R 4 Epidemiology

2024-06-18

Table of contents

Welcome **5**
Acknowledgements . . . 5

Introduction **6**
Goals . . . 6
Text conventions used in this book . . . 7
Other reading . . . 7

Contributing **8**
Typos . . . 8
Issues . . . 15
License Information . . . 15

About the Authors **16**
Brad Cannell . . . 16
Melvin Livingston . . . 17

I Getting Started **18**

1 Installing R and RStudio **19**
1.1 Download and install on a Mac . . . 19
1.2 Download and install on a PC . . . 27

2 What is R? **35**
2.1 What is data? . . . 35
2.2 What is R? . . . 40
2.2.1 Transferring data . . . 41
2.2.2 Managing data . . . 42
2.2.3 Analyzing data . . . 43
2.2.4 Presenting data . . . 44

3 Navigating the RStudio Interface **46**
3.1 The console . . . 47
3.2 The environment pane . . . 50
3.3 The files pane . . . 53

3.4 The source pane . . . 55
3.5 RStudio preferences . . . 58

4 Speaking R's Language 63
4.1 R is a *language* . . . 63
4.2 The R interpreter . . . 64
4.3 Errors . . . 64
4.4 Functions . . . 65
4.4.1 Passing values to function arguments . . . 67
4.5 Objects . . . 70
4.6 Comments . . . 72
4.7 Packages . . . 73
4.8 Programming style . . . 75

5 Let's Get Programming 76
5.1 Simulating data . . . 76
5.2 Vectors . . . 77
5.2.1 Vector types . . . 78
5.2.2 Double vectors . . . 79
5.2.3 Integer vectors . . . 79
5.2.4 Logical vectors . . . 80
5.2.5 Factor vectors . . . 80
5.3 Data frames . . . 84
5.4 Tibbles . . . 86
5.4.1 The as_tibble function . . . 87
5.4.2 The tibble function . . . 88
5.4.3 The tribble function . . . 89
5.4.4 Why use tibbles . . . 91
5.5 Missing data . . . 92
5.6 Our first analysis . . . 94
5.6.1 Manual calculation of the mean . . . 94
5.6.2 Dollar sign notation . . . 95
5.6.3 Bracket notation . . . 95
5.6.4 The sum function . . . 96
5.6.5 Nesting functions . . . 97
5.6.6 The length function . . . 99
5.6.7 The mean function . . . 100
5.7 Some common errors . . . 101
5.8 Summary . . . 102

II Coding Tools and Best Practices **103**

6 Quarto Files **104**

III Collaboration **105**

7 Using git and GitHub **106**

IV References **107**

References **108**

Appendices **109**

A Glossary **109**

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

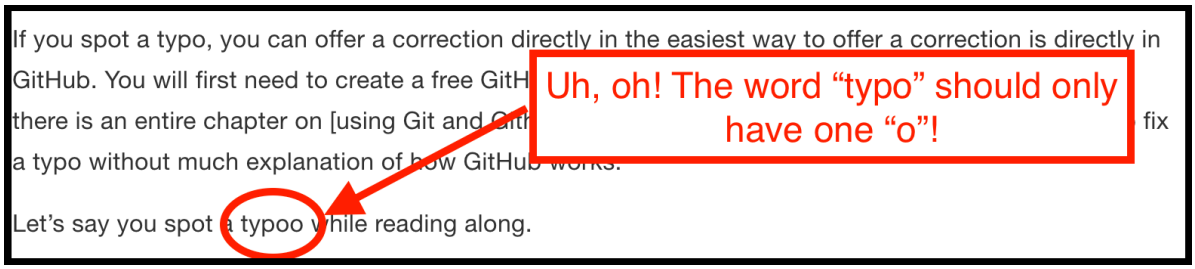
Over the years, we have learned so much from our students and colleagues, and we anticipate that there is much more we can learn from you – our readers. Therefore, we welcome and appreciate all constructive contributions to R4Epi!

Typos

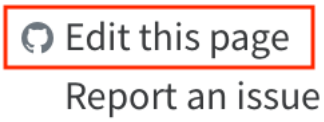
The easiest way for you to contribute is to help us clean up the little typos and grammatical errors that inevitably sneak into the text.

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 7). Here, we're just going to walk you through how to fix a typo without much explanation of how GitHub works.

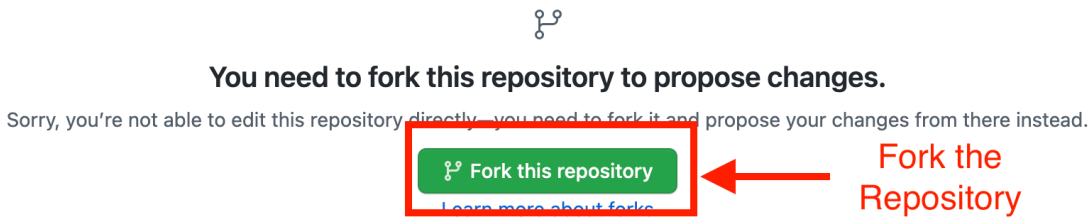
Let's say you spot a typo while reading along.



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



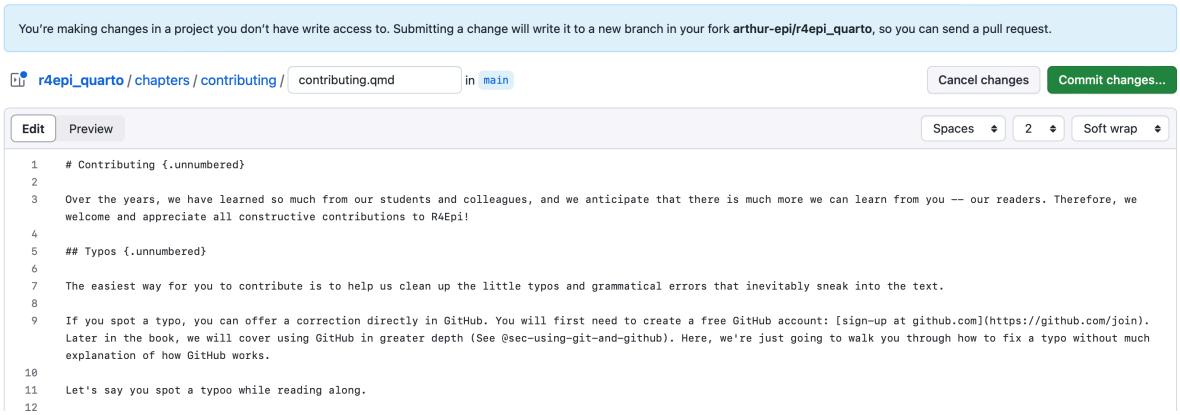
The first time you click the icon, you will be taken to the R4Epi repository on GitHub and asked to fork it. For our purposes, you can think of a GitHub repository as being similar to a shared folder on Dropbox or Google Drive.



"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.

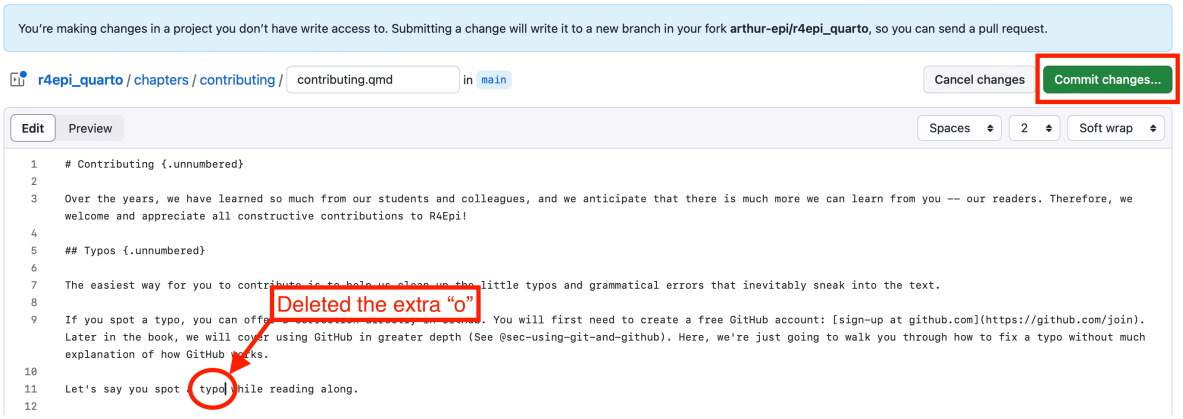
> **Note**
>
> Forking the R4Epi repository does not cost any money or add any files to your computer.

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

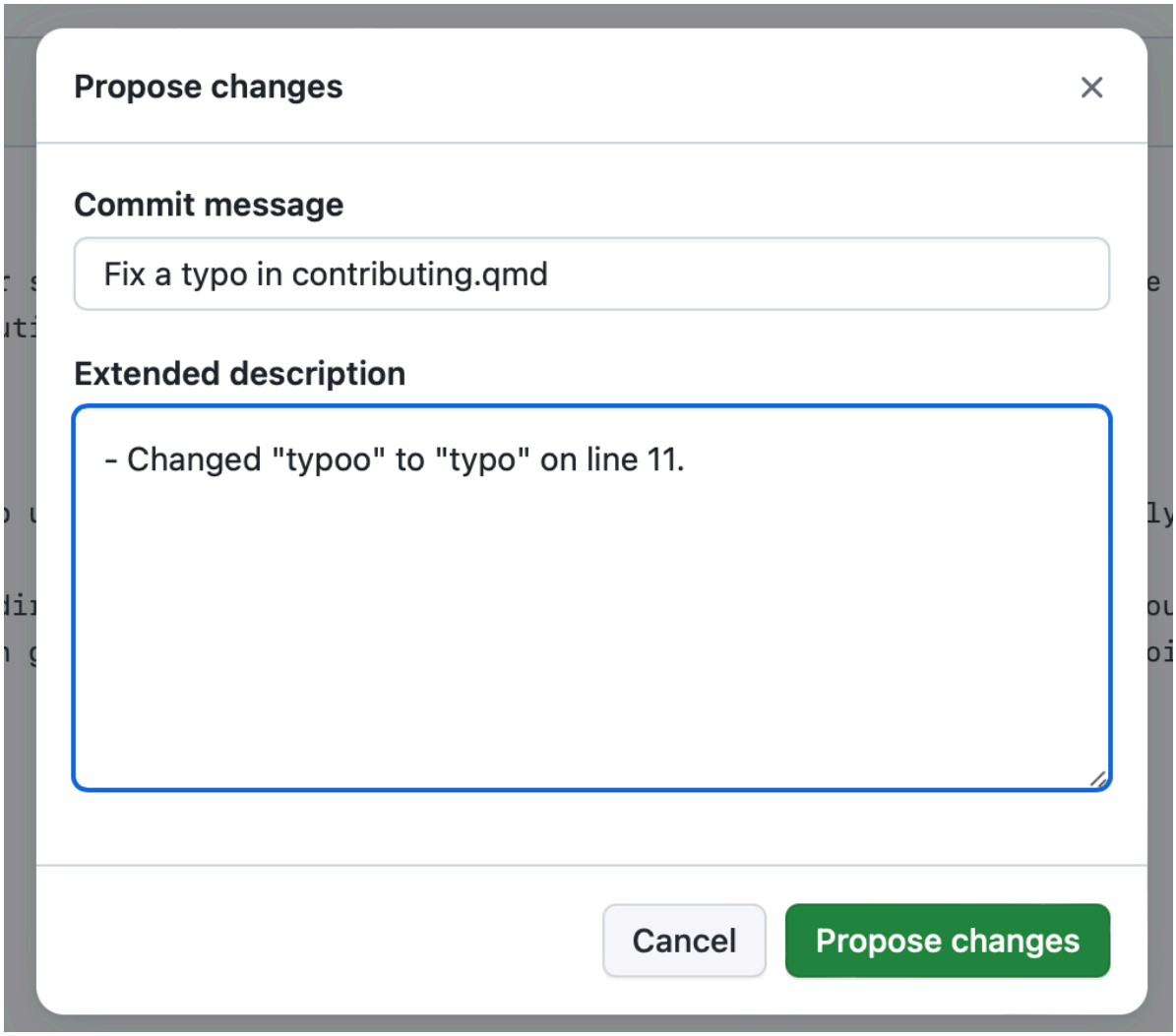


The text editor will display the contents of the file used to make the chapter you were looking at when you clicked the `edit` button. In this example, it was a file named `contributing.qmd`. The `.qmd` file extension means that the file is a Quarto file. We will learn more about Quarto files in Chapter 6, 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.

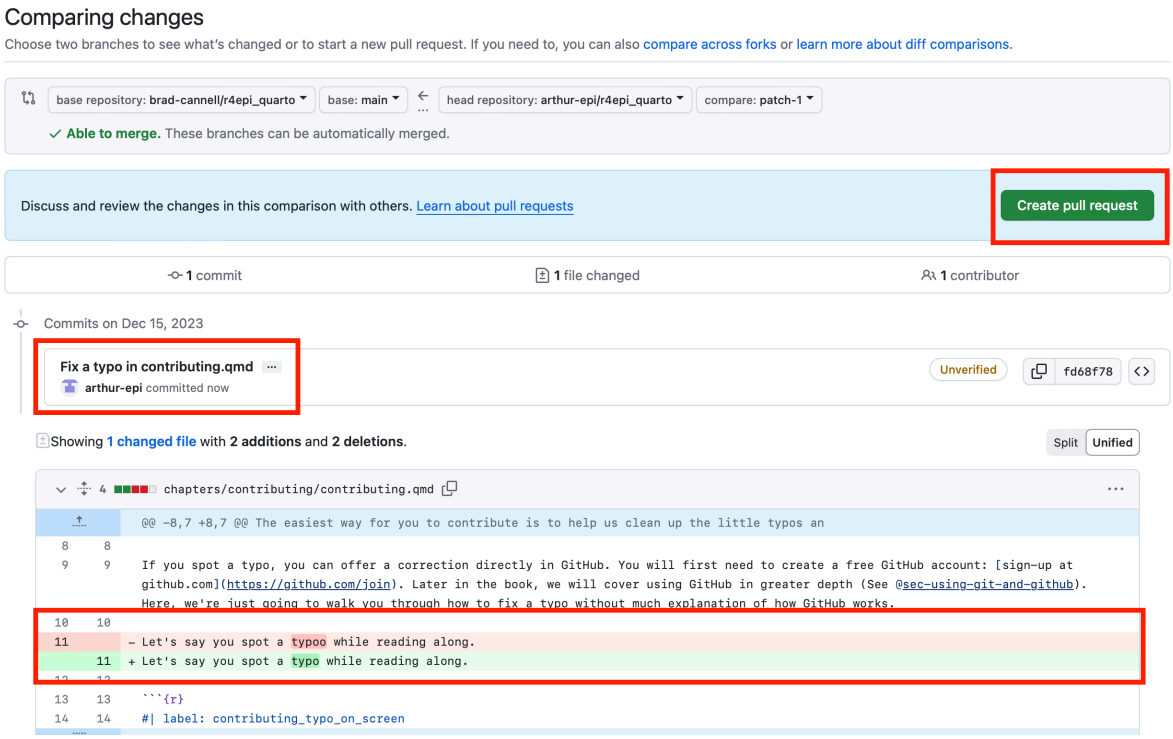
Next, scroll down through the text until you find the typo and fix it. In this case, line 11 contains the word "typoo". To fix it, you just need to click in the editor window and begin typing. In this case, you would click next to the word "typoo" and delete the second "o".



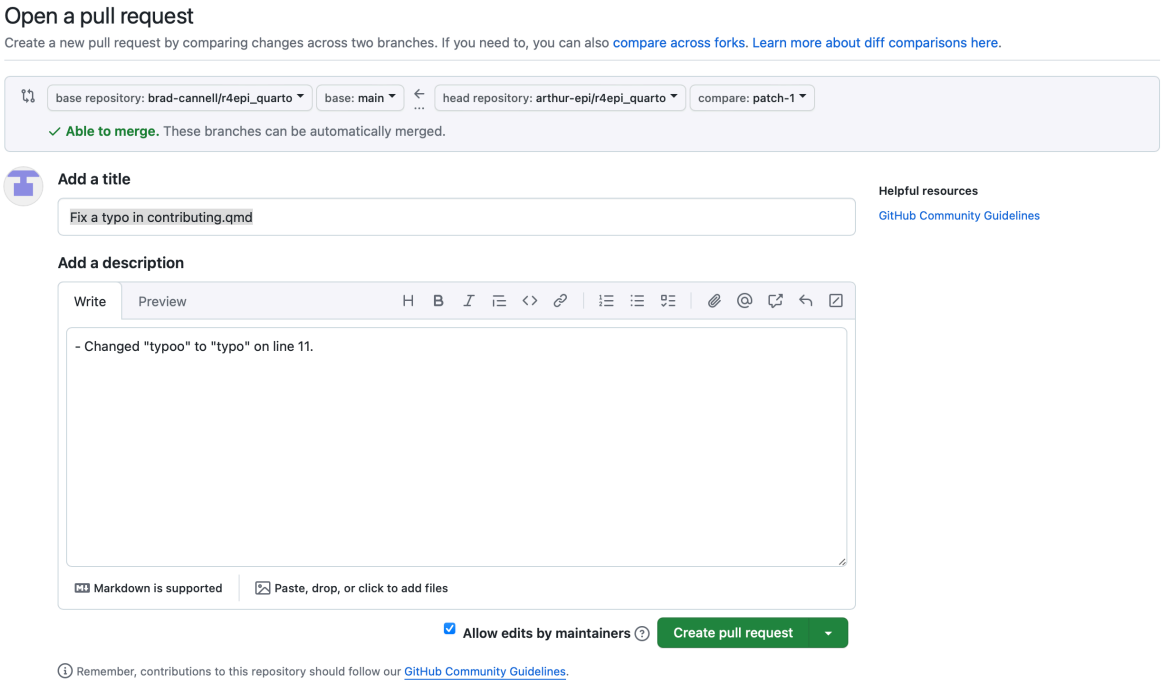
Now, the only thing left to do is propose your typo fix to the authors. To do so, click the green `Commit changes...` button on the right side of the screen above the text editor (surrounded with a red box in the screenshot above). When you click it, a new `Propose changes` box will appear on your screen. 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.

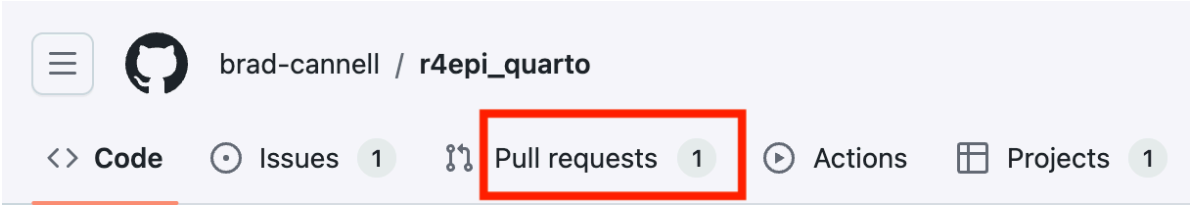


For now, we will focus on the three different sections of the screen that are highlighted with a red outline. We will start at the bottom and work our way up. The red box that is closest to the bottom of the screenshot shows us that the change that made was on line 11. The word "typoo" (highlighted in 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.

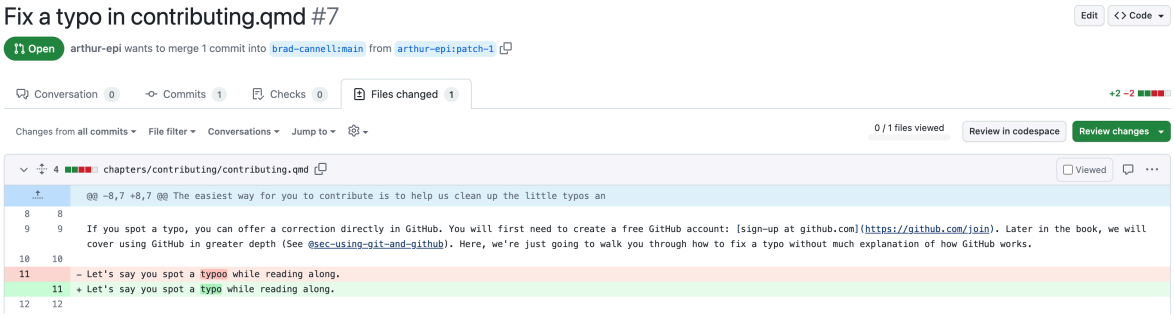


After doing so, you will get one final chance to amend the description of your proposed changes. If you are happy with the commit message and description, then click the `Create pull request` button one more time. At this point, your job is done! It is now up to the authors to review the changes you've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

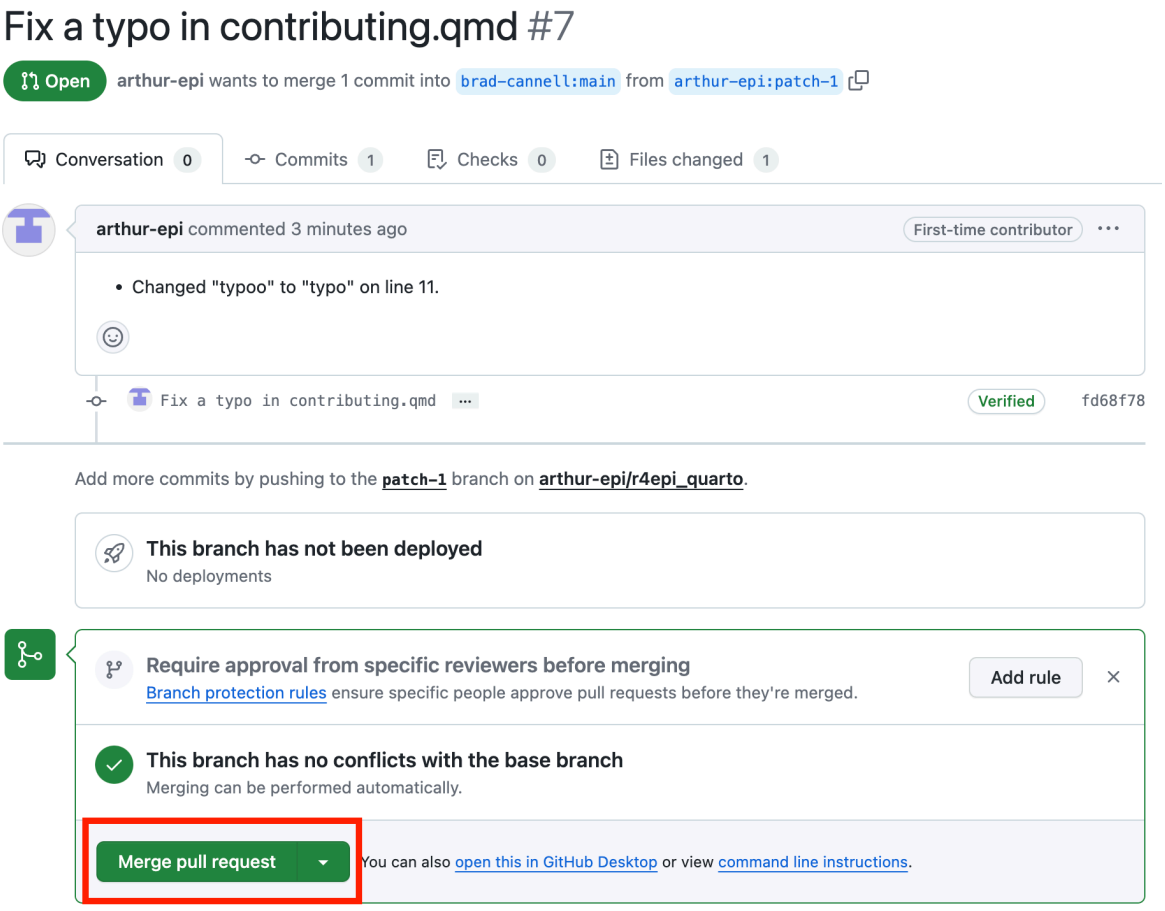
In case you are curious, here is what the process looks like on the authors' end. First, when we open the R4Epi repository page on GitHub, we will see that there is a new pull request.



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

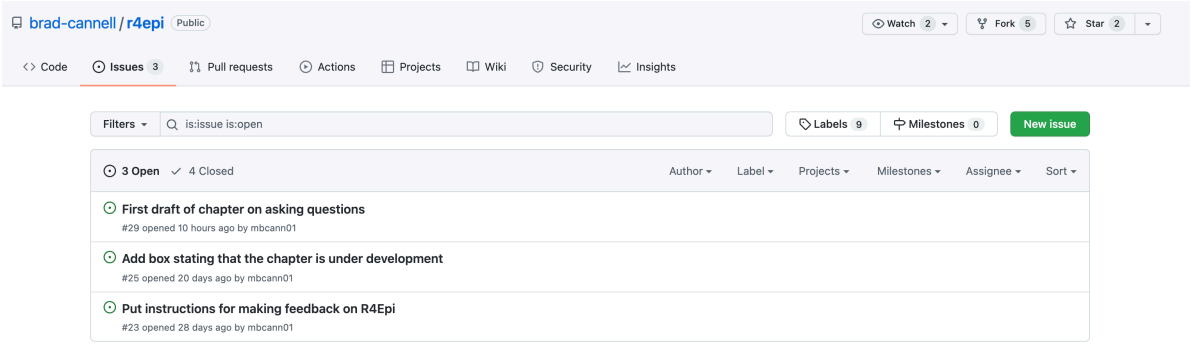


Then, all we have to do is click the `Merge pull request button` and the fixed file is "pulled in" to replace the file with the typo.



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.

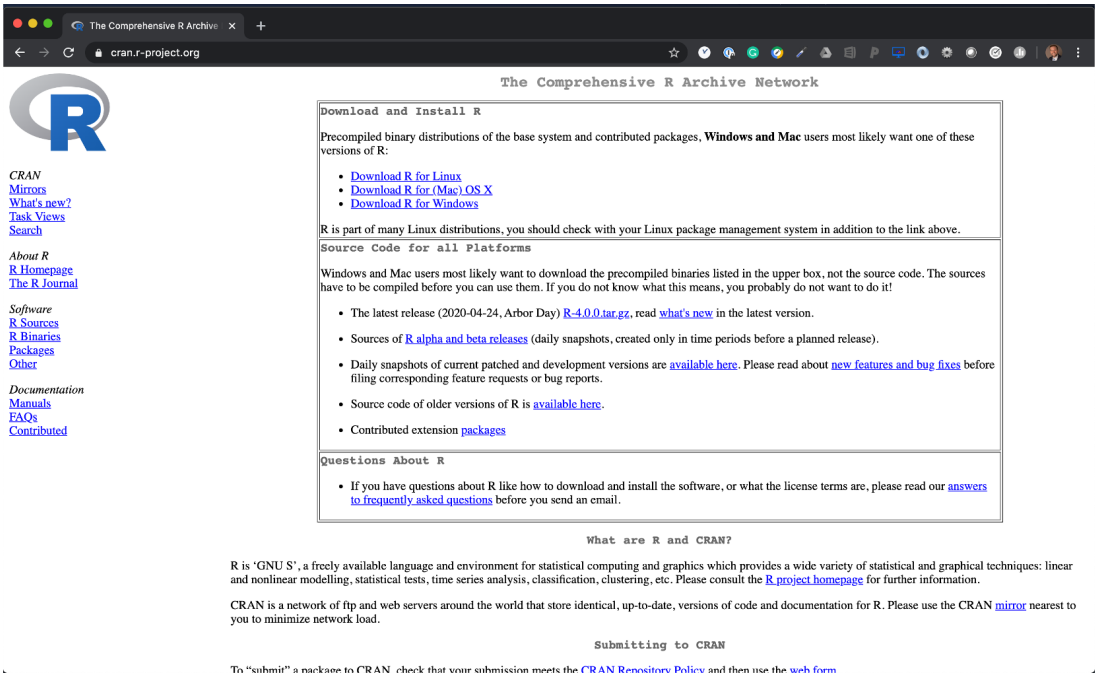
At this point, you may be wondering what R is, what RStudio is, and how they are related. We will answer those questions in the near future. However, in the interest of keeping things brief and simple, We're not going to get into them right now. Instead, all you have to worry about is getting the R programming 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.

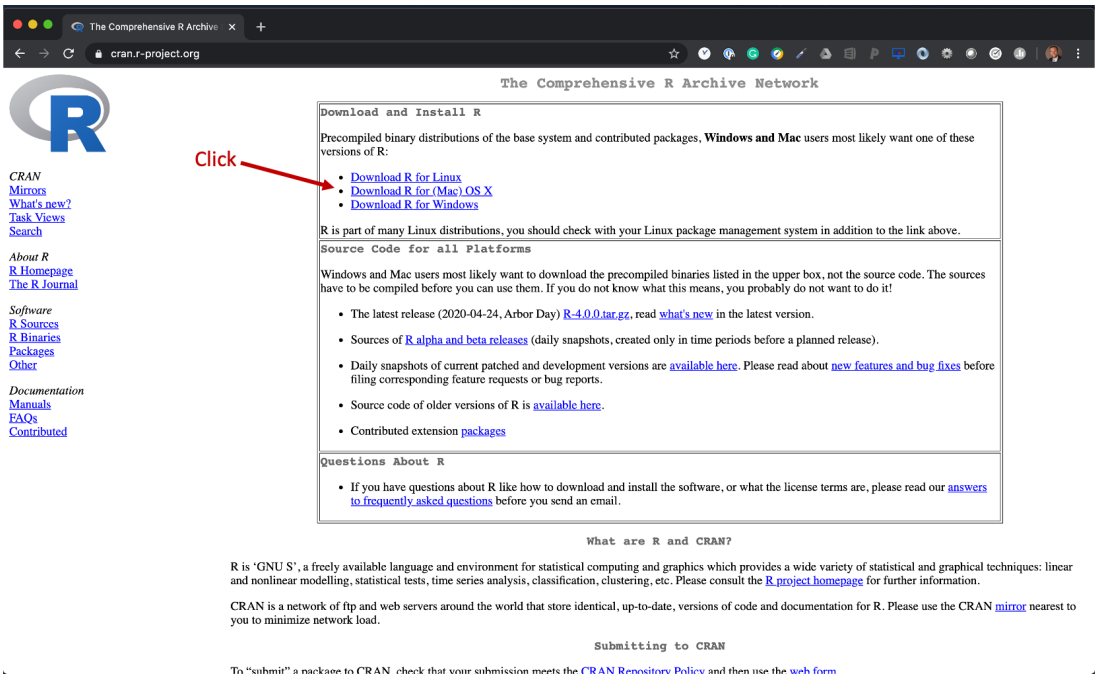
Step 1: Regardless of which operating system you are using, please make sure your computer is on, properly functioning, connected to the internet, and has enough space on your hard drive to save R and RStudio.

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/.

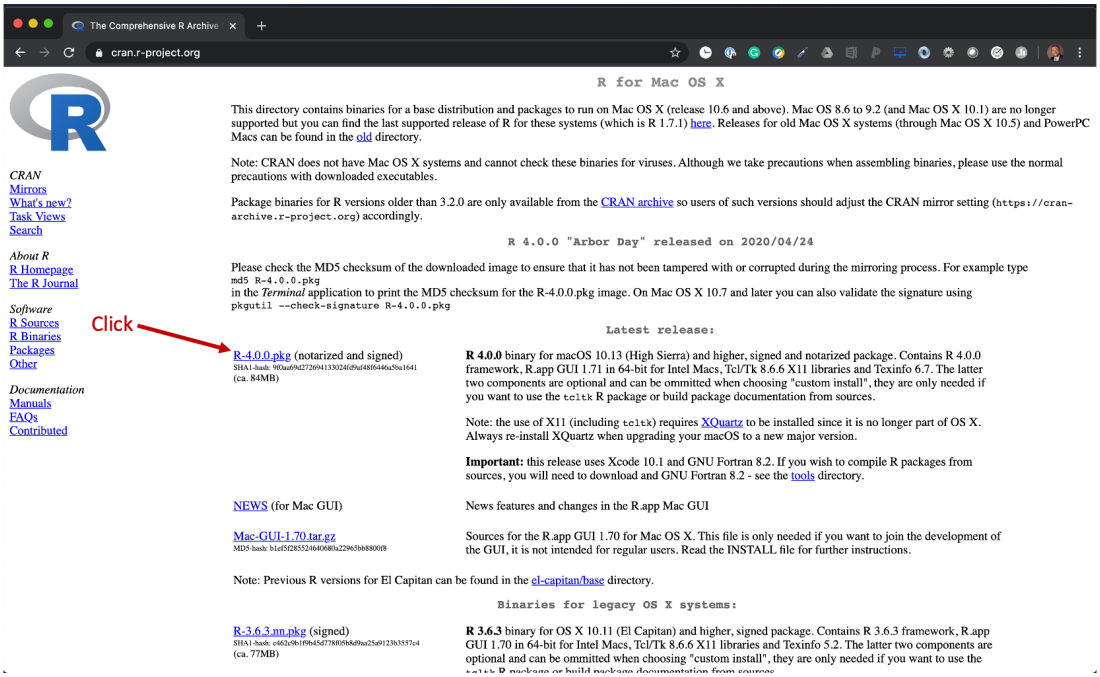


Step 3: Click on Download R for macOS.

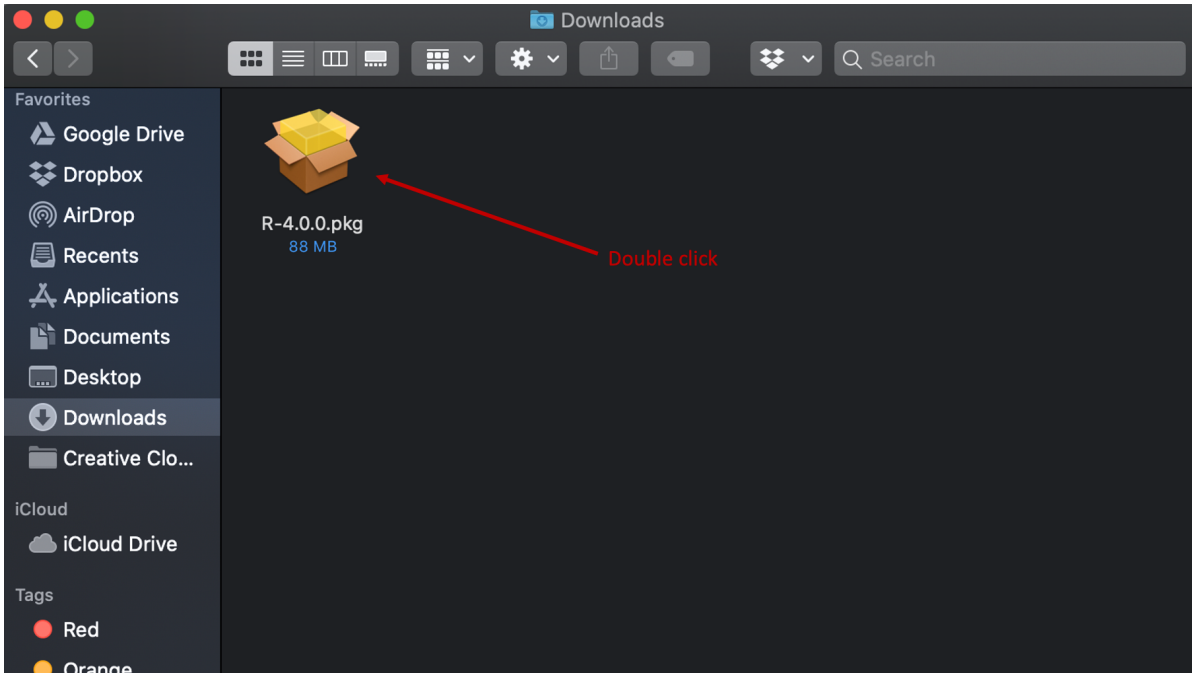


Step 4: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version

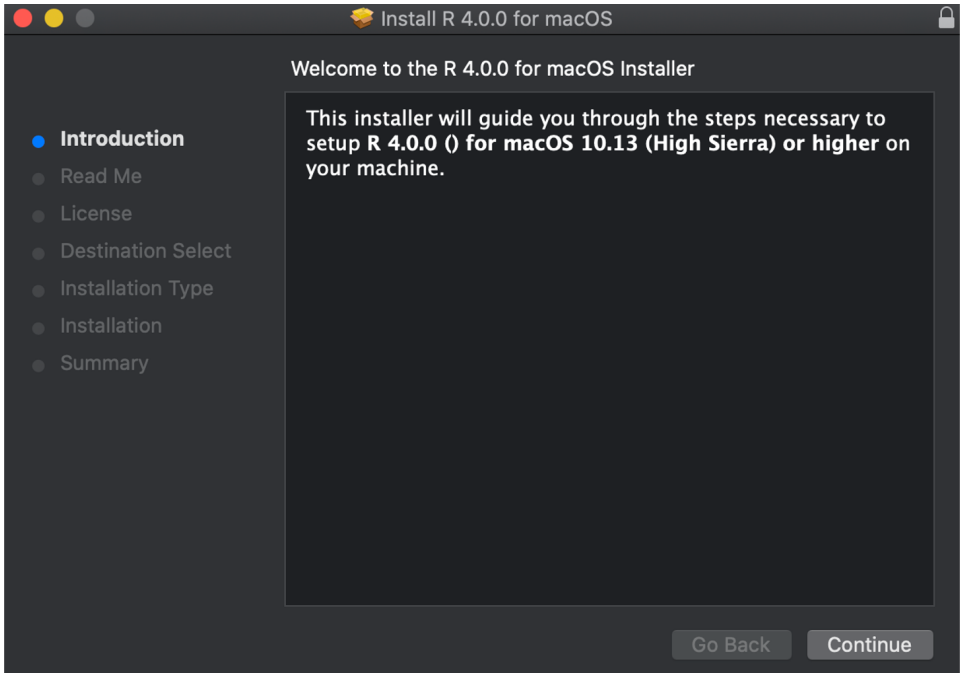
should be roughly in the same place – the middle of the screen under "Latest release:". After clicking the link, R should start to download to your computer automatically.



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.

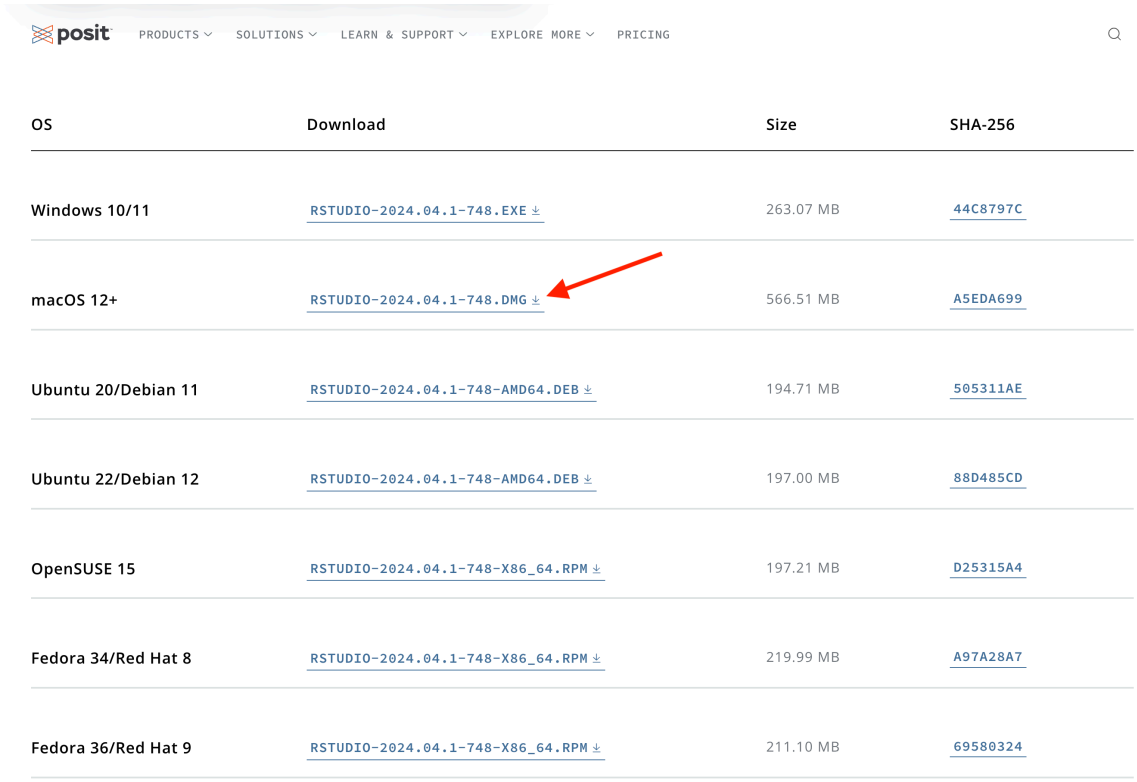


Step 6: A dialogue box will open and ask you to make some decisions about how and where you want to install R on your computer. We typically just click "continue" at every step without changing any of the default options.

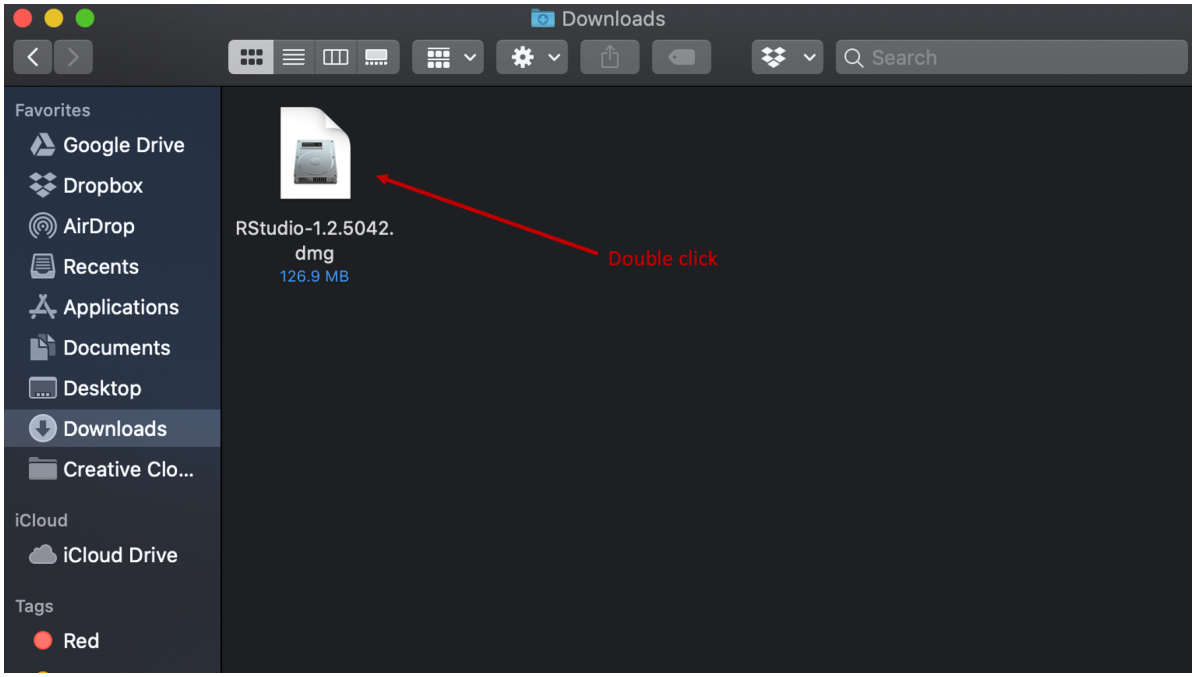


If R installed properly, you should now see it in your applications folder.

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.



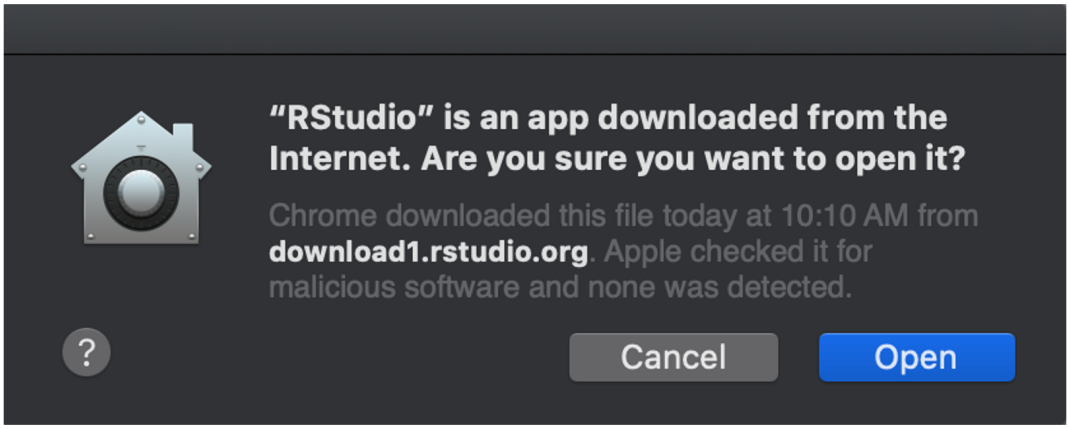
Step 8: Again, locate the DMG file you just downloaded and double click it. Unless you've changed your download settings, this file should be in the same location as the R package file you already downloaded.



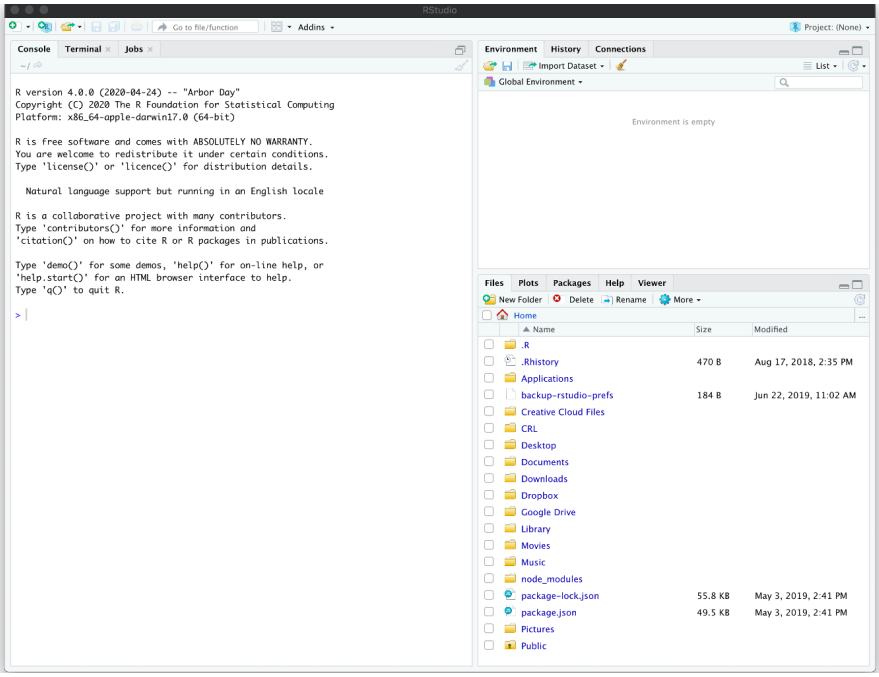
Step 9: A new finder window should automatically pop up that looks like the one you see below. Click on the RStudio icon and drag it into the Applications folder.

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

If this warning pops up, just click Open.

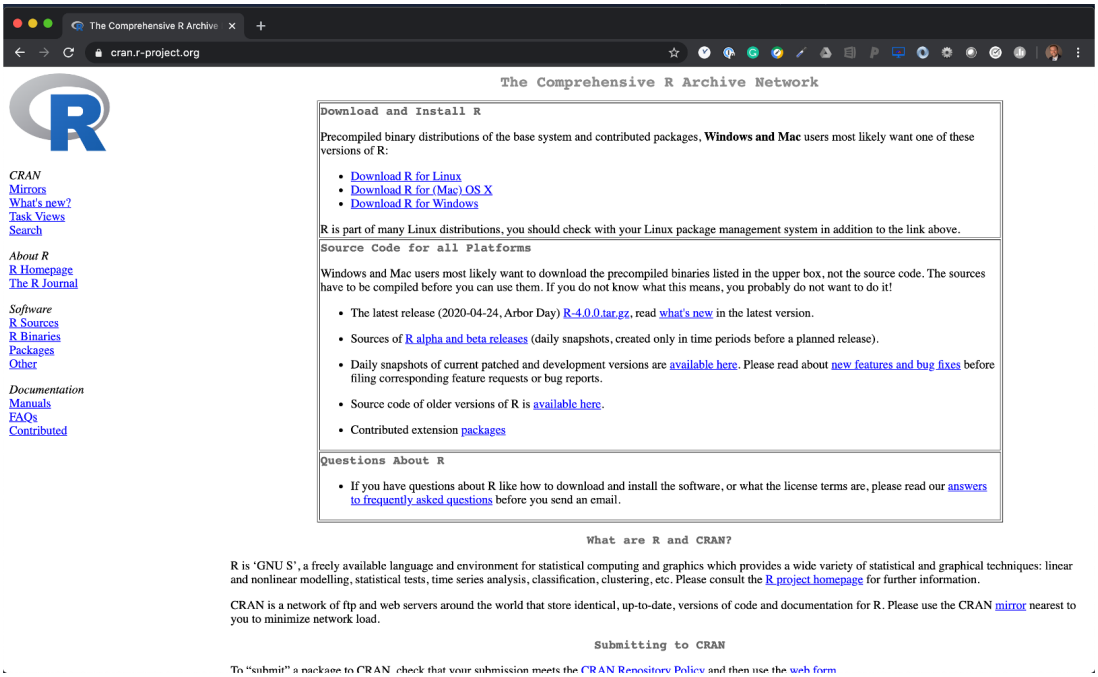


The RStudio IDE should open and look something like the window you see here. If so, you are good to go!

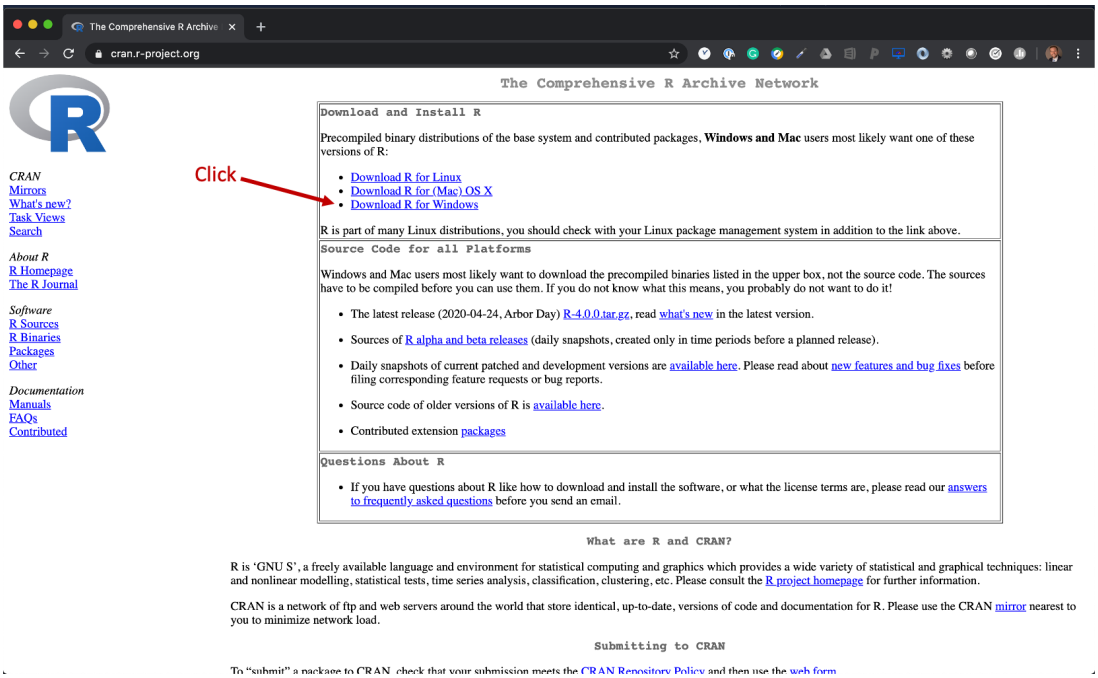


1.2 Download and install on a PC

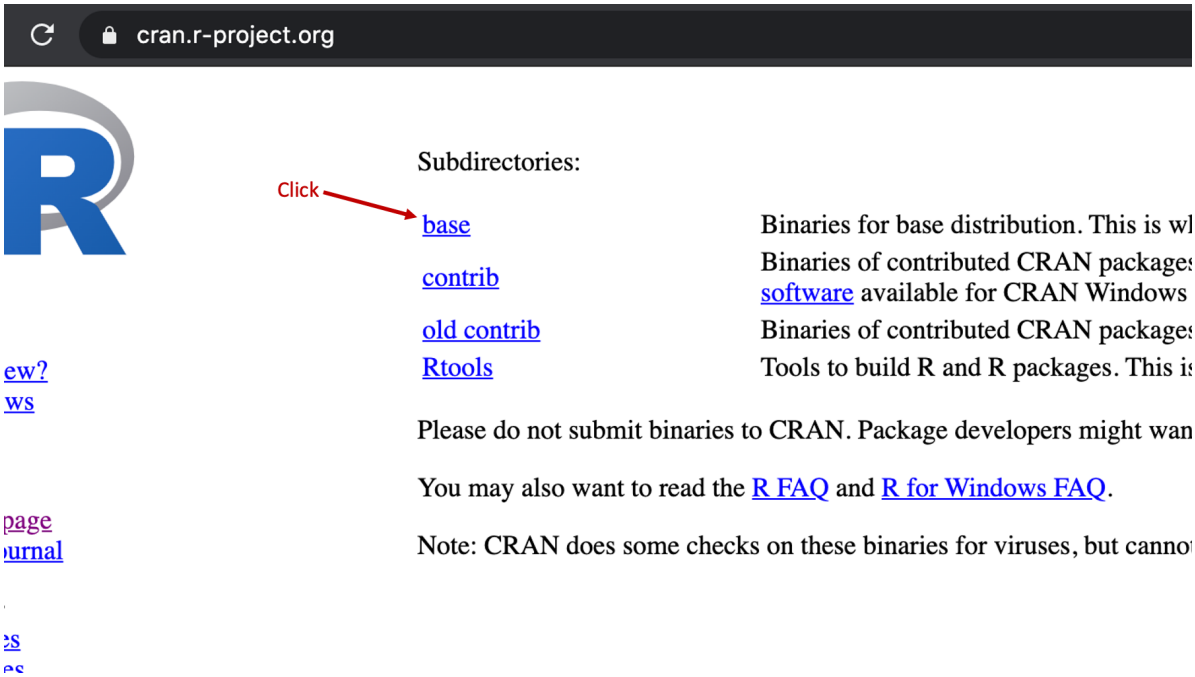
Step 2: Navigate to the Comprehensive R Archive Network (CRAN), which is located at https://cran.r-project.org/.



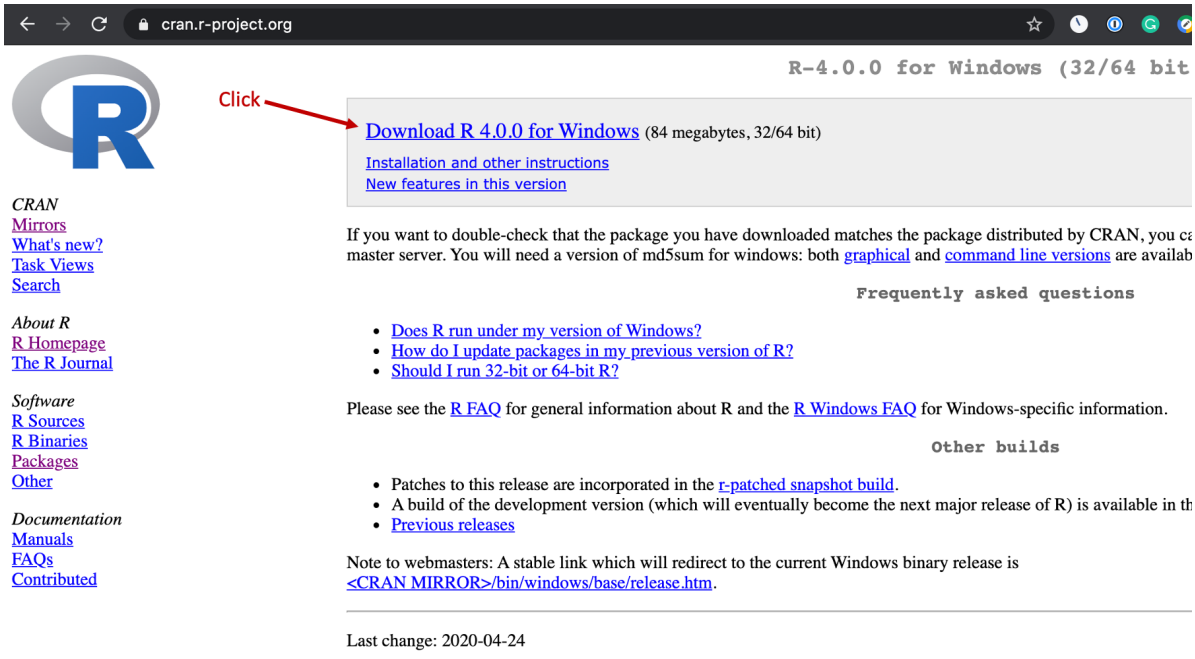
Step 3: Click on Download R for Windows.



Step 4: Click on the base link.

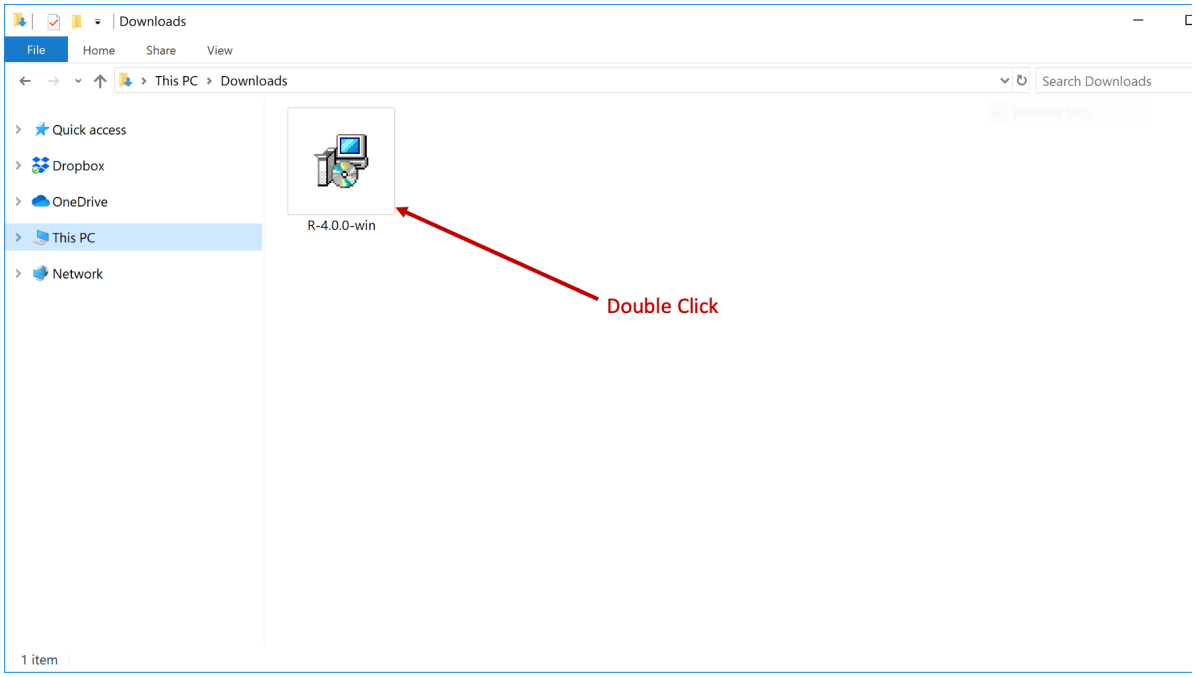


Step 5: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version should be roughly the same. After clicking, R should start to download to your computer.

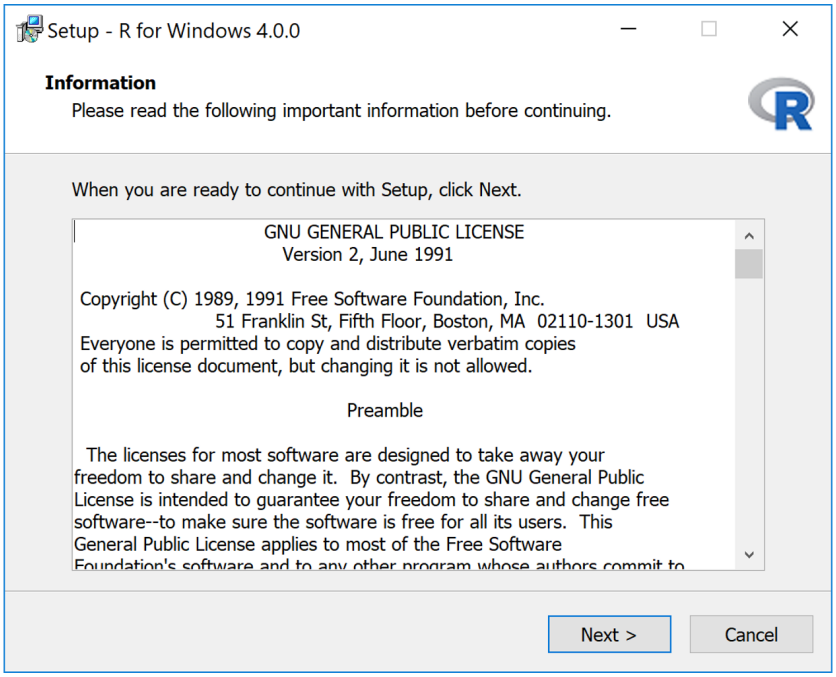


Step 6: Locate the installation file you just downloaded and double click it. Unless you've

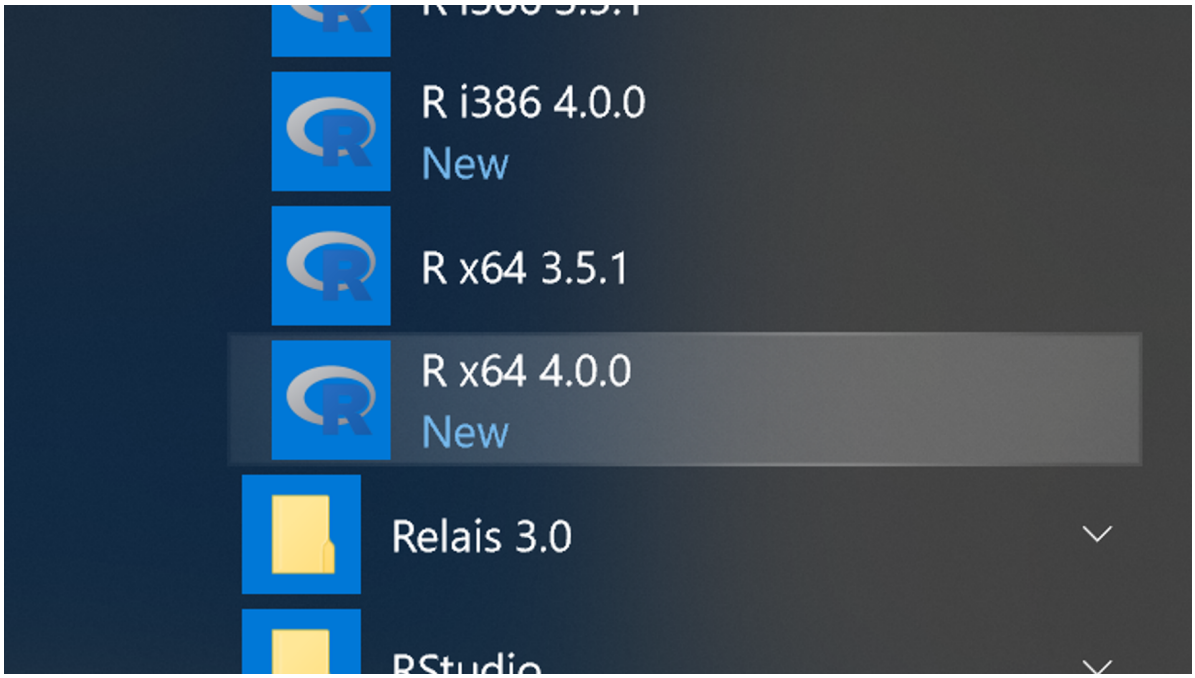
changed your download settings, this file will probably be in your downloads folder. That is the default location for most web browsers.



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.

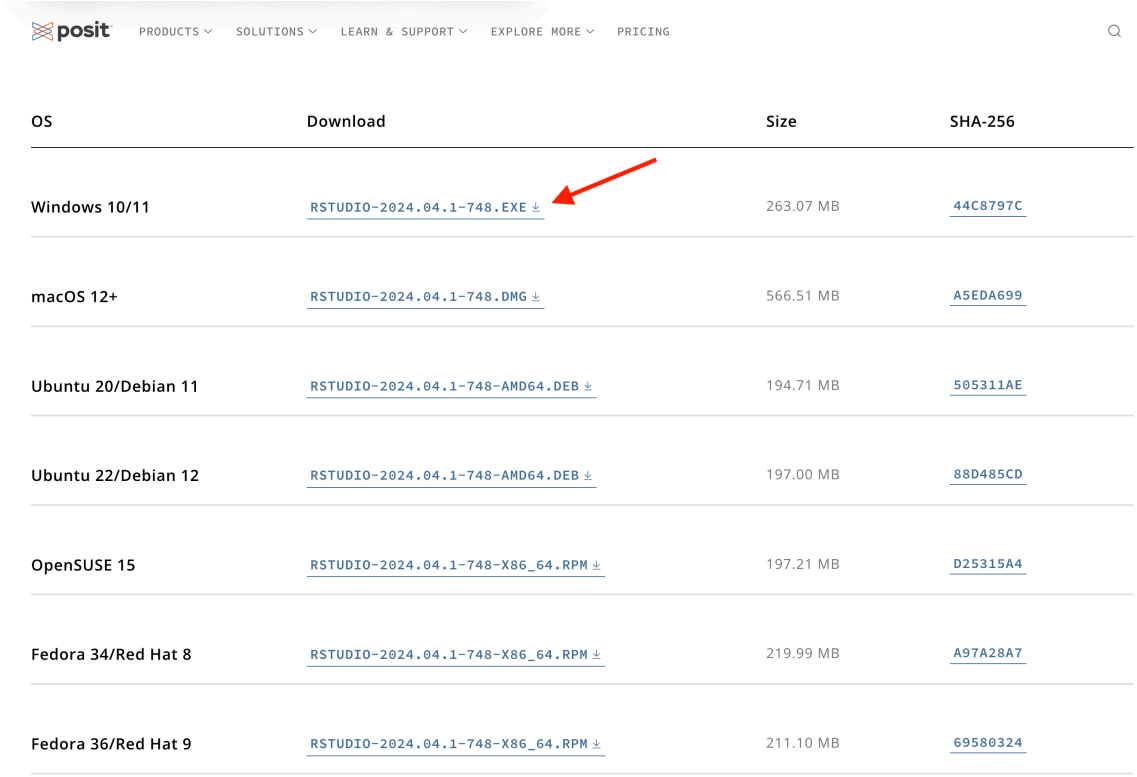


If R installed properly, you should now see it in the Windows start menu.

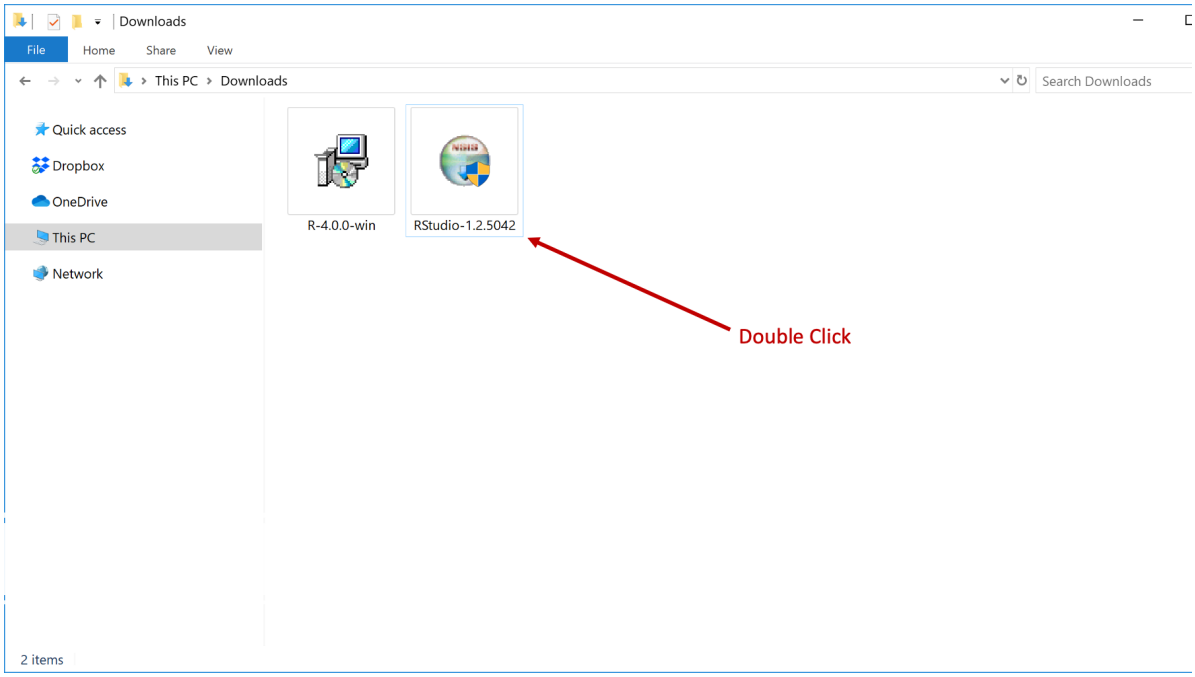


Step 8: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at https://posit.co/download/rstudio-desktop/. On that page, click the button to download the latest version of RStudio for your computer. Note that

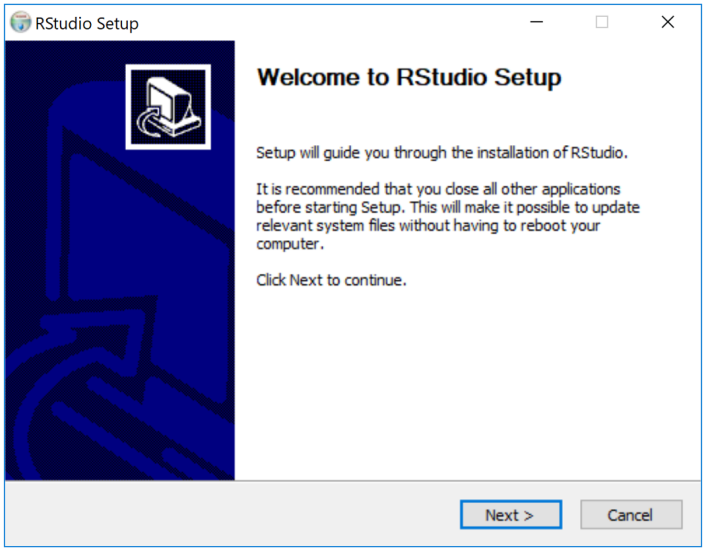
the website may look different that what you see in the screenshot below because websites change over time.



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.

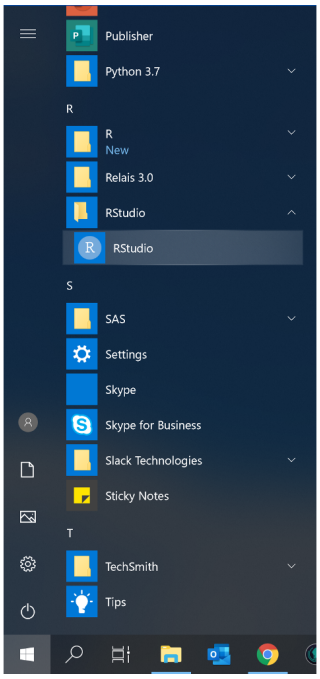


Step 10: Another dialogue box will open and ask you to make some decisions about how and where you want to install RStudio on your computer. We typically just click "Next" at every step without changing any of the default options.

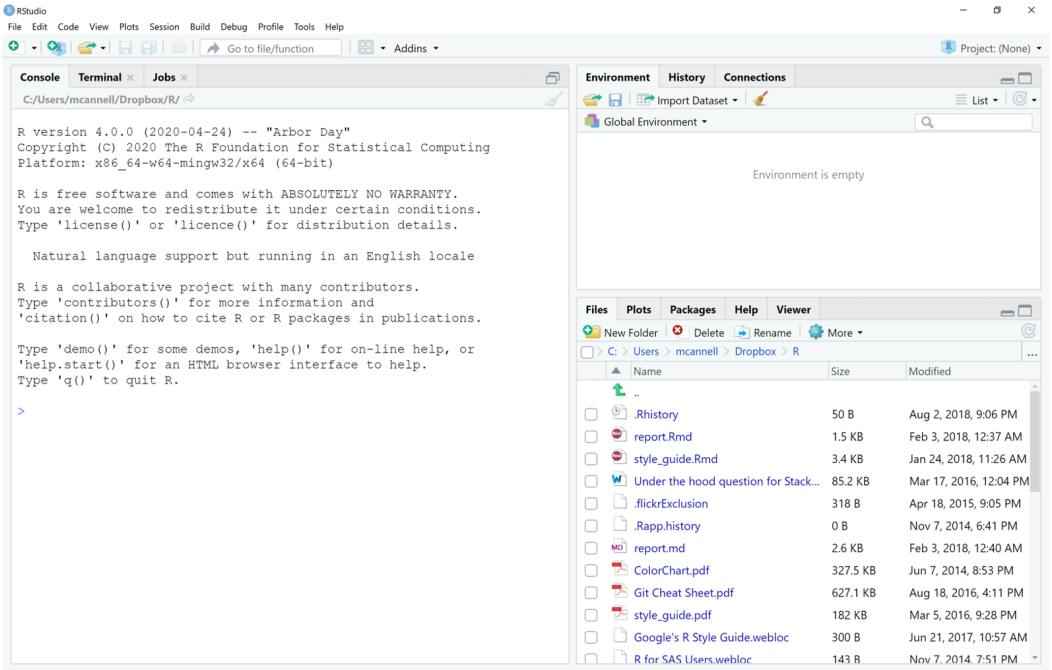


When RStudio is finished installing, you should see RStudio in the Windows start menu. Click

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

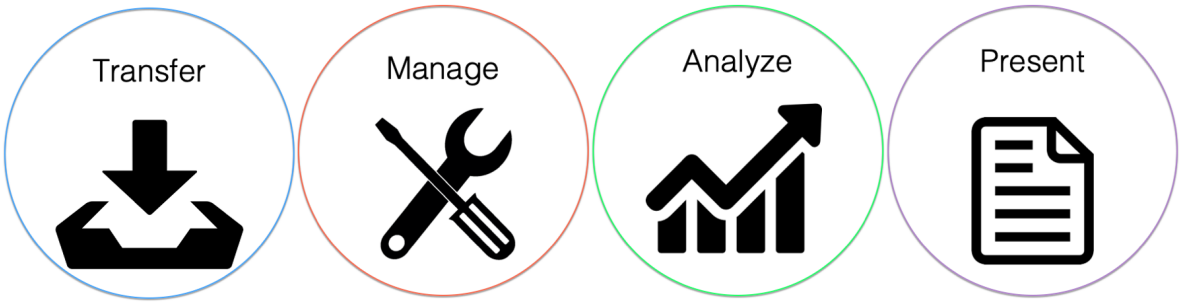
Finally, the contents of each cell are called **values**.

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

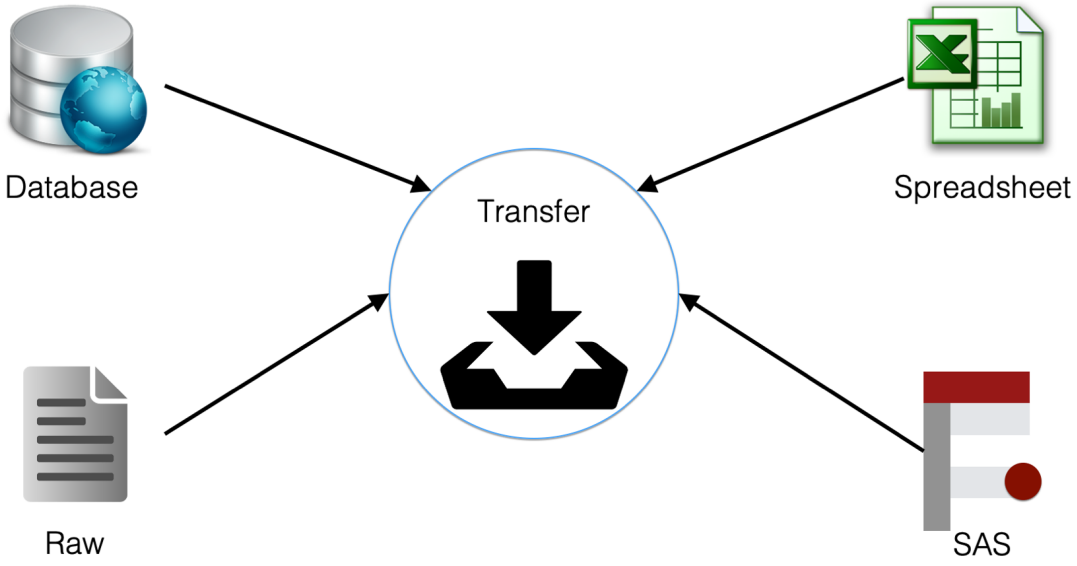
You should now be up to speed on some basic terminology used by R, as well as other analytic, database, and spreadsheet programs. These terms will be used repeatedly throughout the course.

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:

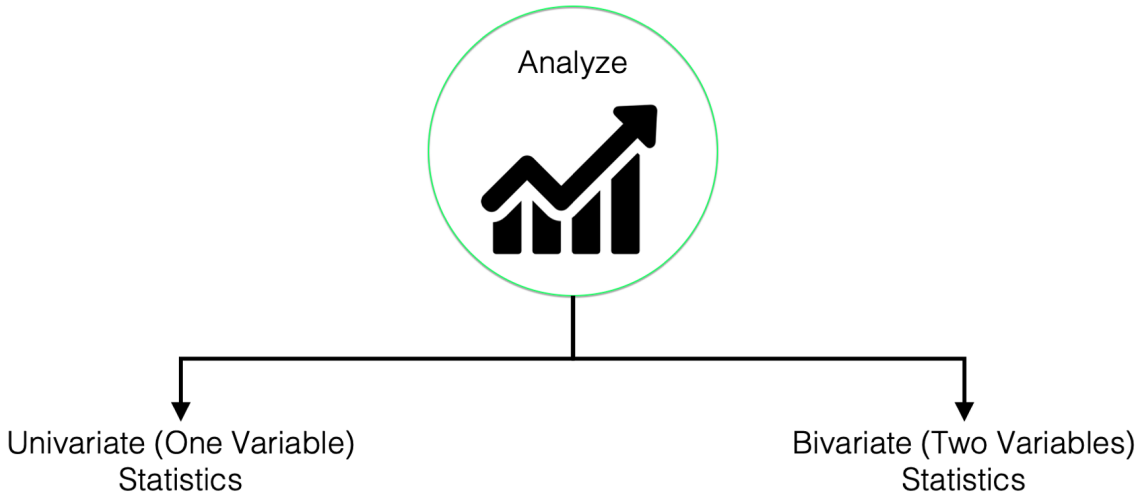
- Validating and cleaning data. In other words, dealing with potential errors in the data.
- Subsetting data. For example, using only some of the columns or some of the rows.
- Creating new variables. For example, creating a BMI variable in a data frame that was sent to you with height and weight columns.
- 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.

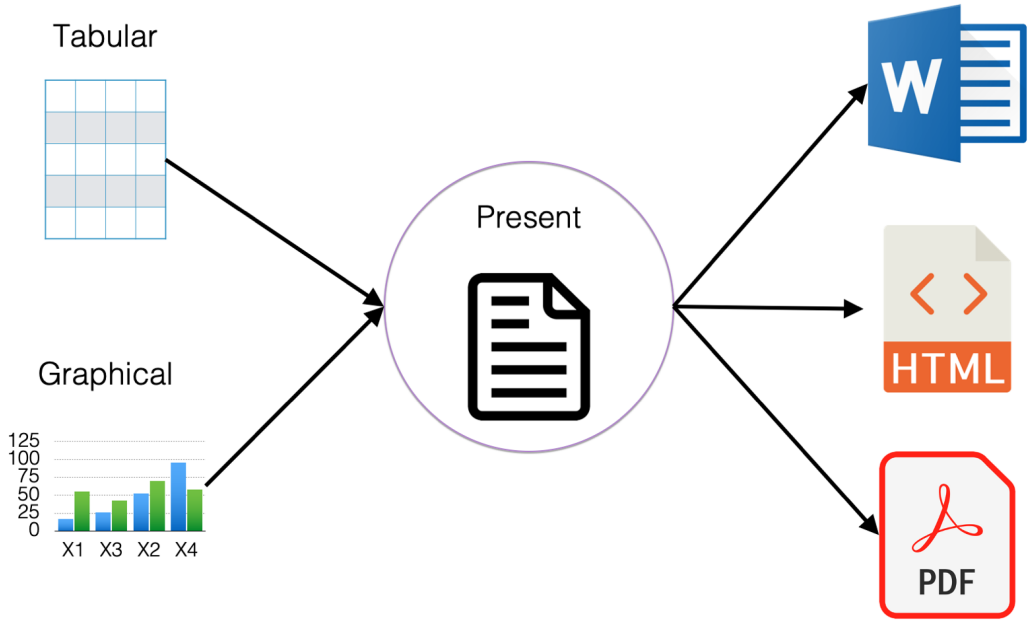
2.2.3 Analyzing data



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

You now have R and RStudio on your computer and you have some idea of what R and RStudio are. At this point, it is really 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 you first load RStudio you should see a screen that looks very similar to what you see in the picture below. @ref(fig:rstudio) In the current view, you see three **panes** and each pane has multiple tabs. Don't beat yourself up if this isn't immediately obvious. I'll make it clearer soon.

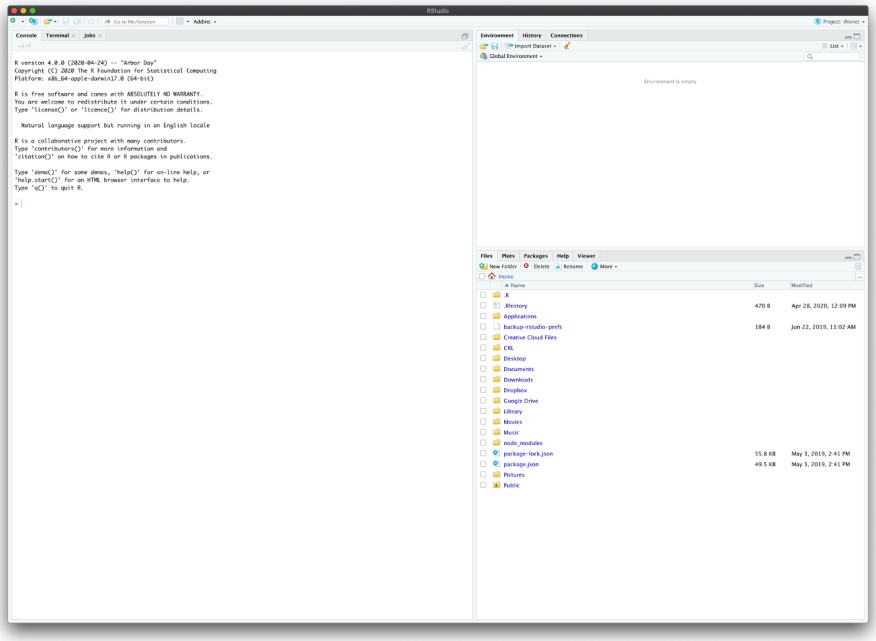


Figure 3.1: The default RStudio user interface.

3.1 The console

The first pane we are going to talk about is the Console/Terminal/Jobs pane. @ref(fig:console)

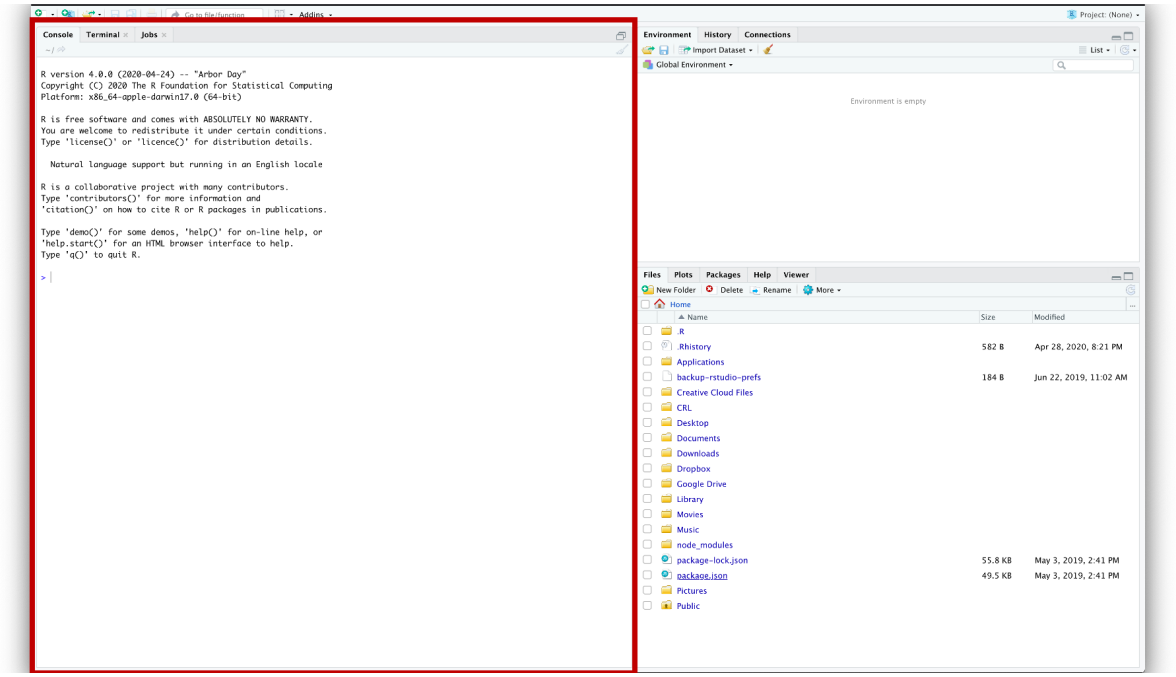


Figure 3.2: The R console.

It's called the Console/Terminal/Jobs pane because it has three tabs you can click on: Console, Terminal, and Jobs. However, we will mostly refer to it as the Console pane and we will mostly ignore the Terminal and Jobs tabs. We aren't ignoring them because they aren't useful; rather, we are ignoring them because using them isn't essential for anything we discuss anytime soon, and we want to keep things as simple as possible.

The **console** is the most basic way to interact with R. You can type a command to R into the console prompt (the prompt looks like ">") and R will respond to what you type. For example, below I've typed "1 plus 1," hit enter, and the R console returned the sum of the numbers 1 and 1. @ref(fig:one-plus-one)

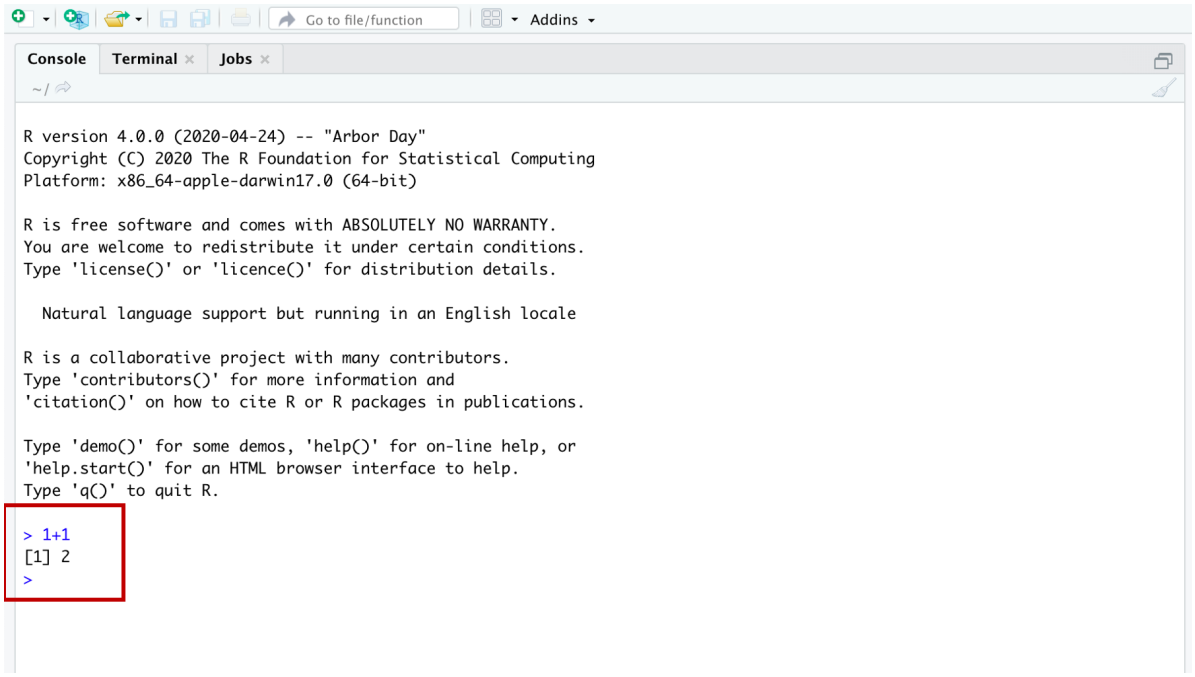


Figure 3.3: Doing some addition in the R console.

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

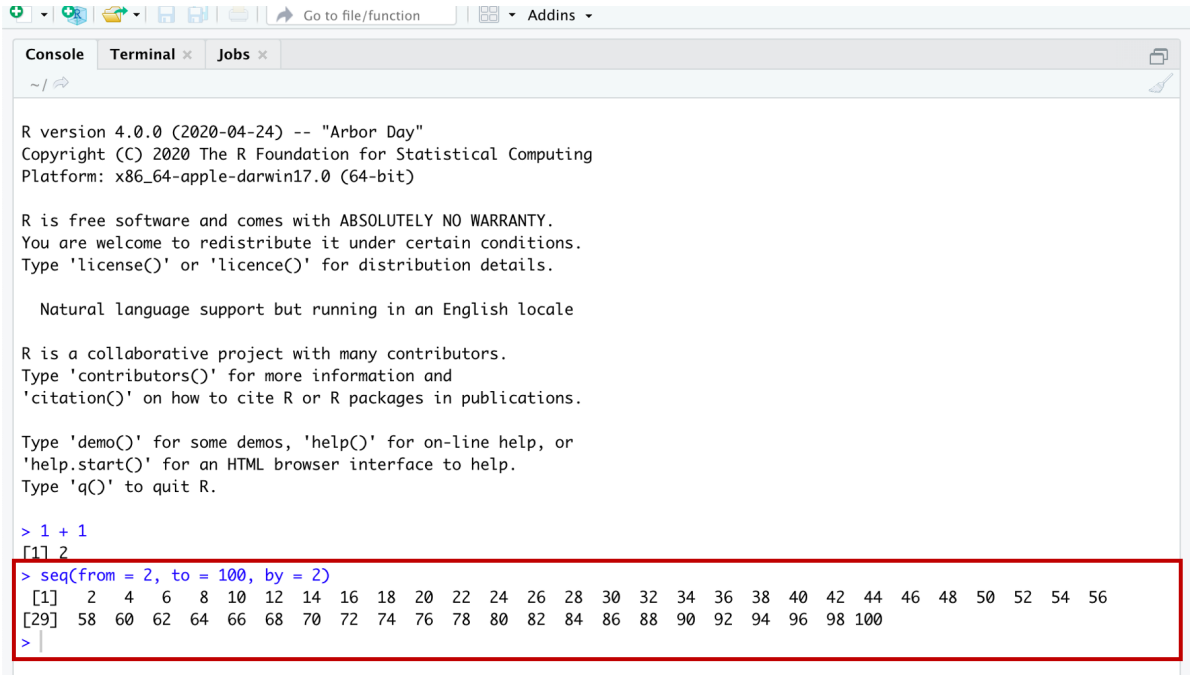


Figure 3.4: Demonstrating a function that returns multiple results.

In the screenshot above we see a couple new things demonstrated. @ref(fig:seq-function)

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 you 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 here that you may have noticed is our use of a **function**. Functions are a **BIG DEAL** in R. So much so that R is called a *functional language*. You don't really need to know all the details of what that means; however, you should know that, in general, everything you *do* in R you will *do* with a function. By contrast, everything you *create* in R will be an *object*. If you wanted to make an analogy between the R language and the English language, functions are verbs – they *do* things – and objects are nouns – they *are* things. This may be confusing right now. Don't worry. It will become clearer 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?

It's the `seq()` function – short for sequence. Inside the function, you may notice that there are three pairs of words, equal symbols, and numbers that are separated by commas. They are, `from = 2`, `to = 100`, and `by = 2`. In this case, `from`, `to`, and `by` are all **arguments** to the `seq()` function.We don't know why they are called arguments, but as far as we are concerned, they just are. We will learn more about functions and arguments later, but 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 gives us a sequence of numbers, but we have to give it information about where that sequence should start, where it should end, and how many steps should be in the middle. Here the sequence begins with the value we gave to the `from` argument (i.e., 2), ends with the value we gave to the `to` argument (i.e., 100), and increases at each step by the number we gave to the `by` argument (i.e., 2). So, 2, 4, 6, 8 … 100.

While it's convenient, let's also learn some programming terminology:

- **Arguments:** Arguments always go *inside* the parentheses of a function and give the function the information it needs to give us the result we want.
- **Pass:** In programming lingo, you *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.
- **Returns:** 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.

> **i** 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. @ref(fig:environment-pane) 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.

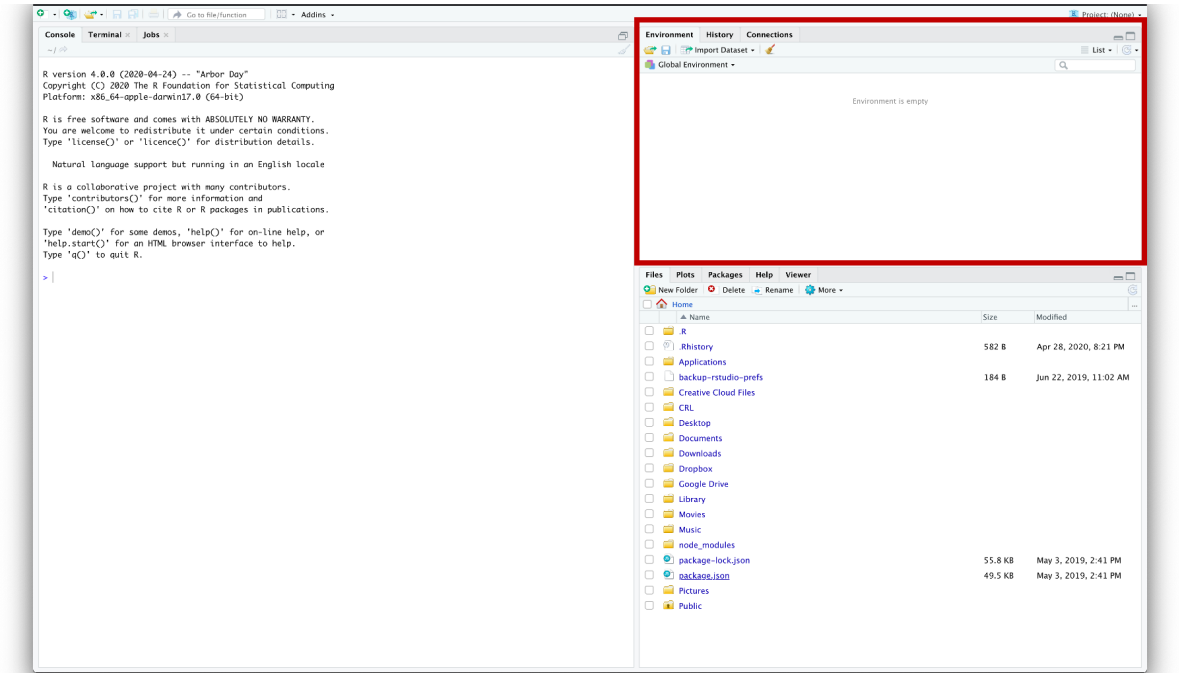


Figure 3.5: The environment pane.

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

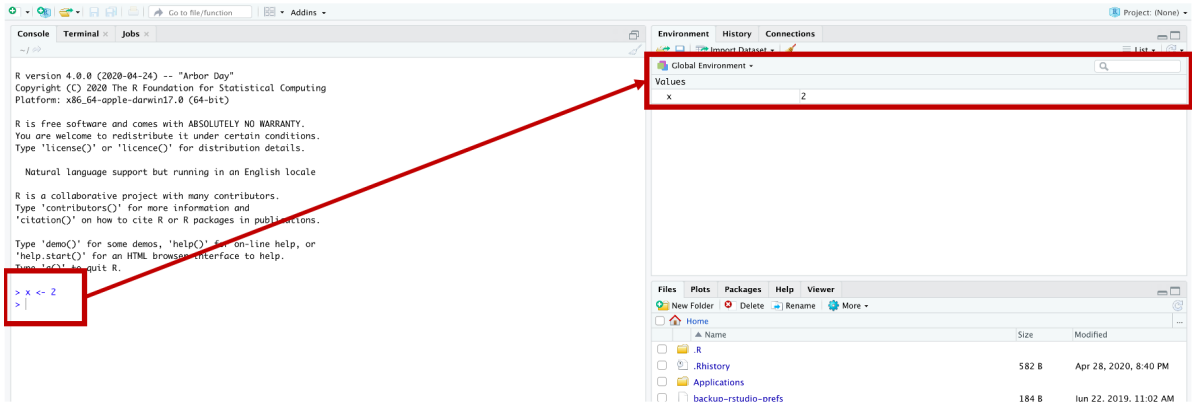


Figure 3.6: The vector x in the global environment.

Here we see that we created a new object called `x`, which now appears in our **Global Environment**. @ref(fig:environment-pane2) 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.

> **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). @ref(fig:files-pane)

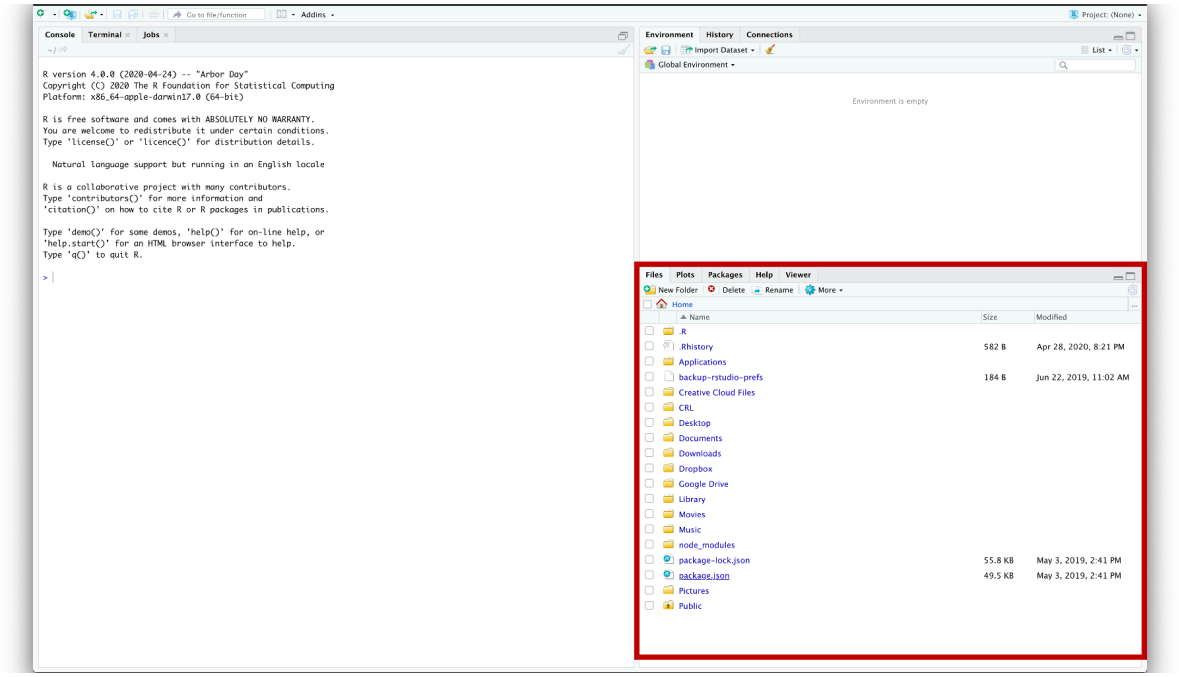


Figure 3.7: The Files/Plots/Packages/Help/Viewer pane.

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.

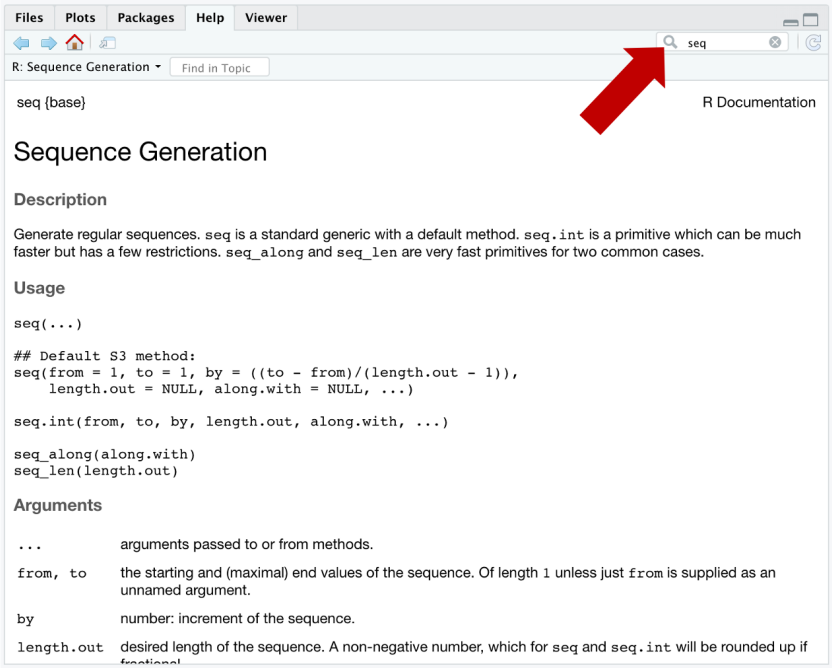


Figure 3.8: The help tab.

For example, in the screenshot above @ref(fig:help) 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*.

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. @ref(fig:source1)

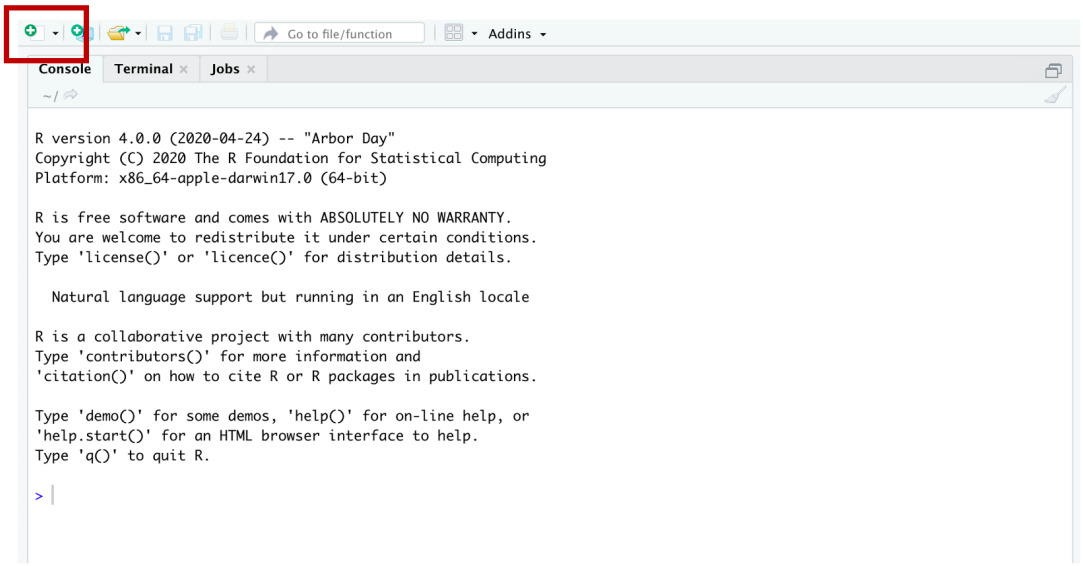


Figure 3.9: Click the new source file icon.

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

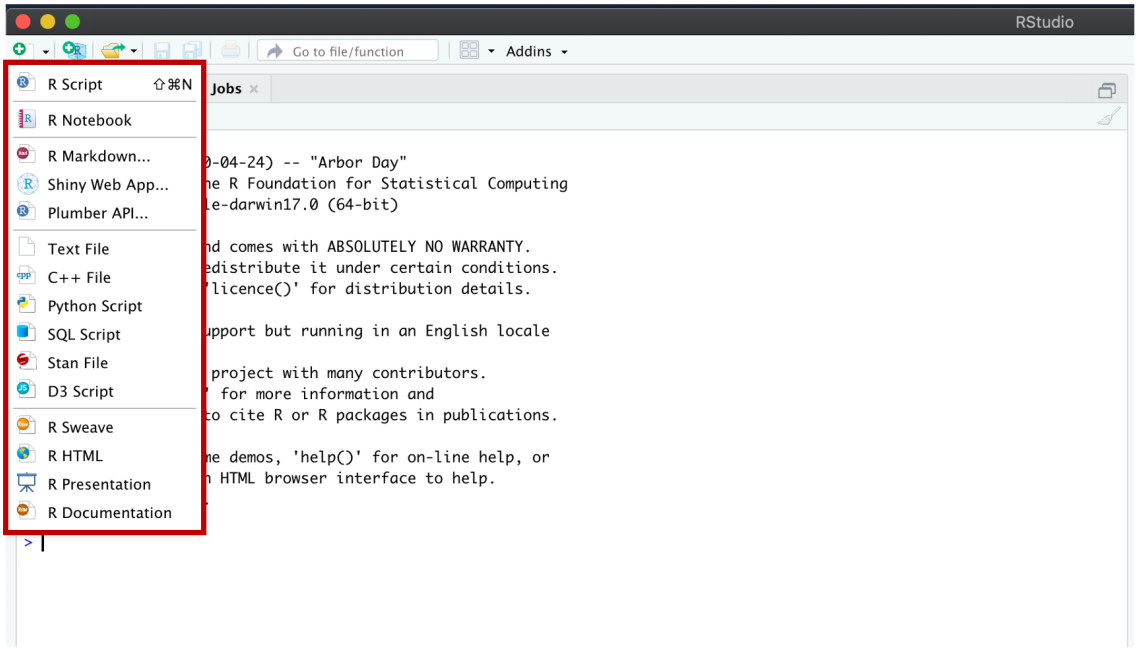


Figure 3.10: New source file options.

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.

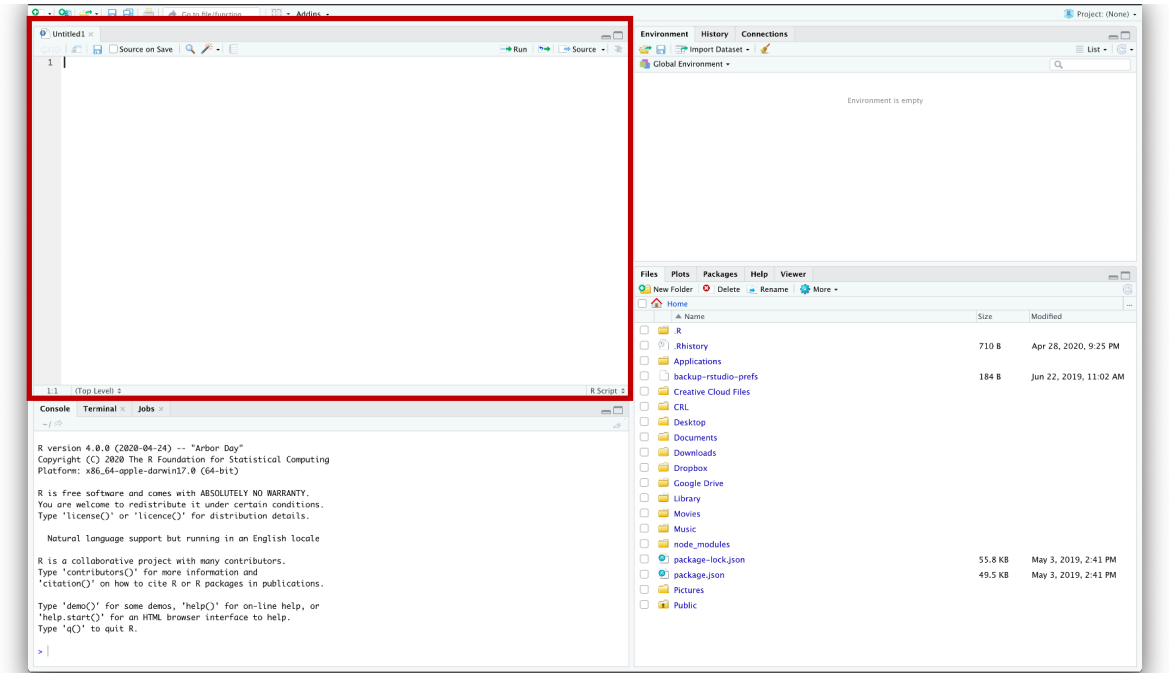


Figure 3.11: A blank R script in the source pane.

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.

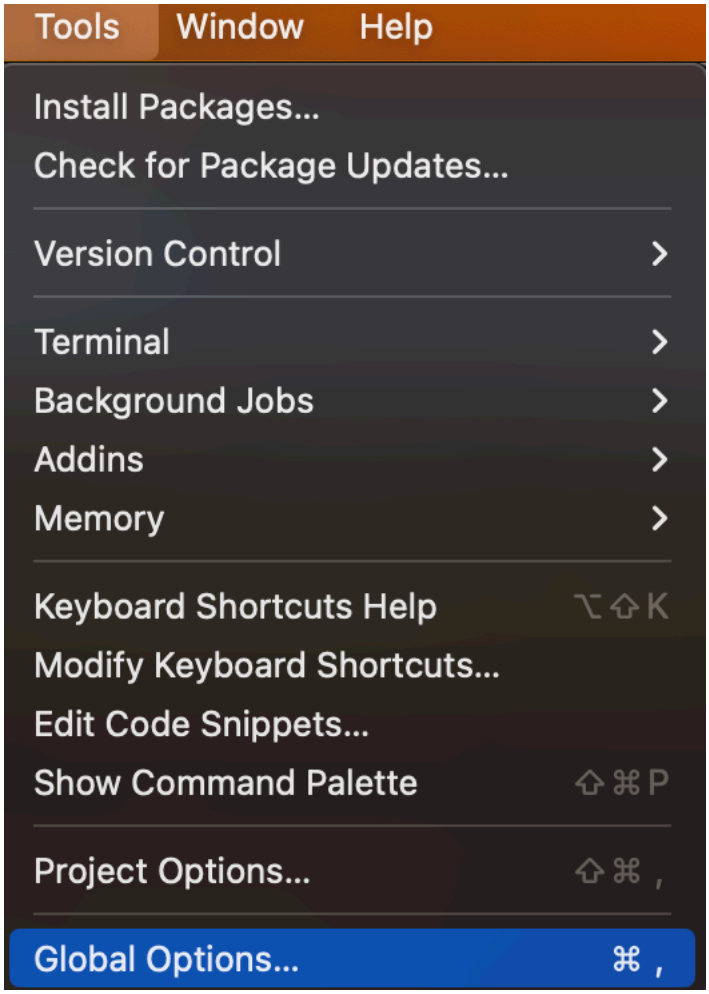


Figure 3.12: Select the preferences menu on Mac.

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.

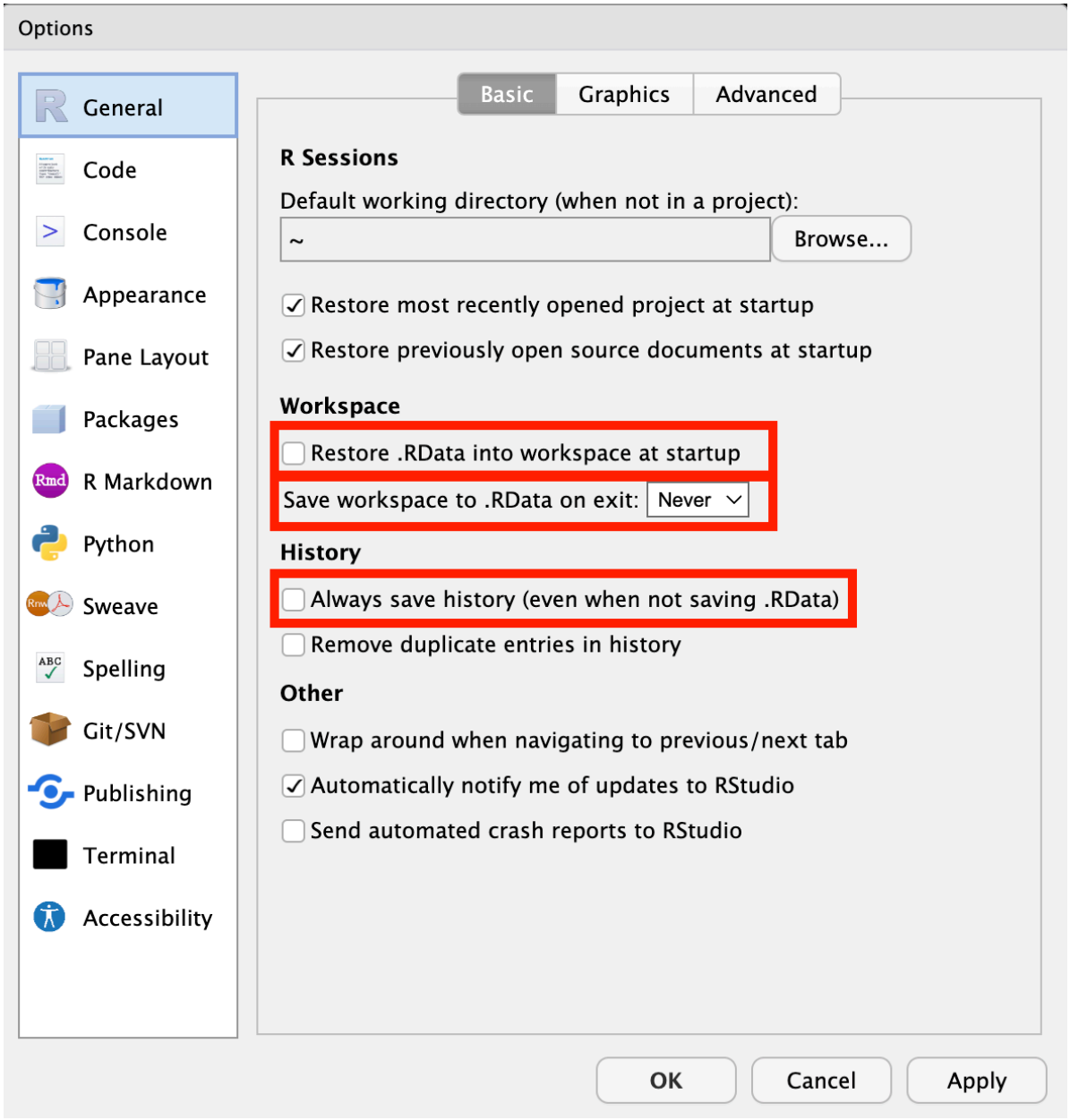


Figure 3.13: General options tab.

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.

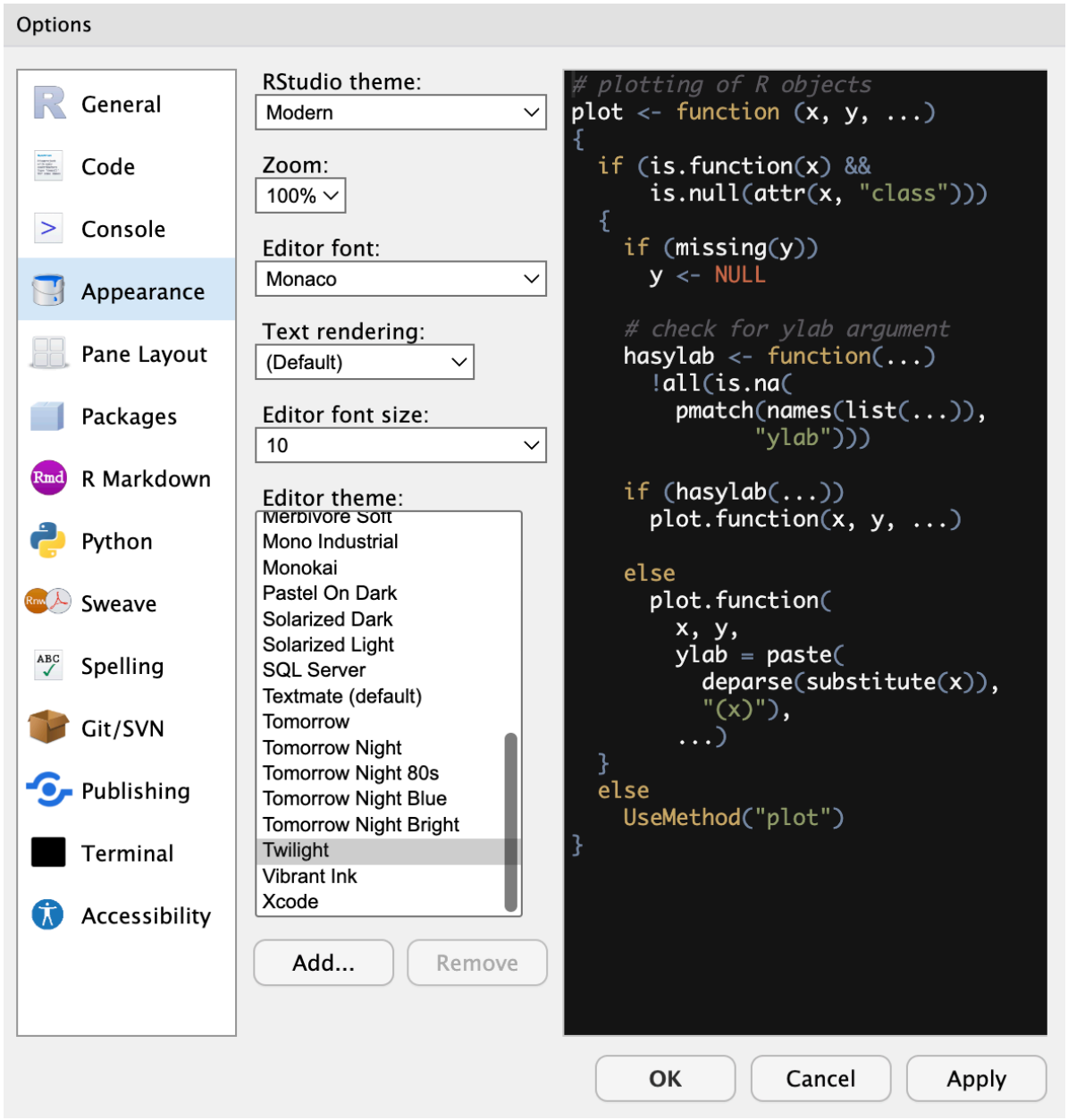


Figure 3.14: Appearance tab.

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?

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.

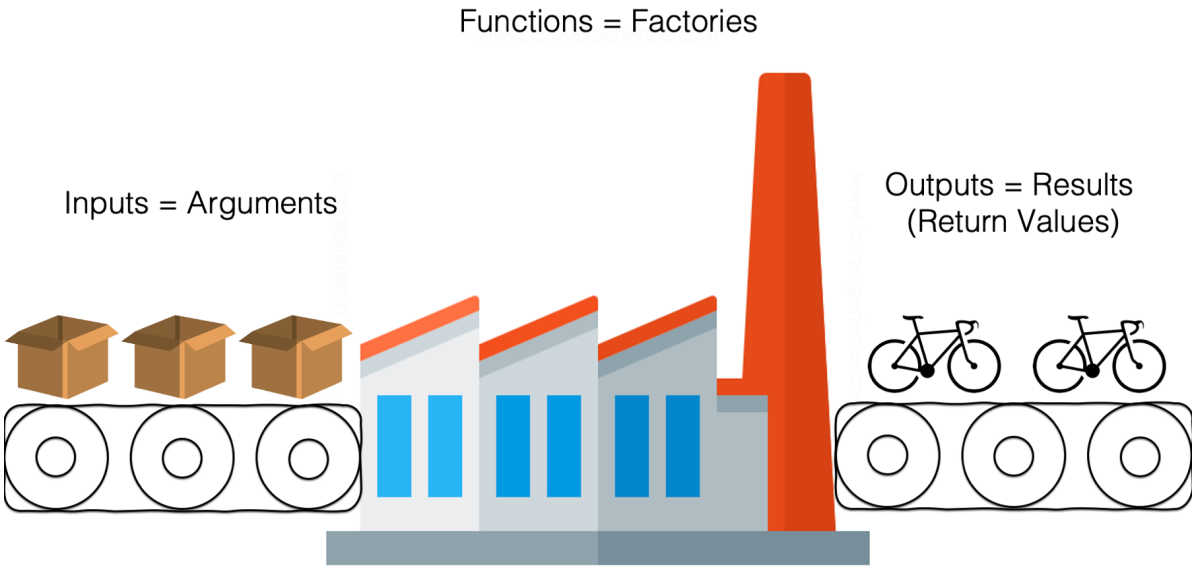


Figure 4.1: A factory making bicycles.

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.

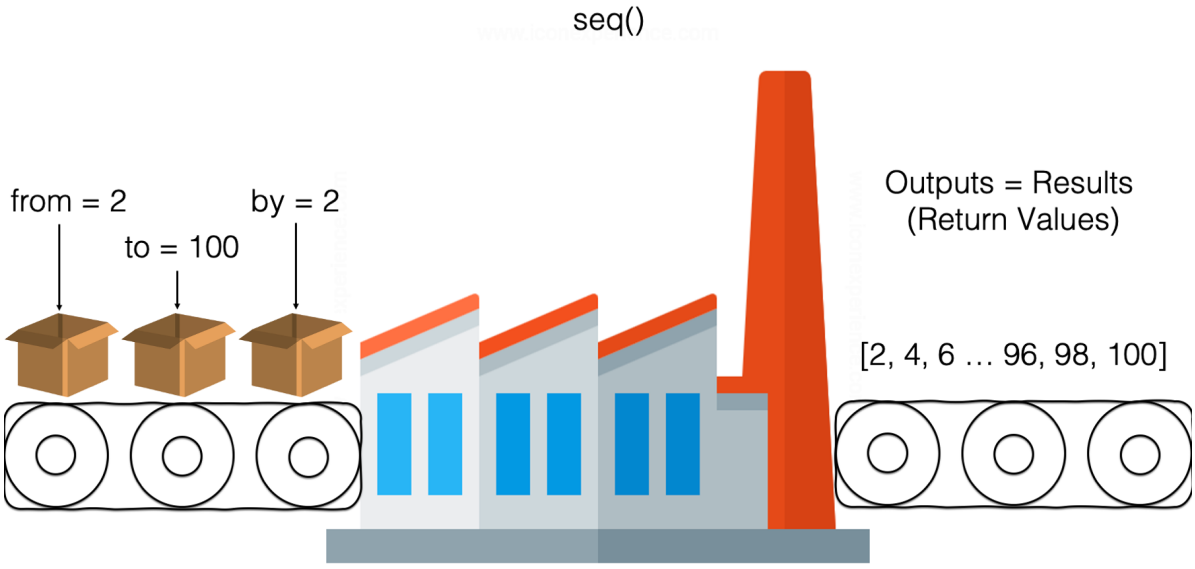


Figure 4.2: A function factory making numbers.

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.

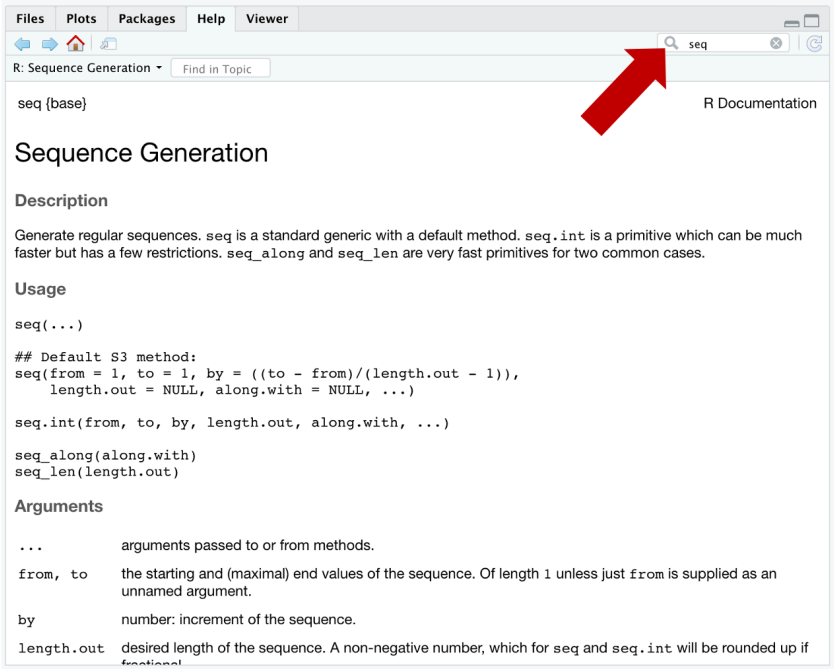


Figure 4.3: The help tab.

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.

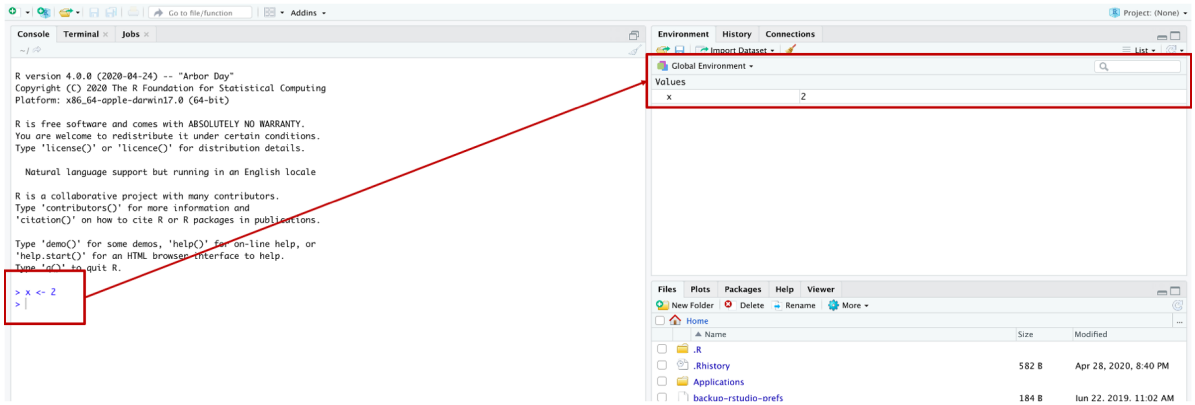
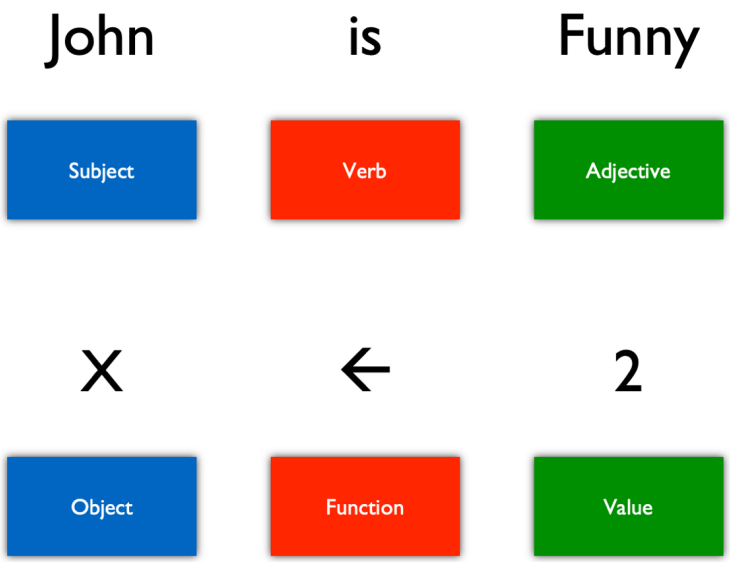


Figure 4.4: Creating the x object.

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.

> ⚠ Warning
>
> **Warning:** By definition, all columns in a data frame must have the same length (i.e., number of rows). That means that each vector you create when building your data frame must have the same number of values in it. For example, the class data frame above has four names and four heights. If we had only entered three heights, we would have gotten the following error: `Error in data.frame(names = c("John", "Sally", "Brad", "Anne"), heights = c(68,  : arguments imply differing number of rows: 4, 3`

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.

Part II

Coding Tools and Best Practices

6 Quarto Files

Part III

Collaboration

7 Using git and GitHub

Part IV

References

References

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

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

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

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

5. RStudio. RStudio. Published online 2020.

A Glossary

Console. Coming soon.

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 go *inside* the parentheses of a function and give the function the information it needs to give us the result we want.
- **Pass:** In programming lingo, you *pass* a value to a function argument. For example, in the funtion 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.
- **Returns:** Instead of saying, "the `seq()` function *gives us* a sequence of numbers…" we could say, "the `seq()` function *returns* us 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."[2]

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."[3] 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."[4] 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).[5]