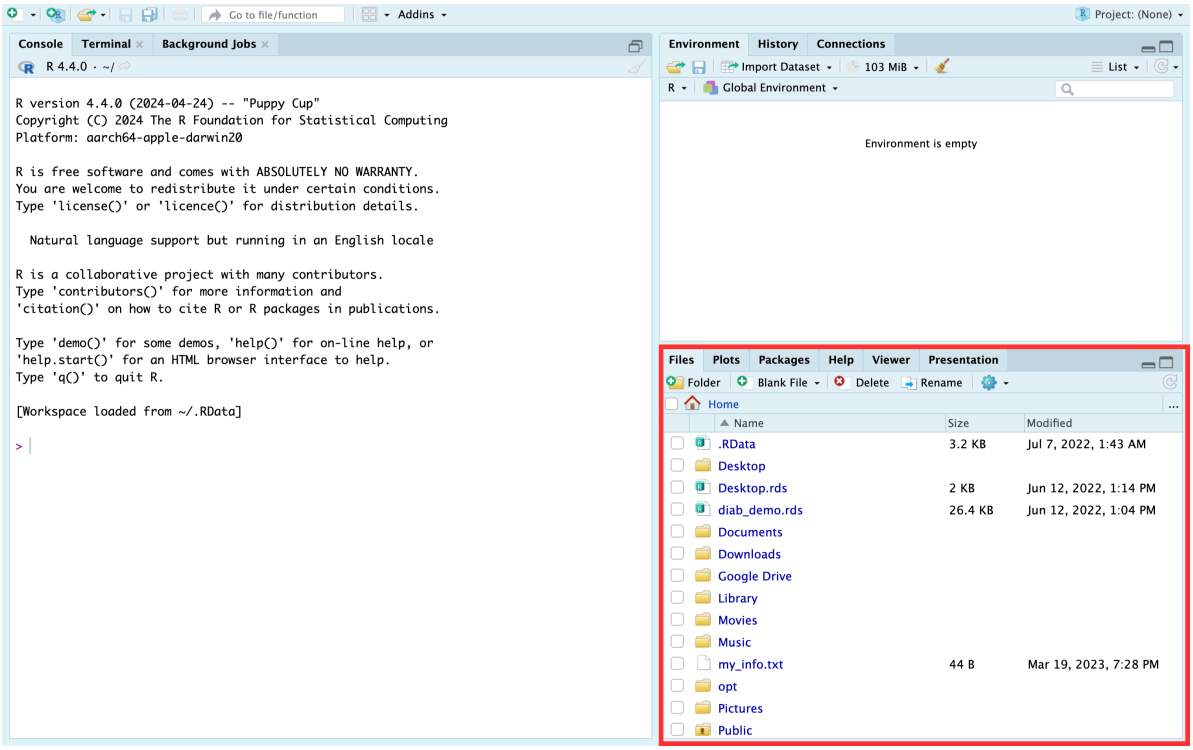


Figure 3.7: The Files/Plots/Packages/Help/Viewer pane.

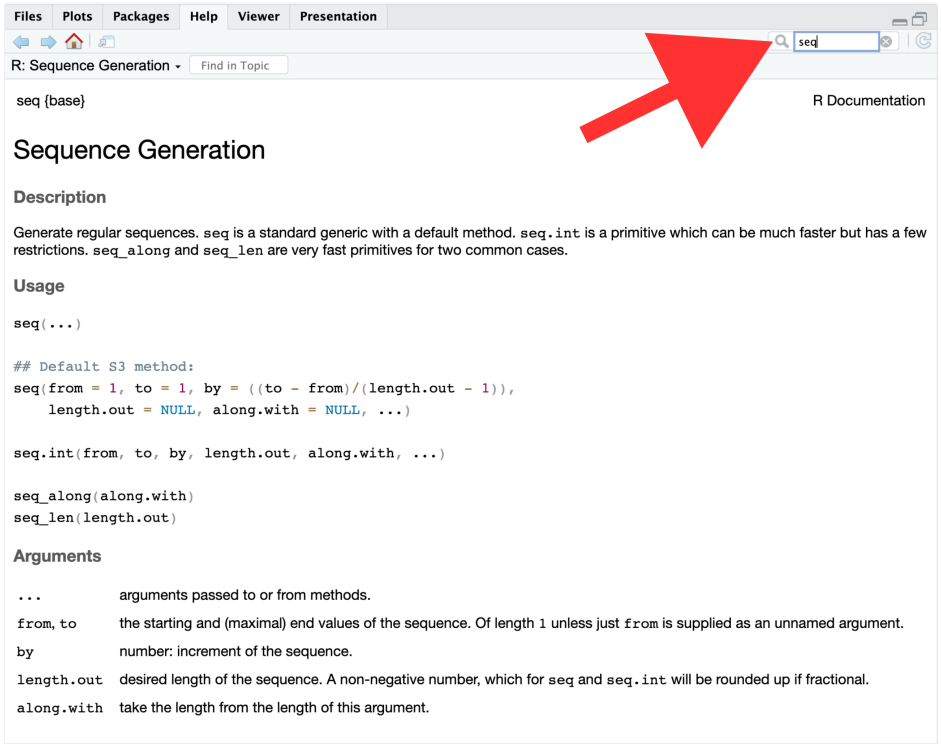


Figure 3.8: The help tab.

3.4 The source pane

There is actually a fourth pane available in RStudio. If you click on the icon shown below you will get the following dropdown box with a list of files you can create. Figure 3.9

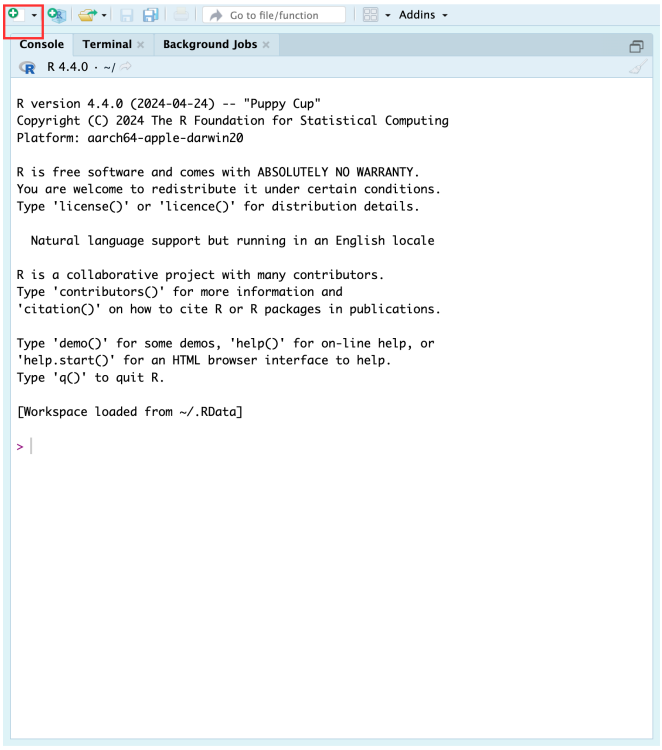


Figure 3.9: Click the new source file icon.

If you click any of these options, a new pane will appear. We will arbitrarily pick the first option – R Script.

When we do, a new pane appears. It's called the **source pane**. In this case, the source pane contains an untitled R Script. We won't get into the details now because we don't want to overwhelm you, but soon you will do the majority of your R programming in the source pane.

3.5 RStudio preferences

Finally, We're going to recommend that you change a few settings in RStudio before we move on. Start by clicking `Tools`, and then `Global Options` in RStudio's menu bar, which probably runs horizontally across the top of your computer's screen.

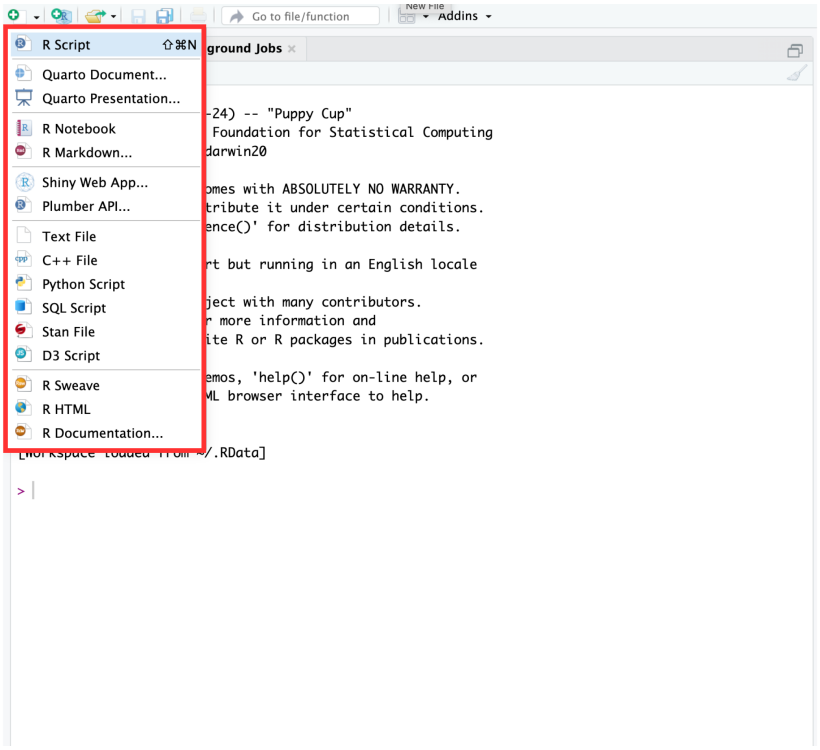


Figure 3.10: New source file options.

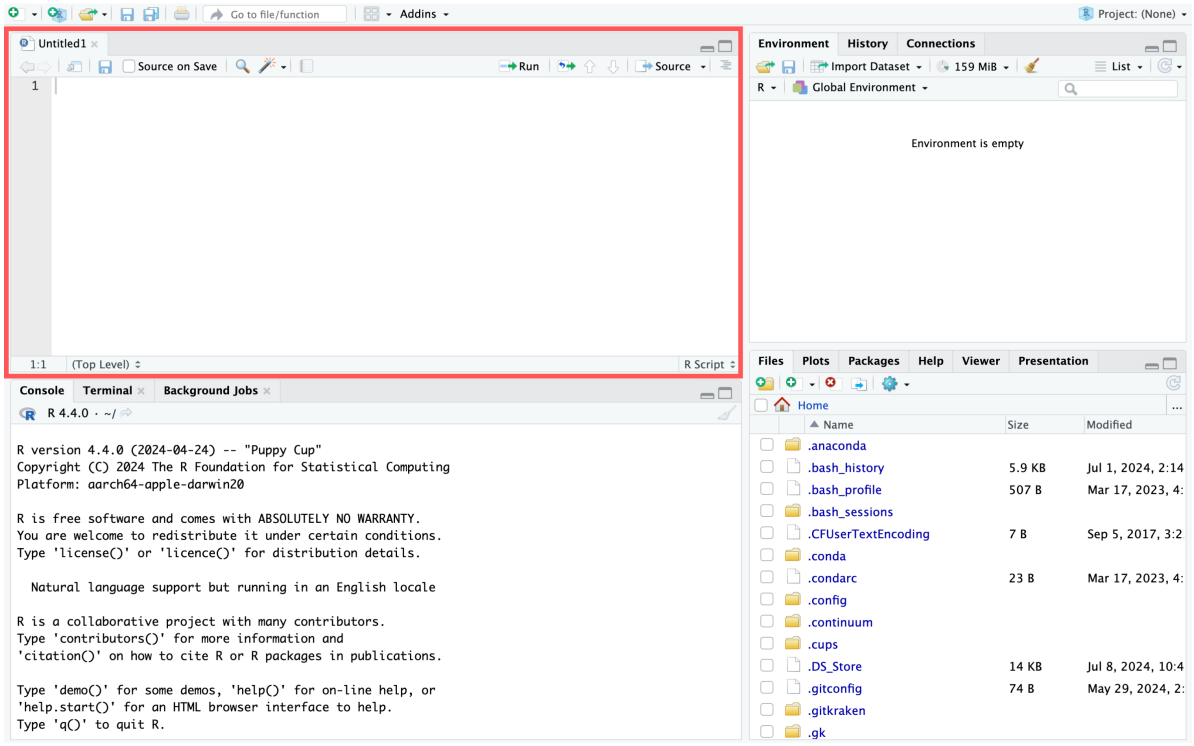


Figure 3.11: A blank R script in the source pane.

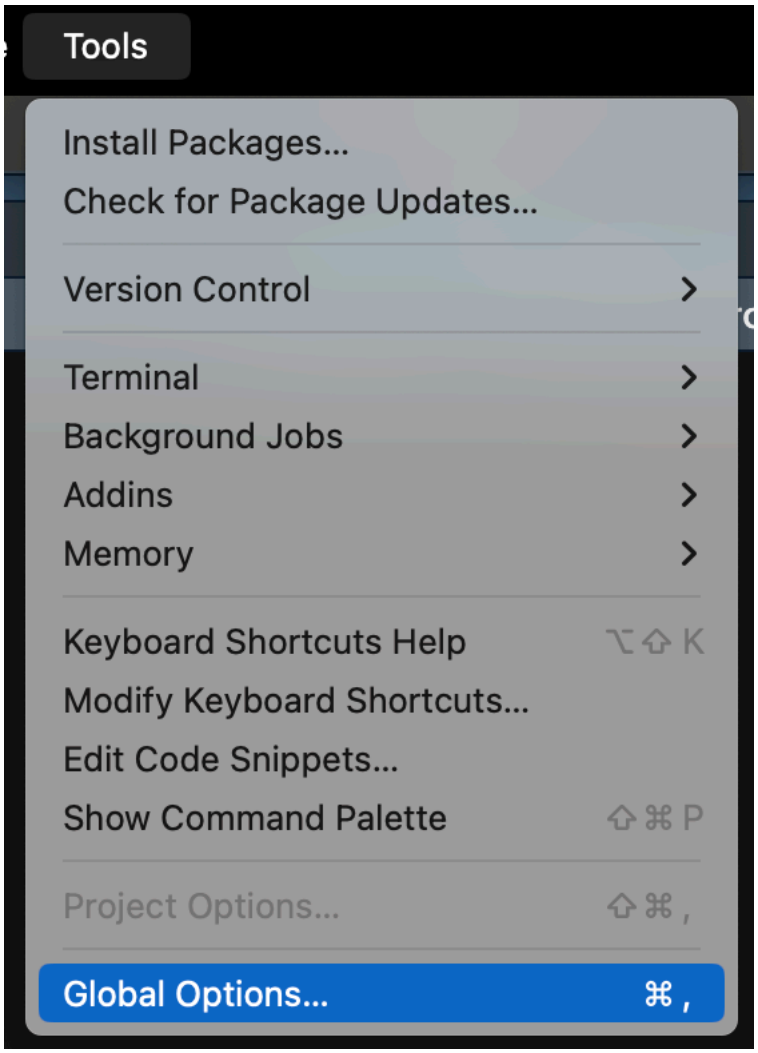


Figure 3.12: Select the preferences menu on Mac.

In the `General` tab, we recommend turning off the `Restore .Rdata into workspace at startup` option. We also recommend setting the `Save workspace .Rdata on exit` dropdown to `Never`. Finally, we recommend turning off the `Always save history (even when not saving .Rdata)` option.

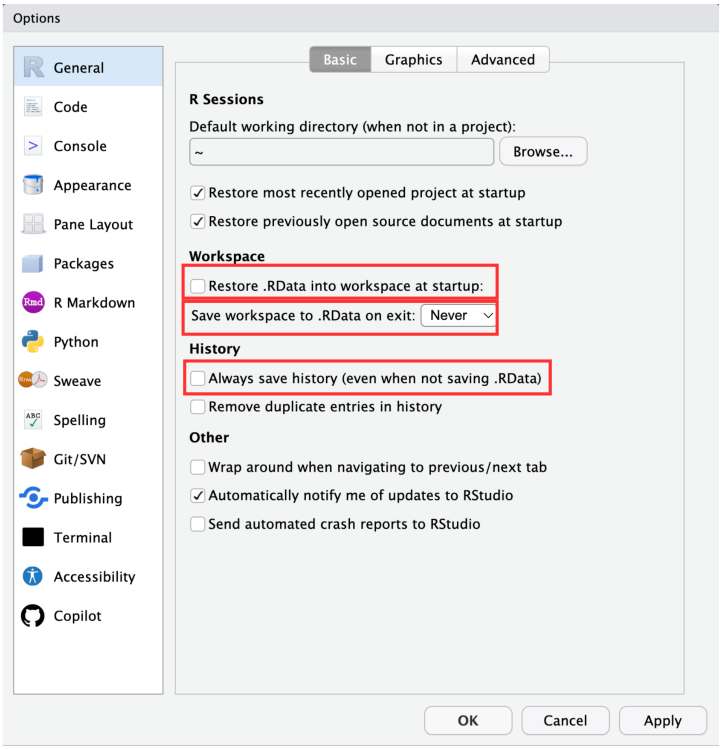


Figure 3.13: General options tab.

We change our editor theme to Twilight in the `Appearance` tab. We aren't necessarily recommending that you change your theme – this is entirely personal preference – we're just letting you know why our screenshots will look different from here on out.

It's likely that you still have lots of questions at this point. That's totally natural. However, we hope you now feel like you have some idea of what you are looking at when you open RStudio. Most of you will naturally get more comfortable with RStudio as we move through the book. For those of you who want more resources now, here are some suggestions.

1. RStudio IDE cheatsheet
2. ModernDive: What are R and RStudio?

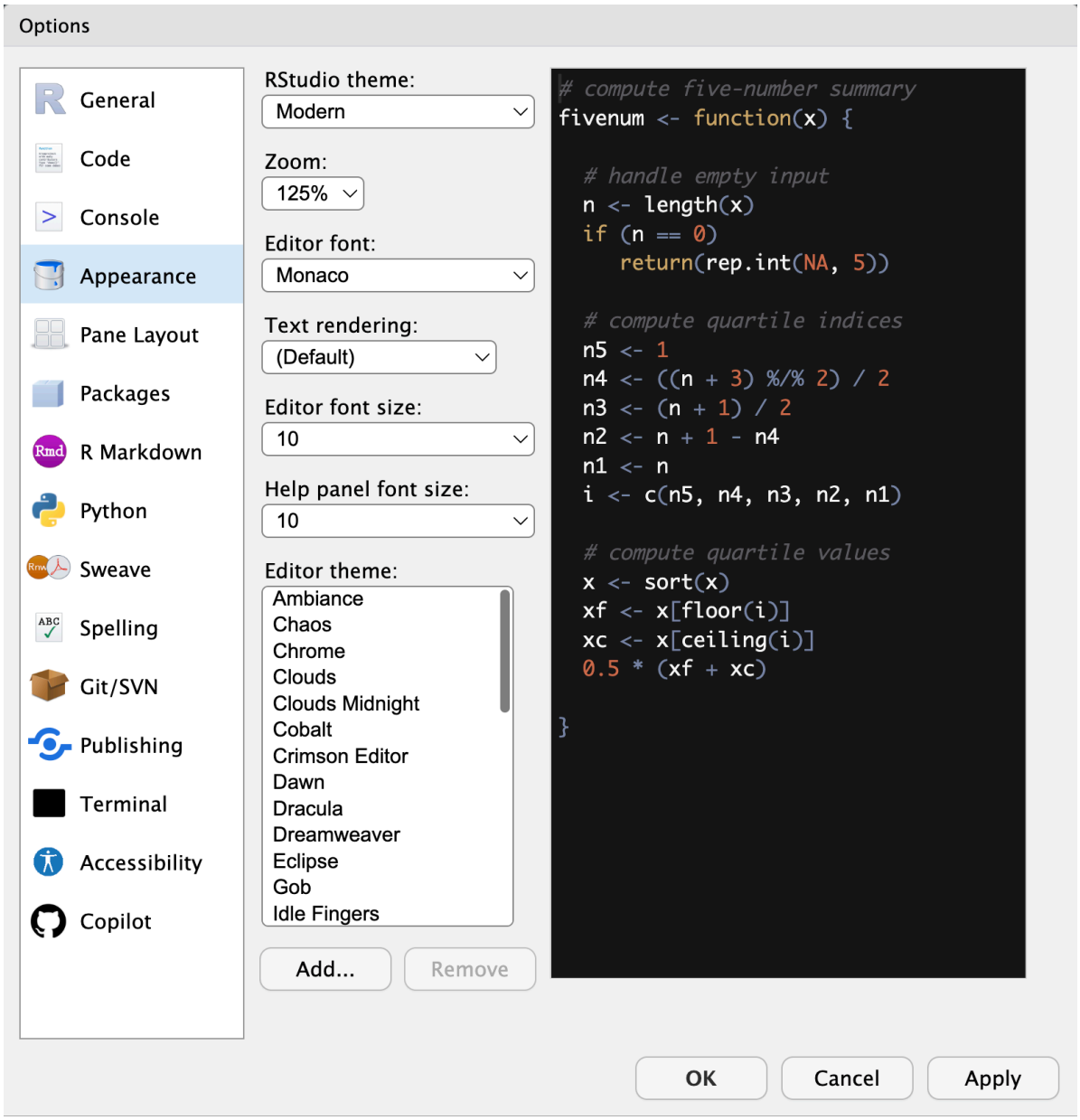


Figure 3.14: Appearance tab.

4 Speaking R's Language

It has been our experience that students often come into statistical programming courses thinking they will be heavy in math or statistics. In reality, our R courses are probably much closer to a foreign language course. There is no doubt that we need a foundational understanding of math and statistics to understand the results we get from R, but R will take care of most of the complicated stuff for us. We only need to learn how to ask R to do what we want it to do. To some extent, this entire book is about learning to communicate with R, but in this chapter we will briefly introduce the R programming language from the 30,000-foot level.

4.1 R is a *language*

In the same way that many people use the English language to communicate with each other, we will use the R programming language to communicate with R. Just like the English language, the R language comes complete with its own structure and vocabulary. Unfortunately, just like the English language, it also includes some weird exceptions and occasional miscommunications. We've already seen a couple examples of commands written to R in the R programming language. Specifically:

```
# Store the value 2 in the variable x
x <- 2
# Print the contents of x to the screen
x
```

```
[1] 2
```

and

```
# Print an example number sequence to the screen
seq(from = 2, to = 100, by = 2)
```

```
 [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96  98 100
```

> **Note**
>
> **Side Note:** The gray boxes you see above are called R code chunks and we created them (and this entire book) using something called Quarto files. Can you believe that you can write an entire book with R and RStudio? How cool is that? You will learn to use Quarto files later in this book. Quarto is great because it allows you to mix R code with narrative text and multimedia content as we've done throughout the page you're currently looking at. This makes it really easy for us to add context and aesthetic appeal to our results.

4.2 The R interpreter

Question: We keep talking about "speaking" to R, but when you speak to R using the R language, who are you actually speaking to?

Well, you are speaking to something called the **R interpreter**. The R interpreter takes the commands we've written in the R language, sends them to our computer to do the actual work (e.g., get the mean of a set of numbers), and then translates the results of that work back to us in a form that we humans can understand (e.g., the mean is 25.5). At this stage, one of the key concepts for you to understand about the R language is that is **extremely literal!** Understanding the literal nature of R is important because it will be the underlying cause of a lot of errors in our R code.

4.3 Errors

No matter what we write next, you are going to get errors in your R code. We still get errors in our R code every single time we write R code. However, our hope is that this section will help you begin to understand *why* you are getting errors when you get them and provide us with a common language for discussing errors.

So, what exactly do we mean when we say that the R interpreter is extremely literal? Well, in the Navigating RStudio chapter, we already told you that R is a **case sensitive** language. Again, that means that uppercase x (X) and lowercase x (x) are different things to R. So, if you assign 2 to lowercase x (`x <- 2`). And then later ask R to tell what number you stored in upper case X; you will get an error (`Error: object 'X' not found`).

```
x <- 2
X
```

```
Error in eval(expr, envir, enclos): object 'X' not found
```

Specifically, this is an example of a logic error. Meaning, R understands what you are *asking* it to do – you want it to print the contents of the uppercase X object to the screen. However, it can't complete your request because you are asking it to do something that doesn't logically make sense – print the contents of a thing that doesn't exist. Remember, R is literal and it will not try to guess that you actually *meant* to ask it to print the contents of lowercase x.

Another general type of error is known as a **syntax error**. In programming languages, syntax refers to the rules of the language. You can sort of think of this as the grammar of the language. In English, we could say something like, "giving dog water drink." This sentence is grammatically completely incorrect; however, most of you would roughly be able to figure out what we're asking you to do based on your life experience and knowledge of the situational context. The R interpreter, as awesome as it is, would not be able to make an assumption about what we want it to do. In this case, the R interpreter would say, "I don't know what you're asking me to do." When the R interpreter says, "I don't know what you're asking me to do," we've made a syntax error.

Throughout the rest of the book, we will try to point out situations where R programmers often encounter errors and how you may be able to address them. The remainder of this chapter will discuss some key components of R's syntax and the data structures (i.e., ways of storing data) that the R syntax interacts with.

4.4 Functions

R is a functional programming language, which simply means that functions play a central role in the R language. But what are functions? Well, factories are a common analogy used to represent functions. In this analogy, arguments are raw material inputs that go into the factory. For example, steel and rubber. The function is the factory where all the work takes place – converting raw materials into the desired output. Finally, the factory output represents the returned results. In this case, bicycles.

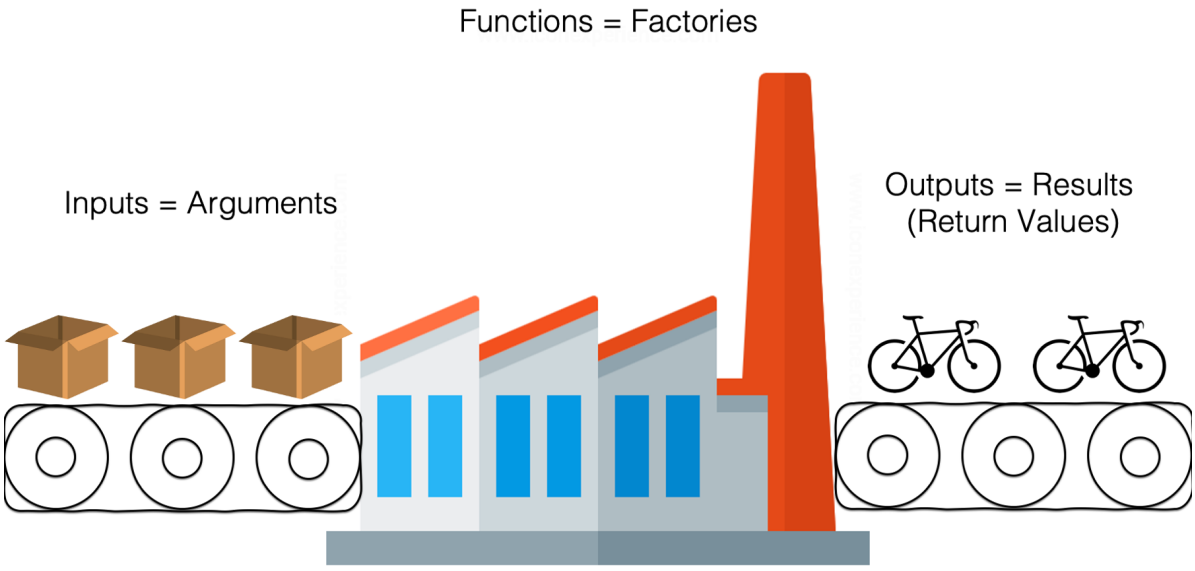


Figure 4.1: A factory making bicycles.

To make this concept more concrete, in the Navigating RStudio chapter we used the `seq()` function as a factory. Specifically, we wrote `seq(from = 2, to = 100, by = 2)`. The inputs (arguments) were `from`, `to`, and `by`. The output (returned result) was a set of numbers that went from 2 to 100 by 2's. Most functions, like the `seq()` function, will be a word or word part followed by parentheses. Other examples are the `sum()` function for addition and the `mean()` function to calculate the average value of a set of numbers.

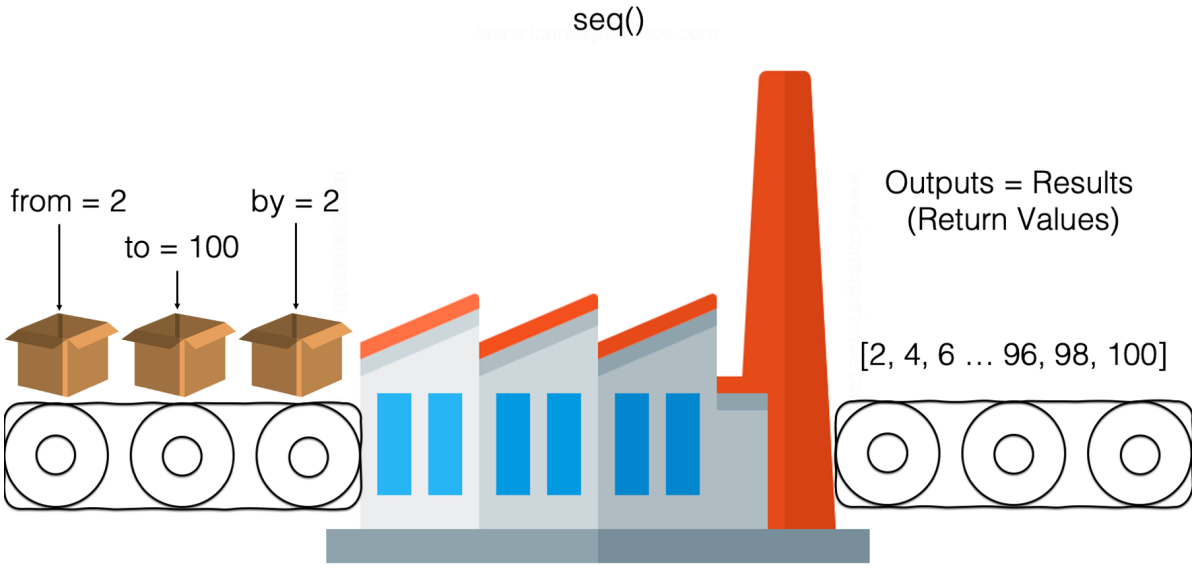


Figure 4.2: A function factory making numbers.

4.4.1 Passing values to function arguments

When we supply a value to a function argument, that is called "passing" a value to the argument. Let's take another look at the sequence function we previously wrote and use it to help us with this discussion.

```
# Create a sequence of numbers beginning at 2 and ending at 100, incremented by 2.
seq(from = 2, to = 100, by = 2)
```

```
 [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96  98 100
```

In the code above, we *passed* the value `2` to the `from` argument, we *passed* the value `100` to the `to` argument, and we *passed* the value `2` to the `by` argument. How do we know we passed the value `2` to the `from` argument? We know because we wrote `from = 2`. To R, this means "pass the value `2` to the `from` argument," and it is an example of passing a value *by name*. Alternatively, we could have also gotten the same result if we had passed the same values to the `seq()` function *by position*. What does that mean? We'll explain, but first take a look at the following R code.

```
# Create a sequence of numbers beginning at 2 and ending at 100, incremented by 2.
seq(2, 100, 2)
```

```
 [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96  98 100
```

How is code different from the code chunk before it? You got it! We didn't explicitly write the names of the function arguments inside of the `seq()` function. So, how did we get the same results? We got the same results because R allows us to pass values to function arguments by name *or* by position. When we pass values to a function *by position*, R will pass the first input value to the first function argument, the second input value to the second function argument, the third input value to the third function argument, and so on.

But how do we know what the first, second, and third arguments to a function are? Do you remember our discussion about RStudio's help tab in the previous chapter? There, we saw the documentation for the `seq()` function.

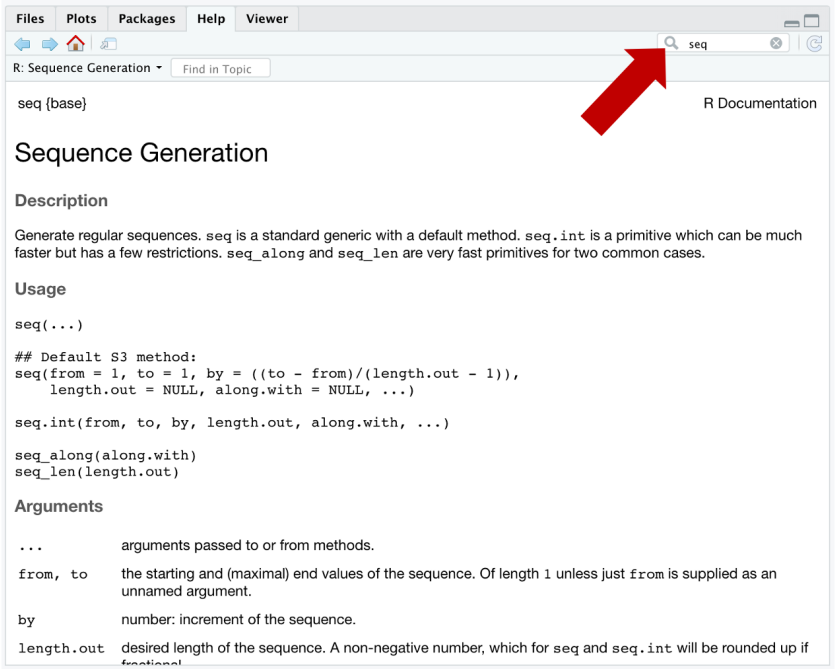


Figure 4.3: The help tab.

In the "Usage" section of the documentation for the `seq()` function, we can see that all of the arguments that the `seq()` function accepts. These documentation files are a little cryptic

until you get used to them but look directly underneath the part that says "## Default S3 method." There, it tells us that the `seq()` function understands the `from`, `to`, `by`, `length.out`, `along.with`, and `...` arguments. The `from` argument is first argument to the `seq()` function because it is listed there first, the `to` argument is second argument to the `seq()` function because it is listed there second, and so on. It is really that simple. Therefore, when we type `seq(2, 100, 2)`, R automatically translates it to `seq(from = 2, to = 100, by = 2)`. And this is called passing values to function arguments by position.

> **Note**
>
> **Side Note:** As an aside, we can view the documentation for any function by typing `?function name` into the R console and then pressing the enter/return key. For example, we can type `?seq` to view the documentation for the `seq()` function.

Passing values to our functions by position has the benefit of making our code more compact, we don't have to write out all the function names. But, as you might have already guessed, passing values to our functions by position also has some potential risks. First, it makes our code harder to read. If we give our code to someone who has never used the `seq()` function before, they will have to guess (or look up) what purpose 2, 100, and 2 serve. When we pass the values to the function by name, their purpose is typically easier to figure out even if we've never used a particular function before. The second, and potentially more important, risk is that we may accidentally pass a value to a different argument than the one we intended. For example, what if we mistakenly think the order of the arguments to the `seq()` function is `from`. `by`, `to`? In that case, we might write the following code:

```
# Create a sequence of numbers beginning at 2 and ending at 100, incremented by 2.
seq(2, 2, 100)
```

```
[1] 2
```

Notice that R still gives us a result, but it isn't the result we want! What happened? Well, we passed the values 2, 2, and 100 to the `seq()` function *by position*, which R translated to `seq(from = 2, to = 2, by = 100)` because `from` is the first argument in the `seq()` function, `to` is the second argument in the `seq()` function, and `by` is the third argument in the `seq()` function.

Quick review: is this an example of a syntax error or a logic error?

This is a logic error. We used perfectly valid R syntax in the code above, but we mistakenly asked R to do something different than we actually wanted it to do. In this simple example, it's easy to see that this result is very different than what we were expecting and try to figure out what we did wrong. But that won't always be the case. Therefore, we need to be really careful when passing values to function arguments by position.

One final note on passing values to functions. When we pass values to R functions *by name*, we can pass them in any order we want. For example:

```
# Create a sequence of numbers beginning at 2 and ending at 100, incremented by 2.
seq(from = 2, to = 100, by = 2)
```

```
 [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96  98 100
```

and

```
# Create a sequence of numbers beginning at 2 and ending at 100, incremented by 2.
seq(to = 100, by = 2, from = 2)
```

```
 [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96  98 100
```

return the exact same values. Why? Because we explicitly told R which argument to pass each value to *by name*. Of course, just because we *can* do something doesn't mean we *should* do it. We really shouldn't rearrange argument order like this unless there is a good reason.

4.5 Objects

In addition to functions, the R programming language also includes objects. In the Navigating RStudio chapter we created an object called `x` with a value of 2 using the `x <- 2` R code. In general, you can think of objects as anything that lives in your R global environment. Objects may be single variables (also called vectors in R) or entire data sets (also called data frames in R).

Objects can be a confusing concept at first. We think it's because it is hard to precisely define exactly what an object is. We'll say two things about this. First, you're probably overthinking it (because we've overthought it too). When we use R, we create and save stuff. We have to call that stuff something in order to talk about it or write books about it. Somebody decided we would call that stuff "objects." The second thing we'll say is that this becomes much less abstract when we finally get to a place where you can really get your hands dirty doing some R programming.

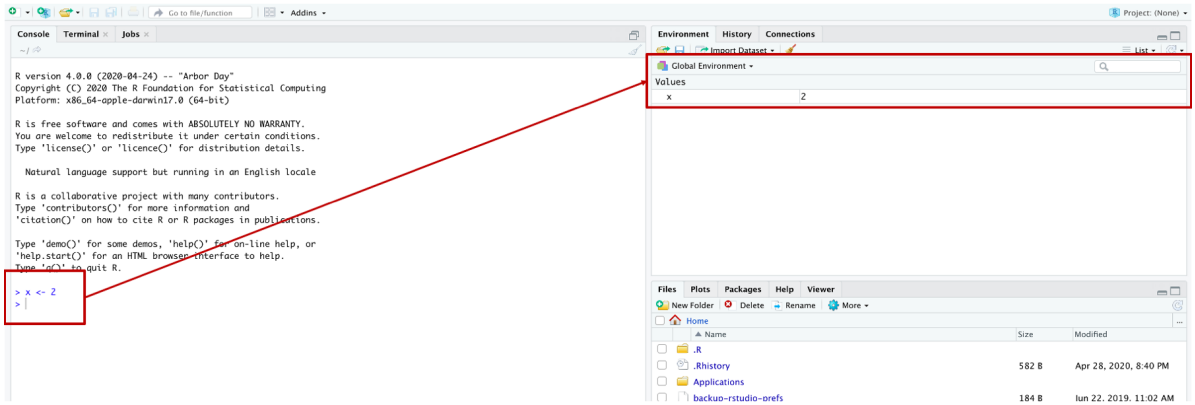
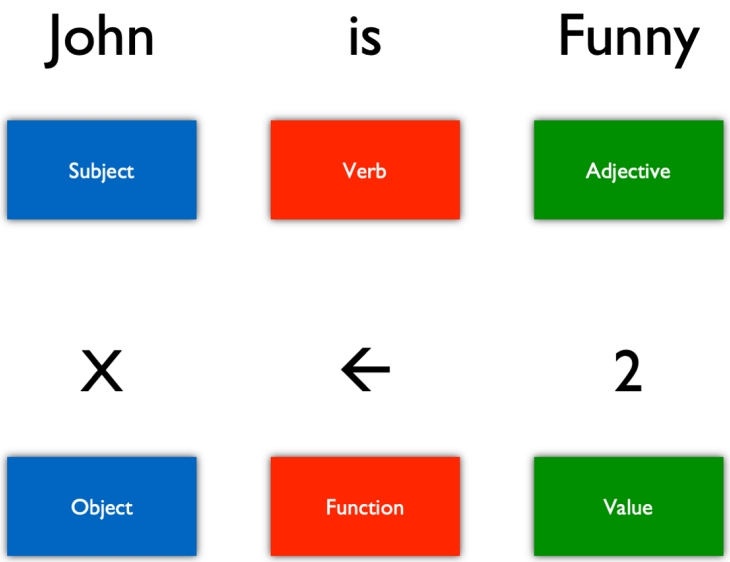


Figure 4.4: Creating the x object.

Sometimes it can be useful to relate the R language to English grammar. That is, when you are writing R code you can roughly think of functions as verbs and objects as nouns. Just like nouns *are* things in the English language, and verbs *do* things in the English language, objects *are* things and functions *do* things in the R language.

So, in the `x <- 2` command `x` is the object and `<-` is the function. "Wait! Didn't you just tell us that functions will be a word followed by parentheses?" Fair question. Technically, we said, "*Most* functions will be a word, or word part, followed by parentheses." Just like English, R has exceptions. All **operators** in R are also functions. Operators are symbols like `+`, `-`, `=`, and `<-`. There are many more operators, but you will notice that they all *do* things. In this case, they add, subtract, and assign values to objects.



4.6 Comments

And finally, there are comments. If our R code is a conversation we are having with the R interpreter, then comments are your inner thoughts taking place during the conversation. Comments don't actually mean anything to R, but they will be extremely important for you. You actually already saw a couple examples of comments above.

```
# Store the value 2 in the variable x
x <- 2
# Print the contents of x to the screen
x
```

```
[1] 2
```

In this code chunk, "# Store the value 2 in the variable x" and "# Print the contents of x to the screen" are both examples of comments. Notice that they both start with the pound or hash sign (#). The R interpreter will ignore anything on the *current line* that comes after the hash sign. A carriage return (new line) ends the comment. However, comments don't have to be written on their own line. They can also be written on the same line as R code as long as put them after the R code, like this:

```
x <- 2 # Store the value 2 in the variable x
x      # Print the contents of x to the screen
```

```
[1] 2
```

Most beginning R programmers underestimate the importance of comments. In the silly little examples above, the comments are not that useful. However, comments will become extremely important as you begin writing more complex programs. When working on projects, you will often need to share your programs with others. Reading R code without any context is really challenging – even for experienced R programmers. Additionally, even if your collaborators can surmise *what* your R code is doing, they may have no idea *why* you are doing it. Therefore, your comments should tell others what your code does (if it isn't completely obvious), and more importantly, what your code is trying to accomplish. Even if you aren't sharing your code with others, you may need to come back and revise or reuse your code months or years down the line. You may be shocked at how foreign the code *you wrote* will seem months or years after you wrote it. Therefore, comments are not just important for others, they are also important for future you!

> **Note**
>
> **Side Note:** RStudio has a handy little keyboard shortcut for creating comments. On a Mac, type shift + command + C. On Windows, Shift + Ctrl + C.

> **Note**
>
> **Side Note:** Please put a space in between the pound/hash sign and the rest of your text when writing comments. For example, `# here is my comment` instead of `#here is my comment`. It just makes the comment easier to read.

4.7 Packages

In addition to being a functional programming language, R is also a type of programming language called an open source programming language. For our purposes, this has two big advantages. First, it means that R is **FREE!** Second, it means that smart people all around the world get to develop new **packages** for the R language that can do cutting edge and/or very niche things.

That second advantage is probably really confusing if this is not a concept you are already familiar with. For example, when you install Microsoft Word on your computer all the code that makes that program work is owned and Maintained by the Microsoft corporation. If you

need Word to do something that it doesn't currently do, your only option is to make a feature request on Microsoft's website. Microsoft may or may not every get around to fulfilling that request.

R works a little differently. When you downloaded R from the CRAN website, you actually downloaded something called **Base R**. Base R is maintained by the R Core Team. However, anybody – *even you* – can write your own code (called packages) that add new functions to the R syntax. Like all functions, these new functions allow you to *do* things that you can't do (or can't do as easily) with Base R.

An analogy that we really like here is used by Ismay and Kim in ModernDive.

> A good analogy for R packages is they are like apps you can download onto a mobile phone. So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn't have everything. R packages are like the apps you can download onto your phone from Apple's App Store or Android's Google Play.[1]

So, when you get a new smart phone it comes with apps for making phone calls, checking email, and sending text messages. But, what if you want to listen to music on Spotify? You may or may not be able to do that through your phone's web browser, but it's way more convenient and powerful to download and install the Spotify app.

In this course, we will make extensive use of packages developed by people and teams outside of the R Core Team. In particular, we will use a number of related packages that are collectively known as the Tidyverse. One of the most popular packages in the tidyverse collection (and one of the most popular R packages overall) is called the `dplyr` package for data management.

In the same way that you have to download and install Spotify on your mobile phone before you can use it, you have to download and install new R packages on your computer before you can use the functions they contain. Fortunately, R makes this really easy. For most packages, all you have to do is run the `install.packages()` function in the R console. For example, here is how you would install the `dplyr` package.

```
# Make sure you remember to wrap the name of the package in single or double quotes.
install.packages("dplyr")
```

Over time, you will download and install a lot of different packages. All those packages with all of those new functions start to create a lot of overhead. Therefore, R doesn't keep them loaded and available for use at all times. Instead, *every time* you open RStudio, you will have to explicitly tell R which packages you want to use. So, when you close RStudio and open it again, the only functions that you will be able to use are Base R functions. If you want to use functions from any other package (e.g., `dplyr`) you will have to tell R that you want to do so using the `library()` function.

```
# No quotes needed here
library(dplyr)
```

Technically, loading the package with the `library()` function is not the only way to use a function from a package you've downloaded. For example, the `dplyr` package contains a function called `filter()` that helps us keep or drop certain rows in a data frame. To use this function, we have to first download the `dplyr` package. Then we can use the filter function in one of two different ways.

```
library(dplyr)
filter(states_data, state == "Texas") # Keeps only the rows from Texas
```

The first way you already saw above. Load all the functions contained in the `dplyr` package using the `library()` function. Then use that function just like any other Base R function.

The second way is something called the **double colon syntax**. To use the double colon syntax, you type the package name, two colons, and the name of the function you want to use from the package. Here is an example of the double colon syntax.

```
dplyr::filter(states_data, state == "Texas") # Keeps only the rows from Texas
```

Most of the time you will load packages using the `library()` function. However, we wanted to show you the double colon syntax because you may come across it when you are reading R documentation and because there are times when it makes sense to use this syntax.

4.8 Programming style

Finally, we want to discuss programming style. R can read any code you write as long as you write it using valid R syntax. However, R code can be much easier or harder for people (including you) to read depending on how it's written. The coding best practices chapter of this book gives complete details on writing R code that is as easy as possible for *people* to read. So, please make sure to read it. It will make things so much easier for all of us!

5 Let's Get Programming

In this chapter, we are going to tie together many of the concepts we've learned so far, and you are going to create your first basic R program. Specifically, you are going to write a program that simulates some data and analyzes it.

5.1 Simulating data

Data simulation can be really complicated, but it doesn't have to be. It is simply the process of *creating* data as opposed to *finding data in the wild*. This can be really useful in several different ways.

1. Simulating data is really useful for getting help with a problem you are trying to solve. Often, it isn't feasible for you to send other people the actual data set you are working on when you encounter a problem you need help with. Sometimes, it may not even be legally allowed (i.e., for privacy reasons). Instead of sending them your entire data set, you can simulate a little data set that recreates the challenge you are trying to address without all the other complexity of the full data set. As a bonus,we have often found that we end up figuring out the solution to the problem we're trying to solve as we recreate the problem in a simulated data set that we intended to share with others.
2. Simulated data can also be useful for learning about and testing statistical assumptions. In epidemiology, we use statistics to draw conclusions about populations of people we are interested in based on samples of people drawn from the population. Because we don't actually have data from *all* the people in the population, we have to make some assumptions about the population based on what we find in our sample. When we simulate data, we know the truth about our population because we *created* our population to have that truth. We can then use this simulated population to play "what if" games with our analysis. *What if we only sampled half as many people? What if their heights aren't actually normally distributed? What if we used a probit model instead of a logit model?* Going through this process and answering these questions can help us understand how much, and under what circumstances, we can trust the answers we found in the real world.

So, let's go ahead and write a complete R program to simulate and analyze some data. As we said, it doesn't have to be complicated. In fact, in just a few lines of R code below we simulate and analyze some data about a hypothetical class.

```r
class <- data.frame(
  names   = c("John", "Sally", "Brad", "Anne"),
  heights = c(68, 63, 71, 72)
)
```

```r
class
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      72
```

```r
mean(class$heights)
```

```
[1] 68.5
```

As you can see, this data frame contains the students' names and heights. We also use the `mean()` function to calculate the average height of the class. By the end of this chapter, you will understand all the elements of this R code and how to simulate your own data.

5.2 Vectors

Vectors are the most fundamental data structure in R. Here, data structure means "container for our data." There are other data structures as well; however, they are all built from vectors. That's why we say vectors are the most fundamental data structure. Some of these other structures include matrices, lists, and data frames. In this book, we won't use matrices or lists much at all, so you can forget about them for now. Instead, we will almost exclusively use data frames to hold and manipulate our data. However, because data frames are built from vectors, it can be useful to start by learning a little bit about them. Let's create our first vector now.

```r
# Create an example vector
names <- c("John", "Sally", "Brad", "Anne")
# Print contents to the screen
names
```

```
[1] "John"  "Sally" "Brad"  "Anne"
```

Here's what we did above:

- We *created* a vector of names with the `c()` (short for combine) function.
  - The vector contains four values: "John", "Sally", "Brad", and "Anne".
  - All of the values are character strings (i.e., words). We know this because all of the values are wrapped with quotation marks.
  - Here we used double quotes above, but we could have also used single quotes. We cannot, however, mix double and single quotes for each character string. For example, `c("John', ...)` won't work.
- We *assigned* that vector of character strings to the word `names` using the `<-` function.
  - R now recognizes `names` as an **object** that we can do things with.
  - R programmers may refer to the names object as "the names object", "the names vector", or "the names variable". For our purposes, these all mean the same thing.
- We *printed* the contents of the `names` object to the screen by typing the word "names".
  - R **returns** (shows us) the four character values ("John" "Sally" "Brad" "Anne") on the computer screen.

Try copying and pasting the code above into the RStudio console on your computer. You should notice the names vector appear in your **global environment**. You may also notice that the global environment pane gives you some additional information about this vector to the right of its name. Specifically, you should see `chr [1:4] "John" "Sally" "Brad" "Anne"`. This is R telling us that `names` is a character vector (`chr`), with four values (`[1:4]`), and the first four values are `"John" "Sally" "Brad" "Anne"`.

5.2.1 Vector types

There are several different vector **types**, but each vector can have only one type. The type of the vector above was character. We can validate that with the `typeof()` function like so:

```
typeof(names)
```

```
[1] "character"
```

The other vector types that we will use in this book are double, integer, and logical. Double vectors hold real numbers and integer vectors hold integers. Collectively, double vectors and integer vectors are known as numeric vectors. Logical vectors can only hold the values TRUE and FALSE. Here are some examples of each:

5.2.2 Double vectors

```
# A numeric vector
my_numbers <- c(12.5, 13.98765, pi)
my_numbers
```

```
[1] 12.500000 13.987650  3.141593
```

```
typeof(my_numbers)
```

```
[1] "double"
```

5.2.3 Integer vectors

Creating integer vectors involves a weird little quirk of the R language. For some reason, and we have no idea why, we must type an "L" behind the number to make it an integer.

```
# An integer vector - first attempt
my_ints_1 <- c(1, 2, 3)
my_ints_1
```

```
[1] 1 2 3
```

```
typeof(my_ints_1)
```

```
[1] "double"
```

```
# An integer vector - second attempt
# Must put "L" behind the number to make it an integer. No idea why they chose "L".
my_ints_2 <- c(1L, 2L, 3L)
my_ints_2
```

```
[1] 1 2 3
```

```
typeof(my_ints_2)
```

```
[1] "integer"
```

5.2.4 Logical vectors

```
# A logical vector
# Type TRUE and FALSE in all caps
my_logical <- c(TRUE, FALSE, TRUE)
my_logical
```

```
[1]  TRUE FALSE  TRUE
```

```
typeof(my_logical)
```

```
[1] "logical"
```

Rather than have an abstract discussion about the particulars of each of these vector types right now, we think it's best to wait and learn more about them when they naturally arise in the context of a real challenge we are trying to solve with data. At this point, just having some vague idea that they exist is good enough.

5.2.5 Factor vectors

Above, we said that we would only work with three vector types in this book: double, integer, and logical. Technically, that is true. Factors aren't technically a vector type (we will explain below) but calling them a vector type is close enough to true for our purposes. We will briefly introduce you to factors here, and then discuss them in more depth later in the chapter on [Numerical Descriptions of Categorical Variables]. We cover them in greater depth there because factors are most useful in the context of working with categorical data – data that is grouped into discrete categories. Some examples of categorical variables commonly seen in public health data are sex, race or ethnicity, and level of educational attainment.

In R, we can represent a categorical variable in multiple different ways. For example, let's say that we are interested in recording people's highest level of formal education completed in our data. The discrete categories we are interested in are:

- 1 = Less than high school
- 2 = High school graduate
- 3 = Some college
- 4 = College graduate

We could then create a numeric vector to record the level of educational attainment for four hypothetical people as shown below.

```
# A numeric vector of education categories
education_num <- c(3, 1, 4, 1)
education_num
```

```
[1] 3 1 4 1
```

But what is less-than-ideal about storing our categorical data this way? Well, it isn't obvious what the numbers in `education_num` mean. For the purposes of this example, we defined them above, but if we didn't have that information then we would likely have no idea what categories the numbers represent.

We could also create a character vector to record the level of educational attainment for four hypothetical people as shown below.

```
# A character vector of education categories
education_chr <- c(
  "Some college", "Less than high school", "College graduate",
  "Less than high school"
)
education_chr
```

```
[1] "Some college"          "Less than high school" "College graduate"
[4] "Less than high school"
```

But this strategy also has a few limitations that we will discuss in in the chapter on [Numerical Descriptions of Categorical Variables]. For now, we just need to quickly learn how to create and identify factor vectors.

Typically, we don't *create* factors from scratch. Instead, we typically convert (or "coerce") an existing numeric or character vector into a factor. For example, we can coerce `education_num` to a factor like this:

```
# Coerce education_num to a factor
education_num_f <- factor(
  x      = education_num,
  levels = 1:4,
  labels = c(
    "Less than high school", "High school graduate", "Some college",
    "College graduate"
```

```
  )
)
education_num_f
```

```
[1] Some college          Less than high school College graduate
[4] Less than high school
4 Levels: Less than high school High school graduate ... College graduate
```

Here's what we did above:

- We used the `factor()` function to create a new factor version of `education_num`.
  - You can type `?factor` into your R console to view the help documentation for this function and follow along with the explanation below.
  - The first argument to the `factor()` function is the `x` argument. The value passed to the `x` argument should be a vector of data. We passed the `education_num` vector to the `x` argument.
  - The second argument to the `factor()` function is the `levels` argument. This argument tells R the unique values that the new factor variable can take. We used the shorthand `1:4` to tell R that `education_num_f` can take the unique values 1, 2, 3, or 4.
  - The third argument to the `factor()` function is the `labels` argument. The value passed to the `labels` argument should be a character vector of labels (i.e., descriptive text) for each value in the `levels` argument. The order of the labels in the character vector we pass to the `labels` argument should match the order of the values passed to the `levels` argument. For example, the ordering of `levels` and `labels` above tells R that `1` should be labeled with "Less than high school", `2` should be labeled with "High school graduate", etc.
- We used the assignment operator (`<-`) to save our new factor vector in our global environment as `education_num_f`.
  - If we had used the name `education_num` instead, then the previous values in the `education_num` vector would have been replaced with the new values. That is sometimes what we want to happen. However, when it comes to creating factors, we typically keep the numeric version of the vector and create an additional factor version of the vector. We just often find that it can be useful to have both versions of the variable hanging around during the analysis process.
  - We also use the `_f` naming convention in our code. That means that when we create a new factor vector, we name it the same thing the original vector was named with the addition of `_f` (for factor) at the end.

- We printed the vector to the screen. The values in `education_num_f` *look* similar to the character strings displayed in `education_chr`. Notice, however, that the values no longer have quotes around them and R displays `Levels: Less than high school High school graduate Some college College graduate` below the data values. This is R telling us the *possible* categorical values that this factor could take on. This is a telltale sign that the vector being printed to the screen is a factor.

Interestingly, although R uses labels to make factors *look* like character vectors, they are still integer vectors under the hood. For example:

```
typeof(education_num_f)
```

```
[1] "integer"
```

And we can still view them as such.

```
as.numeric(education_num_f)
```

```
[1] 3 1 4 1
```

It is also possible to coerce character vectors to factors. For example, we can coerce `education_chr` to a factor like so:

```
# Coerce education_chr to a factor
education_chr_f <- factor(
  x      = education_chr,
  levels = c(
    "Less than high school", "High school graduate", "Some college",
    "College graduate"
  )
)
education_chr_f
```

```
[1] Some college          Less than high school College graduate
[4] Less than high school
4 Levels: Less than high school High school graduate ... College graduate
```

Here's what we did above:

- We coerced a character vector (`education_chr`) to a factor using the `factor()` function.

- Because the levels *are* character strings, there was no need to pass any values to the `labels` argument this time. Keep in mind, though, that the order of the values passed to the `levels` argument matters. It will be the order that the factor levels will be displayed in our analyses.

You might reasonably wonder why we would want to convert character vectors to factors, but we will save that discussion for the chapter on [Numerical Descriptions of Categorical Variables].

5.3 Data frames

Vectors are useful for storing a single characteristic where all the data is of the same type. However, in epidemiology, we typically want to store information about many different characteristics of whatever we happen to be studying. For example, we didn't just want the names of the people in our class, we also wanted the heights. Of course, we can also store the heights in a vector like so:

```
heights <- c(68, 63, 71, 72)
heights
```

```
[1] 68 63 71 72
```

But this vector, in and of itself, doesn't tell us which height goes with which person. When we want to create relationships between our vectors, we can use them to build a data frame. For example:

```
# Create a vector of names
names <- c("John", "Sally", "Brad", "Anne")
# Create a vector of heights
heights <- c(68, 63, 71, 72)
# Combine them into a data frame
class <- data.frame(names, heights)
# Print the data frame to the screen
class
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      72
```

Here's what we did above:

- We *created* a data frame with the `data.frame()` function.
  - The first argument we passed to the `data.frame()` function was a vector of names that we previously created.
  - The second argument we passed to the `data.frame()` function was a vector of heights that we previously created.
- We *assigned* that data frame to the word `class` using the `<-` function.
  - R now recognizes `class` as an **object** that we can do things with.
  - R programmers may refer to this class object as "the class object" or "the class data frame". For our purposes, these all mean the same thing. We could also call it a data set, but that term isn't used much in R circles.
- We *printed* the contents of the `class` object to the screen by typing the word "class".
  - R **returns** (shows us) the data frame on the computer screen.

Try copying and pasting the code above into the RStudio console on your computer. You should notice the `class` data frame appear in your **global environment**. You may also notice that the global environment pane gives you some additional information about this data frame to the right of its name. Specifically, you should see `4 obs. of 2 variables`. This is R telling us that `class` has four rows or observations (`4 obs.`) and two columns or variables (`2 variables`). If you click the little blue arrow to the left of the data frame's name, you will see information about the individual vectors that make up the data frame.

As a shortcut, instead of creating individual vectors and then combining them into a data frame as we've done above, most R programmers will create the vectors (columns) directly inside of the data frame function like this:

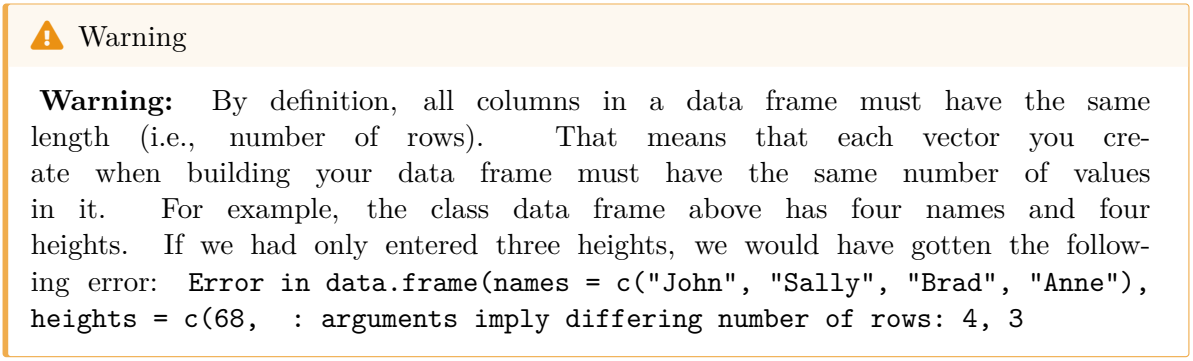
```
# Create the class data frame
class <- data.frame(
  names   = c("John", "Sally", "Brad", "Anne"),
  heights = c(68, 63, 71, 72)
) # Closing parenthesis down here.

# Print the data frame to the screen
class
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      72
```

As you can see, both methods produce the exact same result. The second method, however, requires a little less typing and results in fewer objects cluttering up your global environment. What we mean by that is that the `names` and `heights` vectors won't exist independently in your global environment. Rather, they will only exist as columns of the `class` data frame.

You may have also noticed that when we created the `names` and `heights` vectors (columns) directly inside of the `data.frame()` function we used the equal sign (`=`) to assign values instead of the assignment arrow (`<-`). This is just one of those quirky R exceptions we talked about in the chapter on speaking R's language. In fact, `=` and `<-` can be used interchangeably in R. It is only by convention that we usually use `<-` for assigning values, but use `=` for assigning values to columns in data frames. we don't know why this is the convention. If it were up to me, we wouldn't do this. We would just pick `=` or `<-` and use it in all cases where we want to assign values. But, it isn't up to me and we gave up on trying to fight it a long time ago. Your R programming life will be easier if you just learn to assign values this way – even if it's dumb.



5.4 Tibbles

Tibbles are a data structure that come from another tidyverse package – the `tibble` package. Tibbles *are* data frames and serve the same purpose in R that data frames serve; however, they are enhanced in several ways. You are welcome to look over the tibble documentation or the tibbles chapter in R for Data Science if you are interested in learning about all the differences between tibbles and data frames. For our purposes, there are really only a couple things we want you to know about tibbles right now.

First, tibbles are a part of the `tibble` package – NOT base R. Therefore, we have to install and load either the `tibble` package or the `dplyr` package (which loads the tibble package for us behind the scenes) before we can create tibbles. we typically just load the `dplyr` package.

```
# Install the dplyr package. YOU ONLY NEED TO DO THIS ONE TIME.
install.packages("dplyr")
```

```
# Load the dplyr package. YOU NEED TO DO THIS EVERY TIME YOU START A NEW R SESSION.
library(dplyr)
```

Second, we can create tibbles using one of three functions: `as_tibble()`, `tibble()`, or `tribble()`. I'll show you some examples shortly.

Third, try not to be confused by the terminology. Remember, tibbles *are* data frames. They are just enhanced data frames.

5.4.1 The as_tibble function

We use the `as_tibble()` function to turn an already existing basic data frame into a tibble. For example:

```
# Create a data frame
my_df <- data.frame(
  name = c("john", "alexis", "Steph", "Quiera"),
  age  = c(24, 44, 26, 25)
)

# Print my_df to the screen
my_df
```

```
    name age
1   john  24
2 alexis  44
3  Steph  26
4 Quiera  25
```

```
# View the class of my_df
class(my_df)
```

```
[1] "data.frame"
```

Here's what we did above:

- We used the `data.frame()` function to create a new data frame called `my_df`.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is a data frame.

```
# Use as_tibble() to turn my_df into a tibble
my_df <- as_tibble(my_df)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 2
  name     age
  <chr>  <dbl>
1 john      24
2 alexis    44
3 Steph     26
4 Quiera    25
```

```
# View the class of my_df
class(my_df)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Here's what we did above:

- We used the `as_tibble()` function to turn `my_df` into a tibble.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is still a data frame, but it is also a tibble. That's what "tbl_df" and "tbl" mean.

5.4.2 The tibble function

We can use the `tibble()` function in place of the `data.frame()` function when we want to create a tibble from scratch. For example:

```r
# Create a data frame
my_df <- tibble(
  name = c("john", "alexis", "Steph", "Quiera"),
  age  = c(24, 44, 26, 25)
)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 2
  name     age
  <chr>  <dbl>
1 john      24
2 alexis    44
3 Steph     26
4 Quiera    25
```

```r
# View the class of my_df
class(my_df)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Here's what we did above:

- We used the `tibble()` function to create a new tibble called `my_df`.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is still a data frame, but it is also a tibble. That's what "tbl_df" and "tbl" mean.

5.4.3 The tribble function

Alternatively, we can use the `tribble()` function in place of the `data.frame()` function when we want to create a tibble from scratch. For example:

```r
# Create a data frame
my_df <- tribble(
  ~name,    ~age,
  "john",   24,
  "alexis", 44,
```

```
  "Steph",  26,
  "Quiera", 25
)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 2
  name     age
  <chr>  <dbl>
1 john      24
2 alexis    44
3 Steph     26
4 Quiera    25
```

```
# View the class of my_df
class(my_df)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Here's what we did above:

- We used the `tribble()` function to create a new tibble called `my_df`.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is still a data frame, but it is also a tibble. That's what "tbl_df" and "tbl" mean.
- There is absolutely no difference between the tibble we created above with the `tibble()` function and the tibble we created above with the `tribble()` function. The only difference between the two functions is the syntax we used to pass the column names and data values to each function.
  - When we use the `tibble()` function, we pass the data values to the function horizontally as vectors. This is the same syntax that the `data.frame()` function expects us to use.
  - When we use the `tribble()` function, we pass the data values to the function vertically instead. The only reason this function exists is because it can sometimes be more convenient to type in our data values this way. That's it.
  - Remember to type a tilde ("~") in front of your column names when using the `tribble()` function. For example, type `~name` instead of `name`. That's how R knows you're giving it a column name instead of a data value.

5.4.4 Why use tibbles

At this point, some students wonder, "If tibbles are just data frames, why use them? Why not just use the `data.frame()` function?" That's a fair question. As we have said multiple times already, tibbles are enhanced. However, we don't believe that going into detail about those enhancements is going to be useful to most of you at this point – and may even be confusing. But, we will show you one quick example that's pretty self-explanatory.

Let's say that we are given some data that contains four people's age in years. We want to create a data frame from that data. However, let's say that we also want a column in our new data frame that contains those same ages in months. Well, we could do the math ourselves. We could just multiply each age in years by 12 (for the sake of simplicity, assume that everyone's age in years is gathered on their birthday). But, we'd rather have R do the math for us. We can do so by asking R to multiply each value of the the column called `age_years` by 12. Take a look:

```
# Create a data frame using the data.frame() function
my_df <- data.frame(
  name       = c("john", "alexis", "Steph", "Quiera"),
  age_years  = c(24, 44, 26, 25),
  age_months = age_years * 12
)
```

```
Error in eval(expr, envir, enclos): object 'age_years' not found
```

Uh, oh! We got an error! This error says that the column `age_years` can't be found. How can that be? We are clearly passing the column name `age_years` to the `data.frame()` function in the code chunk above. Unfortunately, the `data.frame()` function doesn't allow us to *create* and *refer to* a column name in the same function call. So, we would need to break this task up into two steps if we wanted to use the `data.frame()` function. Here's one way we could do this:

```
# Create a data frame using the data.frame() function
my_df <- data.frame(
  name       = c("john", "alexis", "Steph", "Quiera"),
  age_years  = c(24, 44, 26, 25)
)

# Add the age in months column to my_df
my_df <- my_df %>% mutate(age_months = age_years * 12)

# Print my_df to the screen
my_df
```

```
    name age_years age_months
1   john        24        288
2 alexis        44        528
3  Steph        26        312
4 Quiera        25        300
```

Alternatively, we can use the `tibble()` function to get the result we want in just one step like so:

```
# Create a data frame using the tibble() function
my_df <- tibble(
  name       = c("john", "alexis", "Steph", "Quiera"),
  age_years  = c(24, 44, 26, 25),
  age_months = age_years * 12
)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 3
  name   age_years age_months
  <chr>      <dbl>      <dbl>
1 john          24        288
2 alexis        44        528
3 Steph         26        312
4 Quiera        25        300
```

In summary, tibbles *are* data frames. For the most part, we will use the terms "tibble" and "data frame" interchangeably for the rest of the book. However, remember that tibbles are *enhanced* data frames. Therefore, there are some things that we will do with tibbles that we can't do with basic data frames.

5.5 Missing data

As indicated in the warning box at the end of the data frames section of this chapter, all columns in our data frames have to have the same length. So what do we do when we are truly missing information in some of our observations? For example, how do we create the `class` data frame if we are missing Anne's height for some reason?

In R, we represent missing data with an `NA`. For example:

```r
# Create the class data frame
data.frame(
  names   = c("John", "Sally", "Brad", "Anne"),
  heights = c(68, 63, 71, NA) # Now we are missing Anne's height
)
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      NA
```

> ⚠ Warning
>
> **Warning:** Make sure you capitalize `NA` and don't use any spaces or quotation marks. Also, make sure you use `NA` instead of writing `"Missing"` or something like that.

By default, R considers `NA` to be a logical-type value (as opposed to character or numeric). for example:

```r
typeof(NA)
```

```
[1] "logical"
```

However, you can tell R to make `NA` a different type by using one of the more specific forms of `NA`. For example:

```r
typeof(NA_character_)
```

```
[1] "character"
```

```r
typeof(NA_integer_)
```

```
[1] "integer"
```

```r
typeof(NA_real_)
```

```
[1] "double"
```

Most of the time, you won't have to worry about doing this because R will take care of converting `NA` for you. What do we mean by that? Well, remember that every vector can have only one type. So, when you add an `NA` (logical by default) to a vector with double values as we did above (i.e., `c(68, 63, 71, NA)`), that would cause you to have three double values and one logical value in the same vector, which is not allowed. Therefore, R will automatically convert the `NA` to `NA_real_` for you behind the scenes.

This is a concept known as "type coercion" and you can read more about it here if you are interested. As we said, most of the time you don't have to worry about type coercion – it will happen automatically. But, sometimes it doesn't and it will cause R to give you an error. we mostly encounter this when using the `if_else()` and `case_when()` functions, which we will discuss later.

5.6 Our first analysis

Congratulations on your new R programming skills. You can now create vectors and data frames. This is no small thing. Basically, everything else we do in this book will start with vectors and data frames.

Having said that, just *creating* data frames may not seem super exciting. So, let's round out this chapter with a basic descriptive analysis of the data we simulated. Specifically, let's find the average height of the class.

You will find that in R there are almost always many different ways to accomplish a given task. Sometimes, choosing one over another is simply a matter of preference. Other times, one method is clearly more efficient and/or accurate than another. This is a point that will come up over and over in this book. Let's use our desire to find the mean height of the class as an example.

5.6.1 Manual calculation of the mean

For starters, we can add up all the heights and divide by the total number of heights to find the mean.

```
(68 + 63 + 71 + 72) / 4
```

```
[1] 68.5
```

Here's what we did above:

- We used the addition operator (+) to add up all the heights.

- We used the division operator (/) to divide the sum of all the heights by 4 - the number of individual heights we added together.
- We used parentheses to enforce the correct order of operations (i.e., make R do addition before division).

This works, but why might it not be the best approach? Well, for starters, manually typing in the heights is error prone. We can easily accidently press the wrong key. Luckily, we already have the heights stored as a column in the `class` data frame. We can *access* or *refer to* a single column in a data frame using the **dollar sign notation**.

5.6.2 Dollar sign notation

```
class$heights
```

```
[1] 68 63 71 72
```

Here's what we did above:

- We used the dollar sign notation to *access* the `heights` column in the `class` data frame.
  - Dollar sign notation is just the data frame name, followed by the dollar sign, followed by the column name.

5.6.3 Bracket notation

Further, we can use **bracket notation** to access each value in a vector. we think it's easier to demonstrate bracket notation than it is to describe it. For example, we could access the third value in the names vector like this:

```
# Create the heights vector
heights <- c(68, 63, 71, 72)

# Bracket notation
# Access the third element in the heights vector with bracket notation
heights[3]
```

```
[1] 71
```

Remember, that data frame columns are also vectors. So, we can combine the dollar sign notation and bracket notation, to access each individual value of the `height` column in the `class` data frame. This will help us get around the problem of typing each individual height value. For example:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
(class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4
```

```
[1] 68.5
```

5.6.4 The sum function

The second method is better in the sense that we no longer have to worry about mistyping the heights. However, who wants to type `class$heights[...]` over and over? What if we had a hundred numbers? What if we had a thousand numbers? This wouldn't work. Luckily, there is a function that adds all the numbers contained in a numeric vector – the `sum()` function. Let's take a look:

```
# Create the heights vector
heights <- c(68, 63, 71, 72)

# Add together all the individual heights with the sum function
sum(heights)
```

```
[1] 274
```

Remember, that data frame columns are also vectors. So, we can combine the dollar sign notation and `sum()` function, to add up all the individual heights in the `heights` column of the `class` data frame. It looks like this:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4
```

```
# Third way. Use dollar sign notation and sum function so that we don't have
# to type as much
sum(class$heights) / 4
```

```
[1] 68.5
```

Here's what we did above:

- We passed the numeric vector `heights` from the `class` data frame to the `sum()` function using dollar sign notation.
- The `sum()` function returned the total value of all the heights added together.
- We divided the total value of the heights by four – the number of individual heights.

5.6.5 Nesting functions

!! Before we move on, we want to point out something that is actually kind of a big deal. In the third method above, we didn't manually add up all the individual heights - R did this calculation for us. Further, we didn't store the sum of the individual heights somewhere and then divide that stored value by 4. Heck, we didn't even see what the sum of the individual heights were. Instead, the returned value from the sum function (274) was used *directly* in the next calculation (`/ 4`) by R without us seeing the result. In other words, `(68 + 63 + 71 + 72) / 4`, `274 / 4`, and `sum(class$heights) / 4` are all exactly the same thing to R. However, the third method (`sum(class$heights) / 4`) is much more **scalable** (i.e., adding a lot more numbers doesn't make this any harder to do) and much less error prone. Just to be clear, the BIG DEAL is that we now know that the values returned by functions can be *directly* passed to other functions in exactly the same way as if we typed the values ourselves.

This concept, functions passing values to other functions is known as **nesting functions**. It's called nesting functions because we can put functions inside of other functions.

"But, Brad, there's only one function in the command `sum(class$heights) / 4` – the `sum()` function." Really? Is there? Remember when we said that operators are also functions in R? Well, the division operator is a function. And, like all functions it can be written with parentheses like this:

```
# Writing the division operator as a function with parentheses
`/`(8, 4)
```

```
[1] 2
```

Here's what we did above:

- We wrote the division operator in its more function-looking form.
  - Because the division operator isn't a letter, we had to wrap it in backticks (`).
  - The backtick key is on the top left corner of your keyboard near the escape key (esc).
  - The first argument we passed to the division function was the dividend (The number we want to divide).
  - The second argument we passed to the division function was the divisor (The number we want to divide by).

So, the following two commands mean exactly the same thing to R:

```
8 / 4
```

```
`/`(8, 4)
```

And if we use this second form of the division operator, we can clearly see that one function is *nested* inside another function.

```
`/`(sum(class$heights), 4)
```

```
[1] 68.5
```

Here's what we did above:

- We calculated the mean height of the class.
  - The first argument we passed to the division function was the returned value from the `sum()` function.
  - The second argument we passed to the division function was the divisor (4).

This is kind of mind-blowing stuff the first time you encounter it. we wouldn't blame you if you are feeling overwhelmed or confused. The main points to take away from this section are:

1. Everything we *do* in R, we will *do* with functions. Even operators are functions, and they can be written in a form that looks function-like; however, we will almost never actually write them in that way.

2. Functions can be **nested**. This is huge because it allows us to directly pass returned values to other functions. Nesting functions in this way allows us to do very complex operations in a scalable way and without storing a bunch of unneeded values that are created in the intermediate steps of the operation.
3. The downside of nesting functions is that it can make our code difficult to read - especially when we nest many functions. Fortunately, we will learn to use the pipe operator (`%>%`) in the workflow basics part of this book. Once you get used to pipes, they will make nested functions much easier to read.

Now, let's get back to our analysis...

5.6.6 The length function

We think most of us would agree that the third method we learned for calculating the mean height is preferable to the first two methods for most situations. However, the third method still requires us to know how many individual heights are in the `heights` column (i.e., 4). Luckily, there is a function that tells us how many individual values are contained in a vector – the `length()` function. Let's take a look:

```
# Create the heights vector
heights <- c(68, 63, 71, 72)

# Return the number of individual values in heights
length(heights)
```

```
[1] 4
```

Remember, that data frame columns are also vectors. So, we can combine the dollar sign notation and `length()` function to automatically calculate the number of values in the `heights` column of the `class` data frame. It looks like this:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4

# Third way. Use dollar sign notation and sum function so that we don't have
# to type as much
# sum(class$heights) / 4
```

```
# Fourth way. Use dollar sign notation with the sum function and the length
# function
sum(class$heights) / length(class$heights)
```

```
[1] 68.5
```

Here's what we did above:

- We passed the numeric vector `heights` from the `class` data frame to the `sum()` function using dollar sign notation.
- The `sum()` function returned the total value of all the heights added together.
- We passed the numeric vector `heights` from the `class` data frame to the `length()` function using dollar sign notation.
- The `length()` function returned the total number of values in the `heights` column.
- We divided the total value of the heights by the total number of values in the `heights` column.

5.6.7 The mean function

The fourth method above is definitely the best method yet. However, this need to find the mean value of a numeric vector is so common that someone had the sense to create a function that takes care of all the above steps for us – the `mean()` function. And as you probably saw coming, we can use the mean function like so:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4

# Third way. Use dollar sign notation and sum function so that we don't have
# to type as much
# sum(class$heights) / 4

# Fourth way. Use dollar sign notation with the sum function and the length
# function
# sum(class$heights) / length(class$heights)
```

```
# Fifth way. Use dollar sign notation with the mean function
mean(class$heights)
```

```
[1] 68.5
```

Congratulations again! You completed your first analysis using R!

5.7 Some common errors

Before we move on, we want to briefly discuss a couple common errors that will frustrate many of you early in your R journey. You may have noticed that we went out of our way to differentiate between the `heights` vector and the `heights` column in the `class` data frame. As annoying as that may have been, we did it for a reason. The `heights` vector and the `heights` column in the `class` data frame are two separate things to the R interpreter, and you have to be very specific about which one you are referring to. To make this more concrete, let's add a `weight` column to our `class` data frame.

```
class$weight <- c(160, 170, 180, 190)
```

Here's what we did above:

- We created a new column in our data frame – `weight` – using dollar sign notation.

Now, let's find the mean weight of the students in our class.

```
mean(weight)
```

```
Error in eval(expr, envir, enclos): object 'weight' not found
```

Uh, oh! What happened? Why is R saying that `weight` doesn't exist? We clearly created it above, right? Wrong. We didn't create an *object* called weight in the code chunk above. We created a *column* called `weight` in the *object* called `class` in the code chunk above. Those are *different things* to R. If we want to get the mean of `weight` we have to tell R that `weight` is a column in `class` like so:

```
mean(class$weight)
```

```
[1] 175
```

A related issue can arise when you have an object and a column with the same name but different values. For example:

```
# An object called scores
scores <- c(5, 9, 3)

# A columnn in the class data frame called scores
class$scores <- c(95, 97, 93, 100)
```

If you ask R for the mean of `scores`, R will give you an answer.

```
mean(scores)
```

```
[1] 5.666667
```

However, if you wanted the mean of the `scores` column in the `class` data frame, this won't be the *correct* answer. Hopefully, you already know how to get the correct answer, which is:

```
mean(class$scores)
```

```
[1] 96.25
```

Again, the `scores` object and the `scores` column of the `class` object are different things to R.

5.8 Summary

Wow! We covered a lot in this first part of the book on getting started with R and RStudio. Don't feel bad if your head is swimming. It's a lot to take-in. However, you should feel proud of the fact that you can already do some legitimately useful things with R. Namely, simulate and analyze data. In the next part of this book, we are going to discuss some tools and best practices that will make it easier and more efficient for you to write and share your R code. After that, we will move on to tackling more advanced programming and data analysis challenges.

6 Asking Questions

Sooner or later, all of us will inevitably have questions while writing `R` programs. This is true for novice `R` users and experienced `R` veterans alike. Getting useful answers to programming questions can be really complicated under the best conditions (i.e., where someone with experience can physically sit down next to you to interactively work through your code with you). In reality, getting answers to our coding questions is often further complicated by the fact that we don't have access to an experienced R programmer who can sit down next to us and help us debug our code. Therefore, this chapter will provide us with some guidance for seeking `R` programming help remotely. We're not going to lie, this will likely be a frustrating process at times, but we will get through it!

An example

Because we like to start with the end in mind, click here for an example of a real post that we created on Stack Overflow. We will refer back to this post below.

6.1 When should we seek help?

Imagine yourself sitting in front of your computer on a Wednesday afternoon. You are working on a project that requires the analysis of some data. You know that you need to clean up your data a little bit before you can do your analysis. For example, maybe you need to drop all the rows from your data that have a missing value for a set of variables. Before you drop them, you want to take a look at which rows meet this criterion and what information would potentially be lost in the process of dropping those rows. In other words, you just want to view the rows of your data that have a missing value for any variable. Sounds simple enough! However, you start typing out the code to make this happen and that's when you start to run into problems. At this point, the problem you encounter will typically come in one of a few different flavors.

1. As you sit down to write the code, you realize that you don't really even know where to start.
2. You happily start typing out the code that you believe should work, but when you run the code you get an error message.
3. You happily start typing out the code that you believe should work, but when you run the code you don't get the result you were expecting.

4. You happily start typing out the code that you believe should work and it does! However, you notice that your solution seems clunky, inefficient, or otherwise less than ideal.

In any of these cases, you will need to figure out what your next step will be. We believe that there is typically a lot of value in starting out by attempting to solve the problem on your own without directly asking others for help. Doing so will often lead you to a deeper understanding of the solution than you would obtain by simply being given the answer. Further, finding the solution on your own helps you develop problem-solving skills that will be useful for the next coding problem you encounter – even if the details of that problem are completely different than the details of your current problem. Having said that, finding a solution on your own does **not** mean attempting to do so in a vacuum without the use of any resources (e.g., textbooks, existing code, or the internet). By all means, use available resources (we suggest some good ones below)!

On the other hand, we – the authors – have found ourselves stubbornly hacking away on our own solution to a coding problem long after doing so ceased being productive on many occasions. We don't recommend doing this either. We hope that the guidance in this chapter will provide you with some tools for effectively and efficiently seeking help from the broader `R` programming community once you've made a sincere effort to solve the problem on your own.

But, how long should you attempt to solve the problem on your own before reaching out for help? As far as we know, there are no hard-and-fast rules about how long you should wait before seeking help with coding problems from others. In reality, the ideal amount of time to wait is probably dependent on a host of factors including the nature of the problem, your level of experience, project deadlines, all of your little personal idiosyncrasies, and a whole host of other factors. Therefore, the best guidance we can provide is pretty vague. In general, it isn't ideal to reach out to the `R` programming community for help as soon as you encounter a problem, nor is it typically ideal to spend many hours attempting to solve a coding problem that could be solved in few minutes if you were to post a well-written question on Stack Overflow or the RStudio Community (more on these below).

6.2 Where should we seek help?

Where should you turn once you've determined that it is time to seek help for your coding problem? We suggest that you simply start with Google. Very often, a quick Google search will give you the results you need to help you solve your problem. However, Google search results won't *always* have the answer you are looking for.

If you've done a Google search and you still can't figure out how to solve your coding problem, we recommend posting a question on one of the following two websites:

1. **Stack Overflow** (https://stackoverflow.com/). This is a great website where programmers who use many different languages help each other solve programming problems. This website is free, but you will need to create an account.
2. **RStudio Community** (https://community.rstudio.com/). Another great discussion-board-type website from the people who created a lot of the software we will use in this book. This website is also free, but also requires you to create an account.

Side Note: Please remember to cross-link your posts if you happen to create them on both Stack Overflow and RStudio Community. When we say "cross-link" we mean that you should add a hyperlink to your RStudio Community post on your Stack Overflow post and a link to your Stack Overflow post on your RStudio Community post.

Next, let's learn how to make a post.

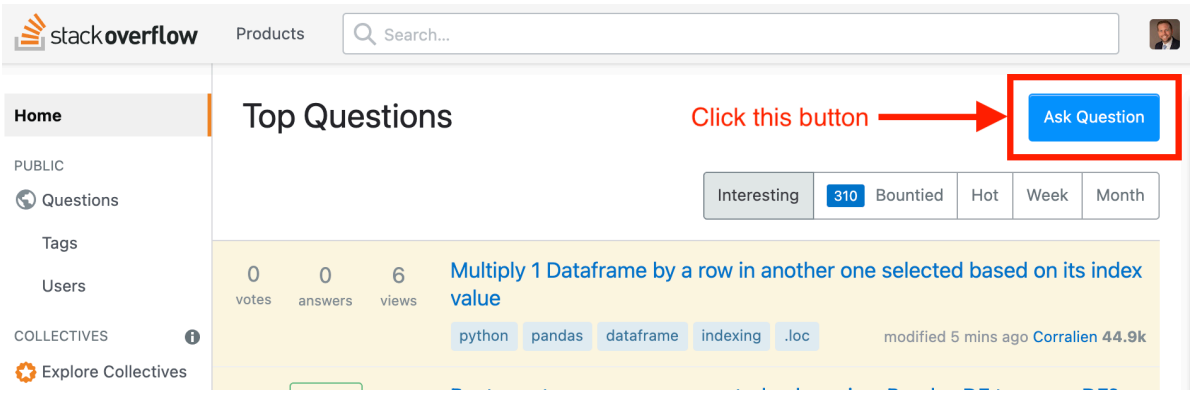
6.3 How should we seek help?

At this point, you've run into a problem, you've spent a little time trying to work out a solution in your head, you've searched Google for a solution to the problem, and you've still come up short. So, you decide to ask the `R` programming community for some help using Stack Overflow. But, how do you do that?

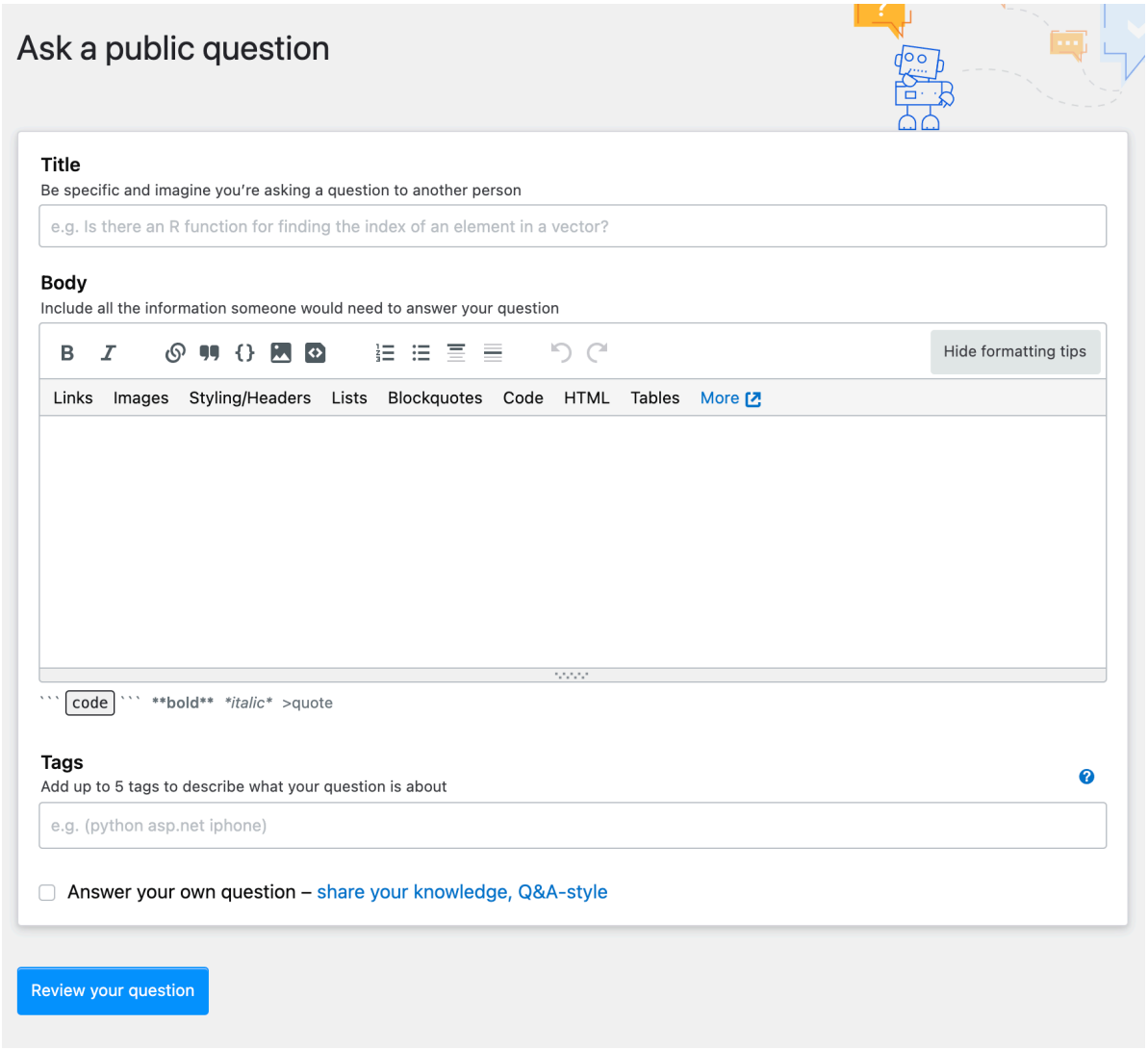
Side Note: We've decided to show you haw to create a post on Stack Overflow in this section, but the process for creating a post in the RStudio Community is very similar. Further, an RStudio Community tutorial is available here: https://community.rstudio.com/t/example-question-answer-topic-thread/70762.

6.3.1 Creating a post on Stack Overflow

The first thing you need to do is navigate to the Stack Overflow website. The homepage will look something like the screenshot below.



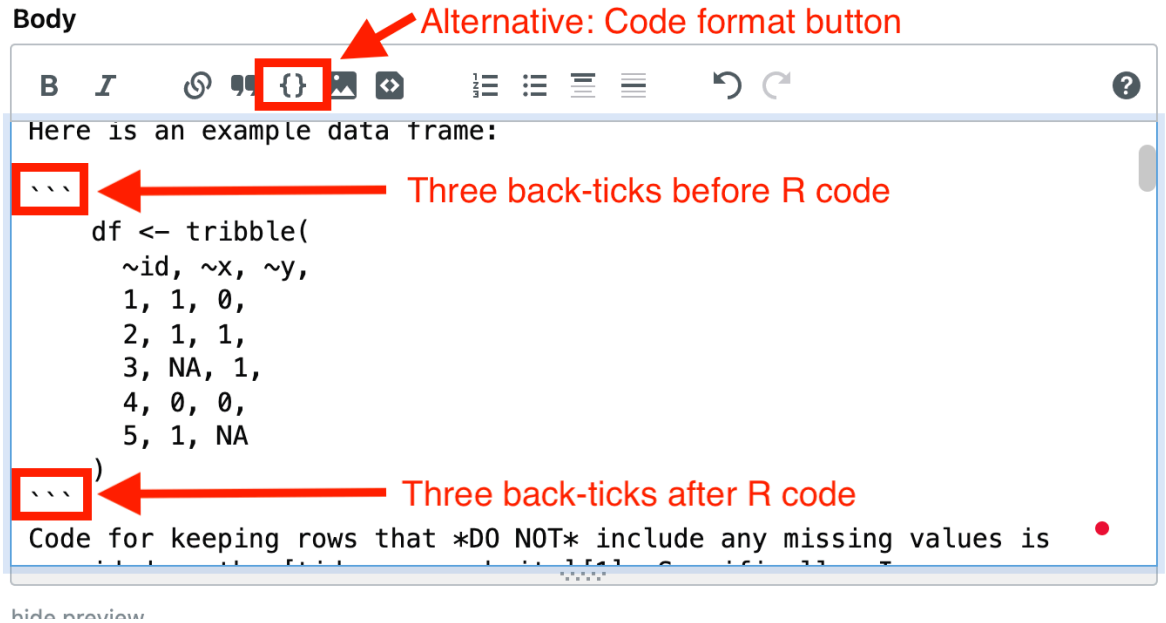
Next, you will click the blue "Ask Question" button. Doing so will take you to a screen like the following.



As you can see, you need to give your post a **title**, you need to post the actual question in the **body** section of the form, and then you can (and should) tag your post. "A tag is simply a word or a phrase that describes the topic of the question."[2] For our `R`-related questions we will want to use the "r" tag. Other examples of tags you may use often if you continue your `R` programming journey may include "dplyr" and "ggplot2". When you have completed the form, you simply click the blue "Review your question" button towards the bottom-left corner of the screen.

6.3.1.1 Inserting R code

To insert `R` code into your post (i.e., in the body), you will need to create **code blocks**. Then, you will type your `R` code inside of the code blocks. You can create code blocks using back-ticks (‘). The back-tick key is the upper-left key of most keyboards – right below the escape key. On our keyboard, the back-tick and the tilde (~) share the same key. We will learn more about code blocks in the chapter on using [R markdown]. For now, let's just take a look at an example of creating a code block in the screenshot below. This screenshot comes from the example Stack Overflow post introduced at the beginning of the chapter.



As you can see, we placed three back-ticks on their own line before our `R` code and three back-ticks on their own line after our R code. Alternatively, we could have used our mouse to highlight our `R` code and then clicked the code format button, which is highlighted in the screenshot above and looks like an empty pair of curly braces ({}).

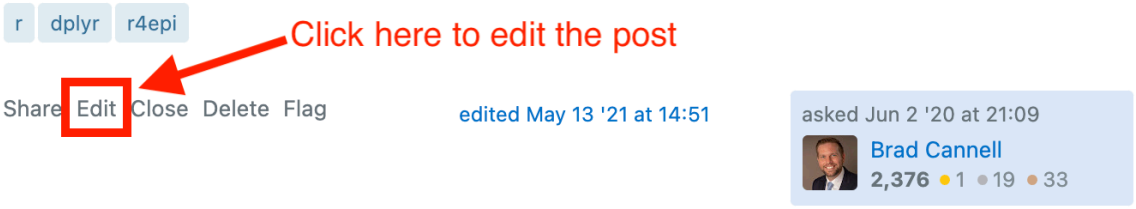
6.3.1.2 Reviewing the post

After you create your post and click the “Review your question” button, you will have an opportunity to check your post for a couple of potential issues.

1. Duplicates. You want to try your best to make sure your question isn't a duplicate question. Meaning, you want to make sure that someone else hasn't already asked the same question or a question that is very similar. As you are typing your post title, Stack Overflow will show you a list of potentially similar questions. It will show you that list again as you are reviewing your post. You should take a moment to look through that

list and make sure you question isn't going to be a duplicate. If it does end up being a duplicate, Stack Overflow moderators may tag it as such and close it.

2. Typos and errors. Of course, you also want to check your post for standard typos, grammatical errors, and coding errors. However, you can always edit your post later if an error does slip through. You just need to click the `edit` text at the bottom of your post. A screenshot from the example post is shown in the screenshot below.



6.3.2 Creating better posts and asking better questions

There are no bad `R` programming questions, but there are definitely ways to ask those questions that will be better received than others. And better received questions will typically result in faster responses and more useful answers. It's important that you ask your questions in a way that will allow the reader to understand what you are trying to accomplish, what you've already tried, and what results you are getting. Further, unless it's something extremely straight forward, **you should always provide a little chunk of data that recreates the problem you are experiencing.** These are known as **reproducible examples** This is so important that there is an R package that does nothing but help you create reproducible examples – Reprex.

Additionally, Stack Overflow and the RStudio community both publish guidelines for posting good questions.

- Stack Overflow guide to asking questions: https://stackoverflow.com/help/how-to-ask
- RStudio Community Tips for writing R-related questions: https://community.rstudio.com/t/faq-tips-for-writing-r-related-questions/6824

You should definitely pause here an take a few minutes to read through these guidelines. If not now, come back and read them before you post your first question on either website. Below, we show you a few example posts and highlight some of the most important characteristics of quality posts.

6.3.2.1 Example posts

Here are a few examples of highly viewed posts on Stack Overflow and the RStudio community. Feel free to look them over. Notice what was good about these posts and what could have been better. The specifics of these questions are totally irrelevant. Instead, look for the elements that make posts easy to understand and respond to.

1. Stack Overflow: How to join (merge) data frames (inner, outer, left, right)
2. RStudio Community: Error: Aesthetics must be either length 1 or the same as the data (2): fill
3. Stack Overflow: How should I deal with "package 'xxx' is not available (for R version x.y.z)" warning?
4. RStudio Community: Could anybody help me! Cannot add ggproto objects together

6.3.2.2 Question title

When creating your posts, you want to make sure they have succinct, yet descriptive, titles. Stack overflow suggests that you pretend you are talking to a busy colleague and have to summarize your issue in a single sentence.[3] The RStudio Community tips for writing questions further suggests that you be specific and use keywords.[4] Finally, if you are really struggling, it may be helpful to write your title last.[3] In our opinion, the titles from the first 3 examples above are pretty good. The fourth has some room for improvement.

6.3.2.3 Explanation of the issue

Make sure your posts have a brief, yet clear, explanation of what you are trying to accomplish. For example, "Sometimes I want to view all rows in a data frame that will be dropped if I drop all rows that have a missing value for any variable. In this case, I'm specifically interested in how to do this with dplyr 1.0's across() function used inside of the filter() verb."

In addition, you may want to **add what you've already tried, what result you are getting, and what result you are expecting**. This information can help others better understand your problem and understand if the solution they offer you does what you are actually trying to do.

Finally, if you've already come across other posts or resources that were similar to the problem you are having, but not quite similar enough for you to solve your problem, it can be helpful to provide links to those as well. The author of example 3 above (i.e., How should I deal with "package 'xxx' is not available (for R version x.y.z)" warning?) does a very thorough job of linking to other posts.

6.3.2.4 Reproducible example

Make sure your question/post includes a small, reproducible data set that helps others recreate your problem. This is so important, and so often overlooked by students in our courses. Notice that we did **NOT** say to post the actual data you are working on for your project. Typically, the actual data sets that we work with will have many more rows and columns than are needed to recreate the problem. All of this extra data just makes the problem harder to clearly see. And more importantly, the real data we often work with contains **protected health information (PHI)** that should **NEVER** be openly published on the internet.

Here is an example of a small, reproducible data set that we created for the example Stack Overflow post introduced at the beginning of the chapter. It only has 5 data rows and 3 columns, but any solution that solves the problem for this small data set will likely solve the problem in our actual data set as well.

```
# Load the dplyr package.
library(dplyr)

# Simulate a small, reproducible example of the problem.
df <- tribble(
  ~id, ~x, ~y,
  1, 1, 0,
  2, 1, 1,
  3, NA, 1,
  4, 0, 0,
  5, 1, NA
)
```

Sometimes you can add reproducible data to your post without simulating your own data. When you download R, it comes with some built in data sets that all other R users have access to as well. You can see an full list of those data sets by typing the following command in your `R` console:

```
data()
```

There are two data sets in particular, `mtcars` and `iris`, that seemed to be used often in programming examples and question posts. You can add those data sets to your global environment and start experimenting with them using the following code.

```
# Add the mtcars data frame your global environment
data(mtcars)

# Add the iris data frame to your global environment
data(iris)
```

In general, you are safe to post a question on Stack Overflow or the RStudio Community using either of these data frames in your example code – assuming you are able to recreate the issue you are trying to solve using these data frames.

6.4 Helping others

Eventually, you may get to a point where you are able to help others with their R coding issues. In fact, spending a little time each day looking through posts and seeing if you can provide answers (whether you officially post them or not) is one way to improve *your* R coding skills. For some of us, this is even a fun way to pass time!

In the same way that there ways to improve the quality and usefulness of your question posts, there are also ways to improve the quality and usefulness of your replies to question posts. Stack Overflow also provides a guide for writing quality answers, which is available here: https://stackoverflow.com/help/how-to-answer. In our opinion, the most important part is to be patient, kind, and respond with a genuine desire to be helpful.

6.5 Summary

In this chapter we discussed when and how to ask for help with R coding problems that will inevitably occur. In short,

1. Try solving the problem on your own first, but don't spend an entire day beating your head against the wall.
2. Start with Google.
3. If you can't find a solution on Google, create a post on Stack Overflow or the RStudio Community.
4. Use best practices to create a high quality posts on Stack Overflow or the RStudio Community. Specifically:
    - Write succinct, yet descriptive, titles.

- Write a a brief, yet clear, explanation of what you are trying to accomplish. Add what you've already tried, what result you are getting, and what result you are expecting.
- Try to always include a reproducable example of the problem you are encountering in the form of data.

5. Be patient, kind, and genuine when posting or responding to posts.

Part II

Coding Tools and Best Practices

7 R Scripts

Up to this point, we've only showed you how to submit your R code to R in the console. Figure 7.1

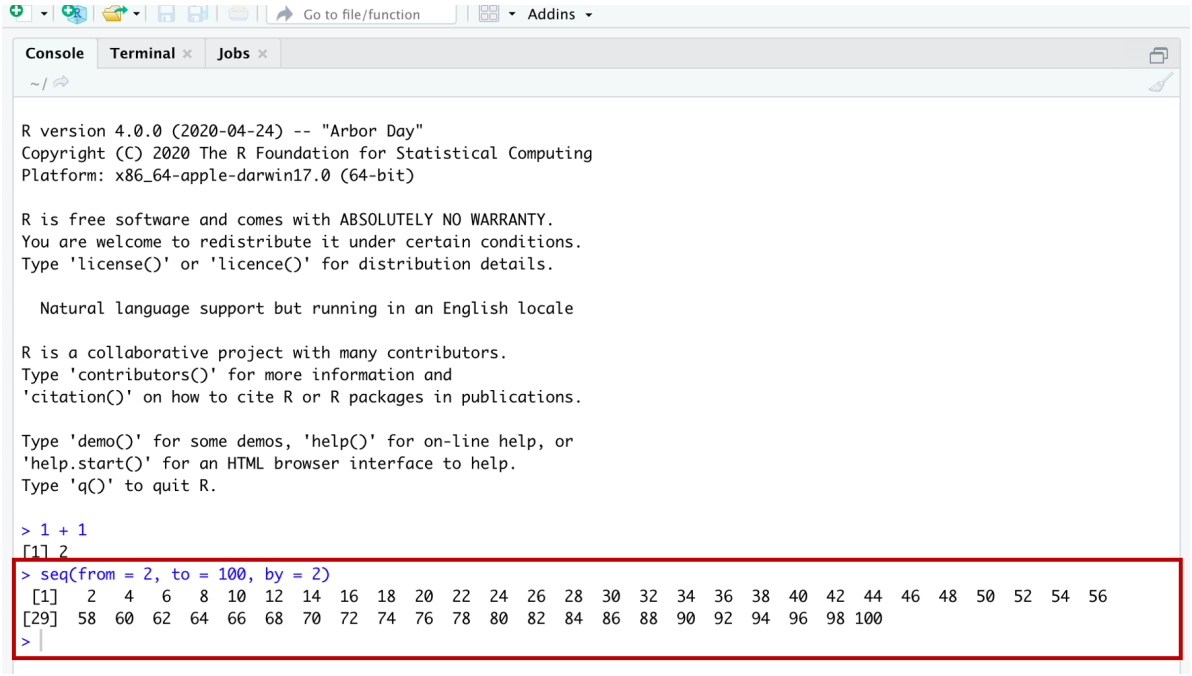


Figure 7.1: Submitting R code in the console.

Submitting code directly to the console in this way works well for quick little tasks and snippets of code. But, writing longer R programs this way has some drawbacks that are probably already obvious to you. Namely, your code isn't saved anywhere. And, because it isn't saved anywhere, you can't modify it, use it again later, or share it with others.

Technically, the statements above are not entirely true. When you submit code to the console, it is copied to RStudio's History pane and from there you can save, modify, and share with others (see figure Figure 7.2. But, this method is much less convenient, and provides you with far fewer whistles and bells than the other methods we'll discuss in this book.

Those of you who have worked with other statistical programs before may be familiar with the idea of writing, modifying, saving, and sharing code scripts. SAS calls these code scripts

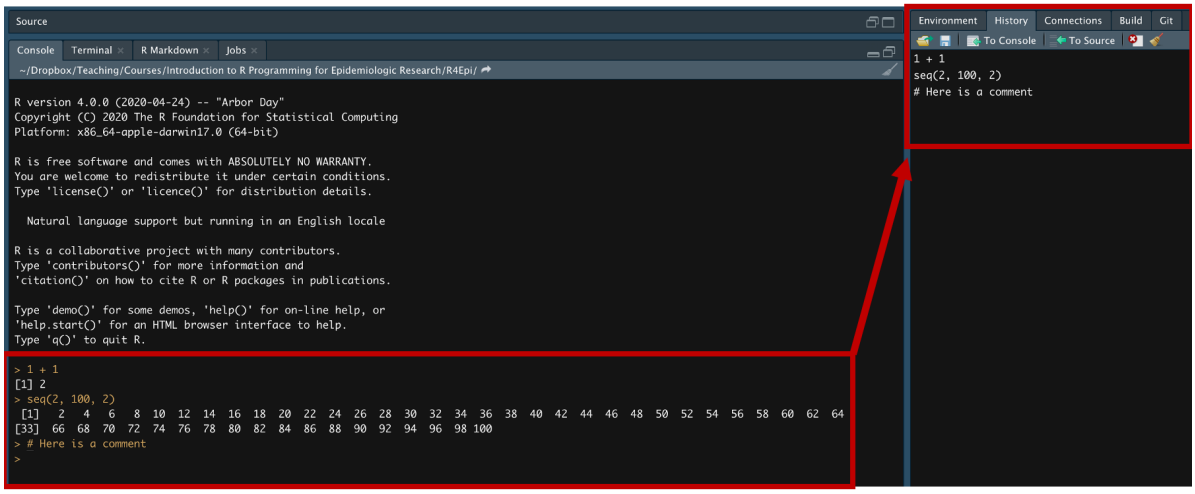


Figure 7.2: Console commands copied to the History pane.

"SAS programs", Stata calls them "DO files", and SPSS calls them "SPSS syntax files". If you haven't created code scripts before, don't worry. There really isn't much to it.

In R, the most basic type of code script is simply called an R script. An R script is just a plain text file that contains R code and comments. R script files end with the file extension `.R`.

Before we dive into giving you any more details about R scripts, we want to say that we're actually going to discourage you from using them for most of what we do in this book. Instead, we're going to encourage you to use Quarto files for the majority of your interactive coding, and for preparing your final products for end users. The next chapter is all about Quarto files. However, we're starting with R scripts because:

1. They are simpler than Quarto files, so they are a good place to start.

2. Some of what we discuss below will also apply to Quarto files.

3. R scripts *are* a better choice than Quarto files in some situations (e.g., writing R packages, creating Shiny apps).

4. Some people just prefer using R scripts.

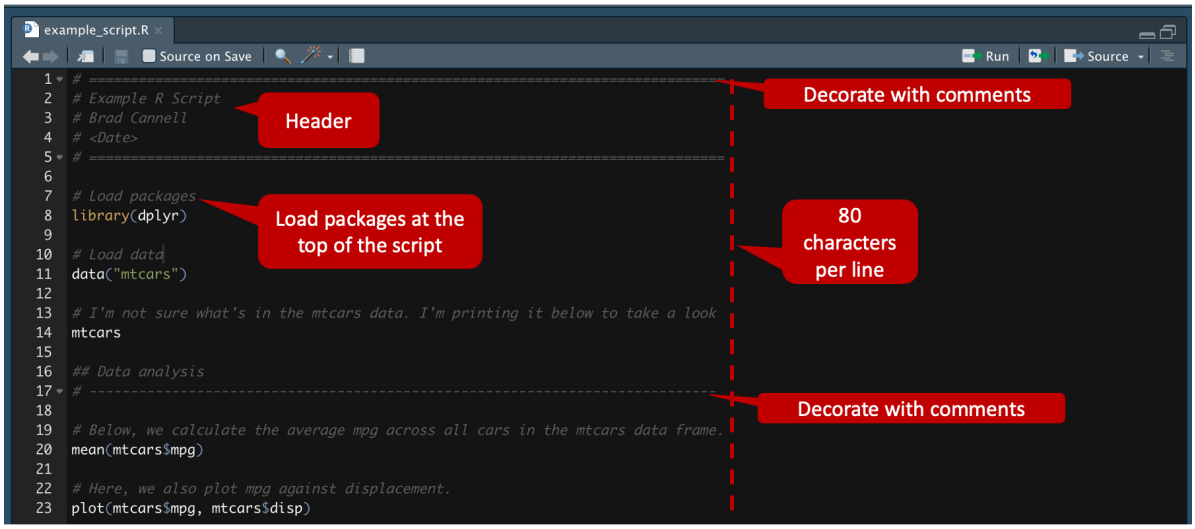


Figure 7.3: Example R script.

With all that said, the screenshot below is of an example R script:

Click here to download the R script

As you can see, I've called out a couple key elements of the R script to discuss. Figure 7.3

First, instead of just jumping into writing R code, lines 1-5 contain a **header** that we've created with comments. Because we've created it with comments, the R interpreter will ignore it. But, it will help other people you collaborate with (including future you) figure out what this script does. Therefore, we suggest that your header includes at least the following elements:

1. A brief description of what the R script does.

2. The author(s) who wrote the R script.

3. Important dates. For example, the date it was originally created and the date it was last modified. You can usually get these dates from your computer's operating system, but they aren't always accurate.

Second, you may notice that we also used comments to create something we're calling **decorations** on lines 1, 5, and 17. Like all comments, they are ignored by the R interpreter. But, they help create visual separation between distinct sections of your R code, which makes your code easier for *humans* to read. We tend to use the equal sign (`# ====`) for separating major

sections and the dash (`# ----`) for separating minor sections; although, "major" and "minor" are admittedly subjective.

we haven't explicitly highlighted it in the screenshot above, but it's probably worth pointing out the use of line breaks (i.e., returns) in the code as well. This is much easier to read…

```
# Load packages
library(dplyr)

# Load data
data("mtcars")

# I'm not sure what's in the mtcars data. I'm printing it below to take a look
mtcars

## Data analysis
# ---------------------------------------------------------------------------

# Below, we calculate the average mpg across all cars in the mtcars data frame.
mean(mtcars$mpg)

# Here, we also plot mpg against displacement.
plot(mtcars$mpg, mtcars$disp)
```

than this…

```
# Load packages
library(dplyr)
# Load data
data("mtcars")
# I'm not sure what's in the mtcars data. I'm printing it below to take a look
mtcars
## Data analysis
# ---------------------------------------------------------------------------
# Below, we calculate the average mpg across all cars in the mtcars data frame.
mean(mtcars$mpg)
# Here, we also plot mpg against displacement.
plot(mtcars$mpg, mtcars$disp)
```

Third, it's considered a best practice to keep each line of code to 80 characters (including spaces) or less. There's a little box at the bottom left corner of your R script that will tell you what row your cursor is currently in and how many characters into that row your cursor is currently at (starting at 1, not 0).

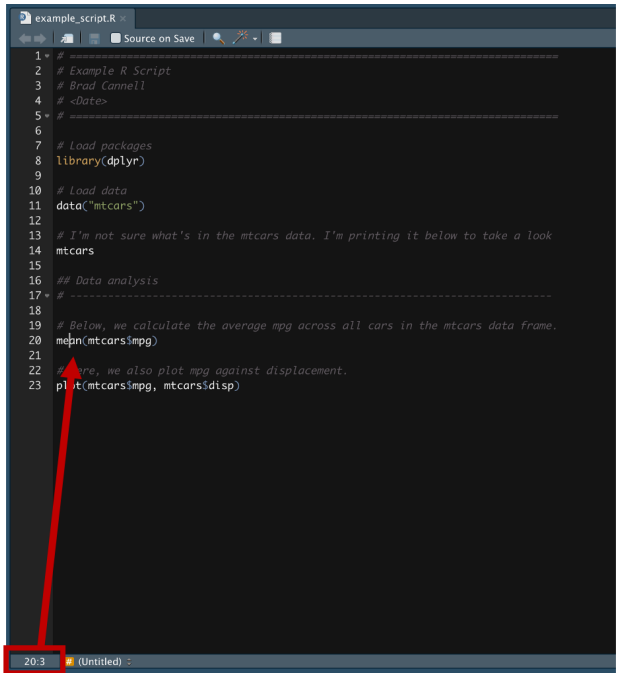


Figure 7.4: Cursor location.

For example, `20:3` corresponds to having your cursor between the "e" and the "a" in `mean(mtcars$mpg)` in the example script above. Figure 7.4

Fourth, it's also considered a best practice to load any packages that your R code will use at the very top of your R script (lines 7 & 8). Figure 7.3 Doing so will make it much easier for others (including future you) to see what packages your R code needs to work properly right from the start.

7.1 Creating R scripts

To create your own R scripts, click on the icon shown below Figure 7.5 and you will get a dropdown box with a list of files you can create. @ref(fig:new-r-script2)

Click the very first option – `R Script`.

When you do, a new untitled R Script will appear in the source pane.

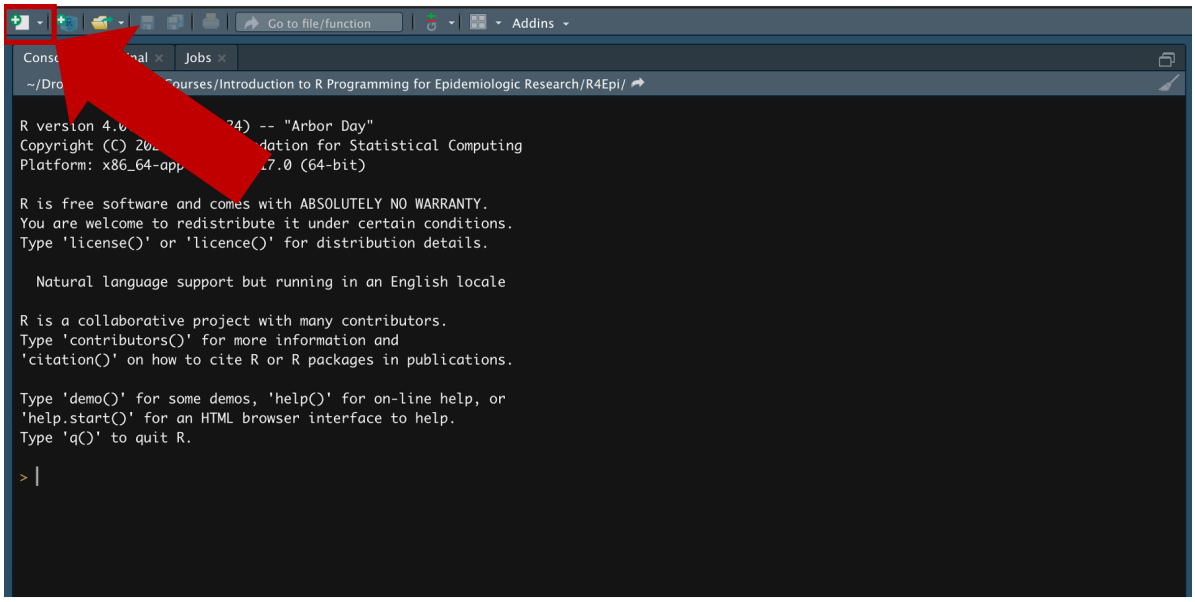


Figure 7.5: Click the new source file icon.

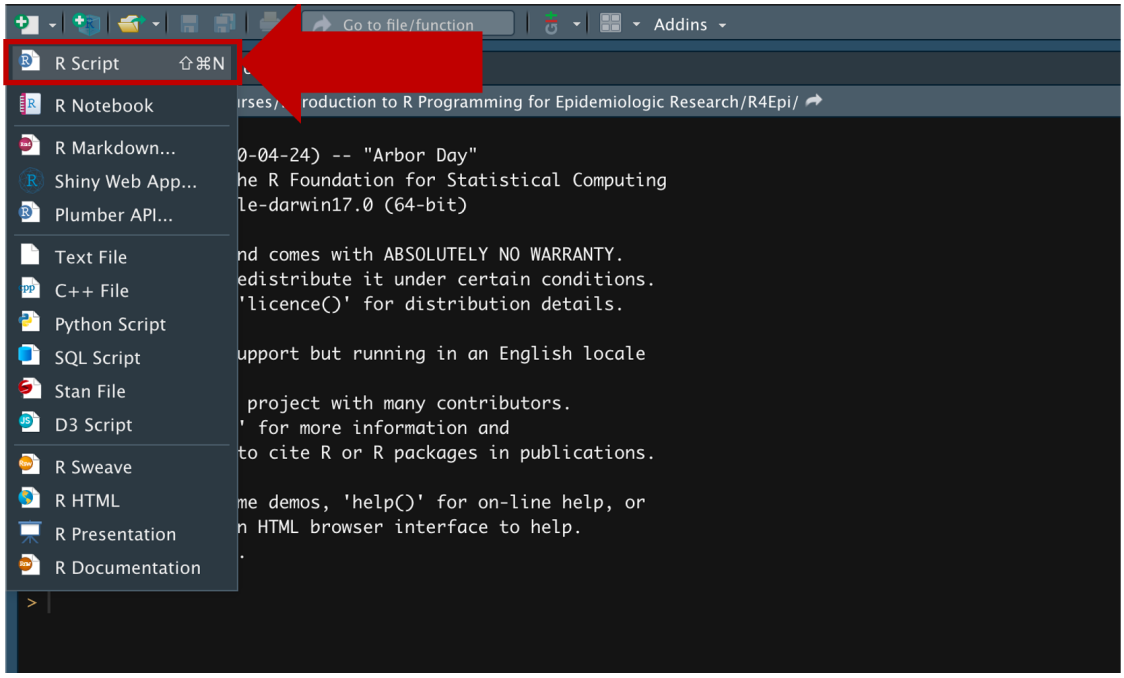


Figure 7.6: New source file options.

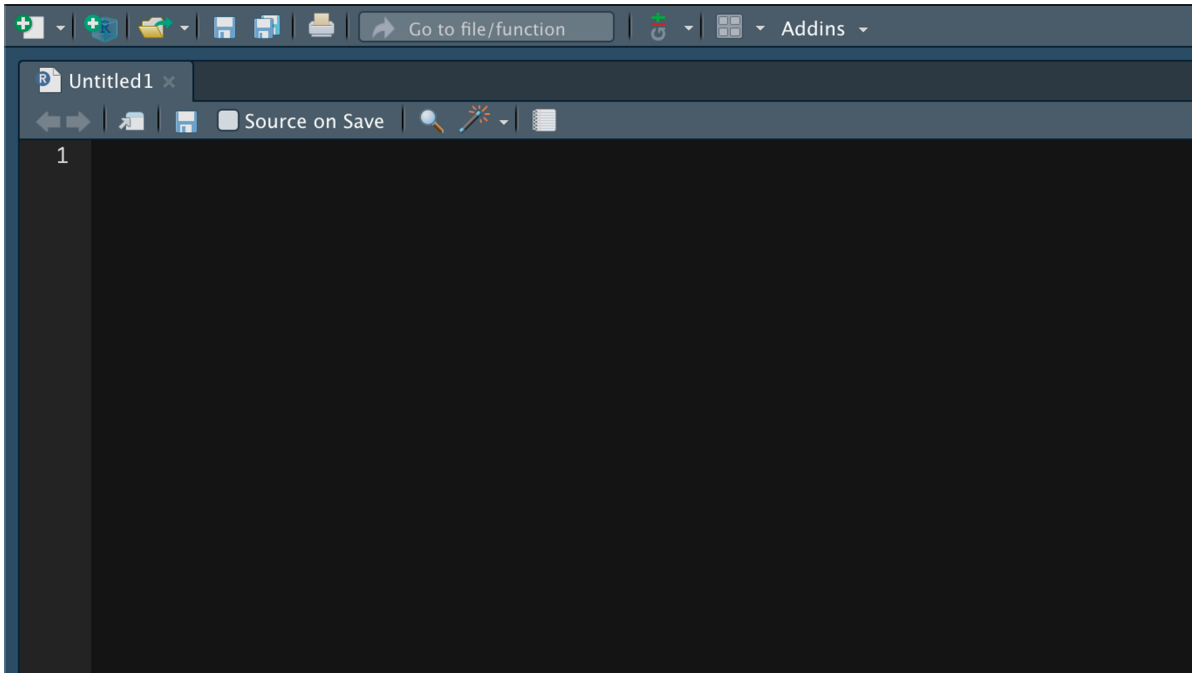


Figure 7.7: A blank R script in the source pane.

And that's pretty much it. Everything else in figure Figure 7.3 is just R code and comments about the R code. But, you can now easily save, modify, and share this code with others. In the next chapter, we are going to learn how to write R code in R markdown files, where we can add a ton of whistles and bells to this simple R script.

8 Quarto Files

In the chapter on R Scripts, you learned how to create R scripts – plain text files that contain R code and comments. These R scripts are kind of a big deal because they give us a simple and effective tool for saving, modifying, and sharing our R code. If it weren't for the existence of Quarto files, we would probably do all of the coding in this book using R scripts. However, Quarto files *do* exist and they are AWESOME! So, we're going to suggest that you use them instead of R scripts the majority of the time.

It's actually kind of difficult for us to *describe* what a Quarto file is if you've never seen or heard of one before. Therefore, we're going to start with an example and work backwards from there. Figure 8.1 below is a Quarto file. It includes the exact same R code and comments as the example we saw in Figure 7.3 in the previous chapter.

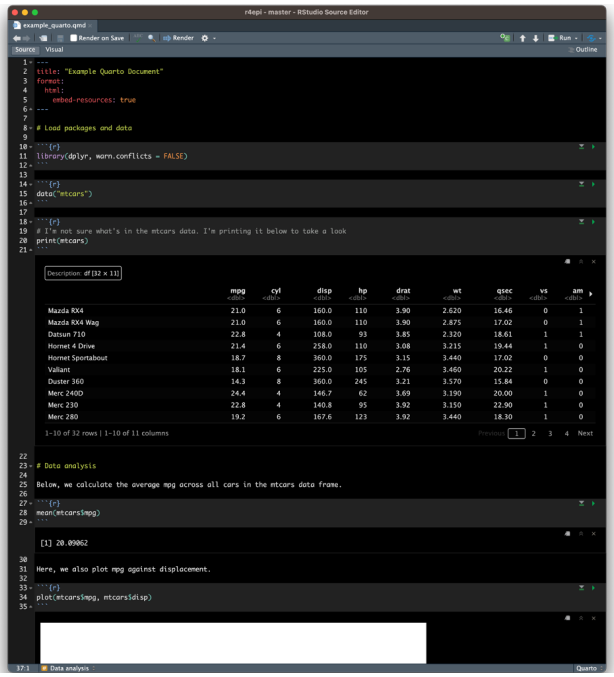
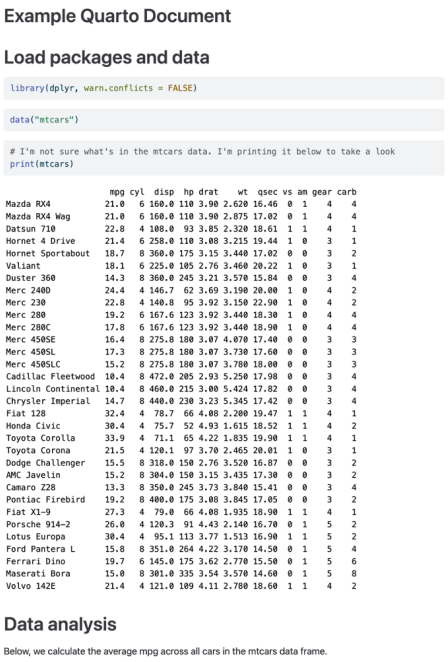


Figure 8.1: Example Quarto file.

Click here to download the Quarto file

Notice that the results are embedded directly in the Quarto file immediately below the R code (e.g., between lines 21 and 22)!

Once rendered, the Quarto file creates the HTML file you see below in Figure 8.2. HTML files are what websites are made out of, and we'll walk you through *how* to create them from Quarto files later in this chapter.



Example Quarto Document

Load packages and data

```
library(dplyr, warn.conflicts = FALSE)
```

```
data("mtcars")
```

```
# I'm not sure what's in the mtcars data. I'm printing it below to take a look
print(mtcars)
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Data analysis

Below, we calculate the average mpg across all cars in the mtcars data frame.

Figure 8.2: Preview of HTML file created from a Quarto file.

Click here to download the rendered HTML file.

Notice how everything is nicely formatted and easy to read!

When you create Quarto files on your computer, as in Figure 8.3, the rendered HTML file is saved in the same folder by default.

In Figure 8.3 above, the HTML file is highlighted with a red box and ends with the `.html` file extension. The Quarto file is below the HTML file and ends with the `.qmd` file extension. Both of these files can be modified, saved, and shared with others.

> ⚠ Warning
>
> **Warning:** HTML documents often require supporting files (e.g., images, CSS style sheets, and JavaScript scripts) to produce the final formatted output you see in the Figure 8.2. Notice that we used the `embed-resources: true` option in our yaml header

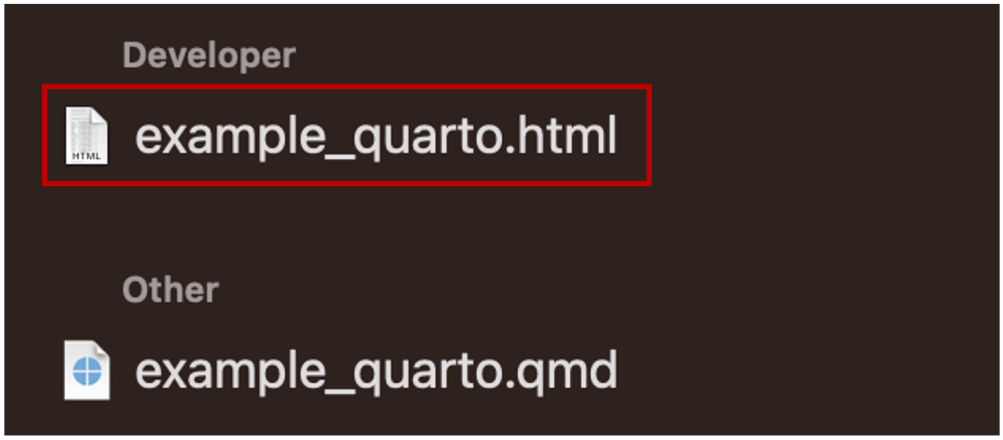


Figure 8.3: Quarto file and rendered HTML file and on MacOS.

(yaml headers are described in more detail below). Including that option makes it possible for us to send a single HTML file to others with all the supporting files embedded. Please see the Quarto documentation for more information about HTML document options.

8.1 What is Quarto?

There are literally entire websites and books about Quarto. Therefore, we're only going to hit some of the highlights in this chapter. As a starting point, you can think of Quarto files as being a mix of R scripts, the R console, and a Microsoft Word or Google Doc document. We say this because:

- The R code that you would otherwise write in R scripts is written in R **code chunks** when you use Quarto files. In Figure 8.1 there are R code chunks at lines 10 to 12, 14 to 16, 18 to 21, 27 to 29, and 33 to 35.
- Instead of having to flip back and forth between your source pane and your console (or viewer) pane in RStudio, the results from your R code are embedded directly in the Quarto file – directly below the code that generated them. In Figure 8.1 there are

embedded results between lines 21 and 22, between lines 29 and 30, and between lines 35 and 36 (not fully visible).

- When creating a document in Microsoft Word or Google Docs, you may format text headings to help organize your document, you may format your text to emphasize *certain* **words**, you may add tables to help organize concepts or data, you may add links to other resources, and you may add pictures or charts to help you clearly communicate ideas to yourself or others. Similarly, Quarto files allow you to surround your R code with formatted text, tables, links, pictures, and charts directly in your document.

Even when we don't share our Quarto files with anyone else, we find that the added functionality described above really helps us organize our data analysis more effectively and helps us understand what we were doing if we come back to the analysis at some point in the future.

But, Quarto *really* shines when we *do* want to share our analysis or results with others. To get an idea of what we're talking about, please take a look at the Quarto gallery and view some of the amazing things you can do with Quarto. As you can see there, Quarto files mix R code with other kinds of text and media to create documents, websites, presentations, and more. In fact, the book you are reading right now is created with Quarto files!

8.2 Why use Quarto?

At this point, you may be thinking "Ok, that Quarto gallery has some cool stuff, but it also looks complicated. Why shouldn't I just use a basic R script for the little R program I'm writing?" If that's what you're thinking, you have a valid point. Quarto files are slightly more complicated than basic R scripts. However, after reading the sections below, we think you will find that getting started with Quarto doesn't have to be super complicated and the benefits provided make the initial investment in learning Quarto worth your time.

8.3 Create a Quarto file

RStudio makes it very easy to create your own Quarto file, of which there are several types. In this chapter, we're going to show you how to create a Quarto file that can be rendered to an HTML file and viewed in your web browser.

The process is actually really similar to the process we used to create an R script. Start by clicking on the icon shown below in Figure 8.4.

As before, we'll be presented with a dropdown box that lists a bunch of different file types for us to choose from. This time, we'll click `Quarto Document` instead of `R script`. Figure 8.5

Next, a dialogue box will pop up with some options for us. For now, we will just give our Quarto document a super creative title – "Text Quarto" – and make sure the default HTML

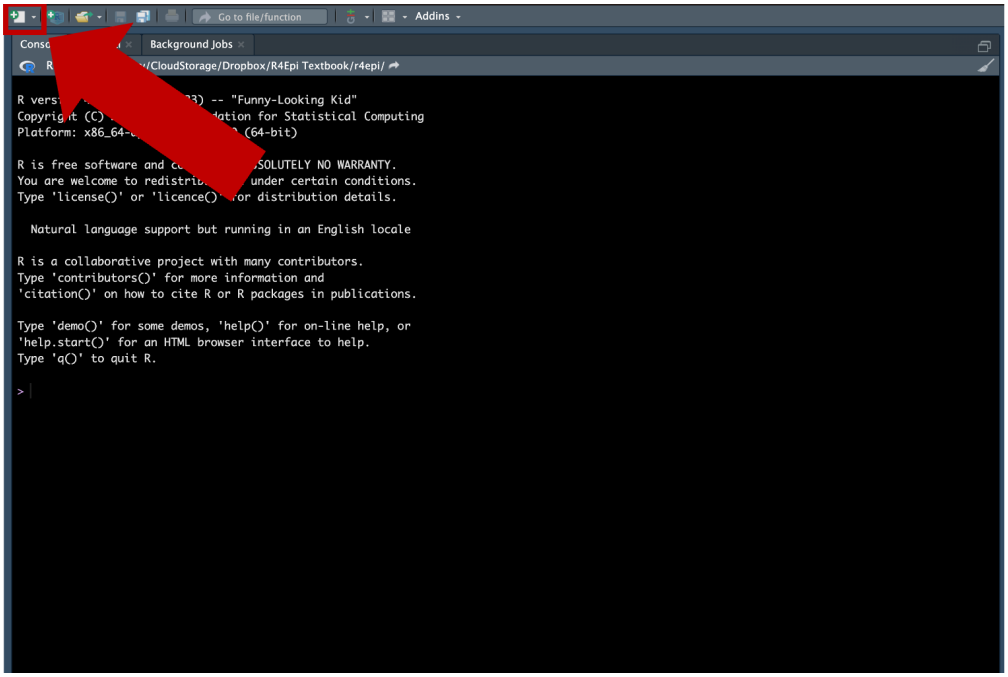


Figure 8.4: Click the new file icon.

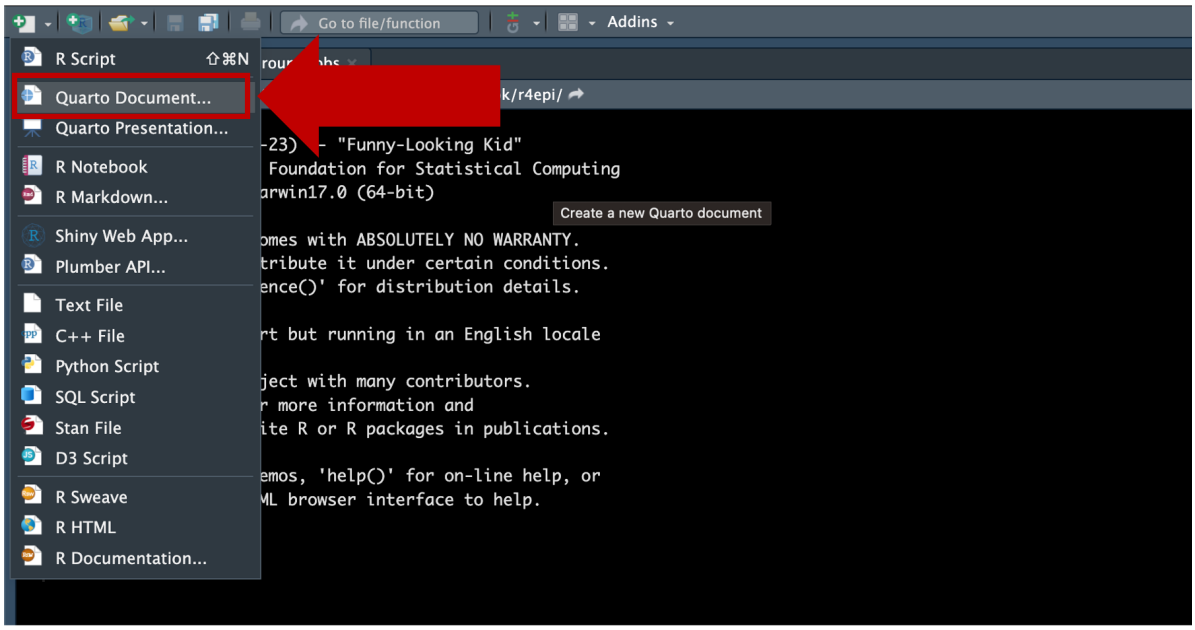


Figure 8.5: New source file options.

format is selected. Finally, we will click the `Create` button in the bottom right-hand corner of the dialogue box.

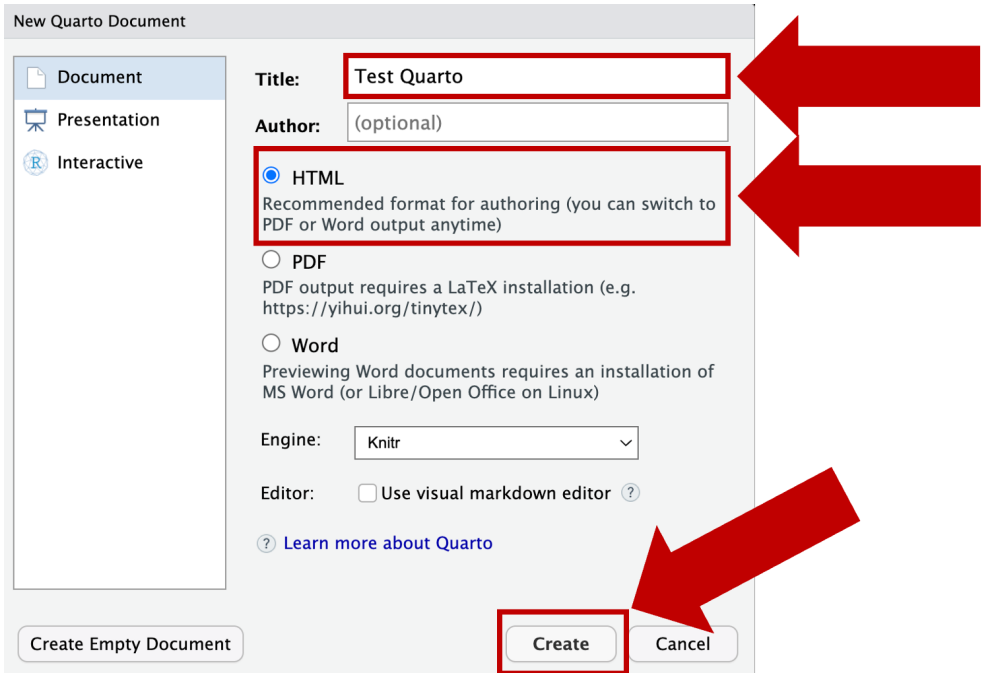


Figure 8.6: New Quarto document options.

A new Quarto file will appear in the RStudio source pane after we click the `Create` button. This Quarto file includes some example text and code meant to help us get started. We are typically going to erase all the example stuff and write our own text and code, but Figure 8.7 highlights some key components of Quarto files for now.

First, notice lines 1 through 6 in the example above. These lines make up something called the **YAML header** (pronounced yamel). It isn't important for us to know what YAML means, but we do need to know that this is one of the defining features of Quarto files. We'll talk more about the details of the YAML header soon.

Second, notice lines 16 through 18. These lines make up something called an **R code chunk**. Code chunks in Quarto files always start with three backticks (‘) and a pair of curly braces ({}), and they always end with three more backticks. We know that this code chunk contains R code because of the "r" inside of the curly braces. We can also create code chunks that will run other languages (e.g., python), but we won't do that in this book. You can think of each R code chunk as a mini R script. We'll talk more about the details of code chunks soon.

Third, all of the other text is called Markdown. In Figure 8.7 above, the markdown text is just filler text with some basic instructions for users. In a real project we would use formatted text like this to add context around our code. For now, you can think of this as being very

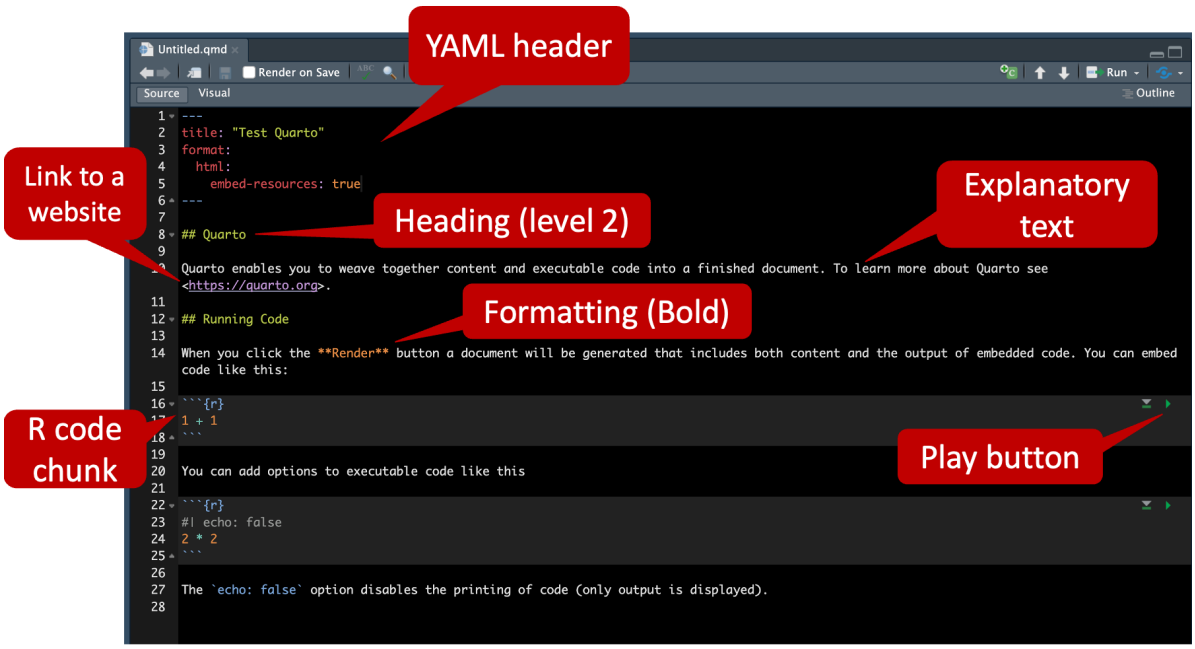


Figure 8.7: The 'Test Quarto' file in the RStudio source pane.

similar to the comments we wrote in our R scripts, but markdown allows us to do lots of cool things that the comments in our R scripts aren't able to do. For example, line 6 has a link to a website embedded in it, line 8 includes a heading (i.e., `## Quarto`), and line 14 includes text that is being formatted (the orange text surrounded by two asterisks). In this case, the text is being bolded.

And that is all we have to do to create a basic Quarto file. Next, we're going to give you a few more details about each of the key components of the Quarto file that we briefly introduced above.

8.4 YAML headers

The YAML header is unlike anything we've seen before. The YAML header always begins and ends with dash-dash-dash (`---`) typed on its own line (1 & 6 in Figure 8.7). The code written inside the YAML header generally falls into two categories:

1. Values to be rendered in the Quarto file. For example, in Figure 8.7 we told Quarto to title our document "Test Quarto". The title is added to the file by adding the `title` keyword, followed by a colon (`:`), followed by a character string wrapped in quotes. Examples of other values we could have added include `author` and `date`.

2. Instructions that tell Quarto how to process the file. What do we mean by that? Well, remember the Quarto gallery you saw earlier? That gallery includes Word documents, PDF documents, websites, and more. But all of those different document types started as Quarto file similar to the one in Figure 8.7. Quarto will create a PDF document, a Word document, or a website from the Quarto file based, in part, on the instructions we give it inside the YAML header. For example, the YAML header in Figure 8.7 tells Quarto to create an HTML file from our Quarto file. This output type is selected by adding the `format` keyword, followed by a colon (`:`), followed by the `html` keyword. Further, we added the `embed-resources: true` option to our HTML format. Including that option makes it possible for us to send a single HTML file to others with all the supporting files embedded.

What does an HTML file look like? Well, if you hit the `Render` button in RStudio:

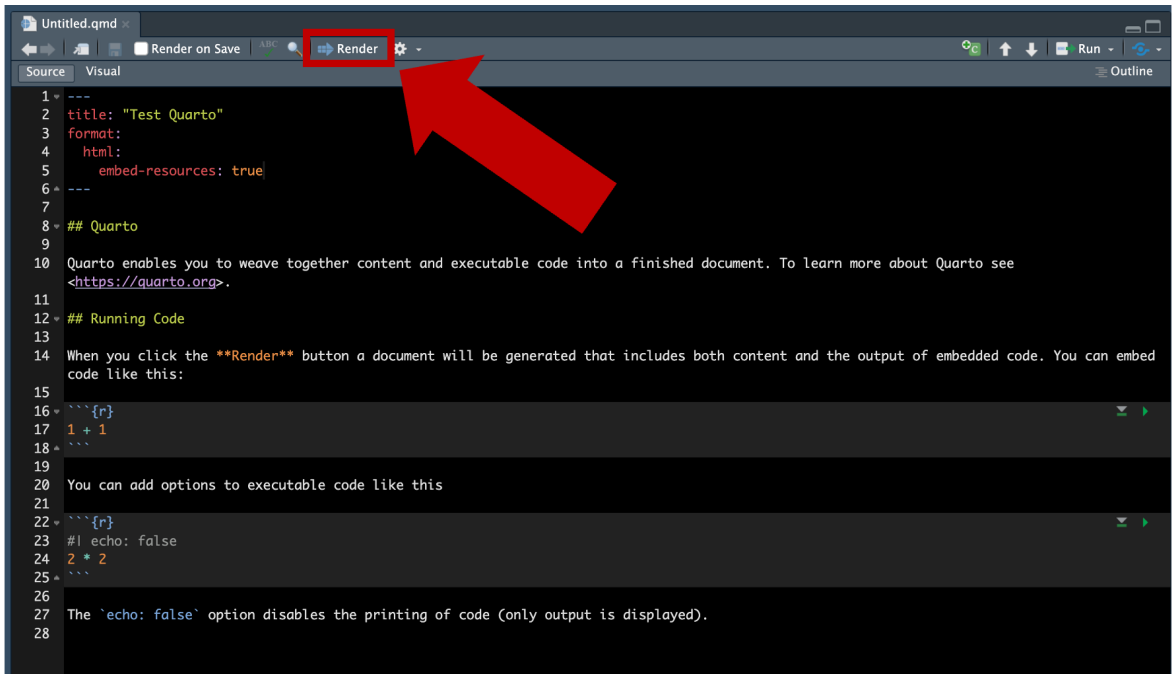


Figure 8.8: RStudio's render button. Only visible when a Quarto file is open.

R will ask you to save your Quarto file. After you save it, R will automatically create (or render) a new HTML file and save it in the same location where your Quarto file is saved. Additionally, a little browser window, like Figure 8.9 will pop up and give you a preview of what the rendered HTML file looks like.

Notice all the formatting that was applied when R rendered the HTML file. For example, the title – "Test Quarto" – is in big bold letters at the top of the screen, The headings – `Quarto` and `Running code` – are also written in a large bold font with a faint line underneath them,

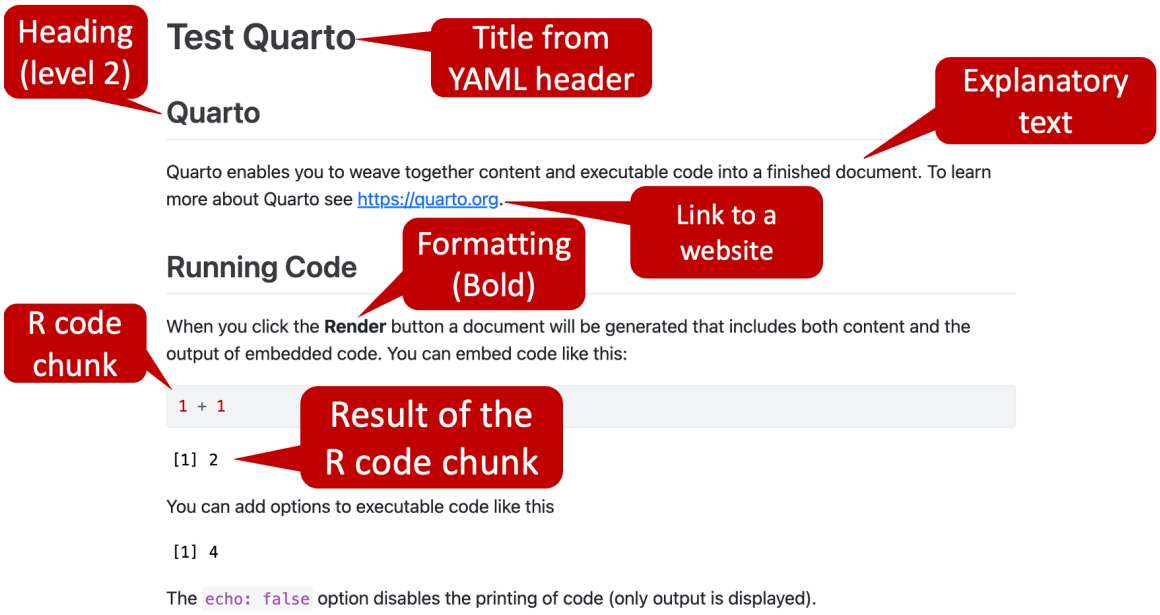


Figure 8.9: An HTML file created using a Quarto file.

the link to the Quarto website is now blue and clickable, and the word "Render" is written in bold font.

We can imagine that this section may seem a little confusing to some readers right now. If so, don't worry. You don't really *need* to understand the YAML header at this point. Remember, when you create a new Quarto file in the manner we described above, the YAML header is already there. You will probably want to change the title, but that may be the only change you make for now.

8.5 R code chunks

As we said above, R code chunks always start out with three backticks (‘) and a pair of curly braces ({}) with an "r" in them ({r}), and they always end with three more backticks. Typing that over and over can be tedious, so RStudio provides a keyboard shortcut for inserting R code chunks into our Quarto files.

On MacOS type `option + command + i`.

On Windows type `control + alt + i`

Inside the code chunk, we can type anything that we would otherwise type in the console or in an R script – including comments. We can then click the little green arrow in the top

right corner of the code chunk to submit it to R and see the result (see the play button in Figure 8.7).

Alternatively, we can run the code in the code chunk by typing `shift + command + return` on MacOS or `shift + control + enter` on Windows. If we want to submit a small section of code in a code chunk, as opposed to all of the code in the code chunk, we can use our mouse to highlight just the section of code we want to run and type `control + return` on MacOS or `control + enter` on Windows. There are also options to run all code chunks in the Quarto file, all code chunks above the current code chunk, and all code chunks below the current chunk. You can access these, and other, run options using the `Run` button in the top right-hand corner of the Quarto file in RStudio (see Figure 8.10 below).

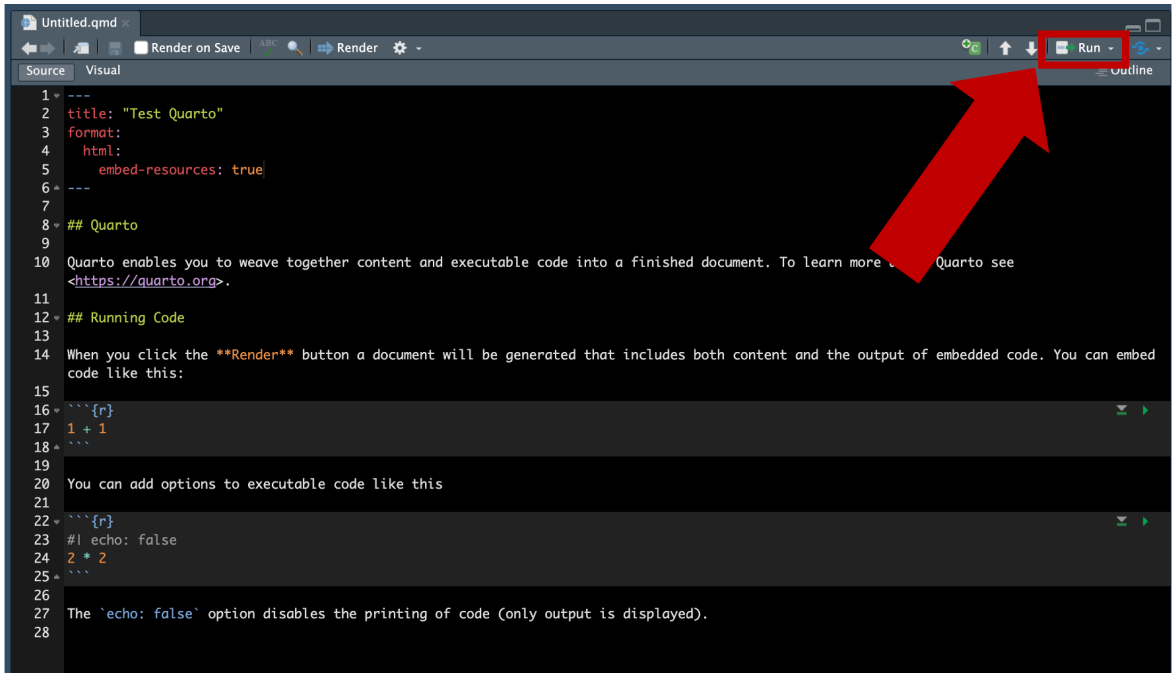


Figure 8.10: The run button in RStudio.

8.6 Markdown

Many readers have probably heard of HTML and CSS before. HTML stands for hypertext markup language and CSS stands for cascading style sheets. Together, HTML and CSS are used to create and style every website you've ever seen. HTML files created from our Quarto files are no different. They will open in any web browser and behave just like any other website. Therefore, we can manipulate and style them using HTML and CSS just like any other website. However, it takes most people a lot of time and effort to learn HTML and CSS.

So, markdown was created as an easier-to-use alternative. Think of it as HTML and CSS lite. It can't fully replace HTML and CSS, but it is much easier to learn, and you can use it to do many of the main things you might want to do with HTML and CSS. For example, Figure 8.7 and Figure 8.9 we saw that wrapping our text with two asterisks (`**`) bolds it.

There are a ton of other things we can do with markdown, and we recommend checking out Quarto's markdown basics website to learn more. The website covers a lot and may feel overwhelming at first. So, we suggest just play around with some of the formatting options and get a feel for what they do. Having said that, it's totally fine if you don't try to tackle learning markdown syntax right now. You don't really *need* markdown to follow along with the rest of the book. However, we still suggest using Quarto files for writing, saving, modifying, and sharing your R code.

8.6.1 Markdown headings

While we are discussing markdown, we would like to call special attention to markdown headings. We briefly glazed over them above, but we find that beginning R users typically benefit from a slightly more detailed discussion. Think back to the `## Quarto` on line 8 of Figure 8.7. This markdown created a heading – text that stands out and breaks our document up into sections. We can create headings by beginning a line in our Quarto document with one or more hash symbols (`#`), followed by a space, and then our heading text. Headings can be nested underneath each other in the same way you might nest topics in a bulleted list. For example:

- Animals
    - Dog
        - Lab
        - Yorkie
    - Cat
- Plants
    - Flowers
    - Trees
        - Oak

Nesting list items this way organizes our list and conveys information that would otherwise require explicitly writing out more text. For example, that a lab is a type of dog and that dogs are a type of animal. Thoughtfully nesting our headings in our Quarto files can have similar benefits. So, how do we nest our headings? Great question! Quarto and RStudio will automatically nest them based on the number of hash symbols we use (between 1 and 6). In the example above, `## Quarto` it is a second-level heading. We know this because the line

begins with two hash symbols. Figure 8.11 below shows how we might organize a Quarto file for a data analysis project into nested sections using markdown headings.

A really important benefit of organizing our Quarto file this way is that it allows us to use RStudio's `document outline` pane to quickly navigate around our Quarto file. In this trivial example, it isn't such a big deal. But it can be a huge time saver in a Quarto file with hundreds, or thousands, of lines of code.

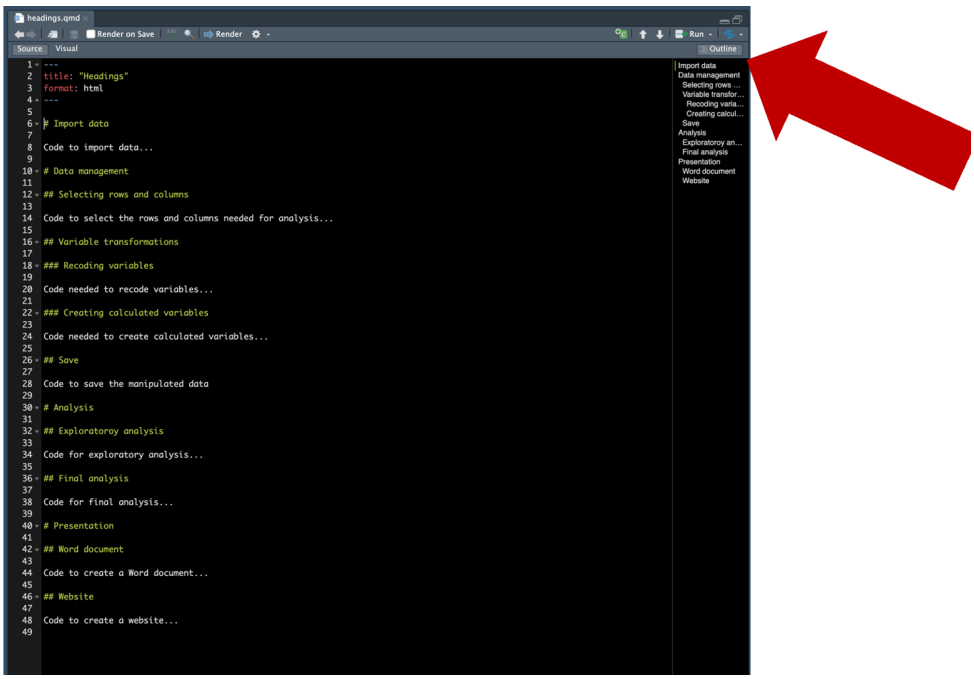


Figure 8.11: A Quarto file with nested headings.

As a final note on markdown headings, we find that new R users sometimes mix up comments and headings. This is a really understandable mistake to make because both start with the hash symbol. So, how do you know when typing a hash symbol will create a comment and when it will create a heading?

- The hash symbol always creates comments in *R scripts*. R scripts don't understand markdown. Therefore, they don't have markdown headings. R scripts only understand comments, which begin with a hash symbol, and R code.
- The hash symbol always creates markdown headings in Quarto files when typed *outside* of an R code chunk. Remember, everything in between the R code chunks in our Quarto files is considered markdown by Quarto, and hash symbols create headings in the markdown language.

- The hash symbol always creates comments in Quarto files when typed *inside* of an R code chunk. Remember, we can think of each R code chunk as a mini R script, and in R scripts, hash symbols create comments.

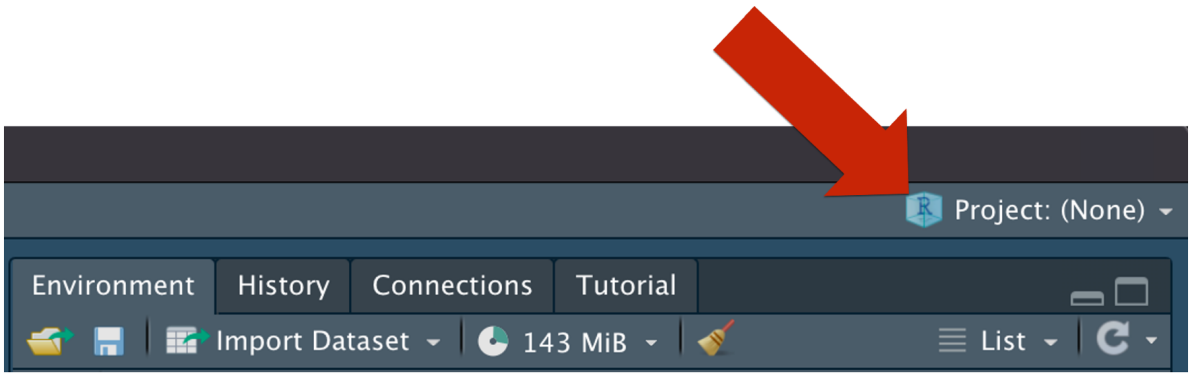
8.7 Summary

Quarto files bring together R code, formatted text, and media in a single file. We can use them to make our lives easier when working on small projects that are just for us, and we can use them to create large complex documents, websites, and applications that are intended for much larger audiences. RStudio makes it easy for us to create and render Quarto files into many different document types, and learning a little bit of markdown can help us format those documents really nicely. We believe that Quarto files are a great default file type to use for most projects and we encourage readers to review the Quarto website for more details (and inspiration)!

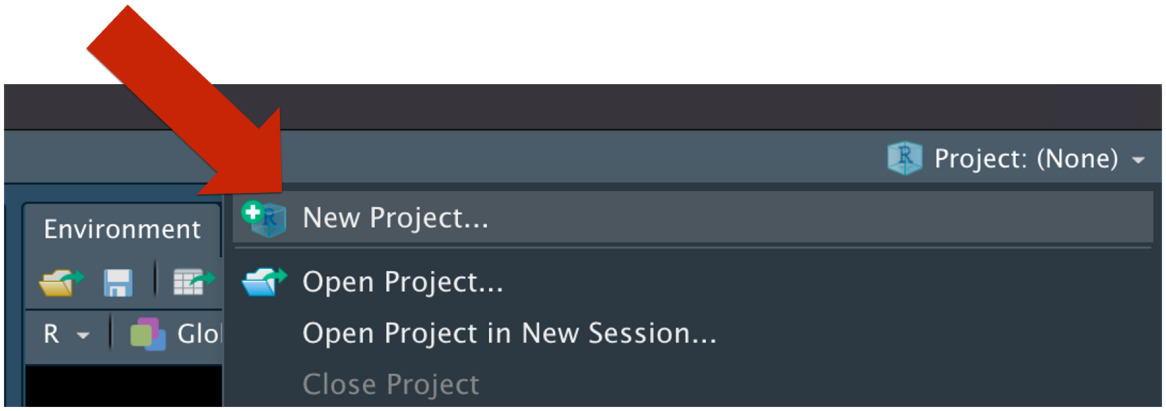
9 R Projects

In previous chapters of this book, we learned how to use R scripts and [R markdown] files to create, modify, save, and share our R code and results. However, in most real-world projects we will actually create *multiple* different R scripts and/or R markdown files. Further, we will often have other files (e.g., images or data) that we want to store alongside our R code files. Over time, keeping up with all of these files can become cumbersome. **R projects** are a great tool for helping us organize and manage collections of files. Another *really* important advantage to organizing our files into R projects is that they allow us to use **relative file paths** instead of **absolute file paths**, which we will [discuss in detail later][File paths].

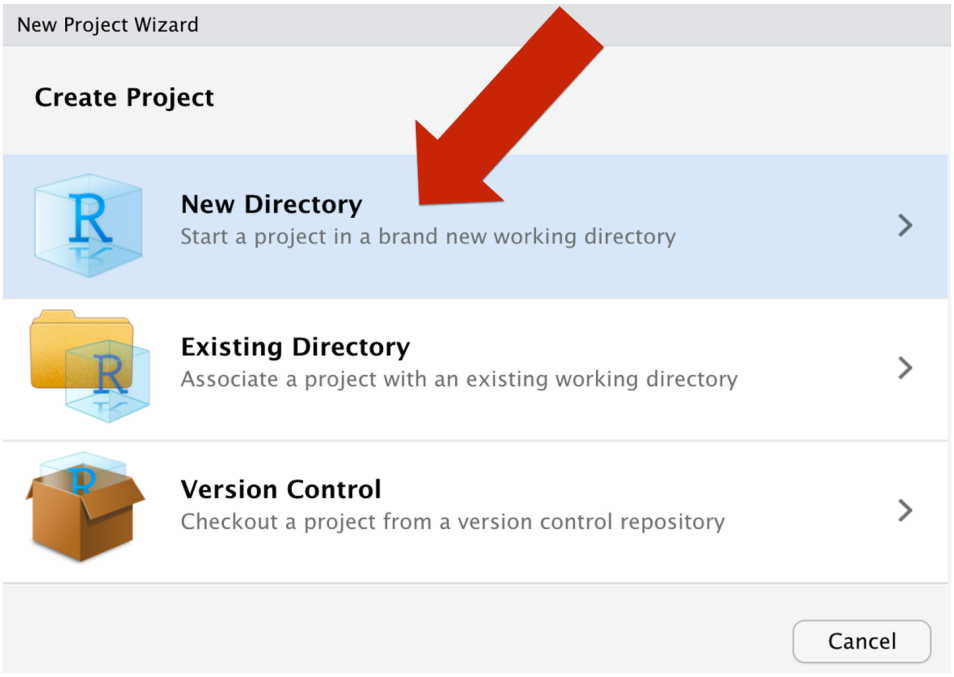
RStudio makes creating R projects really simple. For starters, let's take a look at the top right corner of our RStudio application window. Currently, we see an R project icon that looks like little blue 3-dimensional box with an "R" in the middle. To the right of the R project icon, we see words `Project: (None)`. RStudio is telling us that our current session is not associated with an R project.



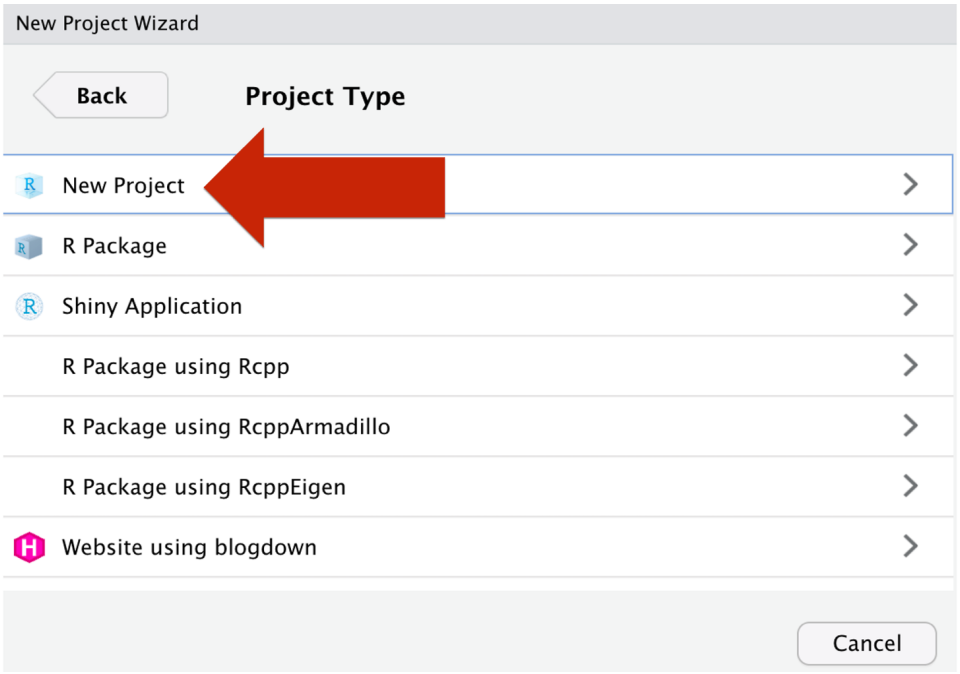
To create a new R project, we just need to click the drop-down arrow next to the words `Project: (None)` to open the projects menu. Then, we will click the `New Project...` option.



Doing so will open the new project wizard. For now, we will select the `New Directory` option. We will discuss the other options later in the book.



Next, we will click the `New Project` option.

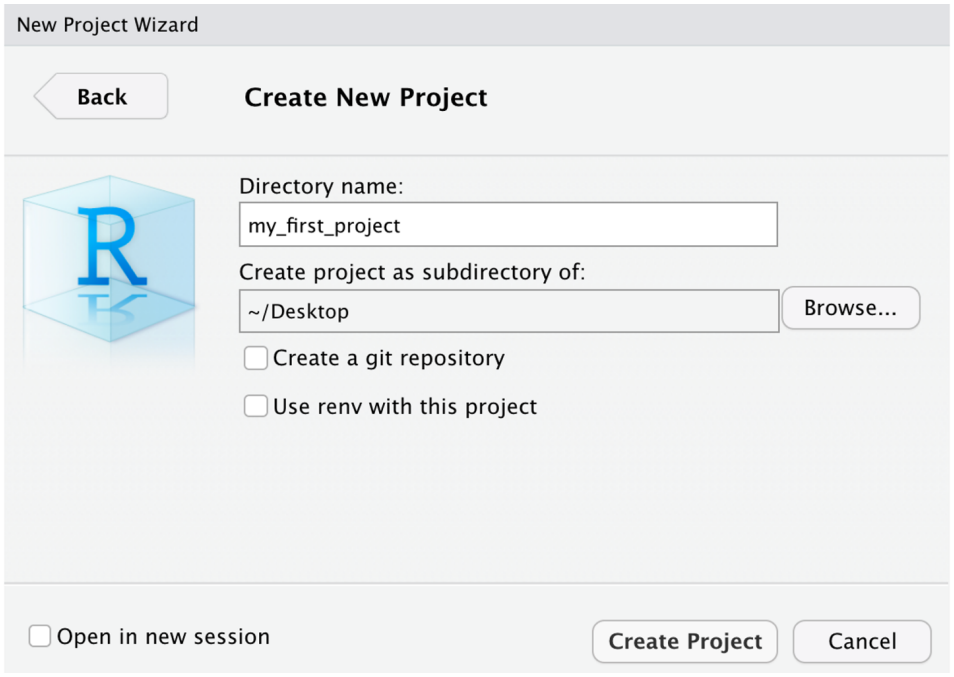


In the next window, we will have to make some choices and enter some information. The fist thing we will have to do is name our project. We do so by entering a value in the `Directory name:` box. Often, we can name our R project directory to match the name of the larger project

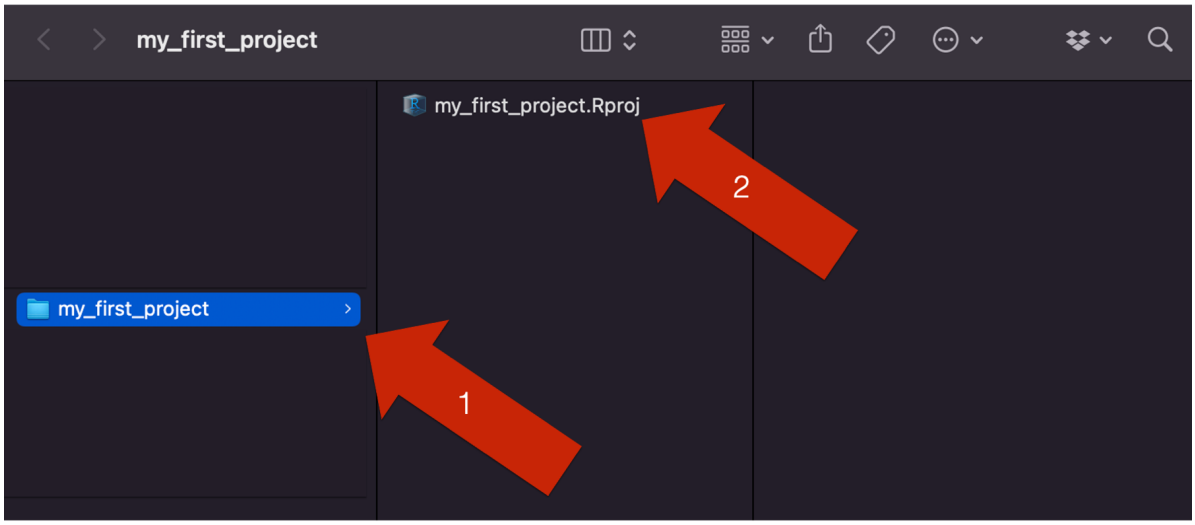
we are working on in a pretty natural way. If not, the name we choose for our project directory should essentially follow the same guidelines that we use for object (variable) names, which we will learn about soon. In this example, we went with the very creative `my_first_project` project name.

When we create our R project in a moment, RStudio will create a folder on our computer where we can keep all of the files we need for our project. That folder will be named using the name we entered in the `Directory name:` box in the previous step. So, the next thing we need to do is tell R where on our computer to put the folder. We do so by clicking the `Browse...` button and selecting a location. For this example, we chose to create the project on our computer's desktop.

Finally, we just click the `Create Project` button near the bottom-right corner of the New Project Wizard.



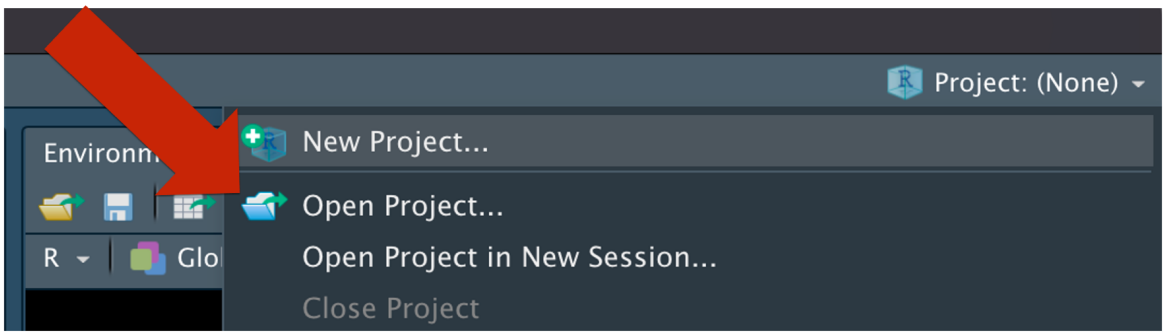
Doing so will create our new R project in the location we selected in the `Create project as subdirectory of:` text box in the new project wizard. In the screenshot below, we can see that a folder was created on our computer's desktop called `my_first_project`. Additionally, there is one file inside of that folder named `my_first_project` that ends with the file extension `.Rproj` (see red arrow 2 in the figure below).



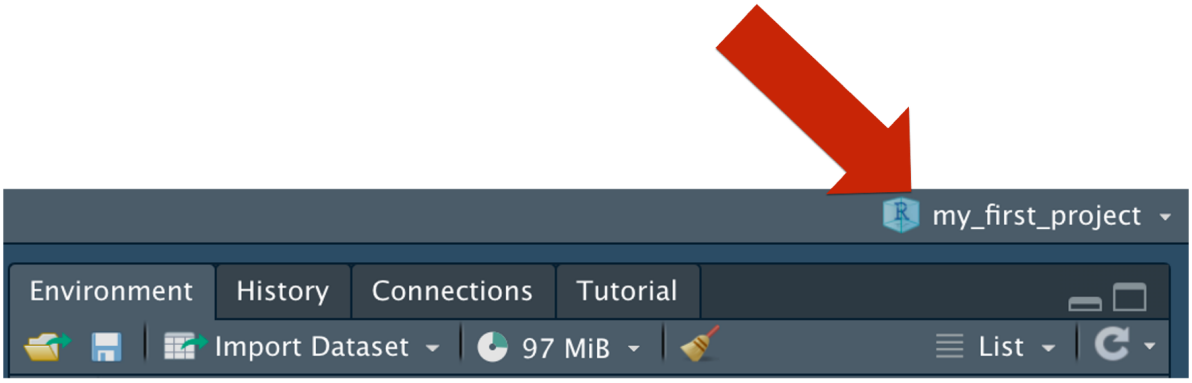
This file is called an R project file. Every time we create an R project, RStudio will create an R project file and add it to our project directory (i.e., the folder) for us. This file helps RStudio track and organize our R project.

To easiest way to open the R project we just created is to double click the R project file – `my_first_project.Rproj`. Doing so will open a new RStudio session along with all of the R code files we had open last time we were working on our R project. Because this is our first time opening our example R project, we won't see any R code files.

Alternatively, we can open our R project by once again clicking the R project icon in the upper right corner of an open RStudio session and then clicking the `Open Project...` option. This will open a file selection window where we can select our R project directory and open it.



Finally, we will know that RStudio understands that we are working in the context of our project because the words `Project: (None)` that we previously saw in the top right corner of the RStudio window will be replaced with the project name. In this case, `my_first_project`.



Now that we've created our R project, there's nothing special we need to do to add other files to it. We only need save files and folders for our project as we typically would. We just need to make sure that we save them in our project directory (i.e., the folder). RStudio will take care of the rest.

R projects are a great tool for organizing our R code and other complimentary files. Should we use them every single time we use R? Probably not. So, when should we use them? Well, the best – albeit somewhat unhelpful – answer is probably to use them whenever they are useful. However, at this point in your R journey you may not have enough experience to know when they will be useful and when they won't. Therefore, we are going to suggest that create an R project for your project if (1) your project will have more than one file and/or (2) more than one person will be working on the R code in your project. As we alluded to earlier, organizing our files into R projects allows us to use **relative file paths** instead of **absolute file paths**, which will make it much easier for us to collaborate with others. [File paths] will be discussed in detail later.

10 Coding Best Practices

At this point in the book, we've talked a little bit about what R is. We've also talked about the RStudio IDE and took a quick tour around its four main panes. Finally, we wrote our first little R program, which simulated and analyzed some data about a hypothetical class. Writing and executing this R program officially made you an *R programmer*.

However, you should know that not all R code is equally "good" – even when it's equally valid. What do we mean by that? Well, we already discussed the R interpreter and R syntax in the chapter on speaking R's language. Any code that uses R syntax that the R interpreter can understand is valid R code. But, is the R interpreter the only one reading your R code? No way! In epidemiology, we collaborate with others *all the time*! That collaboration is going to be much more efficient and enjoyable when there is good communication – including R code that is easy to read and understand. Further, you will often need to read and/or reuse code you wrote weeks, months, or years after you wrote it. You may be amazed at how quickly you forget what you did and/or why you did it that way. Therefore, in addition to writing valid R code, this chapter is about writing "good" R code – code that easily and efficiently communicates ideas to *humans*.

Of course, "good code" is inevitably somewhat subjective. Reasonable people can have a difference of opinion about the best way to write code that is easy to read and understand. Additionally, reasonable people can have a difference of opinion about when code is "good enough." For these reasons, we're going to offer several "suggestions" about writing good R code below, but only two general principles, which we believe most R programmers would agree with.

10.1 General principles

1. **Comment your code**. Whether you intend to share your code with other people or not, make sure to write lots of comments about what you are trying to accomplish in each section of your code and why.
2. **Use a style consistently**. We're going to suggest several guidelines for styling your R code below, but you may find that you prefer to style your R code in a different way. Whether you adopt our suggested style or not, please find or create a style that works for you and your collaborators and use it consistently.

10.2 Code comments

There isn't a lot of specific advice that we can give here because comments are so idiosyncratic to the task at hand. So, we think the best we can do at this point is to offer a few examples for you to think about.

10.2.1 Defining key variables

As we will discuss below, variables should have names that are concise, yet informative. However, the data you receive in the real world will not always include informative variable names. Even when someone has given the variables informative names, there may still be contextual information about the variables that is important to understand for data management and analysis. Some data sets will come with something called a **codebook** or **data dictionary**. These are text files that contain information about the data set that are intended to provide you with some of that more detailed information. For example, the survey questions that were used to capture the values in each variable or what category each value in a categorical variable represents. However, real data sets don't *always* come with a data dictionary, and even when they do, it can be convenient to have some of that contextual information close at hand, right next to your code. Therefore, we will sometimes comment our code with information about variables that are important for the analysis at hand. Here is an example from an administrative data set we ww using for an analysis:

```
* **Case number definition**

    - Case / investigation number.

* **Intake stage definition**

    - An ID number assigned to the Intake. Each Intake (Report) has its
      own number. A case may have more than one intake. For example, case # 12345
      has two intakes associated with it, 9 days apart, each with their own ID
      number. Each of the two intakes associated with this case have multiple
      allegations.

* **Intake start definition**

    - An intake is the submission or receipt of a report - a phone call or
      web-based. The Intake Start Date refers to the date the staff member
      opens a new record to begin recording the report.
```

10.2.2 What this code is trying to accomplish

Sometimes, it is obvious what a section of code literally *does*. but not so obvious why you're doing it. We often try to write some comments around our code about what it's trying to ultimately accomplish and why. For example:

```
## Standardize character strings

# Because we will merge this data with other data sets in the future based on
# character strings (e.g., name), we need to go ahead and standardize their
# formats here. This will prevent mismatches during the merges. Specifically,
# we:

# 1. Transform all characters to lower case
# 2. Remove any special characters (e.g., hyphens, periods)
# 3. Remove trailing spaces (e.g., "John Smith ")
# 4. Remove double spaces (e.g., "John  Smith")

vars <- quos(full_name, first_name, middle_name, last_name, county, address, city)

client_data <- client_data %>%
  mutate_at(vars(!!! vars), tolower) %>%
  mutate_at(vars(!!! vars), stringr::str_replace_all, "[^a-zA-Z\\d\\s]", " ") %>%
  mutate_at(vars(!!! vars), stringr::str_replace, "[[:blank:]]$", "") %>%
  mutate_at(vars(!!! vars), stringr::str_replace_all, "[[:blank:]]{2,}", " ")

rm(vars)
```

10.2.3 Why we chose this particular strategy

In addition to writing comments about why we did something, we sometimes write comments about why we did it *instead of* something else. Doing this can save you from having to relearn lessons you've already learned through trial and error but forgot. For example:

```
### Create exact match dummy variables

* We reshape the data from long to wide to create these variables because it significantly de
```

10.3 Style guidelines

UsInG c_o_n_s_i_s_t_e_n_t STYLE i.s. import-ant!

> Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations… Good style is important because while your code only has one author, it'll usually have multiple readers. This is especially true when you're writing code with others. In that case, it's a good idea to agree on a common style up-front. Since no style is strictly better than another, working with others may mean that you'll need to sacrifice some preferred aspects of your style.[5]

Below, we outline the style that we and our collaborators typically use when writing R code for a research project. It generally follows the Tidyverse style guide, *which we strongly suggest you read*. Outside of our class, you don't have to use our style, but you really should find or create a style that works for you and your collaborators and use it consistently.

10.3.1 Comments

Please put a space in between the pound/hash sign and the rest of your text when writing comments. For example, `# here is my comment` instead of `#here is my comment`. It just makes the comment easier to read.

10.3.2 Object (variable) names

In addition to the object naming guidance given in the Tidyverse style guide, We suggest the following object naming conventions.

10.3.3 Use names that are informative

Using names that are informative and easy to remember will make life easier for everyone who uses your data – including you!

```
# Uninformative names - Don't do this
x1
var1

# Informative names
employed
married
education
```

10.3.3.1 Use names that are concise

You want names to be informative, but you don't want them to be overly verbose. Really long names create more work for you and more opportunities for typos. In fact, we recommend using a single word when you can.

```
# Write out entire name of the study the data comes from - Don't do this
womens_health_initiative

# Write out an acronym for the study the data comes from - assuming everyone
# will be familiar with this acronym - Do this
whi
```

10.3.3.2 Use all lowercase letters

Remember, R is case-sensitive, which means that myStudyData and mystudydata are different things to R. Capitalizing letters in your file name just creates additional details to remember and potentially mess up. Just keep it simple and stick with lowercase letters.

```
# All upper case - so aggressive - Don't use
MYSTUDYDATA

# Camel case - Don't use
myStudyData

# All lowercase - Use
my_study_data
```

10.3.3.3 Separate multiple words with underscores.

Sometimes you really just need to use multiple words to name your object. In those cases, we suggested separating words with an underscore.

```
# Multiple words running together - Hard to read - Don't use
mycancerdata

# Camel case - easier to read, but more to remember and mess up - Don't use
myCancerData

# Separate with periods - easier to read, but doesn't translate well to many
# other languages. For example, SAS won't accept variable names with
```

```
# periods - Don't use
my.cancer.data

# Separate with underscores - Use
my_cancer_data
```

10.3.3.4 Prefix the names of similar variables

When you have multiple related variables, it's good practice to start their variable names with the same word. It makes these related variables easier to find and work with in the future if we need to do something with all of them at once. We can sort our variable names alphabetically to easily find find them. Additionally, we can use variable selectors like `starts_with("name")` to perform some operation on all of them at once.

```
# Don't use
first_name
last_name
middle_name

# Use
name_first
name_last
name_middle

# Don't use
street
city
state

# Use
address_street
address_city
address_state
```

10.3.4 File Names

All the variable naming suggestons above also apply to file names. However, we make a few additional suggestions specific to file names below.

10.3.4.1 Managing multiple files in projects

When you are doing data management and analysis for real-world projects you will typically need to break the code up into multiple files. If you don't, the code often becomes really difficult to read and manage. Having said that, finding the code you are looking for when there are 10, 20, or more separate files isn't much fun either. Therefore, we suggest the following (or similar) file naming conventions be used in your projects.

- Separate *data cleaning* and *data analysis* into separate files (typically, .R or .Rmd).
  - Data cleaning files should be prefixed with the word "data" and named as follows
    - data_[order number]_[purpose]

```
# Examples
data_01_import.Rmd
data_02_clean.Rmd
data_03_process_for_regression.Rmd
```

- Analysis files that do not directly create a table or figure should be prefixed with the word "analysis" and named as follows
  - analysis_[order number]_[brief summary of content]

```
# Examples
analysis_01_exploratory.Rmd
analysis_02_regression.Rmd
```

- Analysis files that *DO* directly create a table or figure should be prefixed with the word "table" or "fig" respectively and named as follows
  - table_[brief summary of content] or
  - fig_[brief summary of content]

```
# Examples
table_network_characteristics.Rmd
fig_reporting_patterns.Rmd
```

> **Note**
>
> **Side Note:** We sometimes do data manipulation (create variables, subset data, reshape data) in an analysis file if that analysis (or table or chart) is the only analysis that uses the modified data. Otherwise, we do the modifications in a separate data cleaning file.

- Images
  - Should typically be exported as png (especially when they are intended for use HTML files).
  - Should typically be saved in a separate "img" folder under the project home directory.
  - Should be given a descriptive name.
    - *Example*: `histogram_heights.png`, *NOT* `fig_02.png`.
  - We have found that the following image sizes typically work pretty well for our projects.
    - 1920 x 1080 for HTML
    - 770 x 360 for Word
- Word and PDF output files
  - We typically save them in a separate "docs" folder under the project home directory.
  - Whenever possible, we try to set the Word or PDF file name to match the name of the R file that it was created in.
    - *Example*: `first_quarter_report.Rmd` creates `docs/first_quarter_report.pdf`
- Exported data files (i.e., RDS, RData, CSV, Excel, etc.)
  - We typically save them in a separate "data" folder under the project home directory.
  - Whenever possible, we try to set the Word or PDF file name to match the name of the R file that it was created in.
    - *Example*: `data_03_texas_only.Rmd` creates `data/data_03_texas_only.csv`

11 Using Pipes

11.1 What are pipes?

What are pipes? This `|>` is the pipe operator. As of version 4.1, the pipe operator is part of base R. Prior to version 4.1, the pipe operator was only available from the magrittr. The pipe imported from the `magrittr` package looked like `%>%` and you may still come across it in R code – including in this book.

What does the pipe operator do? In our opinion, the pipe operator makes your R code *much* easier to read and understand.

How does it do that? It makes your R code easier to read and understand by allowing you to view your nested functions in the order you want them to execute, as opposed to viewing them literally nested inside of each other.

You were first introduced to nesting functions in the Let's get programming chapter. Recall that functions return values, and the R language allows us to directly pass those returned values into other functions for further calculations. We referred to this as nesting functions and said it was a big deal because it allows us to do very complex operations in a scalable way, without storing a bunch of unneeded intermediate objects in our global environment.

In that chapter, we also discussed a potential downside of nesting functions. Namely, our R code can become really difficult to read when we start nesting lots of functions inside one another.

Pipes allow us to retain the benefits of nesting functions without making our code really difficult to read. At this point, we think it's best to show you an example. In the code below we want to generate a sequence of numbers, then we want to calculate the log of each of the numbers, and then find the mean of the logged values.

```
# Performing an operation using a series of steps.
my_numbers <- seq(from = 2, to = 100, by = 2)
my_numbers_logged <- log(my_numbers)
mean_my_numbers_logged <- mean(my_numbers_logged)
mean_my_numbers_logged
```

```
[1] 3.662703
```

Here's what we did above:

- We created a vector of numbers called `my_numbers` using the `seq()` function.
- Then we used the `log()` function to create a new vector of numbers called `my_numbers_logged`, which contains the log values of the numbers in `my_numbers`.
- Then we used the `mean()` function to create a new vector called `mean_my_numbers_logged`, which contains the mean of the log values in `my_numbers_logged`.
- Finally, we printed the value of `mean_my_numbers_logged` to the screen to view.

The obvious first question here is, "why would I ever want to do that?" Good question! You probably won't ever want to do what we just did in the code chunk above, but we haven't learned many functions for working with real data yet and we don't want to distract you with a bunch of new functions right now. Instead, we want to demonstrate what pipes do. So, we're stuck with this silly example.

What's nice about the code above? We would argue that it is pretty easy to read because each line does one thing and it follows a series of steps in logical order. First, create the numbers. Second, log the numbers. Third, get the mean of the logged numbers.

What could be better about the code above? All we really wanted was the mean value of the logged numbers (i.e., `mean_my_numbers_logged`); however, on our way to getting `mean_my_numbers_logged` we also created two other objects that we don't care about – `my_numbers` and `my_numbers_logged`. It took us time to do the extra typing required to create those objects, and those objects are now cluttering up our global environment. It may not seem like that big of a deal here, but in a real data analysis project these things can really add up.

Next, let's try nesting these functions instead:

```
# Performing an operation using nested functions.
mean_my_numbers_logged <- mean(log(seq(from = 2, to = 100, by = 2)))
mean_my_numbers_logged
```

```
[1] 3.662703
```

Here's what we did above:

- We created a vector of numbers called `mean_my_numbers_logged` by nesting the `seq()` function inside of the `log()` function and nesting the `log()` function inside of the `mean()` function.
- Then, we printed the value of `mean_my_numbers_logged` to the screen to view.

What's nice about the code above? It is certainly more efficient than the sequential step method we used at first. We went from using 4 lines of code to using 2 lines of code, and we didn't generate any unneeded objects.

What could be better about the code above? Many people would say that this code is harder to read than than the the sequential step method we used at first. This is primarily due to the fact that each line no longer does one thing, and the code no longer follows a sequence of steps from start to finish. For example, the final operation we want to do is calculate the mean, but the `mean()` function is the first function we see when we read the code.

Finally, let's try see what this code looks like when we use pipes:

```
# Performing an operation using pipes.
mean_my_numbers_logged <- seq(from = 2, to = 100, by = 2) |>
  log() |>
  mean()
mean_my_numbers_logged
```

```
[1] 3.662703
```

Here's what we did above:

- We created a vector of numbers called `mean_my_numbers_logged` by passing the result of the `seq()` function directly to the `log()` function using the pipe operator, and passing the result of the the `log()` function directly to the `mean()` function using the pipe operator.
- Then, we printed the value of `mean_my_numbers_logged` to the screen to view.

As you can see, by using pipes we were able to retain the benefits of performing the operation in a series of steps (i.e., each line of code does one thing and they follow in sequential order) and the benefits of nesting functions (i.e., more efficient code).

The utility of the pipe operator may not be immediately apparent to you based on this very simple example. So, next we're going to show you a little snippet of code from one of our research projects. In the code chunk that follows, the operation we're trying to perform on the data is written in two different ways – without pipes and with pipes. It's very unlikely that you will know what this code does, but that isn't really the point. Just try to get a sense of which version is easier for you to read.

```
# Nest functions without pipes
responses <- select(ungroup(filter(group_by(filter(merged_data, !is.na(incident_number)), in

# Nest functions with pipes
```

```
responses <- merged_data |>
  filter(!is.na(incident_number)) |>
  group_by(incident_number) |>
  filter(row_number() == 1) |>
  ungroup() |>
  select(date_entered, detect_data, validation)
```

What do you think? Even without knowing what this code does, do you feel like one version is easier to read than the other?

11.2 How do pipes work?

Perhaps we've convinced you that pipes are generally useful. But, it may not be totally obvious to you *how* to use them. They are actually really simple. Start by thinking about pipes as having a left side and a right side.

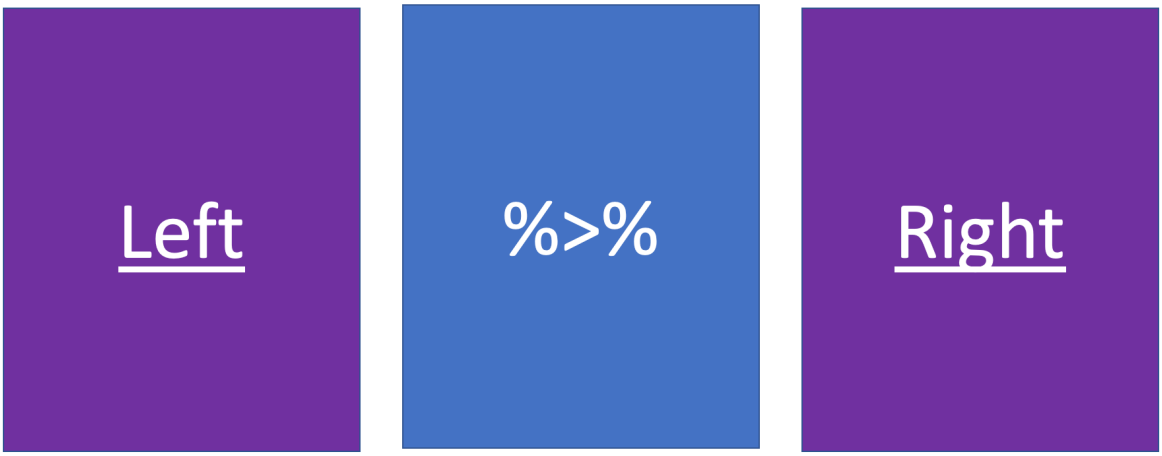


Figure 11.1: Pipes have a left side and a right side.

The thing on the right side of the pipe operator should always be a function.

The thing on the left side of the pipe operator can be a function or an object.

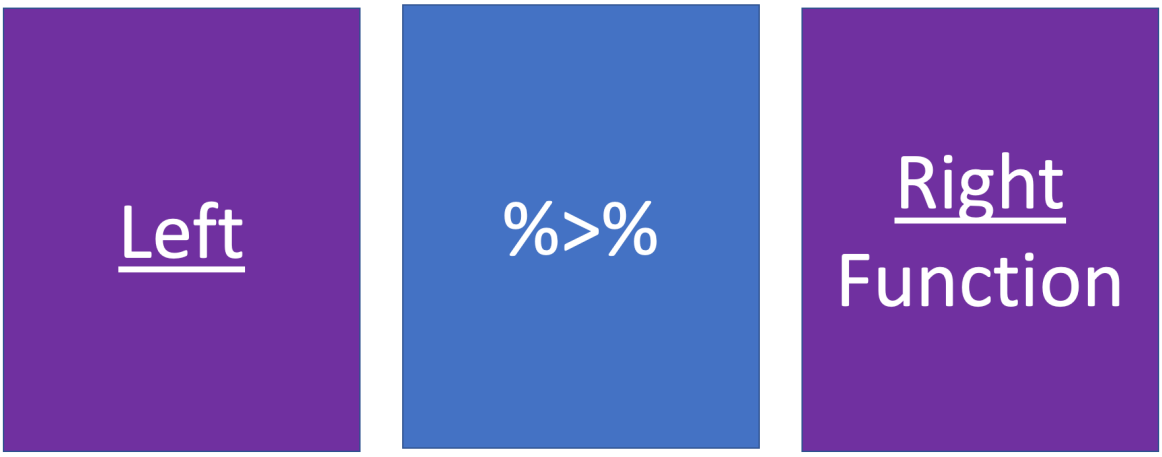


Figure 11.2: A function should always be to the right of the pipe operator.

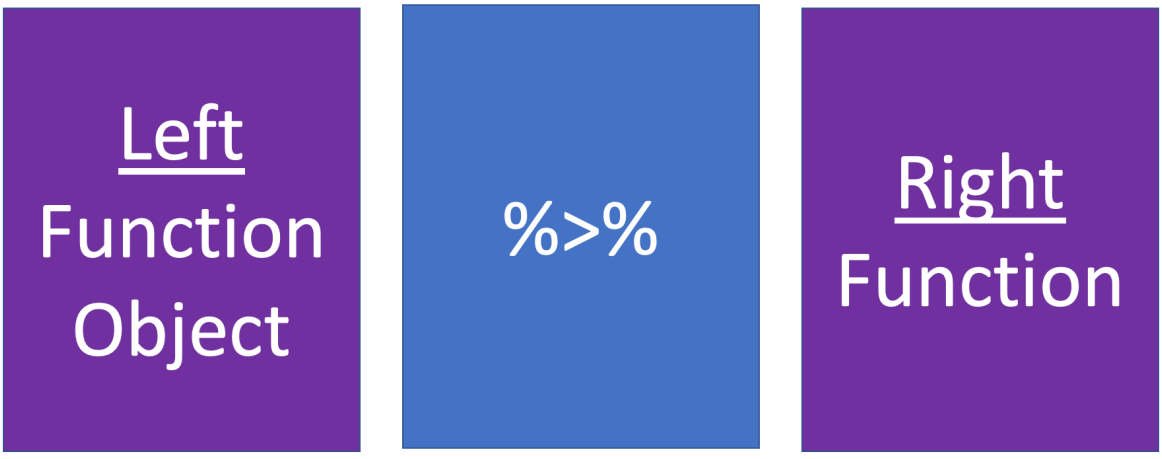


Figure 11.3: A function or an object can be to the left of the pipe operator.

All the pipe operator does is take the thing on the left side and pass it to the first argument of the function on the right side.

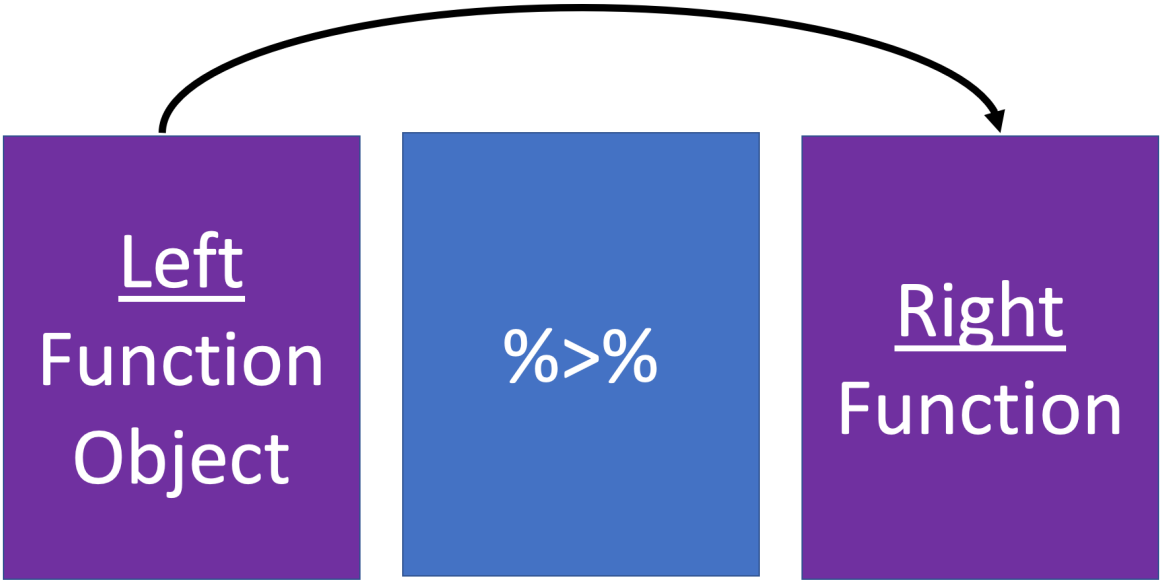


Figure 11.4: Pipe the left side to the first argument of the function on the right side.

It's a really simple concept, but it can also cause people a lot of confusion at first. So, let's take look at a couple more concrete examples.

Below we pass a vector of numbers to the to the `mean()` function, which returns the mean value of those numbers to us.

```
mean(c(2, 4, 6, 8))
```

```
[1] 5
```

We can also use a pipe to pass that vector of numbers to the `mean()` function.

```
c(2, 4, 6, 8) |> mean()
```

```
[1] 5
```

So, the R interpreter took the thing on the left side of the pipe operator, stuck it into the first argument of the function on the right side of the pipe operator, and then executed the function. In this case, the `mean()` function doesn't require any other arguments, so we don't have to write anything else inside of the `mean()` function's parentheses. When we see `c(2, 4, 6, 8) |> mean()`, R sees `mean(c(2, 4, 6, 8))`

Here's one more example. Pretty soon we will learn how to use the `filter()` function from the `dplyr` package to keep only a subset of rows from our data frame. Let's start by simulating some data:

```
# Simulate some data
height_and_weight <- tibble(
  id     = c("001", "002", "003", "004", "005"),
  sex    = c("Male", "Male", "Female", "Female", "Male"),
  ht_in  = c(71, 69, 64, 65, 73),
  wt_lbs = c(190, 176, 130, 154, 173)
)

height_and_weight
```

```
# A tibble: 5 x 4
  id    sex    ht_in wt_lbs
  <chr> <chr>  <dbl>  <dbl>
1 001   Male      71    190
2 002   Male      69    176
3 003   Female    64    130
4 004   Female    65    154
5 005   Male      73    173
```

In order to work, the `filter()` function requires us to pass two values to it. The first value is the name of the data frame object with the rows we want to subset. The second is the condition used to subset the rows. Let's say that we want to do a subgroup analysis using only the females in our data frame. We could use the `filter()` function like so:

```
# First value = data frame name (height_and_weight)
# Second value = condition for keeping rows (when the value of sex is Female)
filter(height_and_weight, sex == "Female")
```

```
# A tibble: 2 x 4
  id    sex    ht_in wt_lbs
  <chr> <chr>  <dbl>  <dbl>
1 003   Female    64    130
2 004   Female    65    154
```

Here's what we did above:

- We kept only the rows from the data frame called `height_and_weight` that had a value of `Female` for the variable called `sex` using `dplyr`'s `filter()` function.

We can also use a pipe to pass the `height_and_weight` data frame to the `filter()` function.

```
# First value = data frame name (height_and_weight)
# Second value = condition for keeping rows (when the value of sex is Female)
height_and_weight |> filter(sex == "Female")
```

```
# A tibble: 2 x 4
  id    sex    ht_in wt_lbs
  <chr> <chr>  <dbl>  <dbl>
1 003   Female    64    130
2 004   Female    65    154
```

As you can see, we get the exact same result. So, the R interpreter took the thing on the left side of the pipe operator, stuck it into the first argument of the function on the right side of the pipe operator, and then executed the function. In this case, the `filter()` function needs a value supplied to two arguments in order to work. So, we wrote `sex == "Female"` inside of the `filter()` function's parentheses. When we see `height_and_weight |> filter(sex == "Female")`, R sees `filter(height_and_weight, sex == "Female")`.

> **Note**
>
> **Side Note:** This pattern – a data frame piped into a function, which is usually then piped into one or more additional functions is something that you will see over and over in this book.

Don't worry too much about how the `filter()` function works. That isn't the point here. The two main takeaways so far are:

1. Pipes make your code easier to read once you get used to them.
2. The R interpreter knows how to automatically take whatever is on the left side of the pipe operator and make it the value that gets passed to the first argument of the function on the right side of the pipe operator.

11.2.1 Keyboard shortcut

Typing `|>` over and over can be tedious! Thankfully, RStudio provides a keyboard shortcut for inserting the pipe operator into your R code.

On Mac type `shift + command + m`.

On Windows type `shift + control + m`

It may not seem totally intuitive at first, but this shortcut is really handy once you get used to it.

11.2.2 Pipe style

As with all the code we write, style is an important consideration. We generally agree with the recommendations given in the Tidyverse style guide. In particular:

1. We tend to use pipes in such a way that each line of code does one, and only one, thing.
2. If a line of code contains a pipe operator, the pipe operator should generally be the last thing typed on the line.
3. The pipe operator should always have a space in front of it.
4. If the pipe operator isn't the last thing typed on the line, then it should be have a space after it too.
5. "If the function you're piping into has named arguments (like `mutate()` or `summarize()`), put each argument on a new line. If the function doesn't have named arguments (like `select()` or `filter()`), keep everything on one line unless it doesn't fit, in which case you should put each argument on its own line."**Wickham2023-ta?**
6. "After the first step of the pipeline, indent each line by two spaces. RStudio will automatically put the spaces in for you after a line break following a `|>` . If you're putting each argument on its own line, indent by an extra two spaces. Make sure `)` is on its own line, and un-indented to match the horizontal position of the function name."**Wickham2023-ta?**

Each of these recommendations are demonstrated in the code below.

```
# Do this...
female_height_and_weight <- height_and_weight |> # Line 1
  filter(sex == "Female") |>                     # Line 2
  summarise(                                     # Line 3
    mean_ht = mean(ht_in),                       # Line 4
    sd_ht   = sd(ht_in)                          # Line 5
  ) |>                                           # Line 6
  print()                                        # Line 7
```

```
# A tibble: 1 x 2
  mean_ht sd_ht
    <dbl> <dbl>
1    64.5 0.707
```

In the code above, we would first like you to notice that each line of code does one, and only one, thing. Line 1 *only* assigns the result of the code pipeline to a new object – `female_height_and_weight`, line 2 *only* keeps the rows in the data frame we want – rows for females, line 3 *only* opens the `summarise()` function, line 4 *only* calculates the mean of the `ht_in` column, line 5 *only* calculates the standard deviation of the `ht_in` column, line 6 *only* closes the `summarise()` function, and line 7 *only* prints the result to the screen.

Second, we'd like you to notice that each line containing a pipe operator (i.e., lines 1, 2, and 6) *ends* with the pipe operator, and the pipe operators all have a space in front of them.

Third, we'd like you to notice that each named argument in the `summarise()` function is written on its own line (i.e., lines 4 and 5).

Finally, we'd like you notice that each step of the pipeline is indented two spaces (i.e., lines 2, 3, 6, and 7), lines 4 and 5 are indented an *additional* two spaces because they contain named arguments to the `summarise()` function, and that the `summarise()` function's closing parenthesis is on its own line (i.e., line 6), horizontally aligned with the "s" in "summarise(".

Now compare that with the code in the code chunk below.

```
# Avoid this...
female_height_and_weight <- height_and_weight |> filter(sex == "Female") |>
  summarise(mean_ht = mean(ht_in), sd_ht = sd(ht_in)) |> print()
```

```
# A tibble: 1 x 2
  mean_ht sd_ht
    <dbl> <dbl>
1    64.5 0.707
```

Although we get the same result as before, most people would agree that the code is harder to quickly glance at and read. Further, most people would also agree that it would be more difficult to add or rearrange steps when the code is written that way. As previously stated, there is a certain amount of subjectivity in what constitutes "good" style. But, we will once again reiterate that it is important to adopt *some* style and use it consistently. If you are a beginning R programmer, why not adopt the tried-and-true styles suggested here and adjust later if you have a compelling reason to do so?

11.3 Final thought on pipes

We think it's important to note that not everyone in the R programming community is a fan of using pipes. We hope that we've made a compelling case for why we use pipes, but we acknowledge that it is ultimately a preference, and that using pipes is not the best choice in all circumstances. Whether or not you choose to use the pipe operator is up to you; however, we will be using them extensively throughout the remainder of this book.

Part III

Collaboration

12 Using git and GitHub

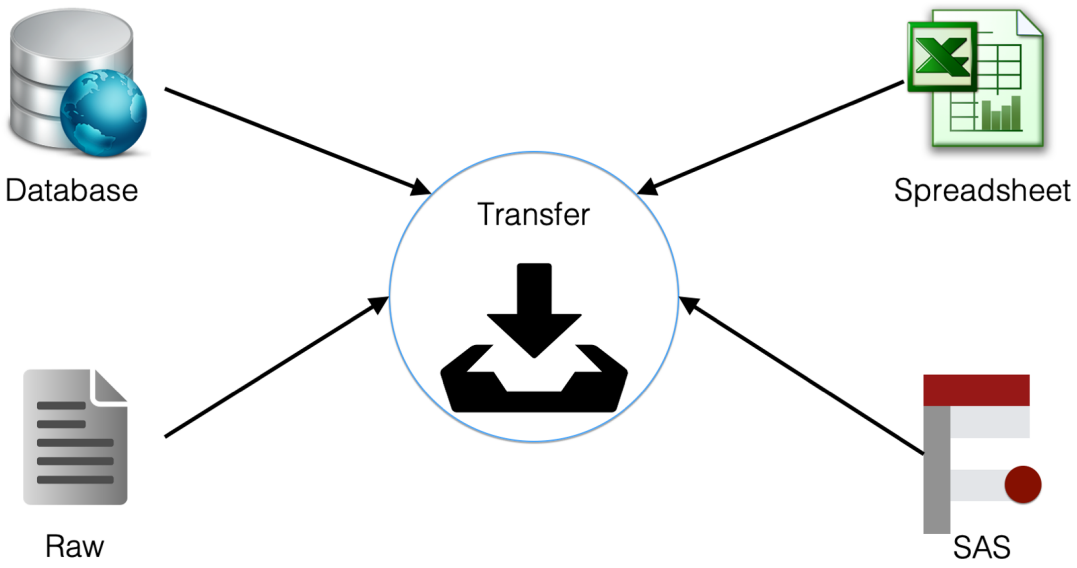
Part IV

Data Transfer

13 Introduction to Data Transfer

In previous chapters, we learned how to write our own simple R programs by directly creating data frames in RStudio with the `data.frame()` function, the `tibble()` function, and the `tribble()` function. We consider this to be a really fundamental skill to master because it allows us to simulate data and it allows us to get data into R regardless of what format that data is stored in (assuming we can "see" the stored data). In other words, if nothing else, we can always resort to creating data frames this way.

In practice, however, this is not how people generally exchange data. You might recall that in Section 2.2.1 Transferring data We briefly mentioned the need to get data into R that others have stored in various different **file types**. These file types are also sometimes referred to as **file formats**. Common examples encountered in epidemiology include database files, spreadsheets, text files, SAS data sets, and Stata data sets.



Further, the data frames we've created so far don't currently live in our global environment from one programming session to the next. We haven't yet learned how to efficiently store our data long-term. We think the limitations of having to manually create a data frame every time we start a new programming session are probably becoming obvious to you at this point.

In this part of the book, we will learn to **import** data stored in various different file types into R for data management and analysis, we will learn to store R data frames in a more permanent way so that we can come back later to modify or analyze them, and we will learn to **export** data so that we may efficiently share it with others.

Part V

References

References

1. Ismay C, Kim AY. Chapter 1 getting started with data in R. Published online November 2019.

2. Stack Overflow. What are tags, and how should I use them? Published online January 2022.

3. Stack Overflow. How do I ask a good question? Published online January 2022.

4. RStudio. FAQ: Tips for writing r-related questions. Published online September 2021.

5. Wickham H. Style guide. In: *Advanced R.*; 2019.

6. GitHub. *About Issues.* Github; 2024.

7. R Core Team. *What Is r?* R Foundation for Statistical Computing; 2024.

8. GitHub. About repositories. Published online December 2023.

9. RStudio. RStudio. Published online 2020.

A Glossary

Console The console is located in RStudio's bottom-right pane by default. The R console is an interactive programming environment where we can enter and execute R commands. It's the the most basic interface for interacting with R, providing immediate feedback and results from the code we enter. The R console is useful for testing small pieces of code and interactive data exploration. However, we recommend using R scripts or Quarto files for all but the simplest programming or data analysis tasks.

Data frame. For our purposes, data frames are just R's term for data set or data table. Data frames are made up of columns (variables) and rows (observations). In R, all columns of a data frame must have the same length.

Functions. Coming soon.

- **Arguments** Arguments always live *inside* the parentheses of R functions and receive information the function needs to generate the result we want.
- **Pass** In programming lingo, we *pass* a value to a function argument. For example, in the function call `seq(from = 2, to = 100, by = 2)` we could say that we *passed* a value of 2 to the `from` argument, we *passed* a value of 100 to the `to` argument, and we *passed* a value of 2 to the `by` argument.
- **Return** Instead of saying, "the `seq()` function *gives us* a sequence of numbers…" we could say, "the `seq()` function *returns* a sequence of numbers…" In programming lingo, functions *return* one or more results.

Global environment. Coming soon.

Issue (GitHub) GitHub's documentation says issues are "items you can create in a repository to plan, discuss and track work. Issues are simple to create and flexible to suit a variety of scenarios. You can use issues to track work, give or receive feedback, collaborate on ideas or tasks, and efficiently communicate with others."[6]

Objects. Coming soon.

R R's documentation says "R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John

Chambers and colleagues."[7] R is open source, and you can download it for free from The Comprehensive R Archive Network (CRAN) at https://cran.r-project.org/.

Repository GitHub's documentation says "a repository contains all of your code, your files, and each file's revision history. You can discuss and manage your work within the repository."[8] A repository can exist *locally* as a set of files on your computer. A repository can also exist *remotely* as a set of files on a sever somewhere, for example, on GitHub.

RStudio RStudio is an integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management. RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro (Debian/Ubuntu, Red Hat/CentOS, and SUSE Linux).[9]