

R 4 Epidemiology

2025-03-12

Table of contents

Welcome	8
Acknowledgements	8
Introduction	9
Goals	9
Text conventions used in this book	10
Other reading	10
Contributing	11
Typos	11
Issues	18
License Information	18
About the Authors	19
Brad Cannell	19
Melvin Livingston	20
I Getting Started	21
1 Installing R and RStudio	22
1.1 Download and install on a Mac	22
1.2 Download and install on a PC	30
2 What is R?	38
2.1 What is data?	38
2.2 What is R?	43
2.2.1 Transferring data	44
2.2.2 Managing data	45
2.2.3 Analyzing data	46
2.2.4 Presenting data	47
3 Navigating the RStudio Interface	49
3.1 The console pane	50
3.2 The environment pane	54
3.3 The files pane	57

3.4	The source pane	60
3.5	RStudio preferences	60
4	Speaking R's Language	66
4.1	R is a <i>language</i>	66
4.2	The R interpreter	67
4.3	Errors	67
4.4	Functions	68
4.4.1	Passing values to function arguments	70
4.5	Objects	73
4.6	Comments	75
4.7	Packages	76
4.8	Programming style	78
5	Let's Get Programming	79
5.1	Simulating data	79
5.2	Vectors	80
5.2.1	Vector types	81
5.2.2	Double vectors	82
5.2.3	Integer vectors	82
5.2.4	Logical vectors	83
5.2.5	Factor vectors	83
5.3	Data frames	87
5.4	Tibbles	89
5.4.1	The <code>as_tibble</code> function	90
5.4.2	The <code>tibble</code> function	91
5.4.3	The <code>tribble</code> function	92
5.4.4	Why use tibbles	94
5.5	Missing data	95
5.6	Our first analysis	97
5.6.1	Manual calculation of the mean	97
5.6.2	Dollar sign notation	98
5.6.3	Bracket notation	98
5.6.4	The <code>sum</code> function	99
5.6.5	Nesting functions	100
5.6.6	The <code>length</code> function	102
5.6.7	The <code>mean</code> function	103
5.7	Some common errors	104
5.8	Summary	105
6	Asking Questions	106
6.1	When should we seek help?	106
6.2	Where should we seek help?	107

6.3	How should we seek help?	108
6.3.1	Creating a post on Stack Overflow	108
6.3.2	Creating better posts and asking better questions	111
6.4	Helping others	114
6.5	Summary	114
II	Coding Tools and Best Practices	116
7	R Scripts	117
7.1	Creating R scripts	121
8	Quarto Files	124
8.1	What is Quarto?	126
8.2	Why use Quarto?	127
8.3	Create a Quarto file	127
8.4	YAML headers	130
8.5	R code chunks	132
8.6	Markdown	133
8.6.1	Markdown headings	134
8.7	Summary	136
9	R Projects	137
10	Coding Best Practices	144
10.1	General principles	144
10.2	Code comments	145
10.2.1	Defining key variables	145
10.2.2	What this code is trying to accomplish	146
10.2.3	Why we chose this particular strategy	146
10.3	Style guidelines	146
10.3.1	Comments	147
10.3.2	Object (variable) names	147
10.3.3	Use names that are informative	147
10.3.4	File Names	149
11	Using Pipes	152
11.1	What are pipes?	152
11.2	How do pipes work?	155
11.2.1	Keyboard shortcut	160
11.2.2	Pipe style	160
11.3	Final thought on pipes	162

III Data Transfer	163
12 Introduction to Data Transfer	164
13 File Paths	166
13.1 Finding file paths	170
13.2 Relative file paths	172
14 Importing Plain Text Files	181
14.1 Packages for importing data	182
14.2 Importing space delimited files	182
14.2.1 Specifying missing data values	185
14.3 Importing tab delimited files	187
14.4 Importing fixed width format files	189
14.4.1 Vector of column widths	193
14.4.2 Paired vector of start and end positions	195
14.4.3 Using named arguments	197
14.5 Importing comma separated values files	199
14.6 Additional arguments	201
15 Importing Binary Files	209
15.1 Packages for importing data	209
15.2 Importing Microsoft Excel spreadsheets	210
15.3 Importing data from other statistical analysis software	215
15.4 Importing SAS data sets	216
15.5 Importing Stata data sets	220
16 RStudio's Data Import Tool	221
17 Exporting Data	227
17.1 Plain text files	228
17.2 R binary files	229
IV Descriptive Analysis	231
18 Introduction to Descriptive Analysis	232
18.1 What is descriptive analysis and why would we do it?	232
18.2 What kind of descriptive analysis should we perform?	232
19 Numerical Descriptions of Categorical Variables	236
19.1 Factors	238
19.1.1 Coerce a numeric variable	241
19.1.2 Coerce a character variable	245

19.2 Height and Weight Data	247
19.2.1 View the data	247
19.3 Calculating frequencies	249
19.3.1 The base R table function	249
19.3.2 The gmodels CrossTable function	249
19.3.3 The tidyverse way	250
19.4 Calculating percentages	253
19.5 Missing data	254
19.6 Formatting results	256
19.7 Using freqtables	257
V Collaboration	260
20 Introduction to git and GitHub	261
20.1 Versioning	262
20.2 Preservation	265
20.3 Reproducibility	265
20.4 Collaboration	266
20.5 Summary	266
21 Using git and GitHub	268
21.1 Install git	268
21.2 Sign up for a GitHub account	269
21.3 Install GitKraken	270
21.4 Example 1: Contribute to R4Epi	274
21.5 Example 2: Create a repository for a research project	274
Step 1: Create a repository on GitHub	275
Step 2: Clone the repository to your computer	288
Step 3: Add an R project file to the repository	296
Step 4: Update and commit gitignore	298
Step 5: Keep adding and committing files	312
21.6 Committing and pushing	318
21.7 Example 3: Contribute to a research project	318
21.7.1 Forking a repository	320
21.7.2 Creating a pull request	325
21.8 Summary	344
VI References	346
22 References	347

Appendices	348
A Glossary	348

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?

¹In 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.
- [Reproducible 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 in see [Using-git-and-Github](#). 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.

If you spot a typo, you can offer a correction directly in the easiest way to offer a correction is 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 in see [Using-git-and-Github](#). Here, we're just going to walk you through how to fix a typo without much explanation of how GitHub works.

Uh, oh! The word “typo” should only have one “o”!

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

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

 [Edit this page](#)

[Report an issue](#)

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.



You need to fork this repository to propose changes.

Sorry, you're not able to edit this repository directly. You need to fork it and propose your changes from there instead.

[Fork this repository](#)
[Learn more about forks](#)

Fork the Repository

“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.

You're making changes in a project you don't have write access to. Submitting a change will write it to a new branch in your fork arthur-epi/r4epi_quarto, so you can send a pull request.

[r4epi_quarto / chapters / contributing / contributing.qmd](#) in [main](#) [Cancel changes](#) [Commit changes...](#)

[Edit](#) [Preview](#) [Spaces](#) [2](#) [Soft wrap](#)

```
1 # Contributing {.unnumbered}
2
3 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
4 welcome and appreciate all constructive contributions to R4Epi!
5
6 ## Typos {.unnumbered}
7
8 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.
9
10 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](https://github.com/join).
11 Later in the book, we will cover using GitHub in greater depth (See @sec-using-git-and-github). Here, we're just going to walk you through how to fix a typo without much
12 explanation of how GitHub works.
13
14 Let's say you spot a typo while reading along.
```

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**, 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”.

You're making changes in a project you don't have write access to. Submitting a change will write it to a new branch in your fork `arthur-epi/r4epi_quarto`, so you can send a pull request.

`r4epi_quarto / chapters / contributing / contributing.qmd` in `main`

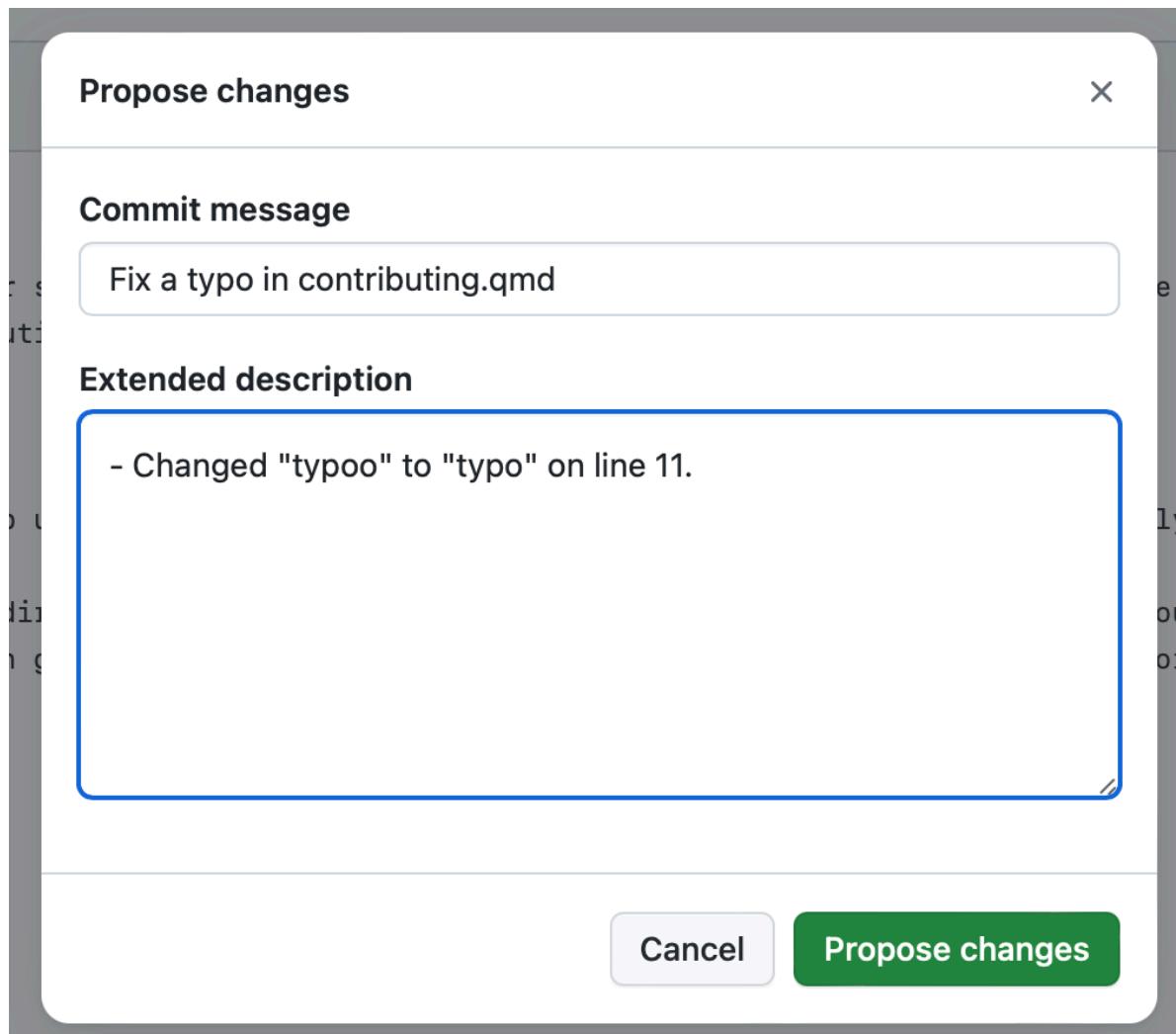
Commit changes...

Edit Preview Spaces 2 Soft wrap

```
1 # Contributing {.unnumbered}
2
3 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
4 welcome and appreciate all constructive contributions to R4Epi!
5
6 ## Typos {.unnumbered}
7 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.
8
9 If you spot a typo, you can off... You will first need to create a free GitHub account: [sign-up at github.com](https://github.com/join). Later in the book, we will cover using GitHub in greater depth (See @sec-using-git-and-github). Here, we're just going to walk you through how to fix a typo without much explanation of how GitHub works.
10
11 Let's say you spot a typo while reading along.
12
```

The word "Deleted the extra 'o'" is highlighted with a red box and circled with a red arrow pointing to the line of code.

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.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about diff comparisons](#).

base repository: brad-cannell/r4epi_quarto ▾ base: main ▾ ⏪ head repository: arthur-epi/r4epi_quarto ▾ compare: patch-1 ▾

✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#)

Create pull request

1 commit 1 file changed 1 contributor

Commits on Dec 15, 2023

Fix a typo in contributing.qmd ... arthur-epi committed now

Showing 1 changed file with 2 additions and 2 deletions.

Unified

Split

...

@@ -8,7 +8,7 @@ The easiest way for you to contribute is to help us clean up the little typos an

8 8

9 9 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](<https://github.com/join>). Later in the book, we will cover using GitHub in greater depth (See [@sec-using-git-and-github](#)). Here, we're just going to walk you through how to fix a typo without much explanation of how GitHub works.

10 10

11 - Let's say you spot a **typoo** while reading along.

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

12

13 13 `'(r)

14 14 #| label: contributing_typo_on_screen

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.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#). Learn more about diff comparisons here.

The screenshot shows the GitHub interface for creating a pull request. At the top, there are dropdown menus for 'base repository' (brad-cannell/r4epi_quarto), 'base' (main), 'head repository' (arthur-epi/r4epi_quarto), and 'compare' (patch-1). A green checkmark indicates 'Able to merge'. Below this, there's a title field with 'Add a title' placeholder and a text area containing 'Fix a typo in contributing.qmd'. The text area has a rich text editor toolbar above it. To the right, there are 'Helpful resources' links to 'GitHub Community Guidelines'. Under the title, there's a 'Add a description' section with a text area containing '- Changed "typoo" to "typo" on line 11.' Below the text area, there are buttons for 'Write' and 'Preview', and a note that 'Markdown is supported'. At the bottom, there's a 'Create pull request' button with a dropdown arrow, and a note that 'Remember, contributions to this repository should follow our [GitHub Community Guidelines](#)'.

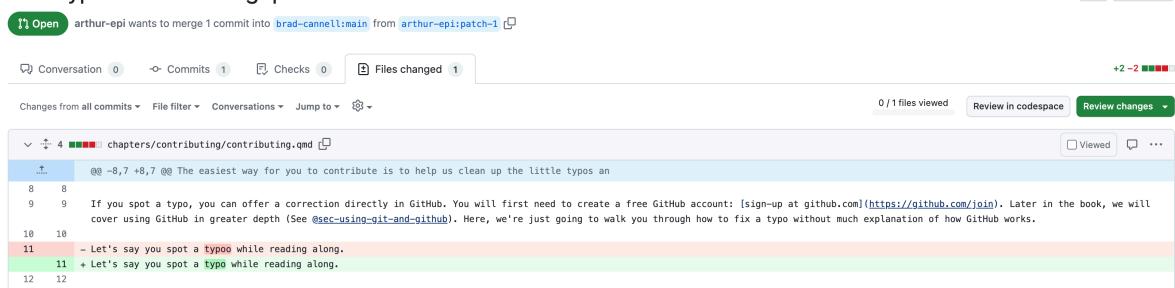
After doing so, you will get one final chance to amend the description of your proposed changes. If you are happy with the commit message and description, then click the **Create pull request** button one more time. At this point, your job is done! It is now up to the authors to review the changes you've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

The screenshot shows the GitHub repository navigation bar for 'brad-cannell / r4epi_quarto'. The bar includes icons for 'Code', 'Issues' (1), 'Pull requests' (1), 'Actions', and 'Projects' (1). The 'Pull requests' button is highlighted with a red box.

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

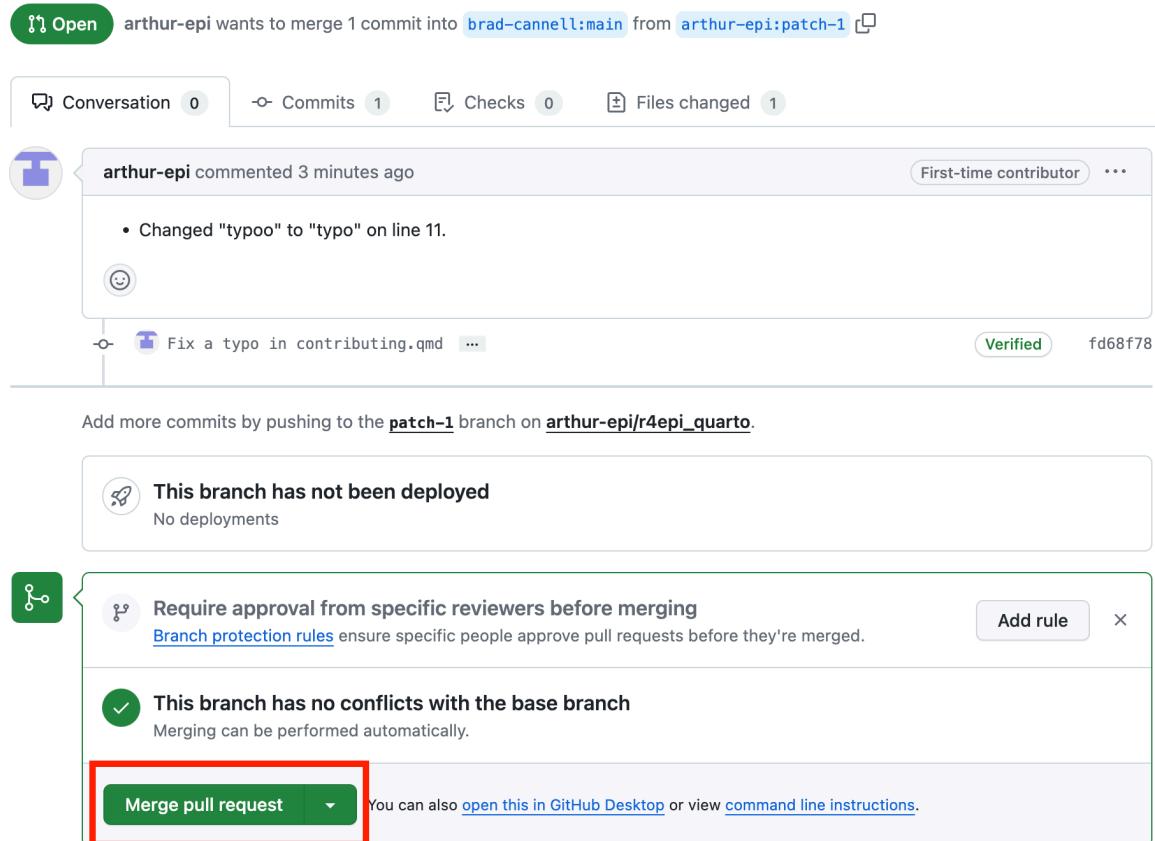
Fix a typo in contributing.qmd #7



The screenshot shows a GitHub pull request interface. At the top, it says "arthur-epi wants to merge 1 commit into brad-cannell:main from arthur-epi:patch-1". Below this is a navigation bar with tabs for Conversation (0), Commits (1), Checks (0), and Files changed (1). The "Files changed" tab is active, showing a diff for "chapters/contributing/contributing.qmd". The diff highlights a change on line 11: "- Let's say you spot a typoo while reading along." is replaced by "+ Let's say you spot a typo while reading along.". The commit message "Fix a typo in contributing.qmd" is visible at the bottom of the pull request page.

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.

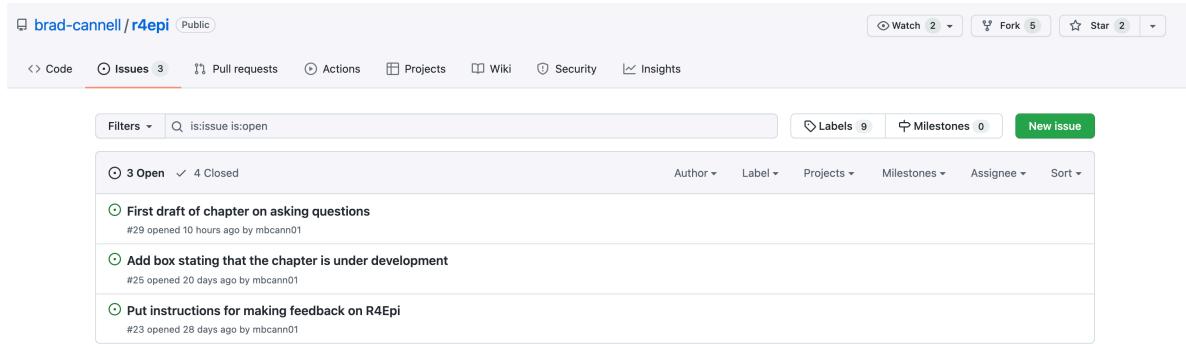
Fix a typo in contributing.qmd #7



The screenshot shows a GitHub pull request interface. At the top, it says "arthur-epi wants to merge 1 commit into brad-cannell:main from arthur-epi:patch-1". Below this is a navigation bar with tabs for Conversation (0), Commits (1), Checks (0), and Files changed (1). A comment from "arthur-epi" is shown, stating: "Changed "typoo" to "typo" on line 11." Below the comment is the commit message "Fix a typo in contributing.qmd". The commit is marked as "Verified" and has a hash "fd68f78". At the bottom, there is a message: "Add more commits by pushing to the patch-1 branch on arthur-epi/r4epi_quarto." A deployment status box says "This branch has not been deployed" with "No deployments". The merge interface shows "Require approval from specific reviewers before merging" and "Branch protection rules ensure specific people approve pull requests before they're merged." It also shows "This branch has no conflicts with the base branch" and "Merging can be performed automatically." A green "Merge pull request" button is highlighted with a red box.

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>.



The screenshot shows the GitHub Issues page for the repository "brad-cannell/r4epi". The page has a header with "Code", "Issues 3", "Pull requests", "Actions", "Projects", "Wiki", "Security", and "Insights". Below the header are buttons for "Watch 2", "Fork 5", and "Star 2". A search bar contains the query "is:issue is:open". There are filters for "Labels 9" and "Milestones 0", and a green "New issue" button. The main area shows three open issues:

- First draft of chapter on asking questions (#29) - opened 10 hours ago by mbcann01
- Add box stating that the chapter is under development (#25) - opened 20 days ago by mbcann01
- Put instructions for making feedback on R4Epi (#23) - opened 28 days ago by mbcann01

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

This book was created by Brad Cannell and is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

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 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/>.

The Comprehensive R Archive Network

Download and Install R

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

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

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

Source Code for all Platforms

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

- The latest release (2020-04-24, Arbor Day) [R-4.0.0.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#)

Questions About R

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

Step 3: Click on Download R for macOS.

The Comprehensive R Archive Network

Download and Install R

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

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

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

Source Code for all Platforms

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

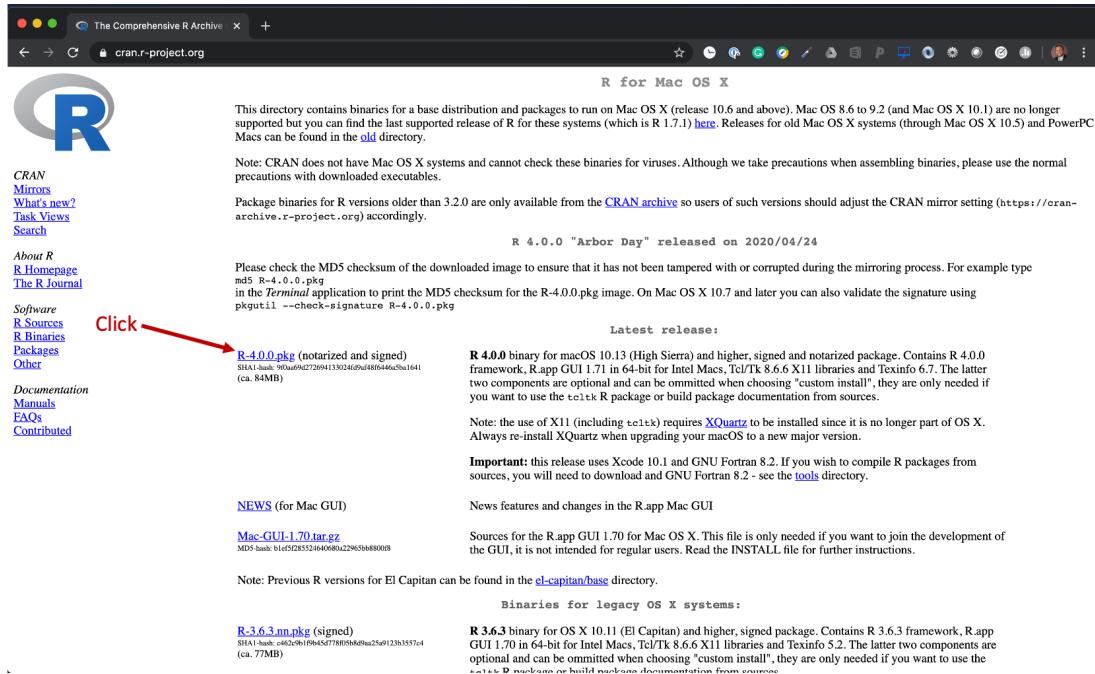
- The latest release (2020-04-24, Arbor Day) [R-4.0.0.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#)

Questions About R

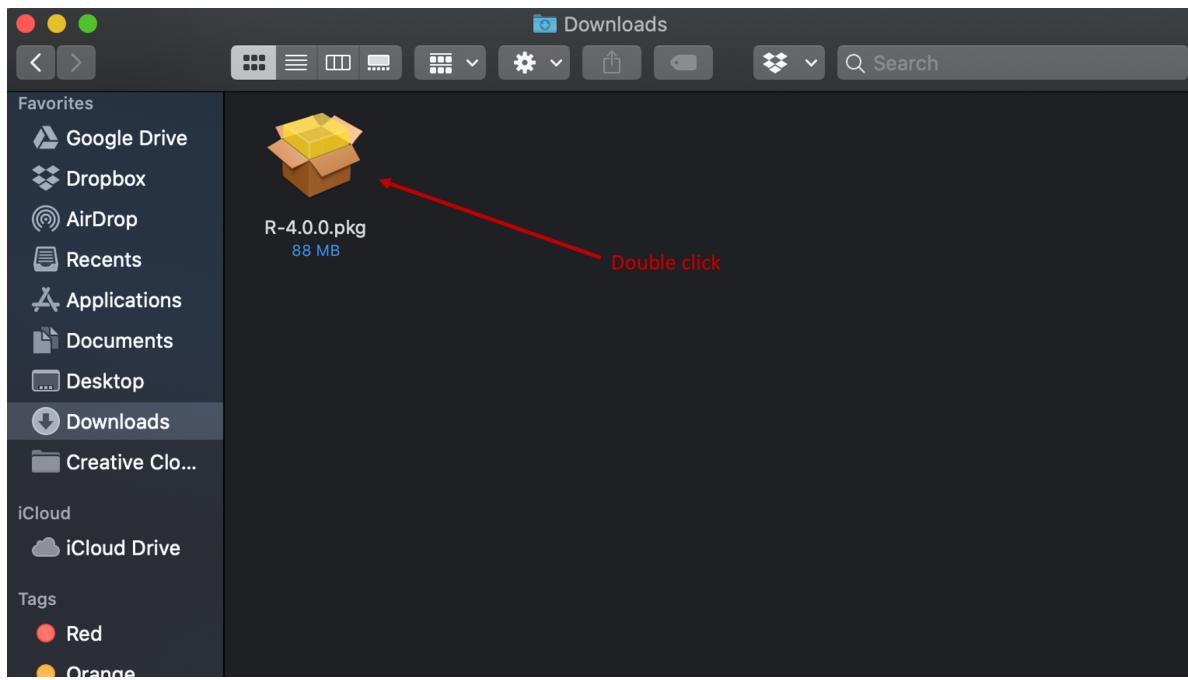
- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

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

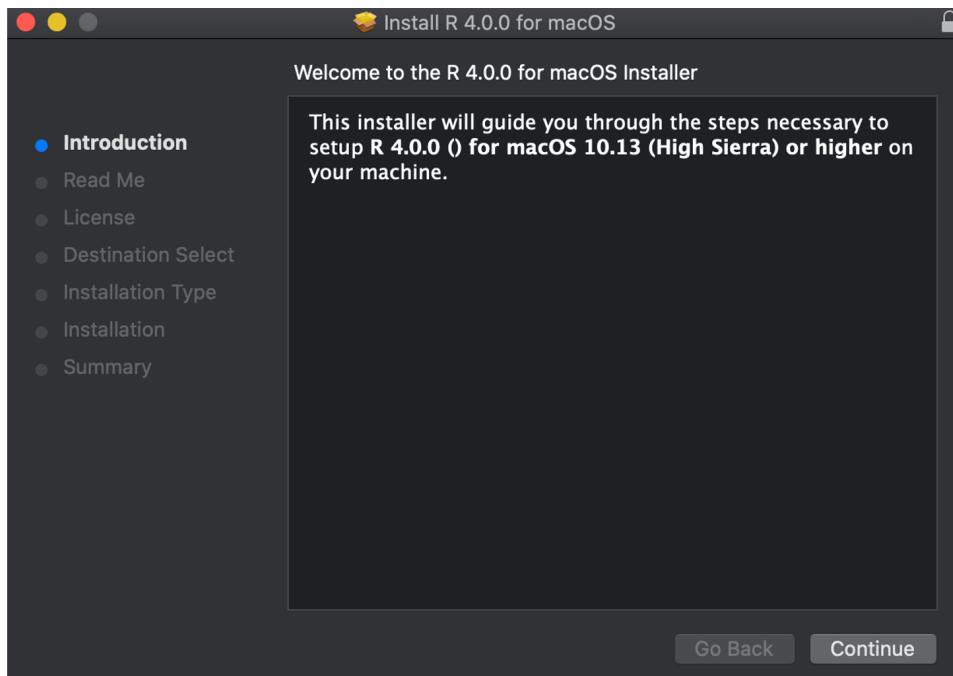
version should be roughly in the same place – the middle of the screen under “Latest release:”. After clicking the link, R should start to download to your computer automatically.



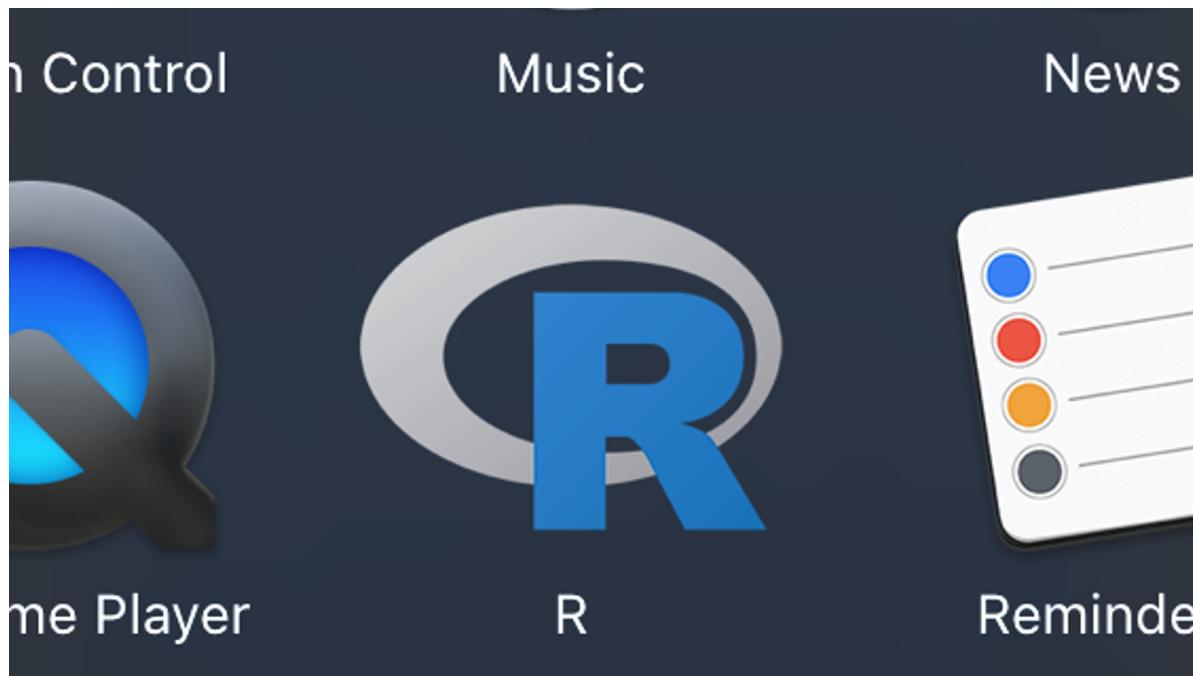
Step 5: Locate the package file you just downloaded and double click it. Unless you've changed your download settings, this file will probably be in your “downloads” folder. That is the default location for most web browsers. After you locate the file, just double click it.



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.



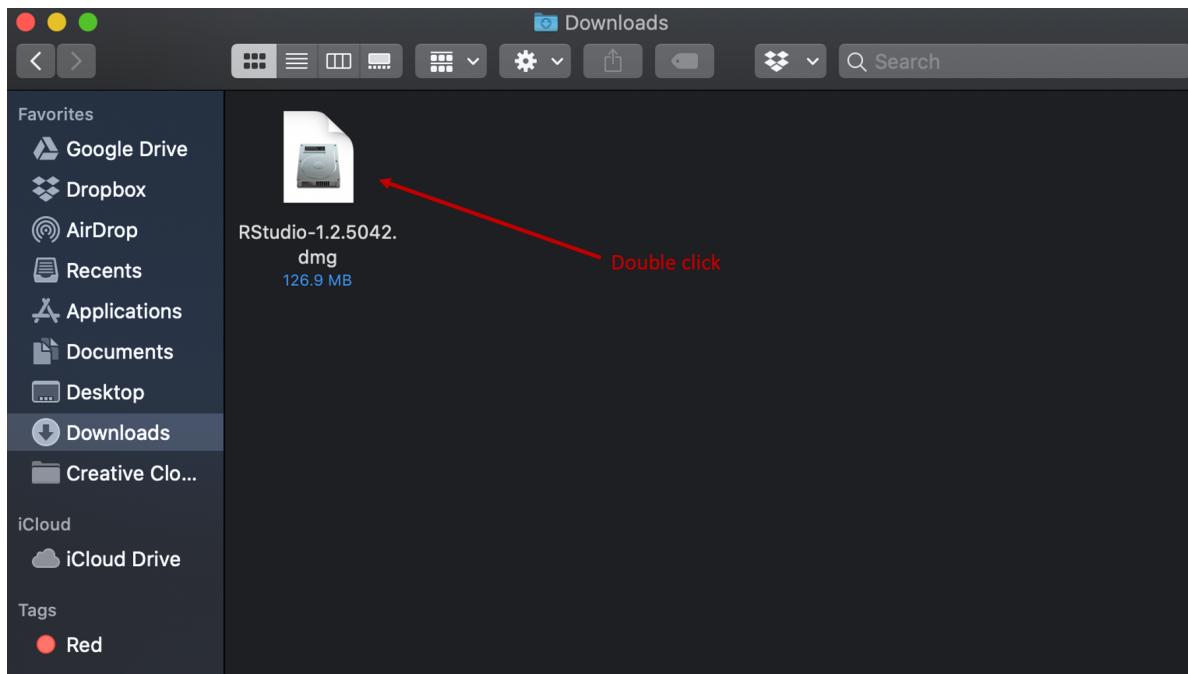
If R installed properly, you should now see it in your applications folder.



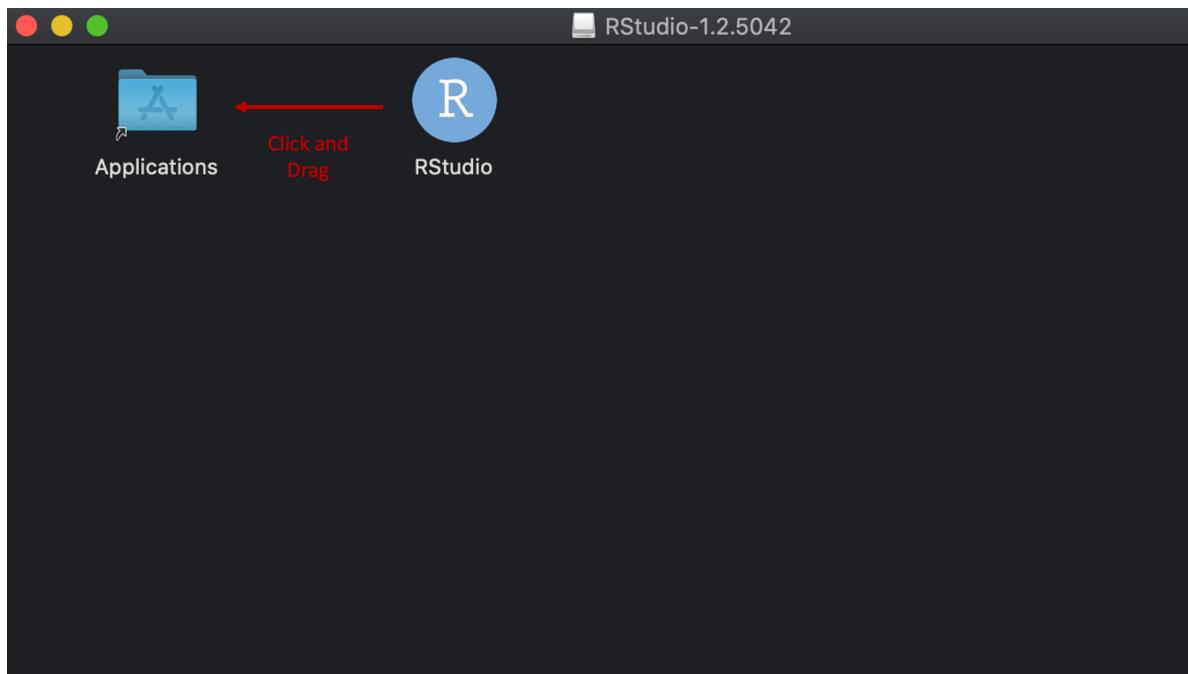
Step 7: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at <https://posit.co/download/rstudio-desktop/>. On that page, click the button to download the latest version of RStudio for your computer. Note that the website may look different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

OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2024.04.1-748.EXE	263.07 MB	44C8797C
macOS 12+	RSTUDIO-2024.04.1-748.DMG	566.51 MB	A5EDA699
Ubuntu 20/Debian 11	RSTUDIO-2024.04.1-748-AMD64.DEB	194.71 MB	505311AE
Ubuntu 22/Debian 12	RSTUDIO-2024.04.1-748-AMD64.DEB	197.00 MB	88D485CD
OpenSUSE 15	RSTUDIO-2024.04.1-748-X86_64.RPM	197.21 MB	D25315A4
Fedora 34/Red Hat 8	RSTUDIO-2024.04.1-748-X86_64.RPM	219.99 MB	A97A28A7
Fedora 36/Red Hat 9	RSTUDIO-2024.04.1-748-X86_64.RPM	211.10 MB	69580324

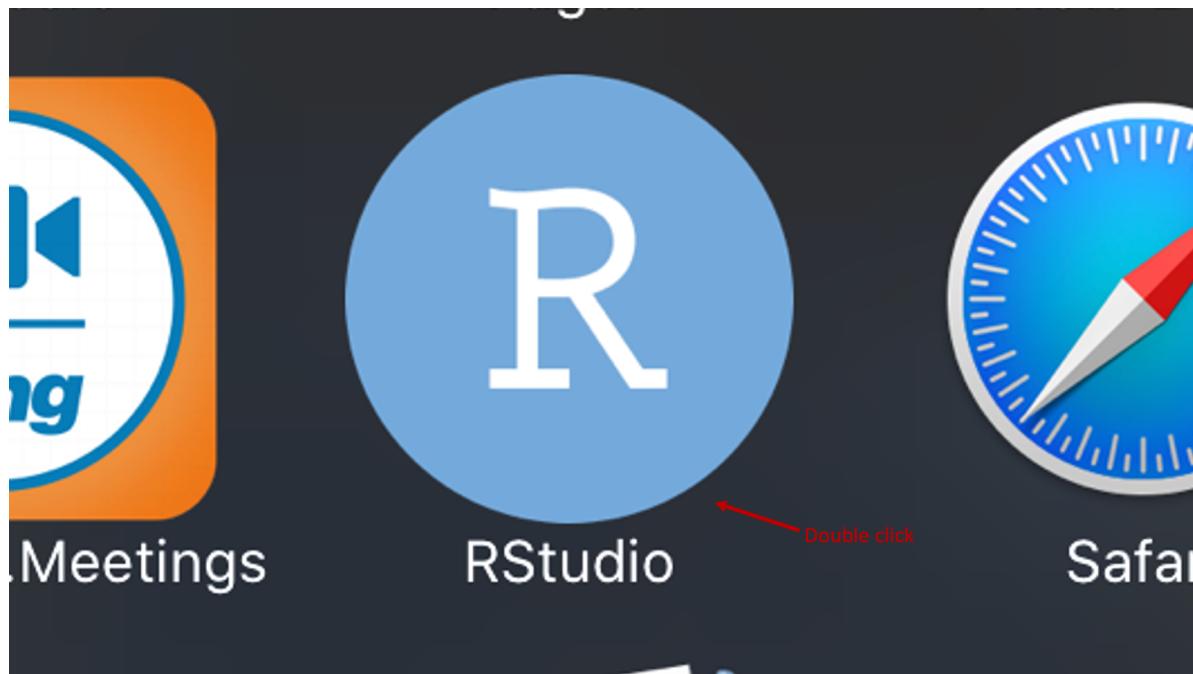
Step 8: Again, locate the DMG file you just downloaded and double click it. Unless you've changed your download settings, this file should be in the same location as the R package file you already downloaded.



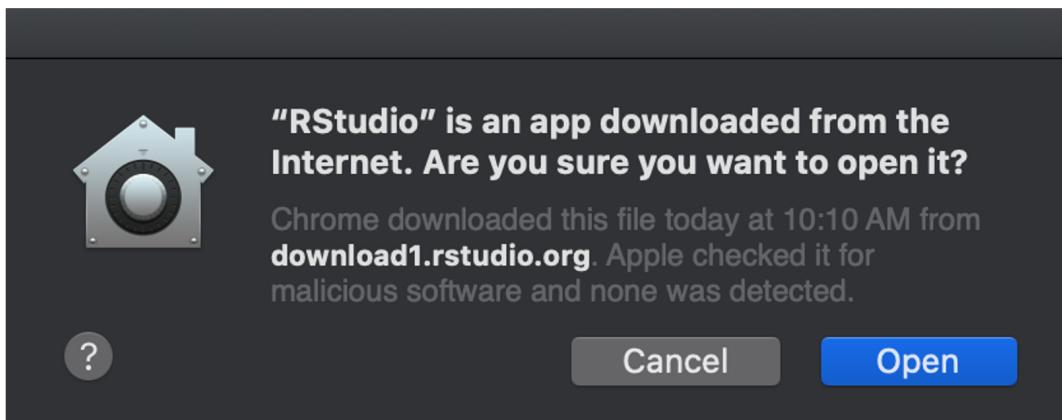
Step 9: A new finder window should automatically pop up that looks like the one you see below. Click on the RStudio icon and drag it into the Applications folder.



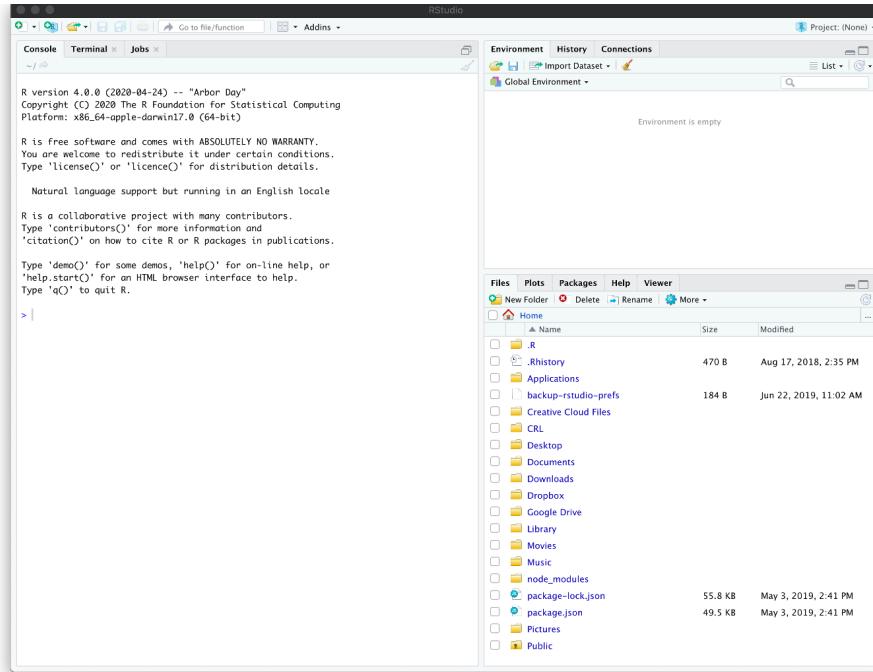
You should now see RStudio in your Applications folder. Double click the icon to open RStudio.



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

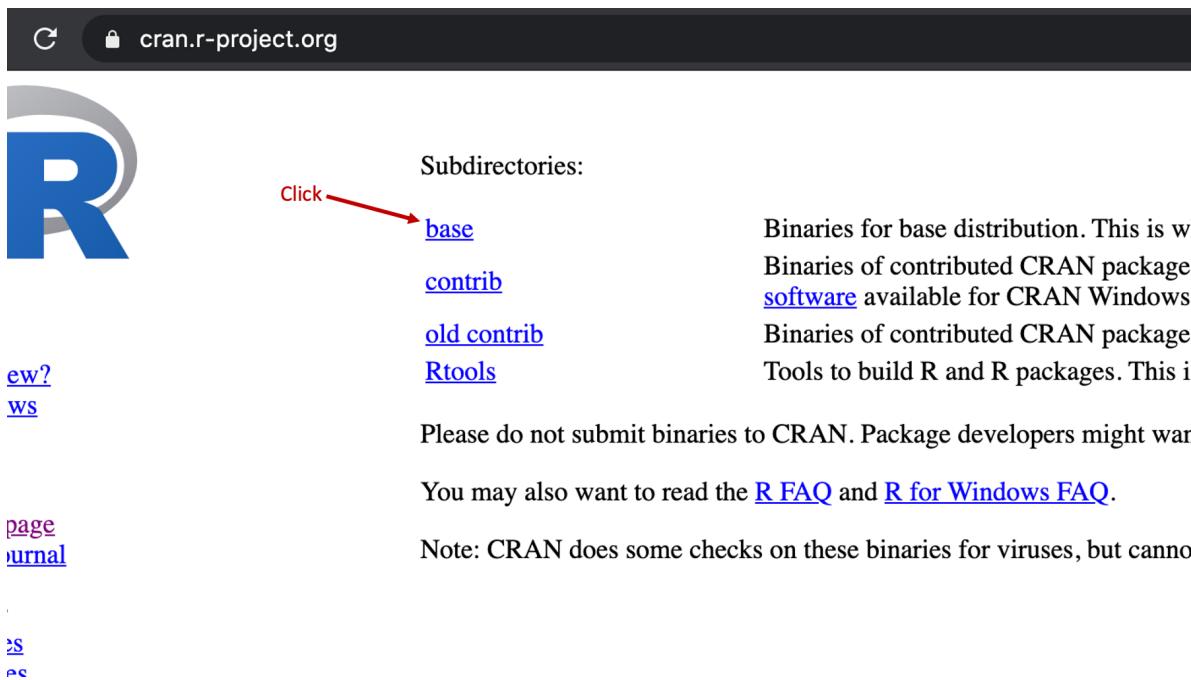
Step 2: Navigate to the Comprehensive R Archive Network (CRAN), which is located at <https://cran.r-project.org/>.

The screenshot shows the main page of the Comprehensive R Archive Network. On the left, there's a sidebar with links like CRAN, Mirrors, What's new?, Task Views, Search, About R, R Homepage, The R Journal, Software, R Sources, R Binaries, Packages, Other, Documentation, Manuals, FAQs, and Contributed. The main content area has a large title "The Comprehensive R Archive Network". Below it, a section titled "Download and Install R" contains a list of precompiled binary distributions for Windows and Mac users. Another section below it discusses Linux distributions and source code. A third section, "Questions About R", provides answers to common questions about R. At the bottom, there's a "What are R and CRAN?" section and a "Submitting to CRAN" note.

Step 3: Click on Download R for Windows.

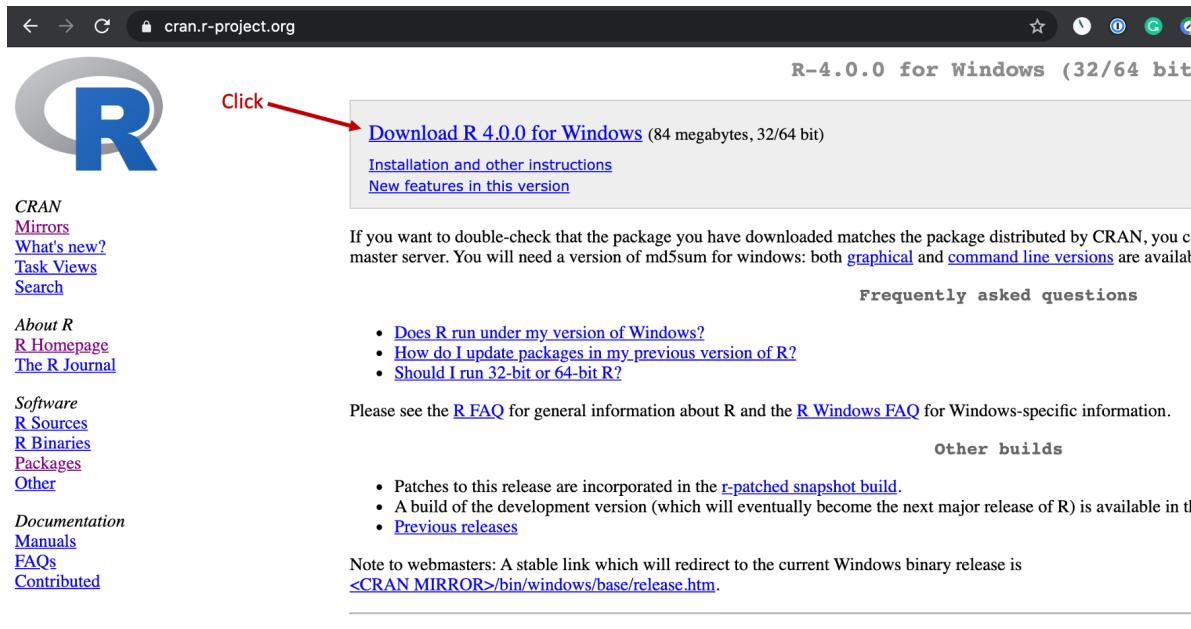
This screenshot is identical to the one above, but it includes a red arrow pointing to the "Download R for Windows" link within the "Download and Install R" section. This indicates the specific action the user needs to take in Step 3.

Step 4: Click on the base link.



The screenshot shows the CRAN homepage. On the left, there is a large blue 'R' logo. To its right, a red arrow points from the word 'Click' to a list of subdirectories: [base](#), [contrib](#), [old_contrib](#), and [Rtools](#). To the right of these links, descriptions are provided: 'Binaries for base distribution. This is w...' for [base](#), 'Binaries of contributed CRAN packages...' for [contrib](#), 'Binaries of contributed CRAN packages...' for [old_contrib](#), and 'Tools to build R and R packages. This is...' for [Rtools](#). Below this, a note says 'Please do not submit binaries to CRAN. Package developers might wan...'. Further down, it says 'You may also want to read the [R FAQ](#) and [R for Windows FAQ](#)'. At the bottom, there is a note: 'Note: CRAN does some checks on these binaries for viruses, but cannot...'.

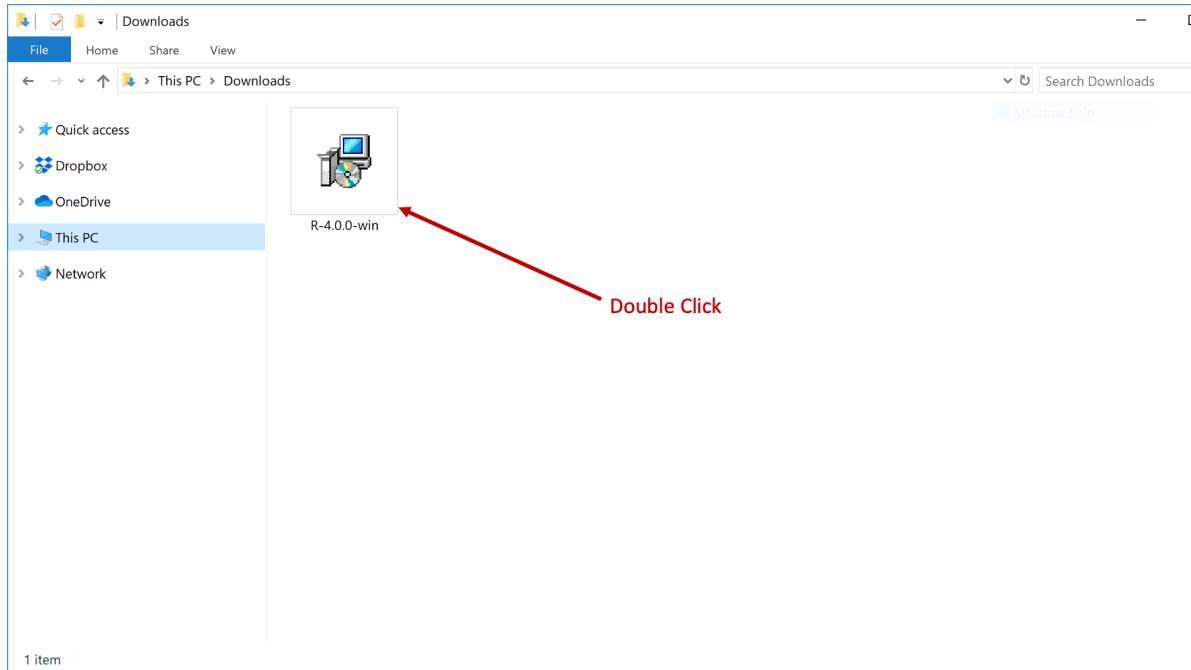
Step 5: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version should be roughly the same. After clicking, R should start to download to your computer.



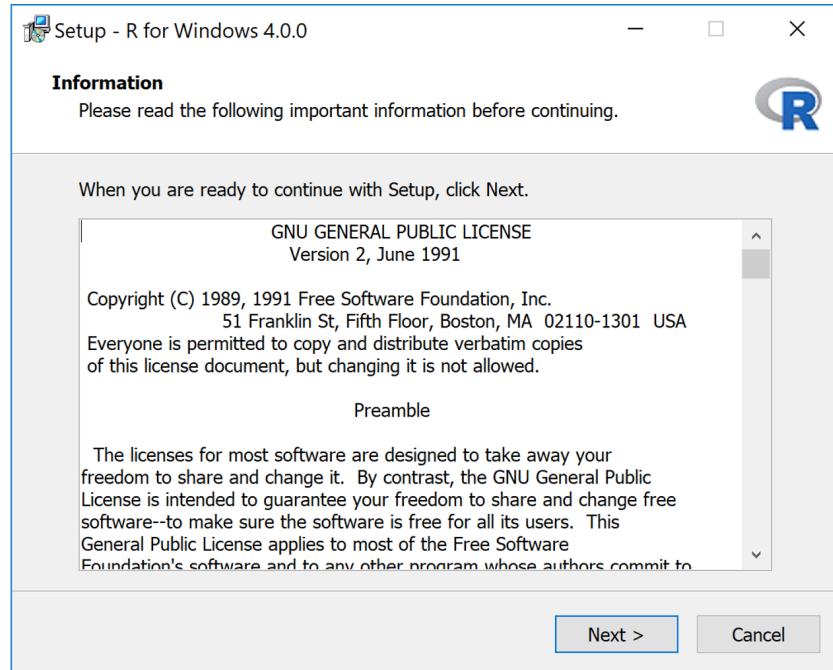
The screenshot shows the 'R-4.0.0 for Windows (32/64 bit)' page. A red arrow points from the word 'Click' to the [Download R 4.0.0 for Windows](#) button. Below the button, there are links for 'Installation and other instructions' and 'New features in this version'. To the left, there is a sidebar with links for 'CRAN', 'About R', 'Software', 'Documentation', and 'Last change: 2020-04-24'. To the right, there are sections for 'Frequently asked questions' (with links to 'Does R run under my version of Windows?', 'How do I update packages in my previous version of R?', and 'Should I run 32-bit or 64-bit R?'), 'Other builds' (with links to 'Patches to this release are incorporated in the r-patched snapshot build.', 'A build of the development version (which will eventually become the next major release of R) is available in th...', and 'Previous releases'), and a note for webmasters about a stable link to the current Windows binary release.

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

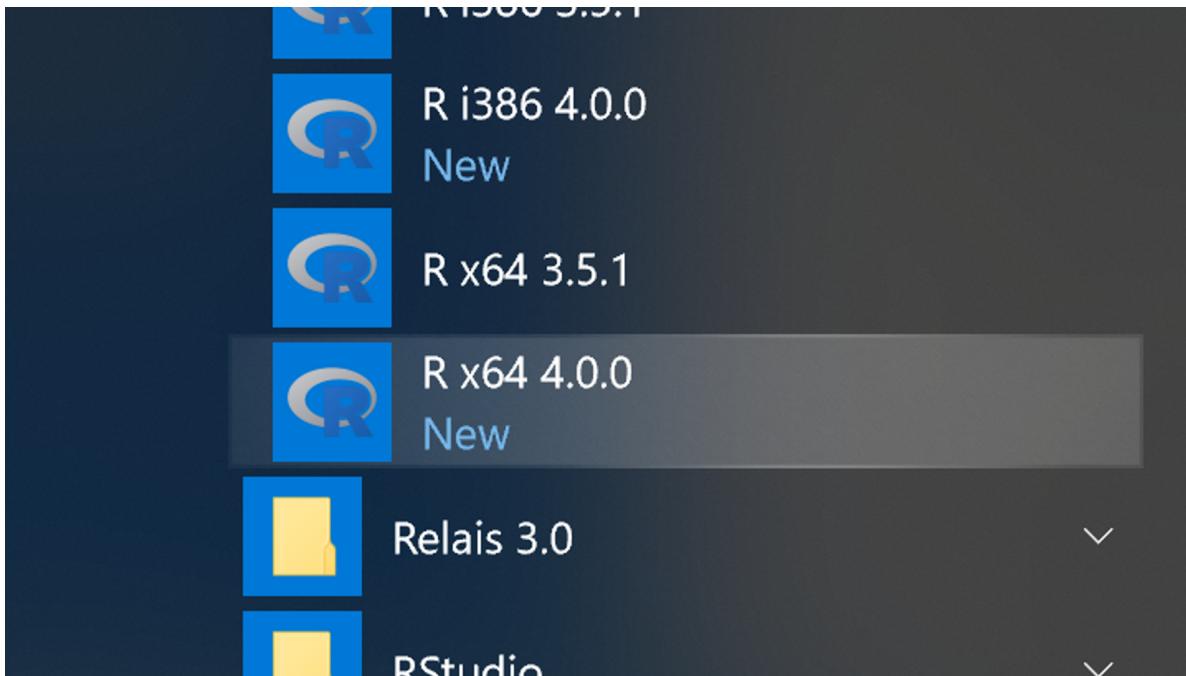
changed your download settings, this file will probably be in your downloads folder. That is the default location for most web browsers.



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.

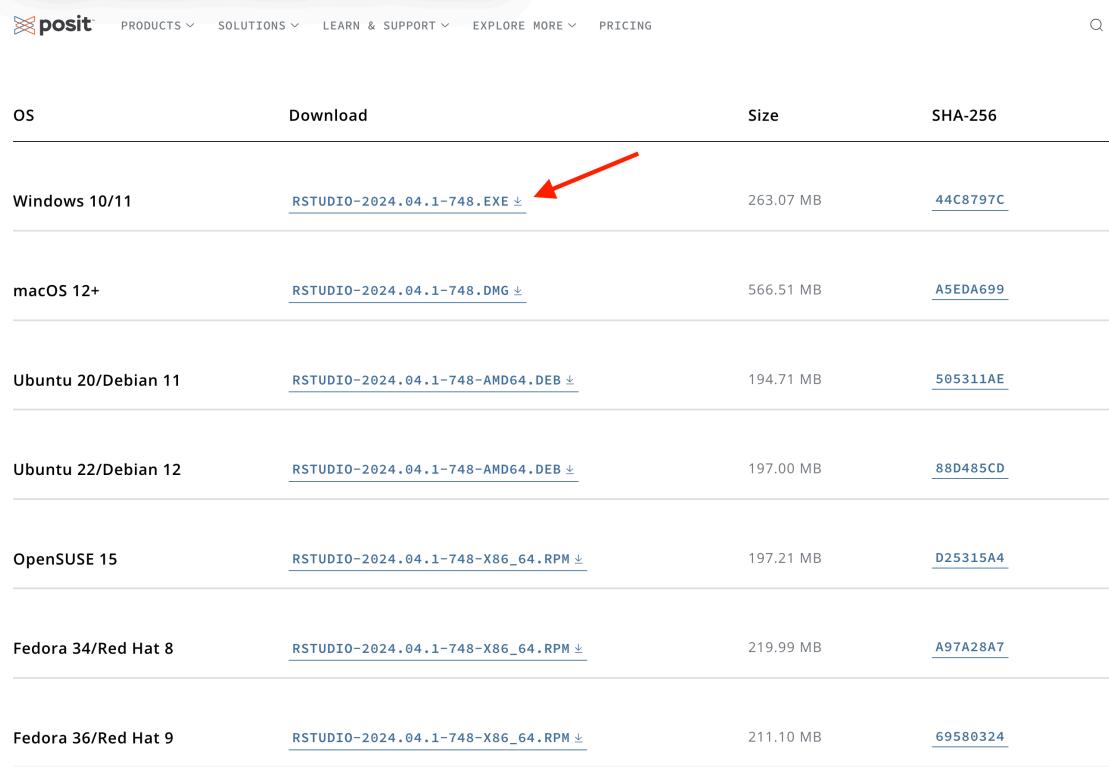


If R installed properly, you should now see it in the Windows start menu.



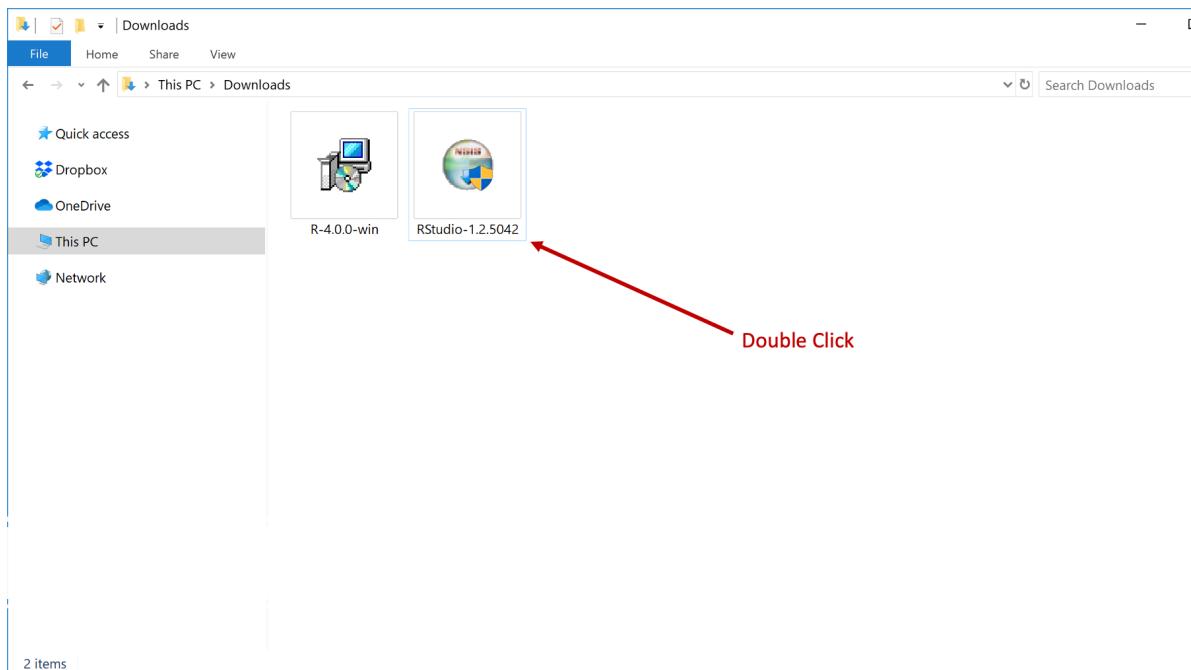
Step 8: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at <https://posit.co/download/rstudio-desktop/>. On that page, click the button to download the latest version of RStudio for your computer. Note that

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

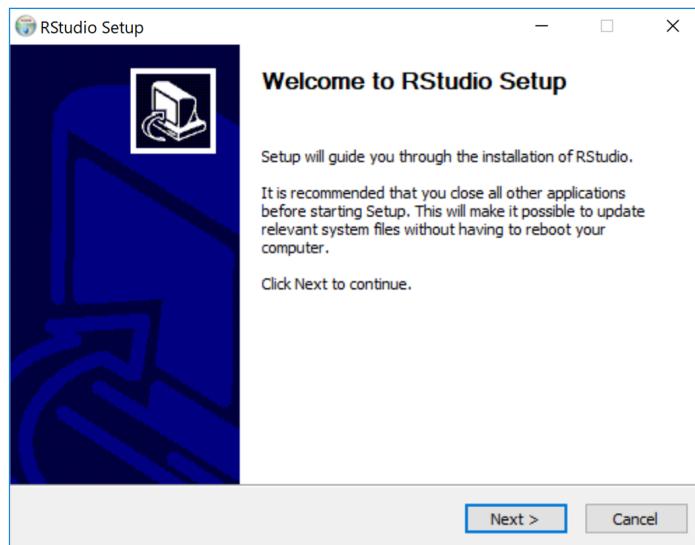


OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2024.04.1-748.EXE	263.07 MB	44C8797C
macOS 12+	RSTUDIO-2024.04.1-748.DMG	566.51 MB	A5EDA699
Ubuntu 20/Debian 11	RSTUDIO-2024.04.1-748-AMD64.DEB	194.71 MB	505311AE
Ubuntu 22/Debian 12	RSTUDIO-2024.04.1-748-AMD64.DEB	197.00 MB	88D485CD
OpenSUSE 15	RSTUDIO-2024.04.1-748-X86_64.RPM	197.21 MB	D25315A4
Fedora 34/Red Hat 8	RSTUDIO-2024.04.1-748-X86_64.RPM	219.99 MB	A97A28A7
Fedora 36/Red Hat 9	RSTUDIO-2024.04.1-748-X86_64.RPM	211.10 MB	69580324

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.

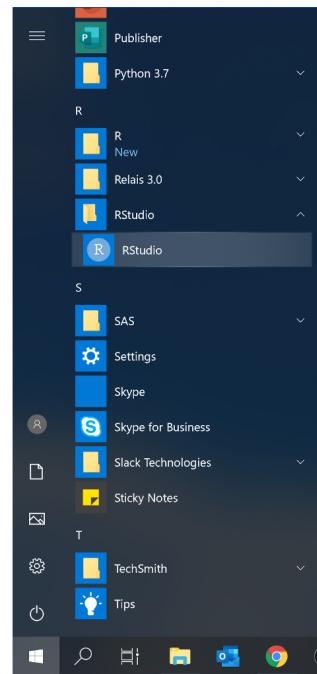


Step 10: Another dialogue box will open and ask you to make some decisions about how and where you want to install RStudio on your computer. We typically just click “Next” at every step without changing any of the default options.

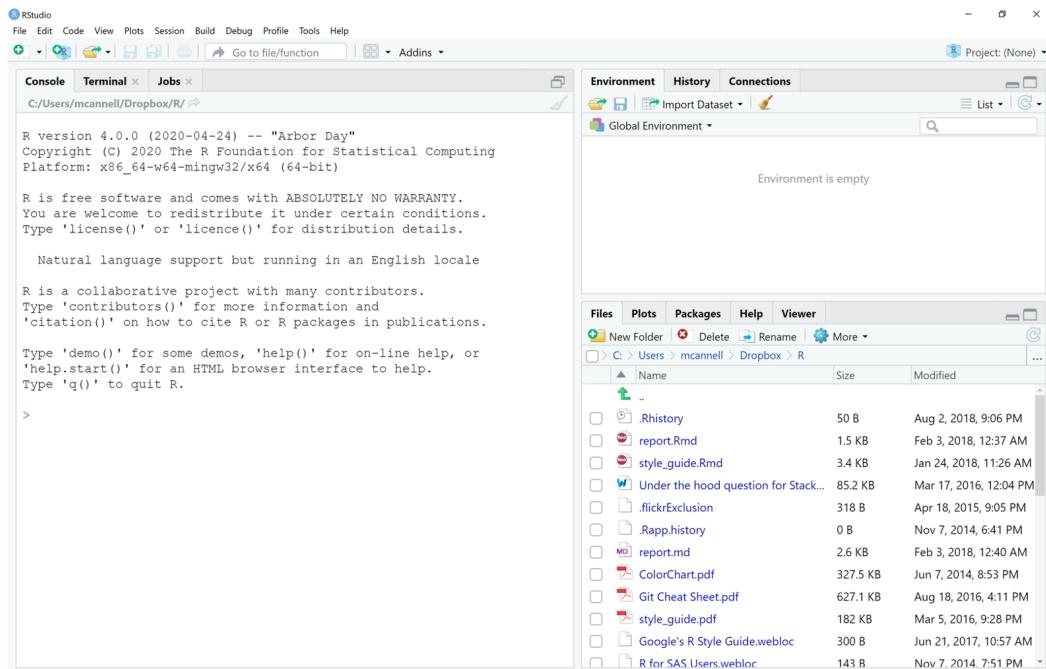


When RStudio is finished installing, you should see RStudio in the Windows start menu. Click

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
1. Numbers and the dot (.) or underscore (_) characters	Male	71	190
2. Begins with a letter or a dot as long as the dot is not followed by a number	Male	69	176
3. No reserved words			
003	Female	64	130
004	Female	65	154

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

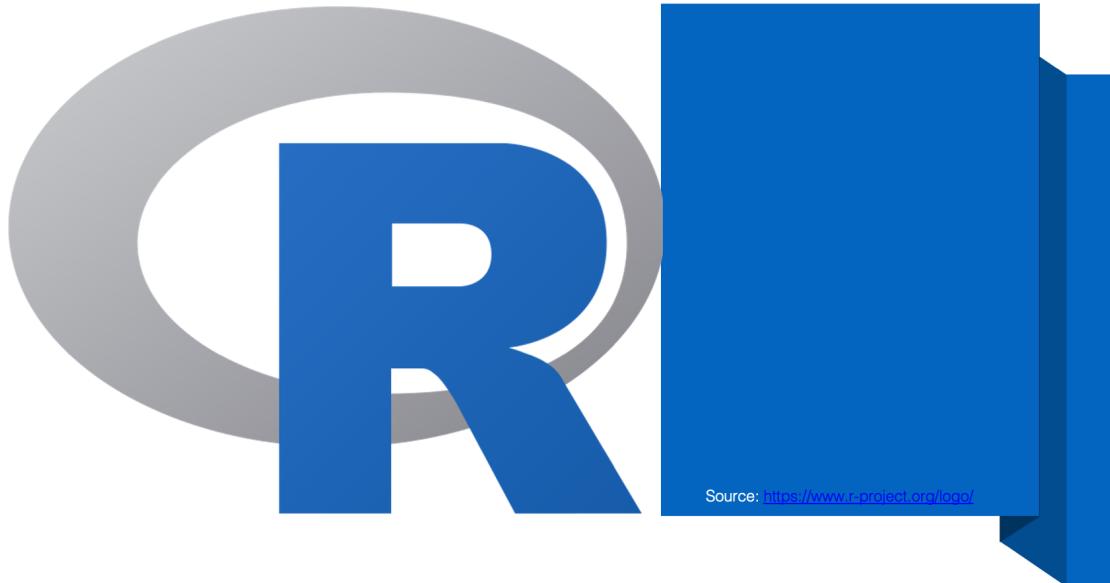
ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

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

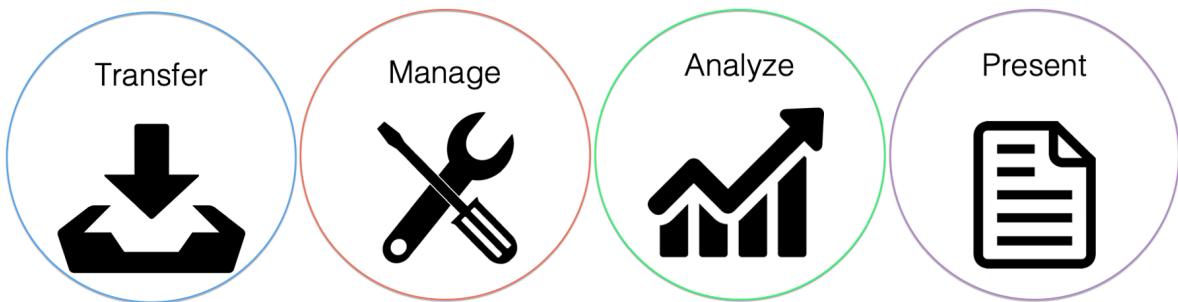
ID	Gender	Height	Weight
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154

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

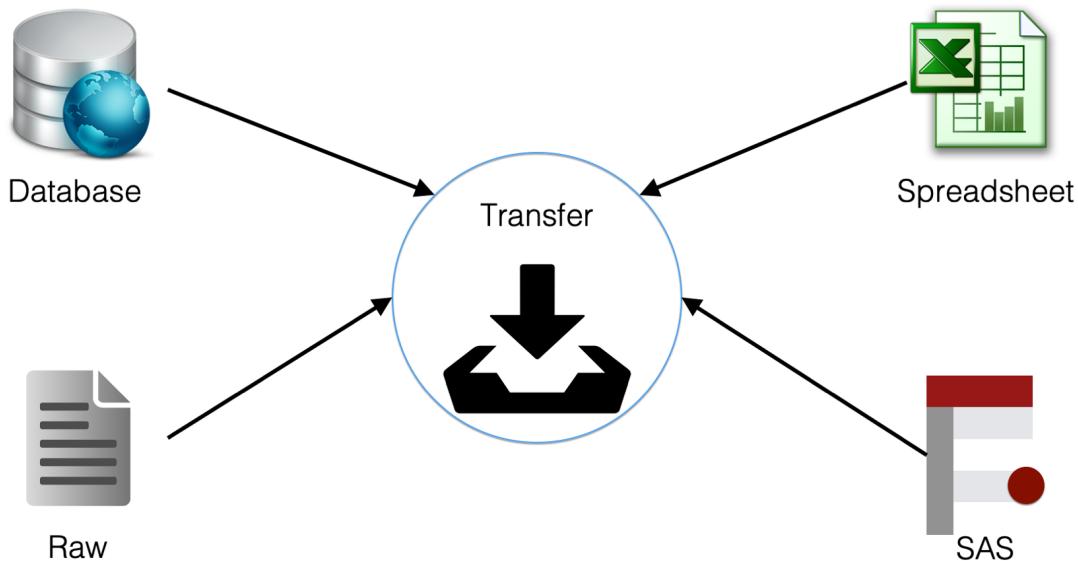
2.2 What is R?



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



2.2.1 Transferring data



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

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

2.2.2 Managing data



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

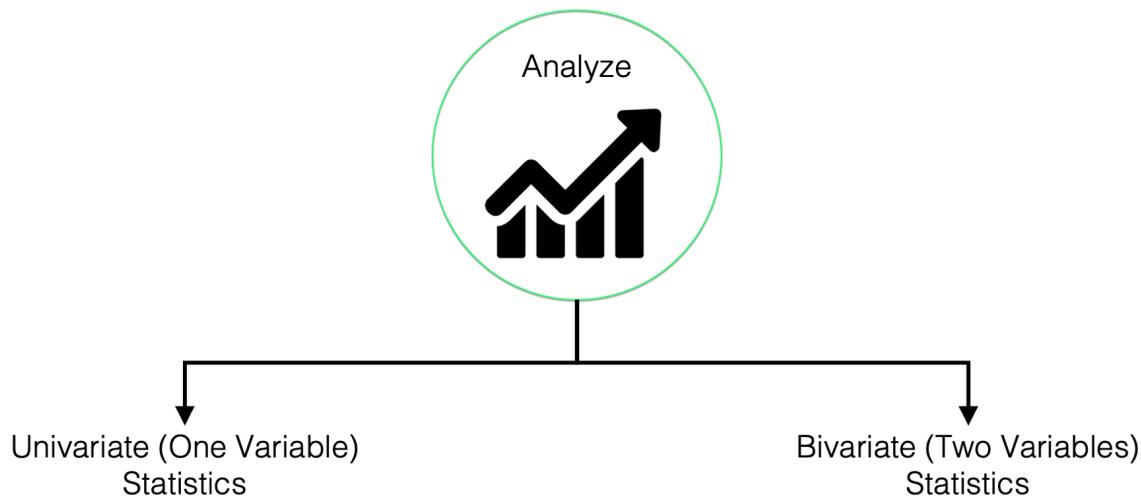
- Validating and cleaning data. In other words, dealing with potential errors in the data.
- Subsetting data. For example, using only some of the columns or some of the rows.
- Creating new variables. For example, creating a BMI variable in a data frame that was sent to you with height and weight columns.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

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

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

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

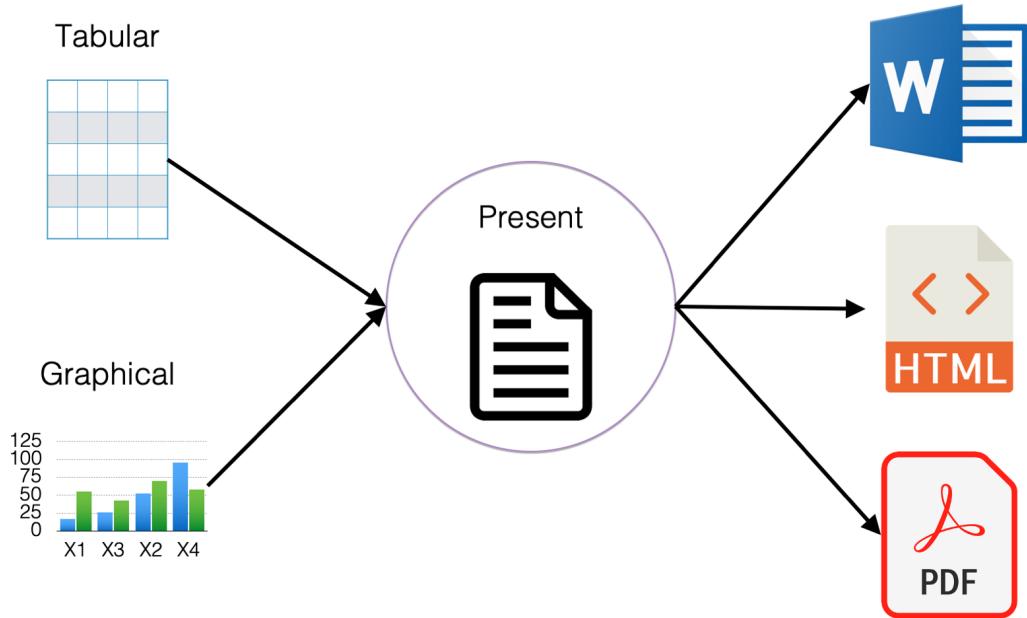
2.2.3 Analyzing data



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

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

2.2.4 Presenting data



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

1. **Microsoft Word documents.** [Click here](#) to view an example report created for one of our research projects in Microsoft Word.
2. **PDF documents.** [Click here](#) to view a data dictionary we created in PDF format.

3. **HTML files.** Hypertext Markup Language (HTML) files are what you are looking at whenever you view a webpage. You can use R to create HTML files that others can view in their web browser. You can email them these files to view in their web browser, or you can make them available for others to view online just like any other website. [Click here](#) to view an example dashboard we created for one of our research projects.
4. **Web applications.** You can even use R to create full-fledged web applications. View the [RStudio website](#) to see some examples.

3 Navigating the RStudio Interface

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

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

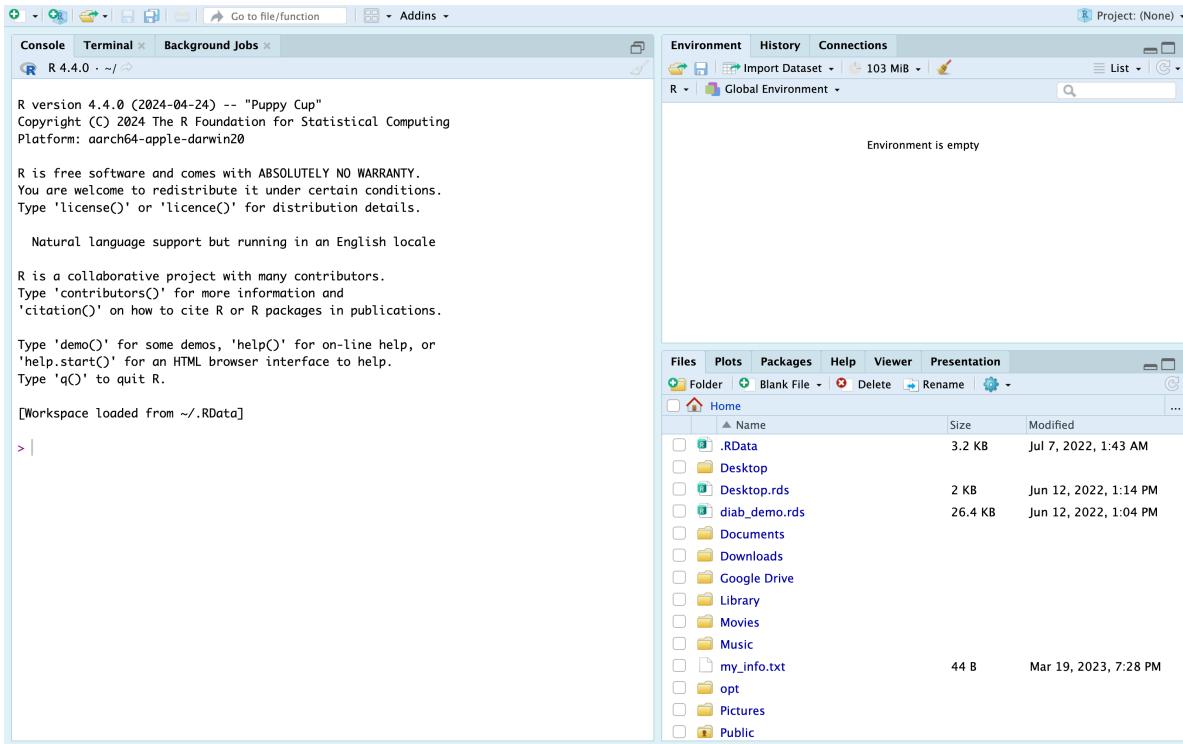


Figure 3.1: The default RStudio user interface.

3.1 The console pane

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

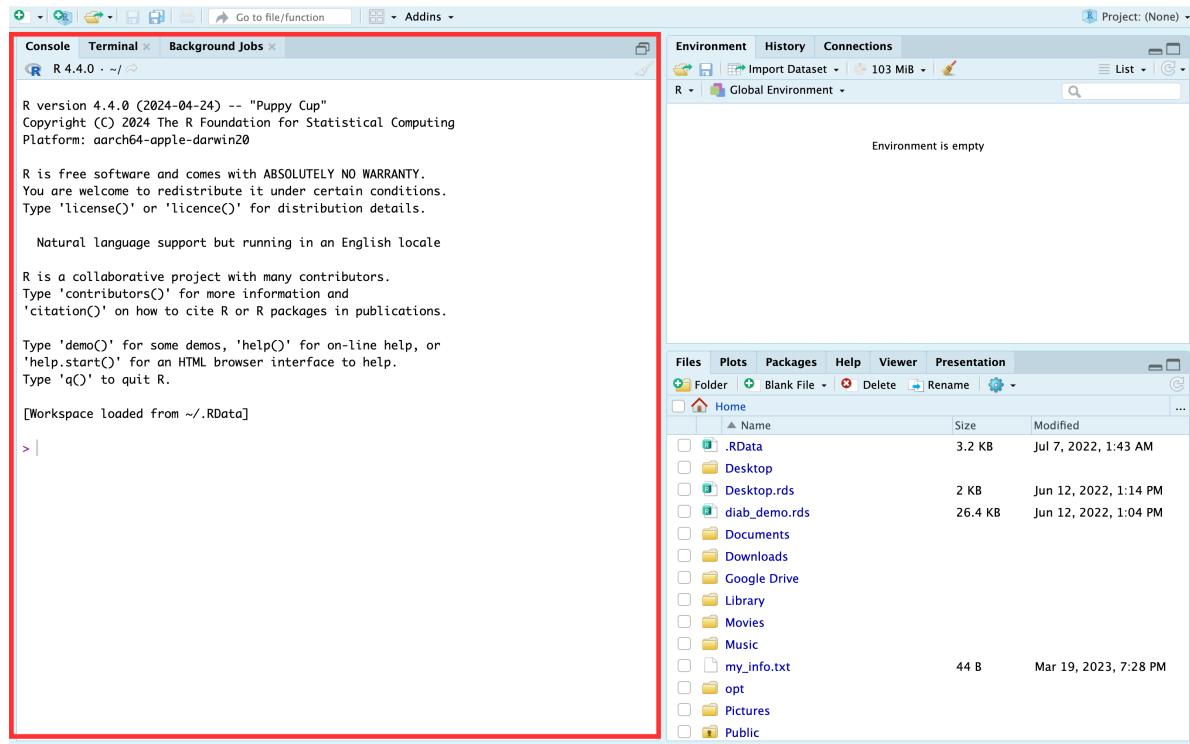


Figure 3.2: The R Console.

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

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

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

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

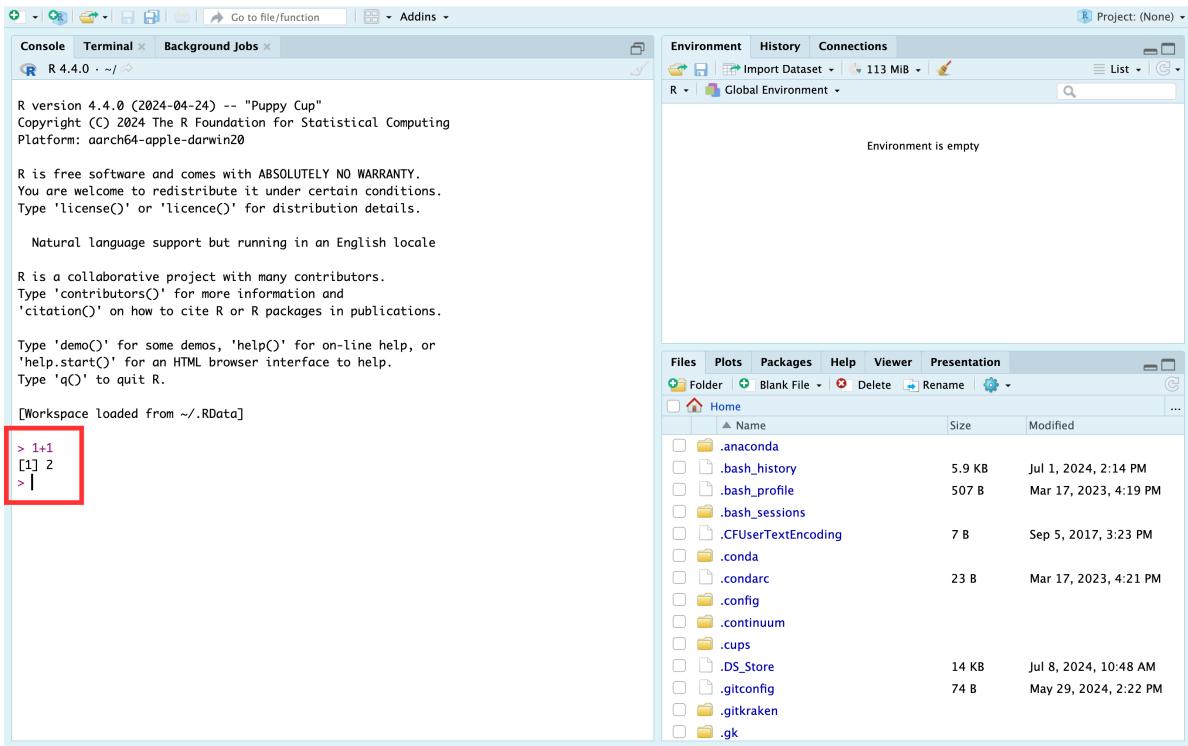
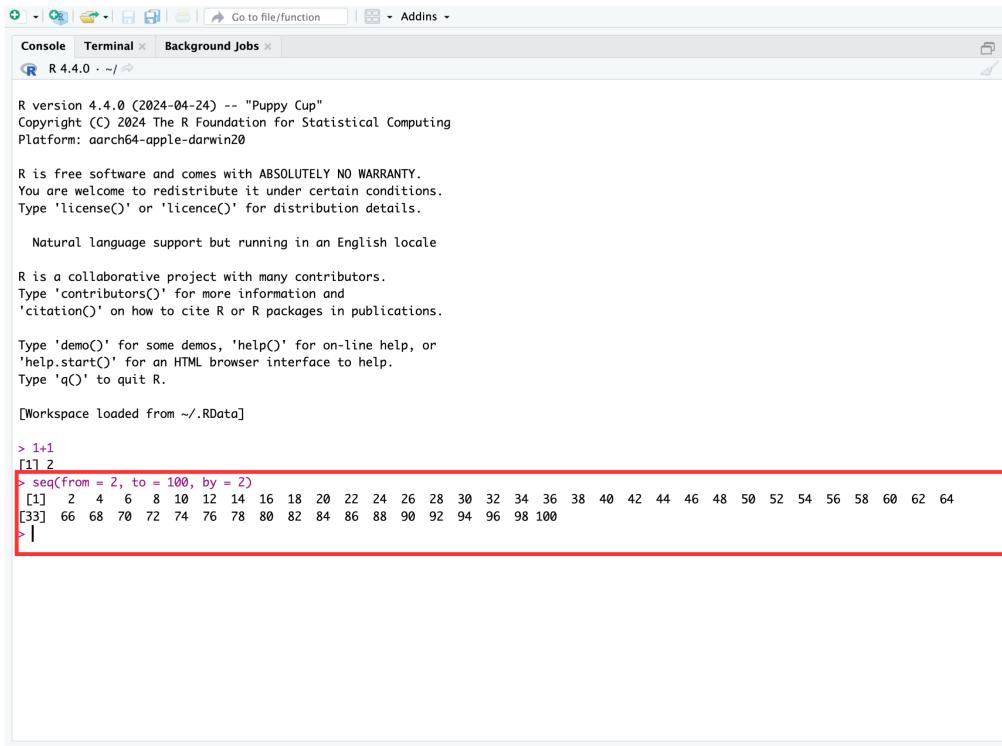


Figure 3.3: Doing some addition in the R console.



```
R version 4.4.0 (2024-04-24) -- "Puppy Cup"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20

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

Natural language support but running in an English locale

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

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

[Workspace loaded from ~/.RData]

> 1+1
[1] 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 40 42 44 46 48 50 52 54 56 58 60 62 64
[33] 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100
> |
```

Figure 3.4: Demonstrating a function that returns multiple results.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

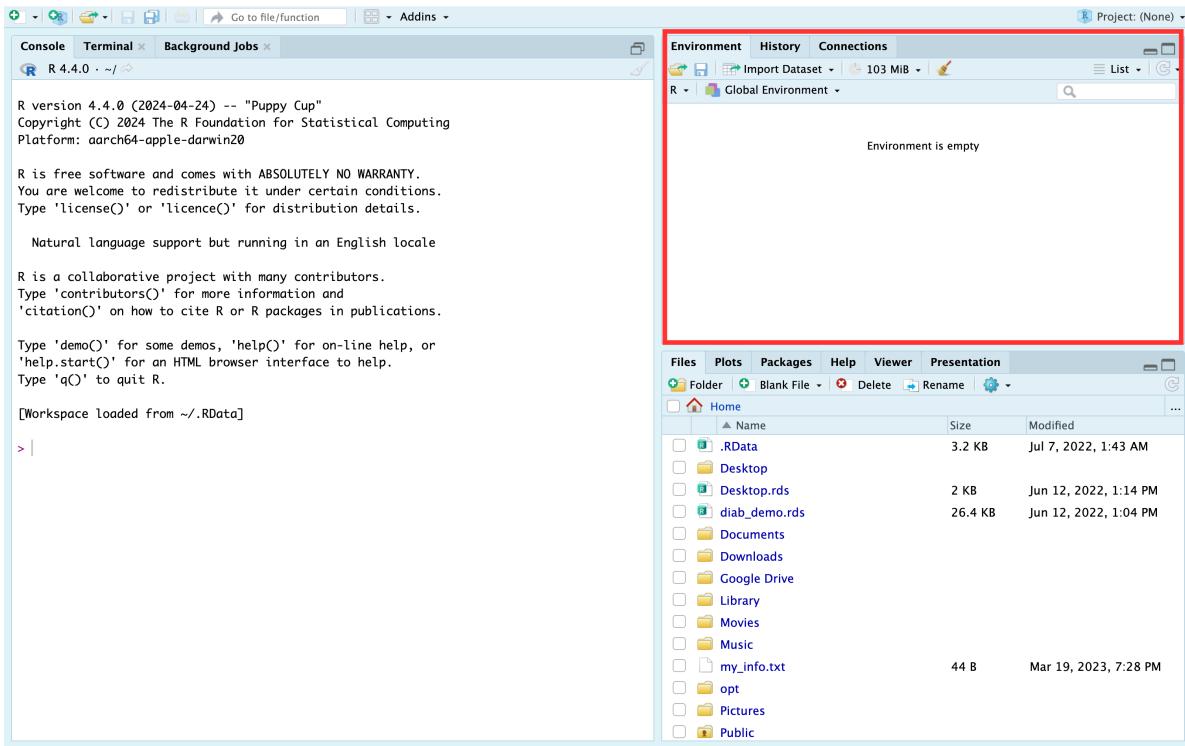


Figure 3.5: The environment pane

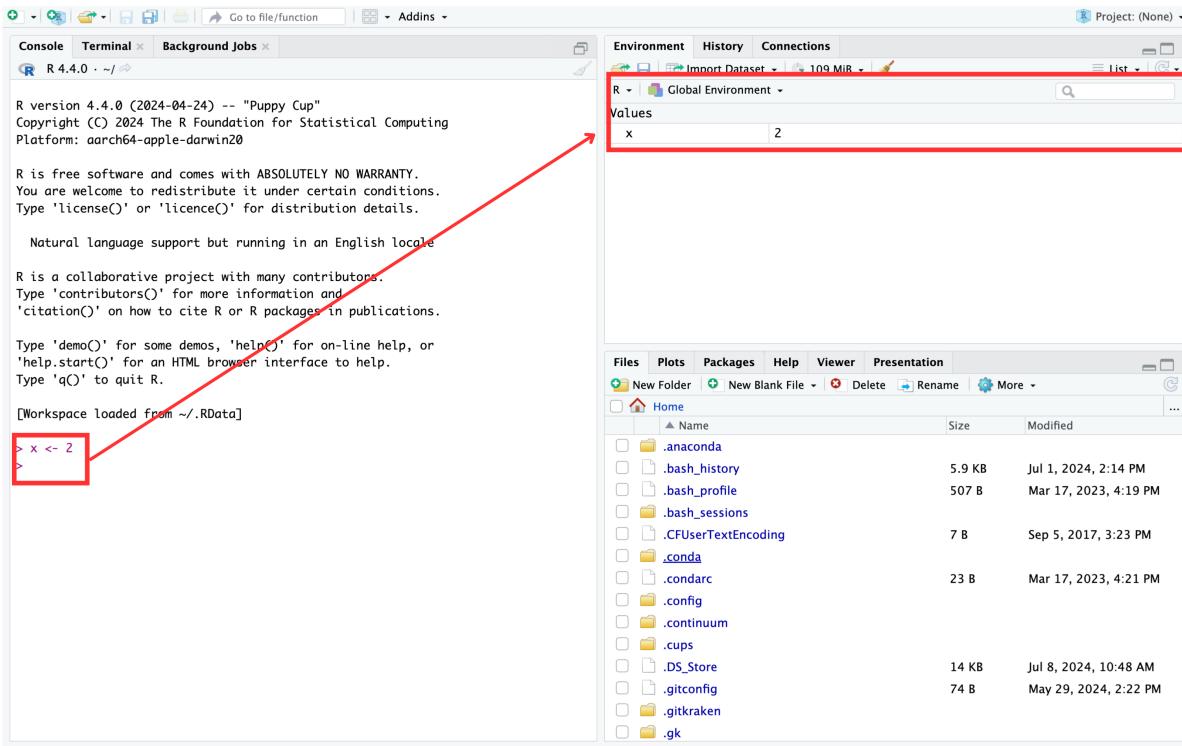


Figure 3.6: The vector `x` in the global environment.

Note

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

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

Warning

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

3.3 The files pane

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

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

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

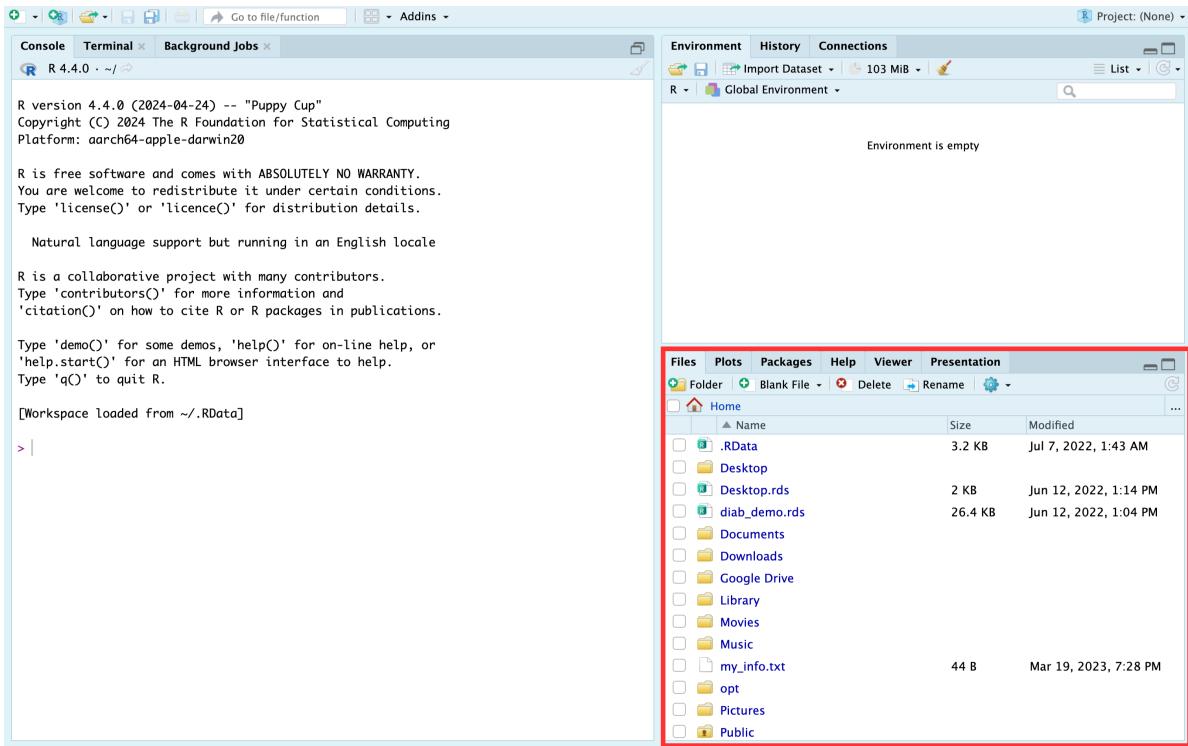


Figure 3.7: The Files/Plots/Packages/Help/Viewer pane.

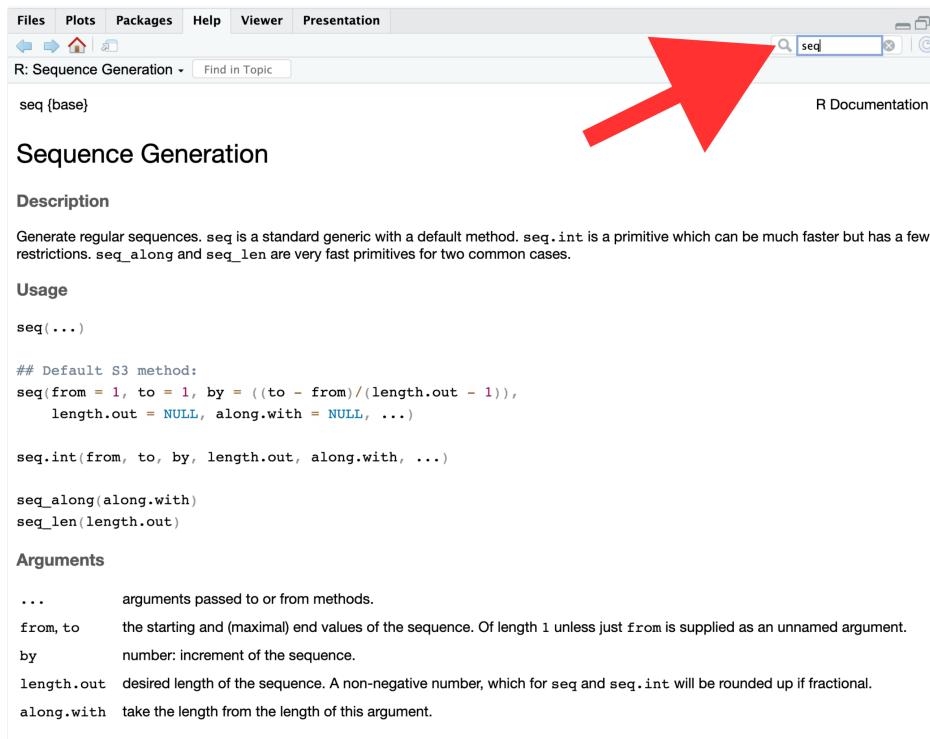


Figure 3.8: The help tab.

3.4 The source pane

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

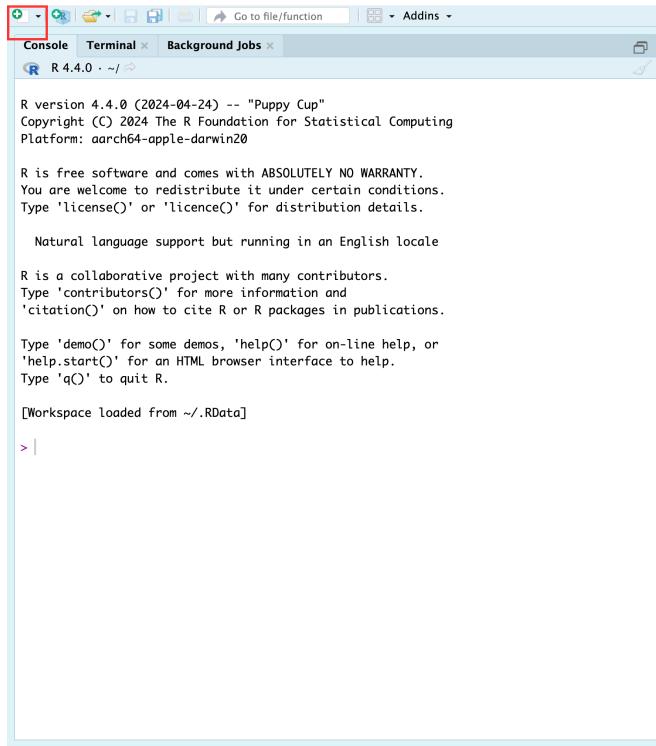


Figure 3.9: Click the new source file icon.

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

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

3.5 RStudio preferences

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

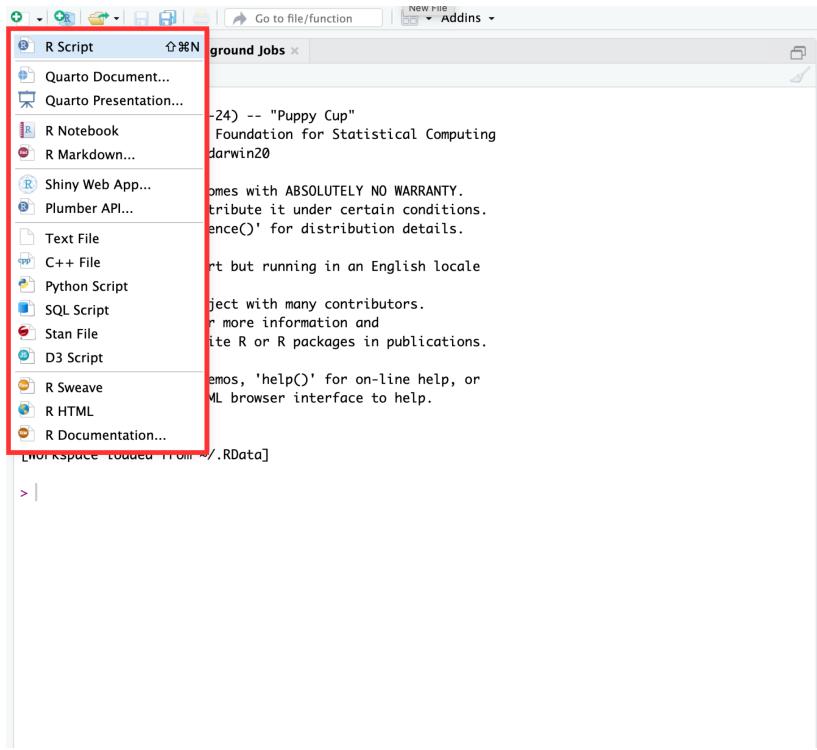


Figure 3.10: New source file options.

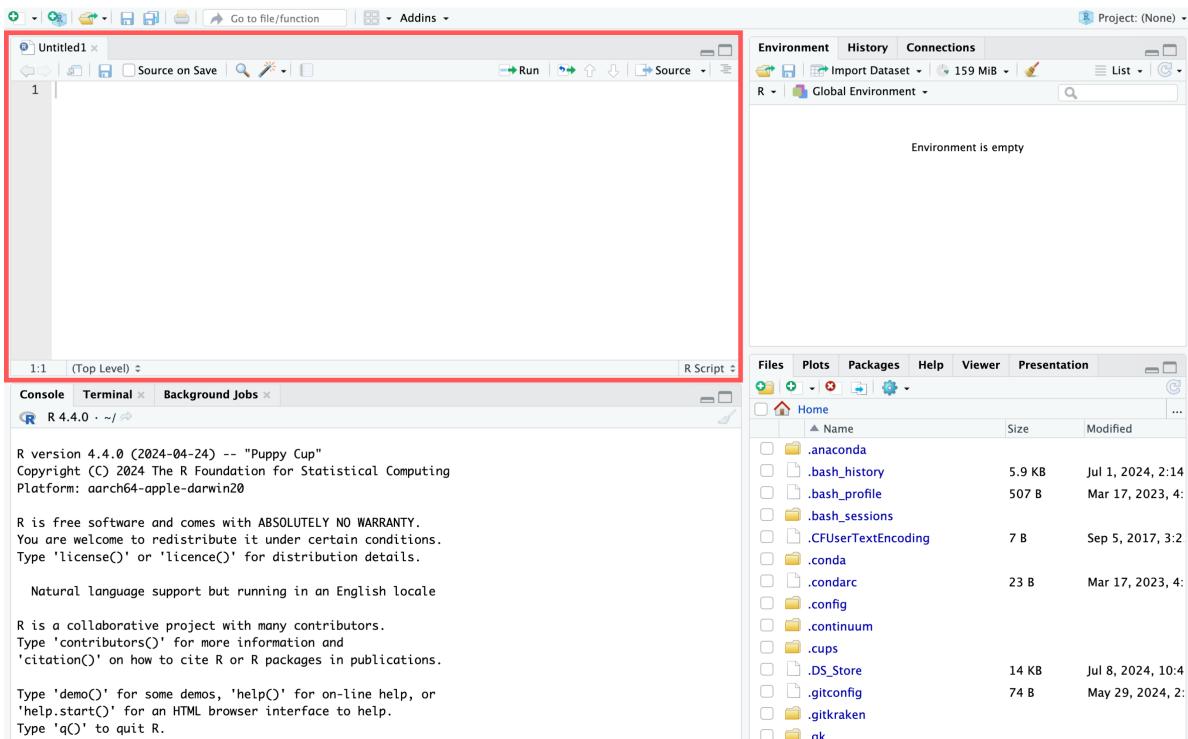


Figure 3.11: A blank R script in the source pane.

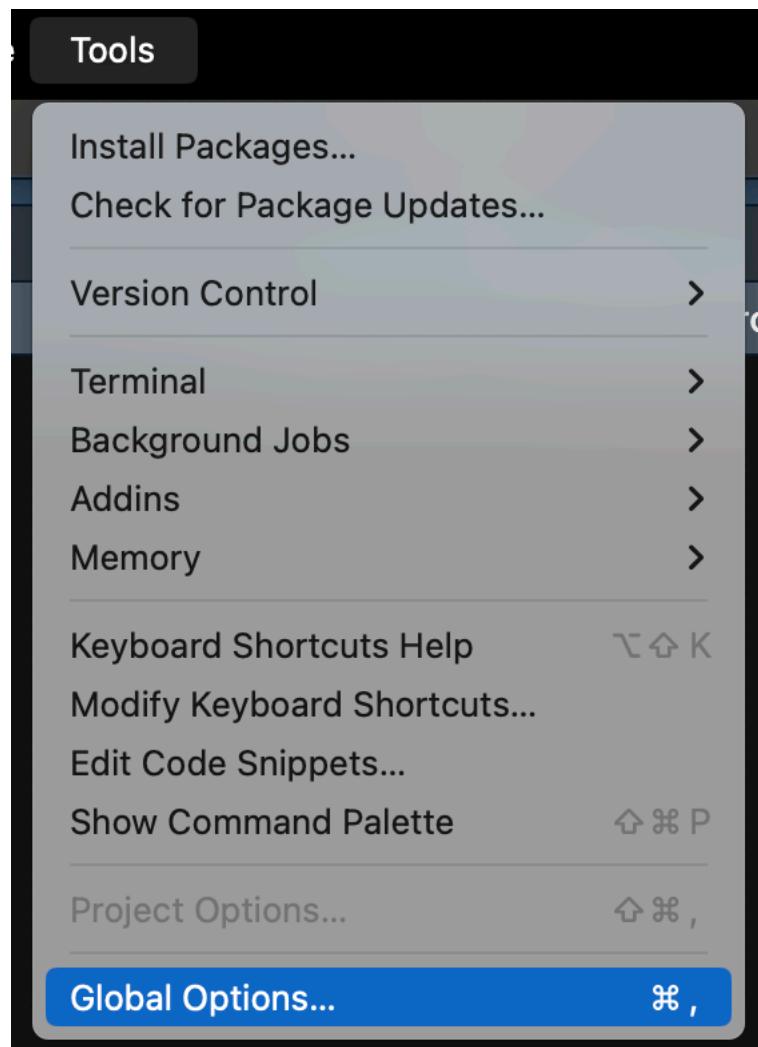


Figure 3.12: Select the preferences menu on Mac.

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

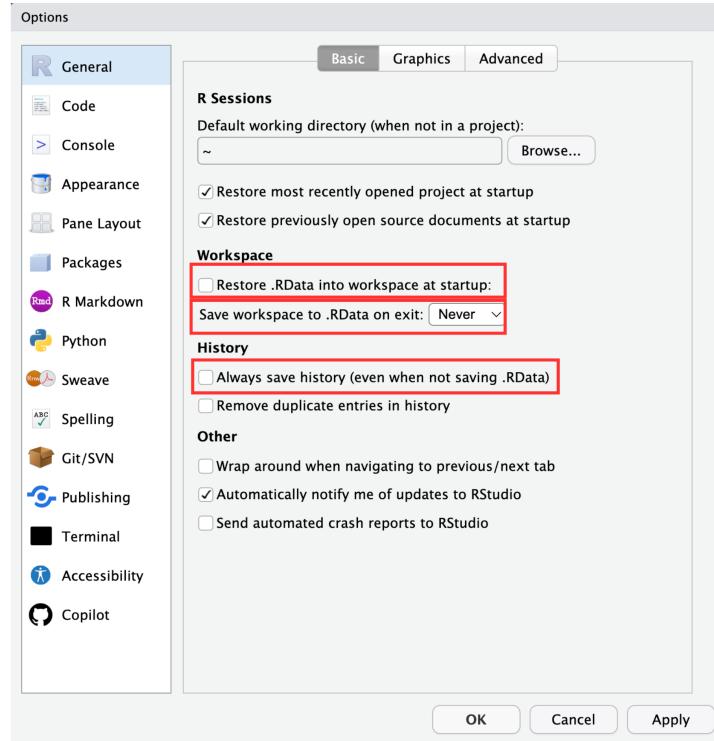


Figure 3.13: General options tab.

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

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

1. [RStudio IDE cheatsheet](#)
2. [ModernDive: What are R and RStudio?](#)

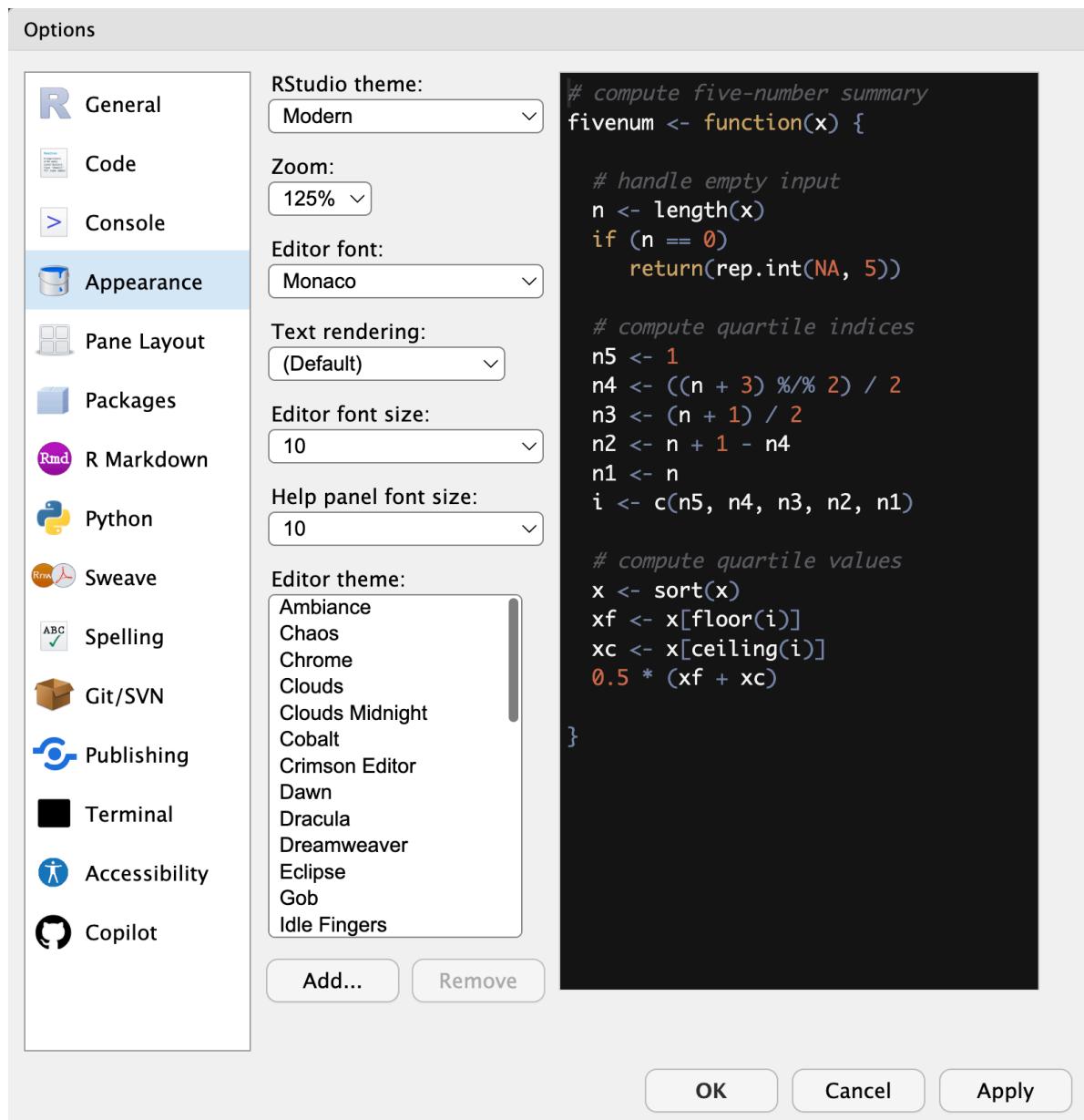


Figure 3.14: Appearance tab.

4 Speaking R's Language

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

4.1 R is a *language*

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

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

```
[1] 2
```

and

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

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

Note

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

4.2 The R interpreter

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

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

4.3 Errors

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

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

```
x <- 2
X
```

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

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

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

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

4.4 Functions

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

Functions = Factories



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.

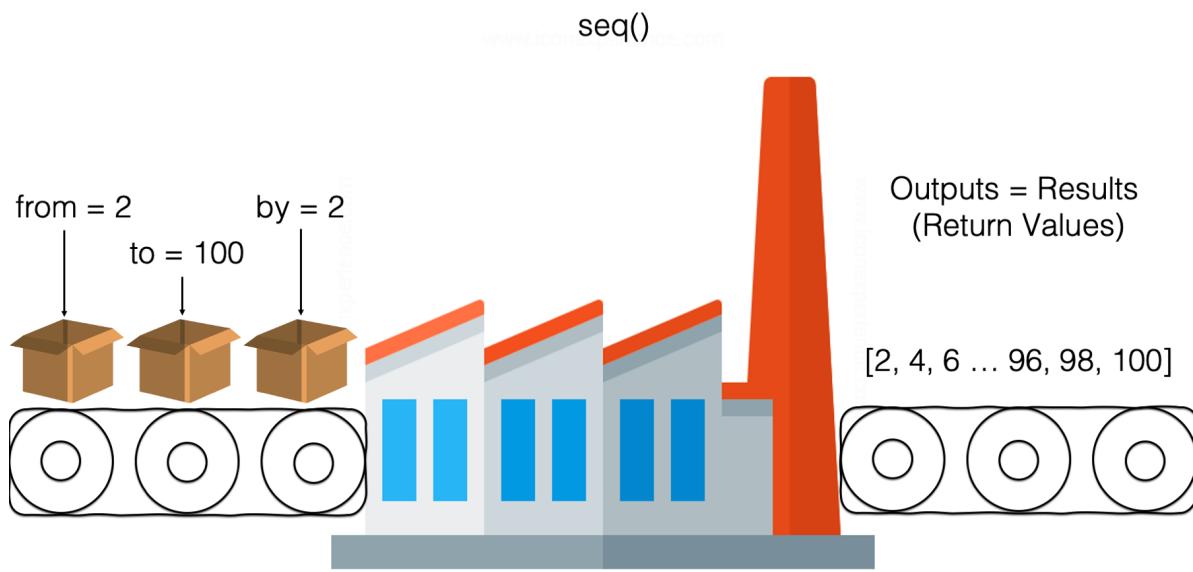


Figure 4.2: A function factory making numbers.

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.

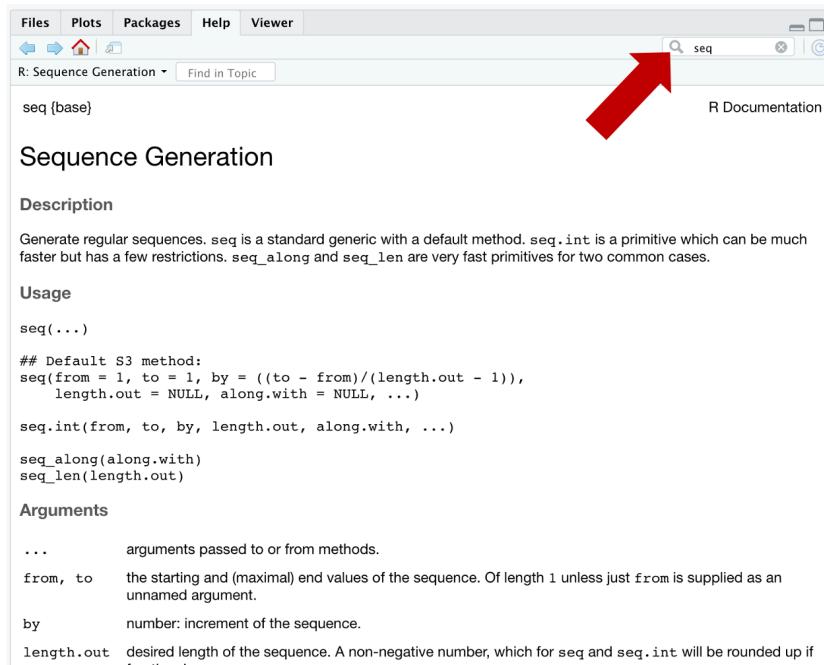


Figure 4.3: The help tab.

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.

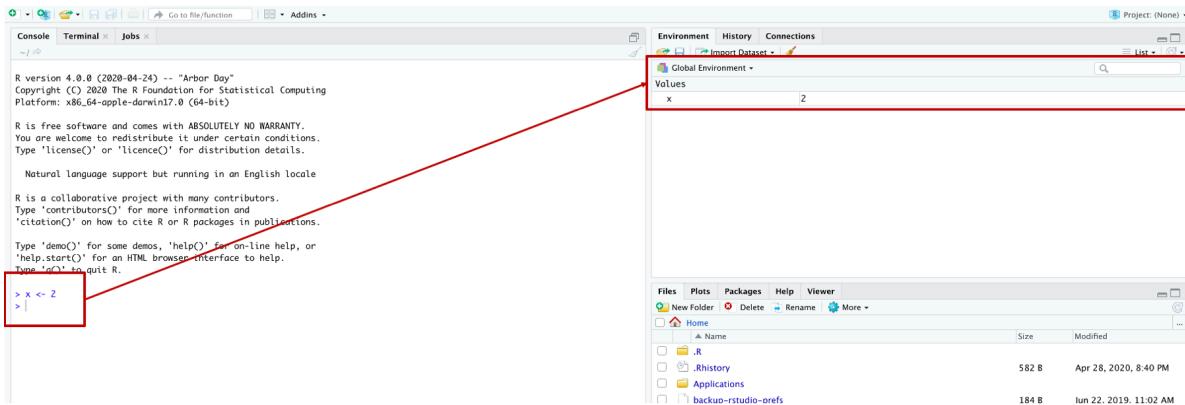


Figure 4.4: Creating the `x` object.

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.

John is Funny



X ← 2



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!

i 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.

i 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.

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

```
class
```

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

```
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.

```
# 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 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

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:

```
# 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	

```
# 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:

```
# 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	john	24
2	alexis	alexis	44
3	Steph	Steph	26
4	Quiera	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:

```
# 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

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:

```
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:

```
typeof(NA_character_)
```

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

```
typeof(NA_integer_)
```

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

```
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 accidentally press the wrong key. Luckily, we already have the heights stored as a column in the `class` data frame. We can *access* or *refer to* a single column in a data frame using the **dollar sign notation**.

5.6.2 Dollar sign notation

```
class$heights
```

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

Here's what we did above:

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

5.6.3 Bracket notation

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

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

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

```
[1] 71
```

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

```
# First way to calculate the mean  
# (68 + 63 + 71 + 72) / 4  
  
# Second way. Use dollar sign notation and bracket notation so that we don't  
# have to type individual heights  
(class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4
```

```
[1] 68.5
```

5.6.4 The `sum` function

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

```
# Create the heights vector  
heights <- c(68, 63, 71, 72)  
  
# Add together all the individual heights with the sum function  
sum(heights)
```

```
[1] 274
```

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

```
# First way to calculate the mean  
# (68 + 63 + 71 + 72) / 4  
  
# Second way. Use dollar sign notation and bracket notation so that we don't  
# have to type individual heights  
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4
```

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

```
[1] 68.5
```

Here's what we did above:

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

5.6.5 Nesting functions

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

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

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

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

```
[1] 2
```

Here's what we did above:

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

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

```
8 / 4
```

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

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

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

```
[1] 68.5
```

Here's what we did above:

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

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

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

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

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

5.6.6 The length function

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

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

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

[1] 4

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

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

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

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

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

```
[1] 68.5
```

Here's what we did above:

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

5.6.7 The mean function

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

```
# First way to calculate the mean  
# (68 + 63 + 71 + 72) / 4  
  
# Second way. Use dollar sign notation and bracket notation so that we don't  
# have to type individual heights  
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4  
  
# Third way. Use dollar sign notation and sum function so that we don't have  
# to type as much  
# sum(class$heights) / 4  
  
# Fourth way. Use dollar sign notation with the sum function and the length  
# function  
# sum(class$heights) / length(class$heights)
```

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

```
[1] 68.5
```

Congratulations again! You completed your first analysis using R!

5.7 Some common errors

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

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

Here's what we did above:

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

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

```
mean(weight)
```

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

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

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

```
[1] 175
```

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

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

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

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

```
mean(scores)
```

```
[1] 5.666667
```

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

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

```
[1] 96.25
```

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

5.8 Summary

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

6 Asking Questions

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

An example

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

6.1 When should we seek help?

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

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

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

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

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

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

6.2 Where should we seek help?

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

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

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

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

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

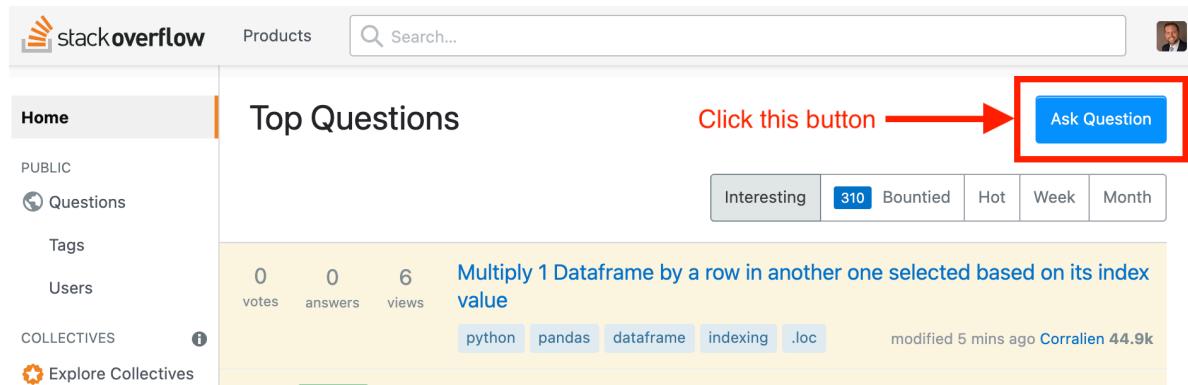
6.3 How should we seek help?

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

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

6.3.1 Creating a post on Stack Overflow

The first thing you need to do is navigate to the [Stack Overflow website](https://stackoverflow.com/). The homepage will look something like the screenshot below.



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

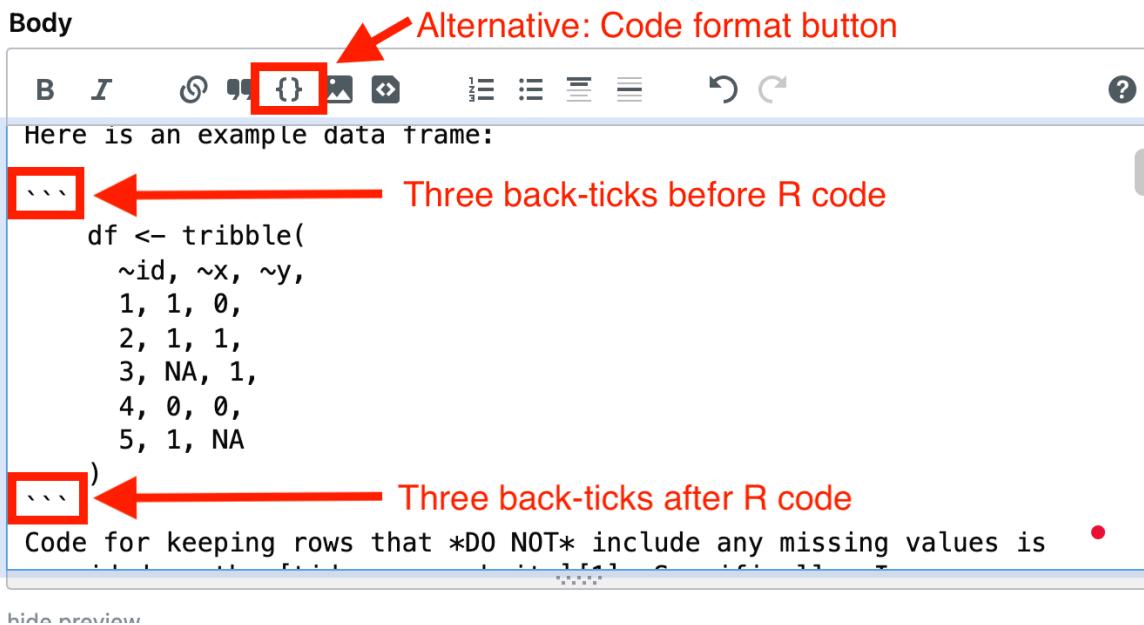
The screenshot shows a web-based form titled "Ask a public question". At the top right is a cartoon robot icon with speech bubbles. The form has several sections:

- Title**: A text input field with placeholder text "Be specific and imagine you're asking a question to another person" and an example "e.g. Is there an R function for finding the index of an element in a vector?".
- Body**: A rich-text editor area with a toolbar containing icons for bold, italic, code, etc. Below the toolbar is a menu bar with links like "Links", "Images", "Styling/Headers", "Lists", "Blockquotes", "Code", "HTML", "Tables", and "More".
- Tags**: A text input field with placeholder text "Add up to 5 tags to describe what your question is about" and an example "e.g. (python asp.net iphone)".
- Checklist**: A checkbox labeled "Answer your own question – share your knowledge, Q&A-style".
- Review button**: A blue button at the bottom-left labeled "Review your question".

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

6.3.1.1 Inserting R code

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



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

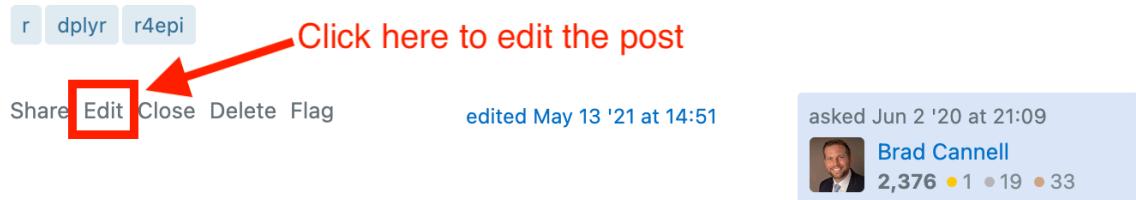
6.3.1.2 Reviewing the post

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

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

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

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



6.3.2 Creating better posts and asking better questions

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

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

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

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

6.3.2.1 Example posts

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

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

6.3.2.2 Question title

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

6.3.2.3 Explanation of the issue

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

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

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

6.3.2.4 Reproducible example

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

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

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

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

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

```
data()
```

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

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

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

6.4 Helping others

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

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

6.5 Summary

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

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

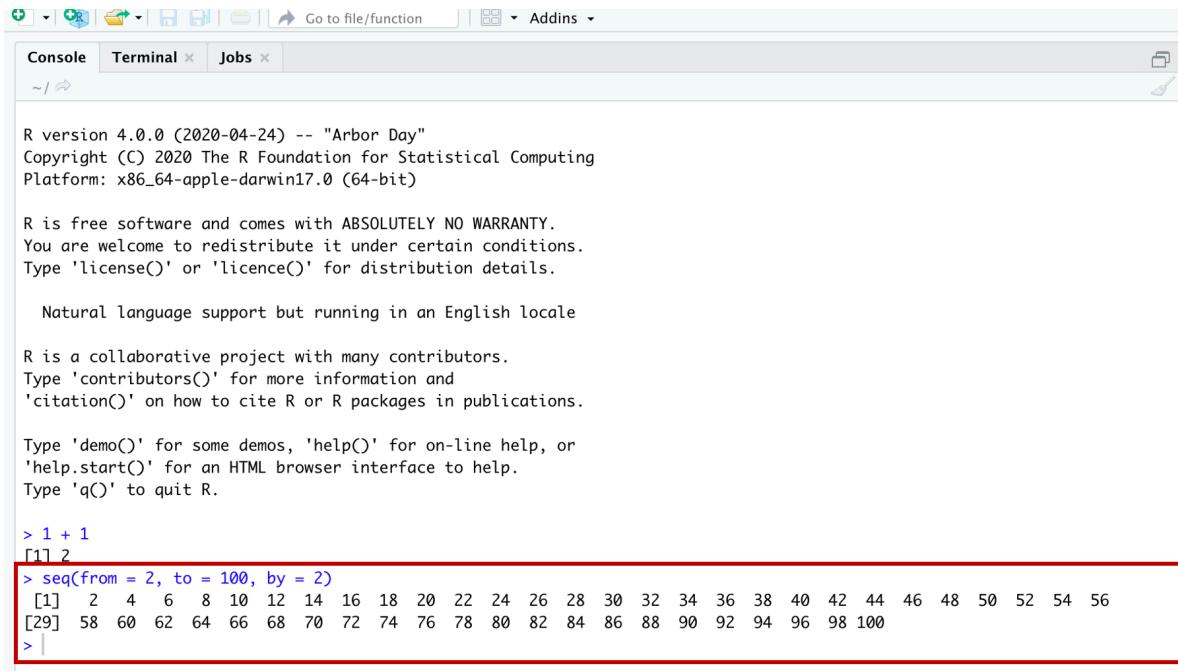
- Write a brief, yet clear, explanation of what you are trying to accomplish. Add what you've already tried, what result you are getting, and what result you are expecting.
 - Try to always include a reproducible example of the problem you are encountering in the form of data.
5. Be patient, kind, and genuine when posting or responding to posts.

Part II

Coding Tools and Best Practices

7 R Scripts

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



The screenshot shows the RStudio interface with the 'Console' tab selected. The R console window displays the standard R startup message, followed by a sequence of numbers generated by the `seq()` function. The output of the `seq()` command is highlighted with a red rectangle.

```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

Natural language support but running in an English locale

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

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

> 1 + 1
[1] 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 40 42 44 46 48 50 52 54 56
[29] 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100
> |
```

Figure 7.1: Submitting R code in the console.

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

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

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

```

Source
Console Terminal < R Markdown < Jobs <
~/Dropbox/Teaching/Courses/Introduction to R Programming for Epidemiologic Research/R4Epi/ ↗

R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

Natural language support but running in an English locale

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

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

> 1 + 1
[1] 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 40 42 44 46 48 50 52 54 56 58 60 62 64
[33] 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100
> # Here is a comment
>

```

Figure 7.2: Console commands copied to the History pane.

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

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

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

1. They are simpler than Quarto files, so they are a good place to start.
2. Some of what we discuss below will also apply to Quarto files.
3. R scripts *are* a better choice than Quarto files in some situations (e.g., writing R packages, creating Shiny apps).
4. Some people just prefer using R scripts.

```

1 # =====
2 # Example R Script
3 # Brad Cannell
4 # <Date>
5 # =====
6
7 # Load packages
8 library(dplyr)
9
10 # Load data
11 data("mtcars")
12
13 # I'm not sure what's in the mtcars data. I'm printing it below to take a look
14 mtcars
15
16 ## Data analysis
17 # -----
18
19 # Below, we calculate the average mpg across all cars in the mtcars data frame.
20 mean(mtcars$mpg)
21
22 # Here, we also plot mpg against displacement.
23 plot(mtcars$mpg, mtcars$disp)

```

The screenshot shows an R script titled "example_script.R" in a code editor. The code itself is as follows:

```

1 # =====
2 # Example R Script
3 # Brad Cannell
4 # <Date>
5 # =====
6
7 # Load packages
8 library(dplyr)
9
10 # Load data
11 data("mtcars")
12
13 # I'm not sure what's in the mtcars data. I'm printing it below to take a look
14 mtcars
15
16 ## Data analysis
17 # -----
18
19 # Below, we calculate the average mpg across all cars in the mtcars data frame.
20 mean(mtcars$mpg)
21
22 # Here, we also plot mpg against displacement.
23 plot(mtcars$mpg, mtcars$disp)

```

Annotations with red callouts explain specific parts of the code:

- A callout labeled "Header" points to the first five lines of the script.
- A callout labeled "Load packages at the top of the script" points to the line `library(dplyr)`.
- A callout labeled "Decorate with comments" points to the lines `# =====` and `# -----`.
- A callout labeled "80 characters per line" points to the line `mean(mtcars\$mpg)`.
- A callout labeled "Decorate with comments" points to the line `# Below, we calculate the average mpg across all cars in the mtcars data frame.`.

Figure 7.3: Example R script.

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

[Click here to download the R script](#)

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

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

1. A brief description of what the R script does.
2. The author(s) who wrote the R script.
3. Important dates. For example, the date it was originally created and the date it was last modified. You can usually get these dates from your computer's operating system, but they aren't always accurate.

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

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

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

```
# Load packages
library(dplyr)

# Load data
data("mtcars")

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

## Data analysis
# ----

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

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

than this...

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

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

A screenshot of the RStudio interface showing an R script named "example_script.R". The code editor pane displays the following R code:

```
1 * # Example R Script
2 # Brad Cannell
3 # <Date>
4 #
5 #
6
7 # Load packages
8 library(dplyr)
9
10 # Load data
11 data("mtcars")
12
13 # I'm not sure what's in the mtcars data. I'm printing it below to take a look
14 mtcars
15
16 ## Data analysis
17 #
18
19 # Below, we calculate the average mpg across all cars in the mtcars data frame.
20 mean(mtcars$mpg)
21
22 # More, we also plot mpg against displacement.
23 plot(mtcars$mpg, mtcars$disp)
```

A red arrow points from the bottom left towards the line number 20, indicating the cursor's position. The status bar at the bottom shows "20:3" and "(Untitled) :".

Figure 7.4: Cursor location.

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

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

7.1 Creating R scripts

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

Click the very first option – R Script.

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

A screenshot of the RStudio interface. The top bar shows tabs for 'Console', 'Global', and 'Jobs'. The main area displays the R startup message:

```
R version 4.0.2 (2020-06-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

Natural language support but running in an English locale

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

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

The left sidebar shows a single item: 'Courses/Introduction to R Programming for Epidemiologic Research/R4Epi/'. A red arrow points to the new source file icon (a plus sign inside a square) in the top-left corner of the toolbar.

Figure 7.5: Click the new source file icon.

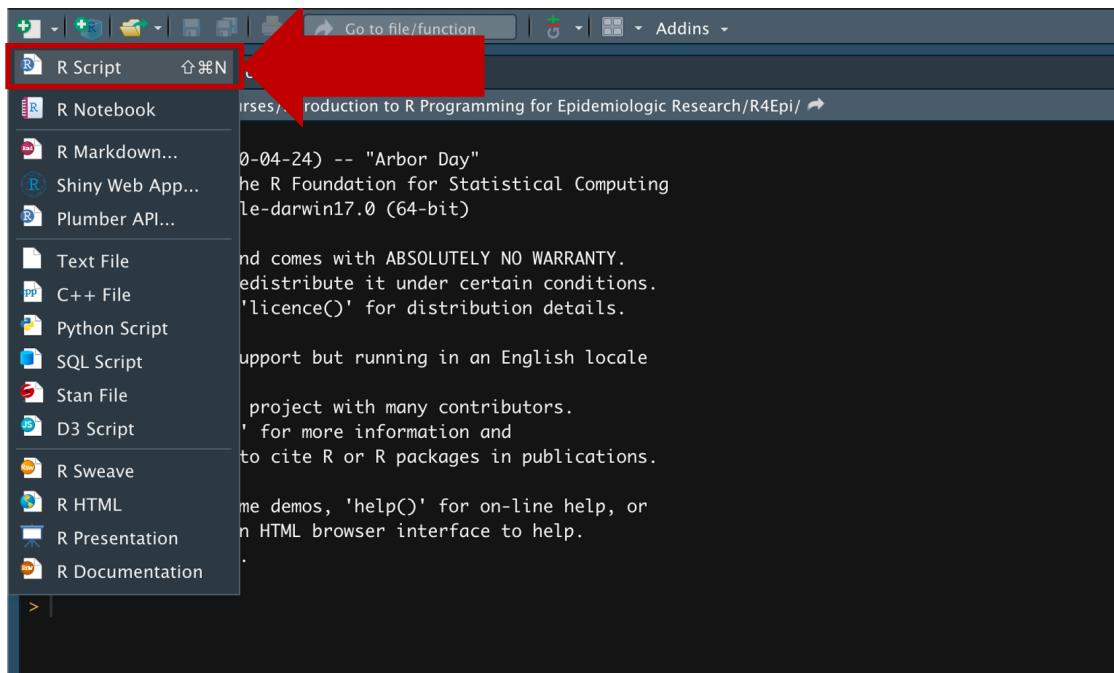


Figure 7.6: New source file options.

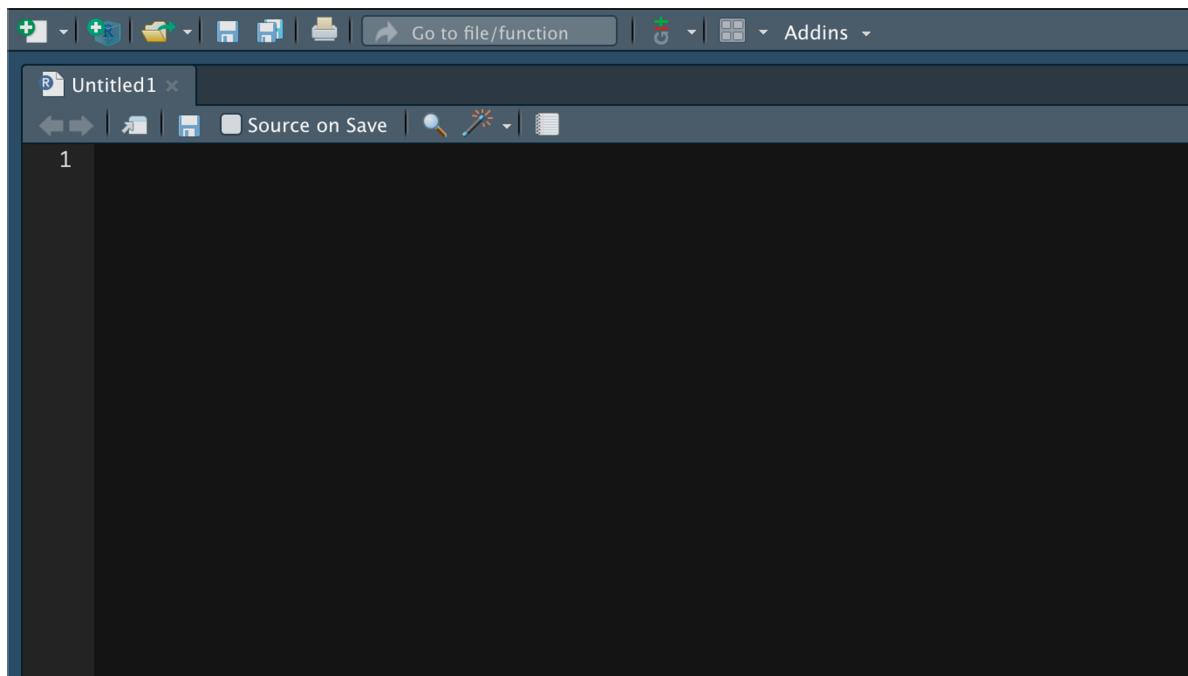


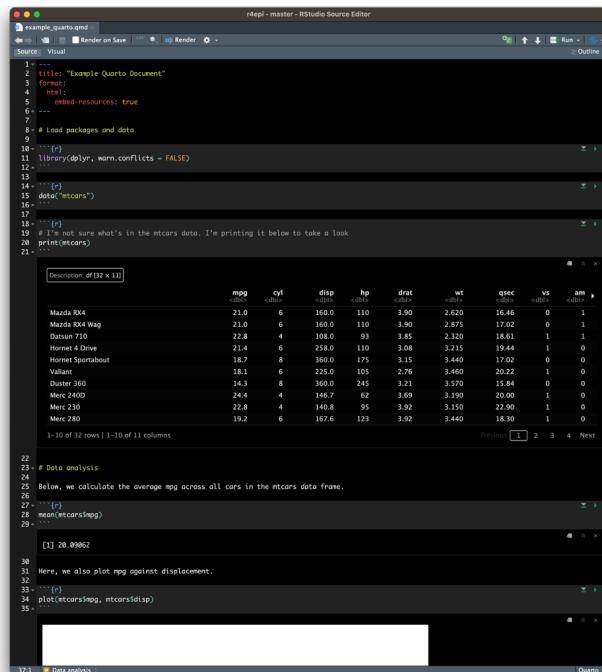
Figure 7.7: A blank R script in the source pane.

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

8 Quarto Files

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

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



```
example_quarto.qmd
Source: Visual
1: #title: "Example Quarto Document"
2: #format:
3: #html:
4: #united-resources: true
5: ---
6: # Load packages and data
7:
8: #> library(dplyr, warn.conflicts = FALSE)
9: #>
10: #> mtcars
11: #>
12: #> (t)
13: #> I'm not sure what's in the mtcars data. I'm printing it below to take a look
14: #> print(mtcars)
15: #>
16: #>
17: #>
18: #> (t)
19: #> [1] 20.09062
20: #> Here, we also plot mpg against displacement.
21: #>
22: # Data analysis
23: # Below, we calculate the average mpg across all cars in the mtcars data frame.
24: #>
25: #> mtcars$mpg
26: #>
27: #> (t)
28: #> mtcars$mpg
29: #>
30: #> [1] 20.09062
31: #> Here, we also plot mpg against displacement.
32: #>
33: #> (t)
34: #> plot(mtcars$mpg, mtcars$displ)
35: #>
```

Description: df [32 x 11]

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.0	6	160.0	110	3.90	2.620	16.46	0	1		
21.0	6	160.0	110	3.90	2.875	17.02	0	1		
22.8	4	108.0	93	3.85	2.320	18.61	1	1		
21.4	6	258.0	110	3.08	3.215	19.44	1	0		
18.7	8	360.0	175	3.73	3.440	17.02	0	0		
18.1	6	225.0	105	2.76	3.460	20.22	1	0		
14.3	8	360.0	245	3.21	3.570	15.84	0	0		
24.4	4	146.7	62	3.69	3.190	20.00	1	0		
22.8	4	140.8	95	3.92	3.150	22.90	1	0		
19.2	6	167.6	123	3.92	3.440	18.30	1	0		

Figure 8.1: Example Quarto file.

[Click here to download the Quarto file](#)

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

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

Example Quarto Document

Load packages and data

```
library(dplyr, warn.conflicts = FALSE)
data("mtcars")

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

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.083	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	160	3.435	3.850	17.02	0	0	3	2
Valiant	14.3	8	225.0	105	2.710	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.210	3.570	15.84	0	0	3	4
Merc 2400	24.4	4	146.7	62	3.693	3.198	28.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.930	3.150	22.98	1	0	4	2
Merc 280	19.2	6	167.6	123	3.90	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.90	3.440	18.98	1	0	4	4
Merc 450SE	16.4	8	275.8	188	3.074	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	188	3.074	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	188	3.074	3.730	17.60	0	0	3	3
Cadillac Fleetwood	18.4	8	472.0	205	2.95	3.250	17.98	0	0	3	4
Lincoln Continental	18.4	8	460.0	215	3.800	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	238	3.230	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.280	19.47	1	1	4	1
Honda Civic	38.4	4	75.7	52	4.931	3.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	55	4.22	1.830	19.98	1	1	4	1
Toyota Corona	21.5	4	120.3	90	3.900	3.050	18.35	0	0	3	1
Dodge Challenger	15.5	8	318.0	158	2.76	3.520	16.97	0	0	3	2
AMC Javelin	15.2	8	304.0	158	3.130	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.773	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.600	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.938	18.98	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.42	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	3.151	16.98	1	1	5	2
Ford Mustang L	15.1	8	302.0	158	3.050	3.250	17.40	0	1	5	4
Ferrari Dino	19.7	8	145.0	95	3.600	2.770	15.58	0	0	5	6
Maserati Bora	15.0	8	301.0	235	3.543	3.570	14.68	0	1	5	0
Volvo 142E	21.4	4	121.0	189	4.112	2.780	18.68	1	1	4	2

Data analysis

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

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

[Click here to download the rendered HTML file.](#)

Notice how everything is nicely formatted and easy to read!

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

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

⚠ Warning

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

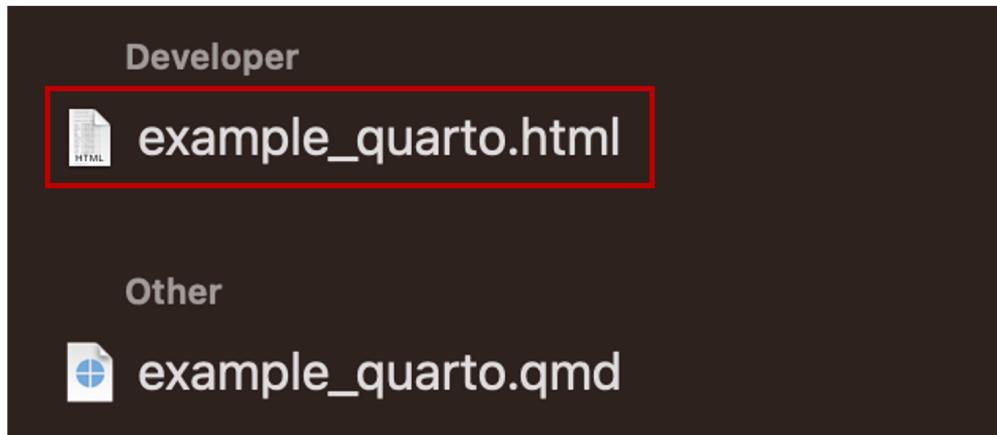


Figure 8.3: Quarto file and rendered HTML file and on MacOS.

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

8.1 What is Quarto?

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

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

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

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

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

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

8.2 Why use Quarto?

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

8.3 Create a Quarto file

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

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

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

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

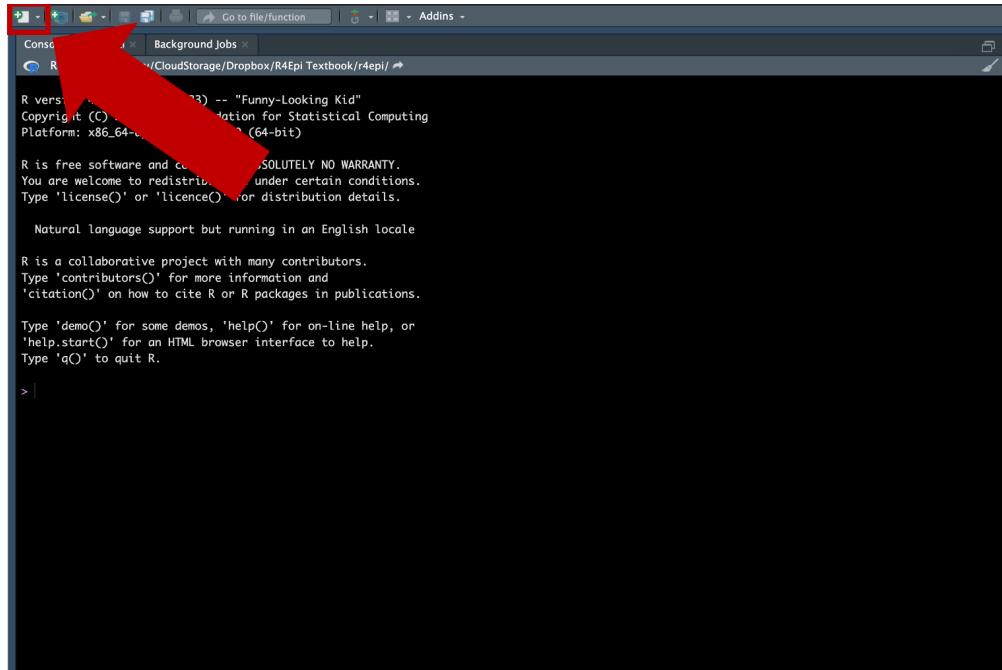


Figure 8.4: Click the new file icon.

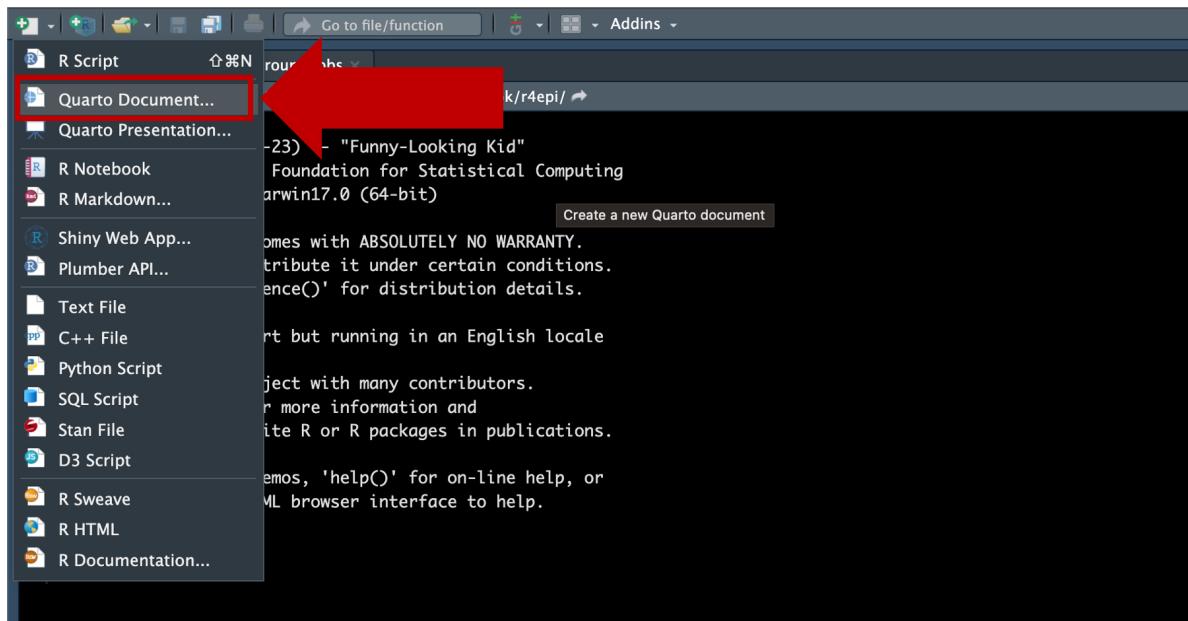


Figure 8.5: New source file options.

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

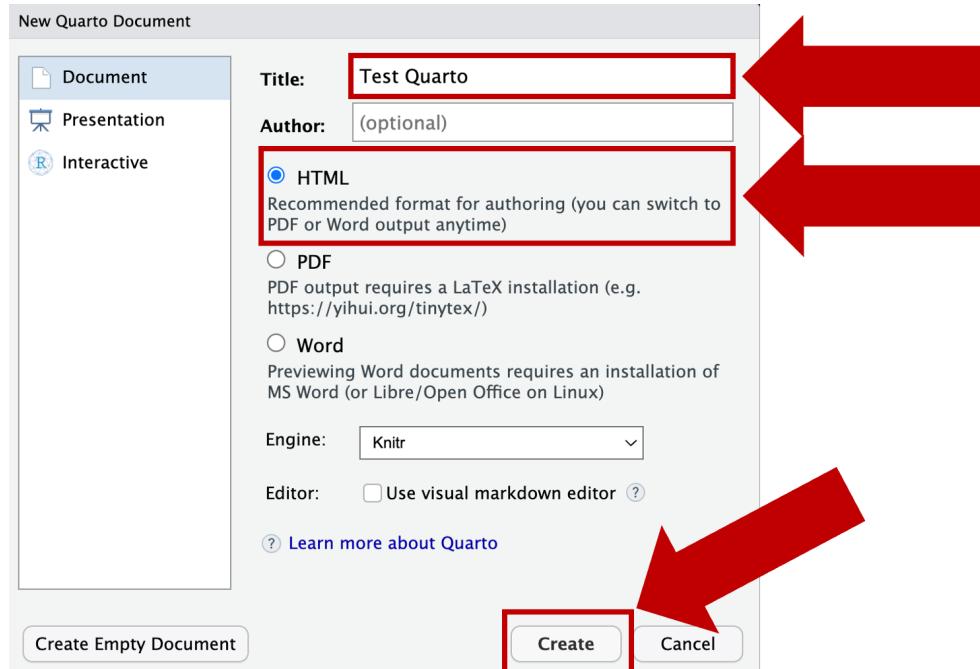


Figure 8.6: New Quarto document options.

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

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

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

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

The screenshot shows the RStudio interface with a document titled 'Untitled.qmd'. The code editor pane displays the following content:

```

1 ---  

2 title: "Test Quarto"  

3 format:  

4   html:  

5     embed-resources: true  

6 ---  

7  

8 ## Quarto  

9  

10 Quarto enables you to weave together content and executable code into a finished document. To learn more about Quarto see  

<https://quarto.org>.  

11  

12 ## Running Code  

13  

14 When you click the **Render** button a document will be generated that includes both content and the output of embedded code. You can embed  

code like this:  

15  

16 ````{r}  

17 1 + 1  

18 ````  

19  

20 You can add options to executable code like this  

21  

22 ````{r}  

23 #| echo: false  

24 2 * 2  

25 ````  

26  

27 The `echo: false` option disables the printing of code (only output is displayed).  

28

```

Annotations with red arrows and callouts point to specific parts of the code:

- YAML header**: Points to the first two lines of the file.
- Link to a website**: Points to the explanatory text starting with 'Quarto enables you to weave together content and executable code into a finished document...'. It includes a link to <https://quarto.org>.
- Heading (level 2)**: Points to the line `## Quarto`.
- Explanatory text**: Points to the explanatory text starting with 'When you click the **Render** button a document will be generated that includes both content and the output of embedded code...'. It includes a bolded section 'You can add options to executable code like this'.
- Formatting (Bold)**: Points to the bolded text '2 * 2' in the R code chunk.
- R code chunk**: Points to the line `````{r}`` and the code block it contains.
- Play button**: Points to the green triangle icon in the RStudio toolbar, which is used to run the code chunk.

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

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

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

8.4 YAML headers

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

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

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

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

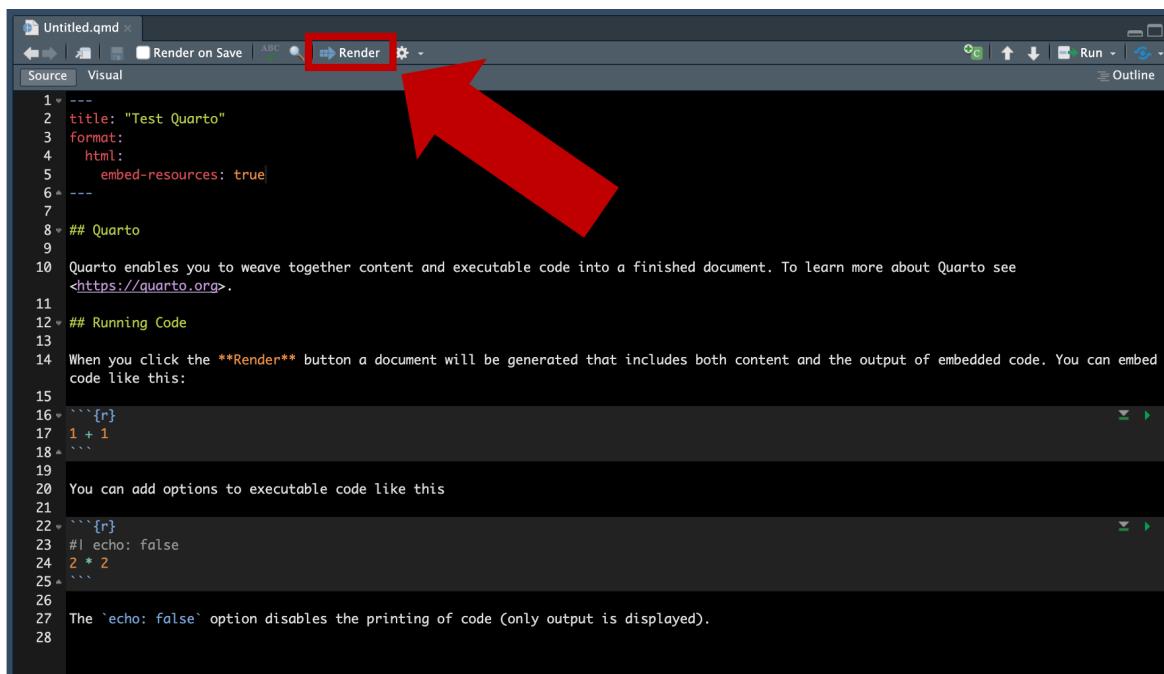


Figure 8.8: RStudio's render button. Only visible when a Quarto file is open.

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

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

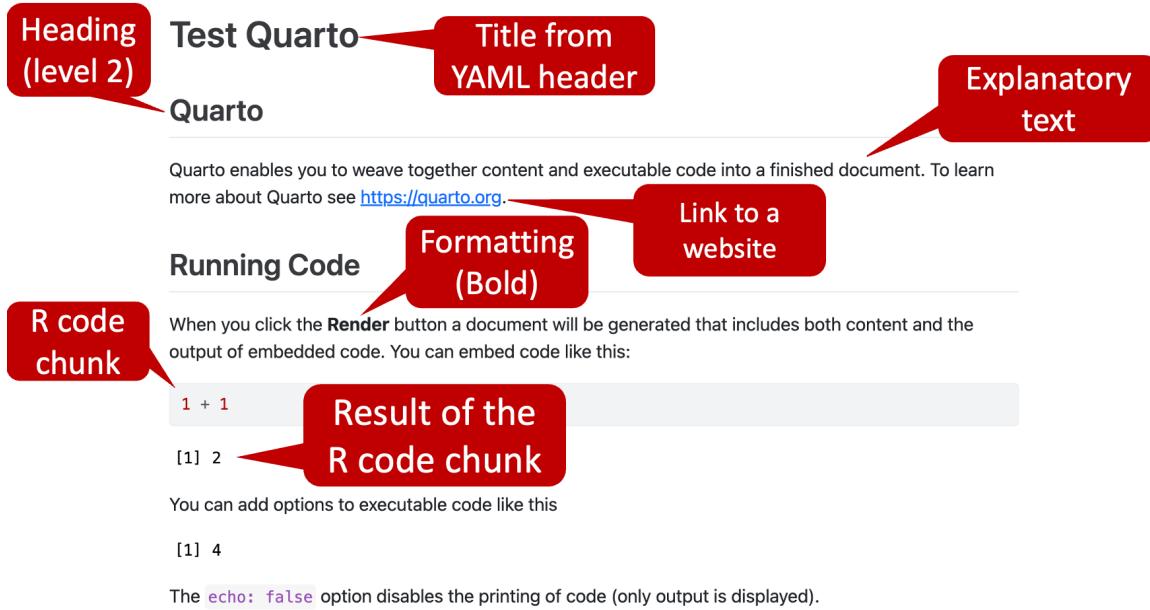


Figure 8.9: An HTML file created using a Quarto file.

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

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

8.5 R code chunks

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

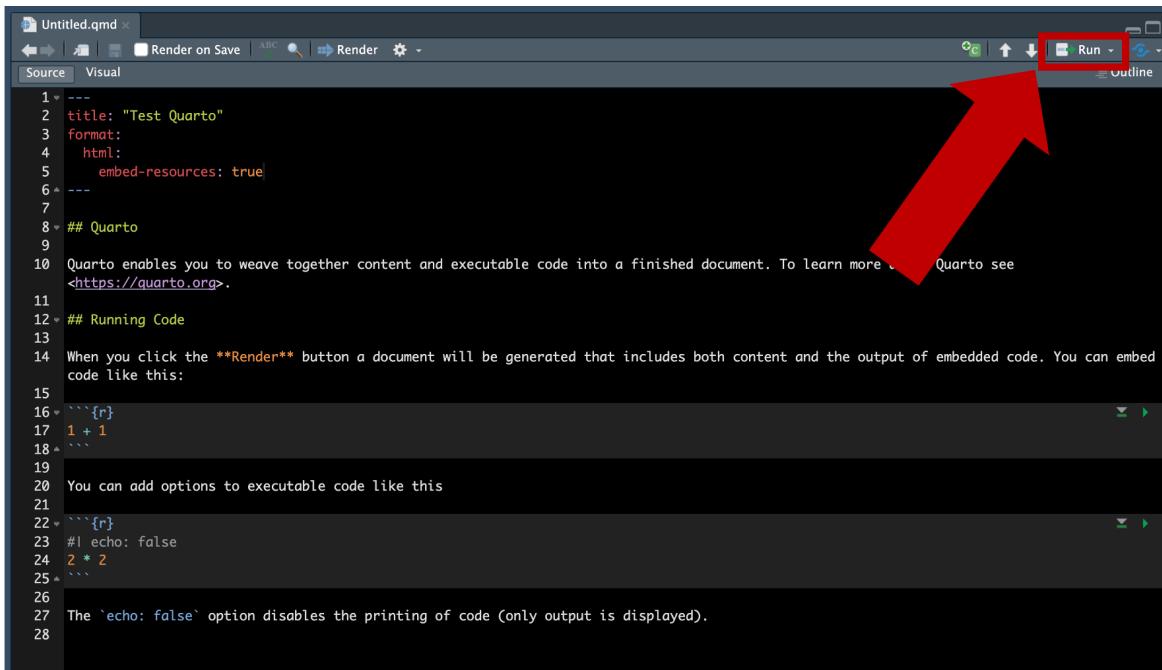
On MacOS type `option + command + i`.

On Windows type `control + alt + i`

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

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

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



A screenshot of the RStudio interface showing a Quarto file named "Untitled.qmd". The code editor displays the following content:

```
1 ---  
2 title: "Test Quarto"  
3 format:  
4   html:  
5     embed-resources: true  
6 ---  
7  
8 ## Quarto  
9  
10 Quarto enables you to weave together content and executable code into a finished document. To learn more about Quarto see  
<https://quarto.org>.  
11  
12 ## Running Code  
13  
14 When you click the **Render** button a document will be generated that includes both content and the output of embedded code. You can embed  
code like this:  
15  
16 ``{r}  
17 1 + 1  
18 ``  
19  
20 You can add options to executable code like this  
21  
22 ``{r}  
23 #| echo: false  
24 2 * 2  
25 ``  
26  
27 The `echo: false` option disables the printing of code (only output is displayed).  
28
```

A large red arrow points to the "Run" button in the top right corner of the RStudio window.

Figure 8.10: The run button in RStudio.

8.6 Markdown

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

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

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

8.6.1 Markdown headings

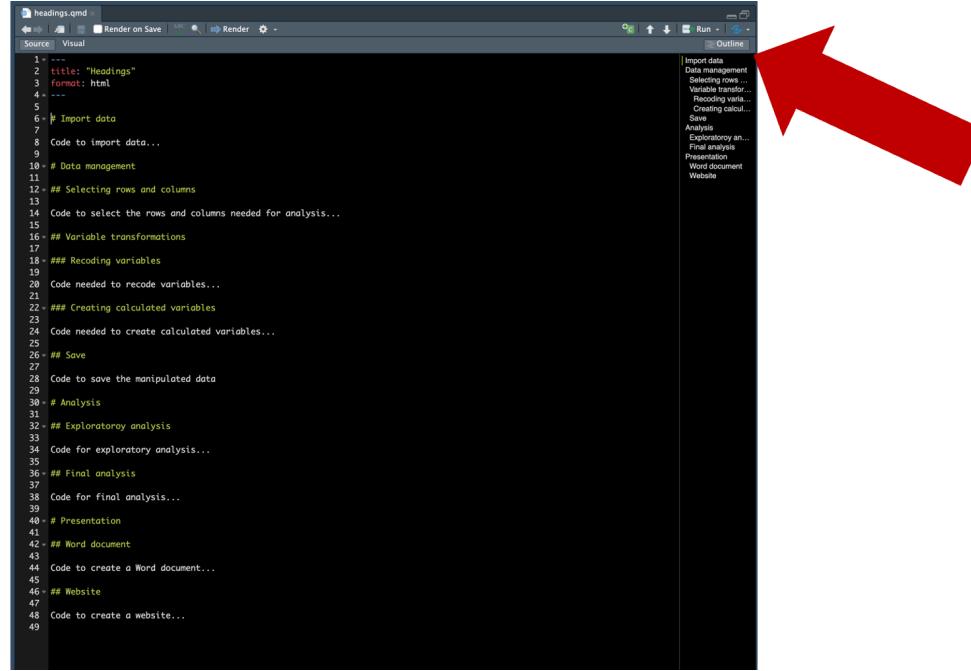
While we are discussing markdown, we would like to call special attention to markdown headings. We briefly glazed over them above, but we find that beginning R users typically benefit from a slightly more detailed discussion. Think back to the `## Quarto` on line 8 of Figure 8.7. This markdown created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

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

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

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

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

A screenshot of the RStudio interface showing a Quarto file named "headings.qmd". The main editor window displays the following R code with various sections and comments:

```
1 ---  
2 title: "Headings"  
3 format: html  
4 ---  
5 # Import data  
6 # Data management  
7 # Selecting rows and columns  
8 # Code to select the rows and columns needed for analysis...  
9 # Variable transformations  
10 # Recoding variables  
11 # Code needed to recode variables...  
12 # Creating calculated variables  
13 # Code needed to create calculated variables...  
14 # Save  
15 # Code to save the manipulated data  
16 # Analysis  
17 # Exploratory analysis  
18 # Code for exploratory analysis...  
19 # Final analysis  
20 # Code for final analysis...  
21 # Presentation  
22 # Word document  
23 # Code to create a Word document...  
24 # Website  
25 # Code to create a website...
```

A red arrow points from the text "outline" in the caption to the "Outline" tab in the top right corner of the RStudio interface, which is currently selected. The "Outline" tab is part of the "Source" tab bar.

Figure 8.11: A Quarto file with nested headings.

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

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

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

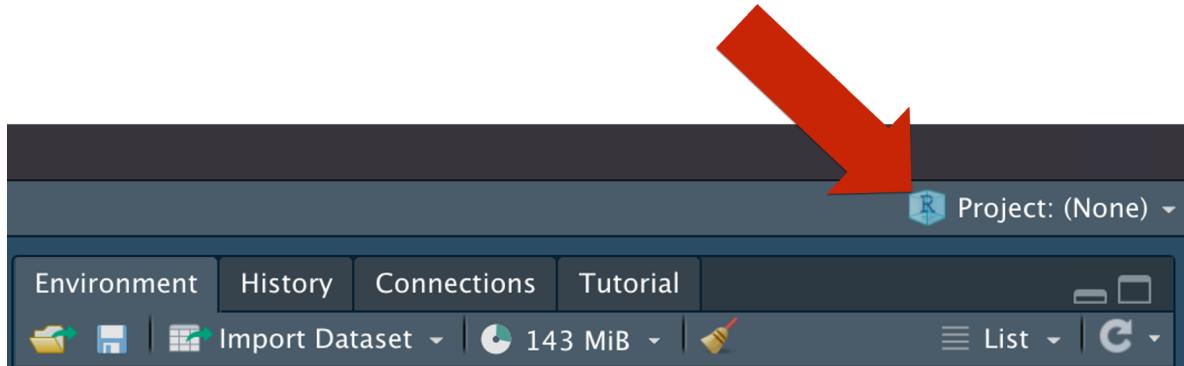
8.7 Summary

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

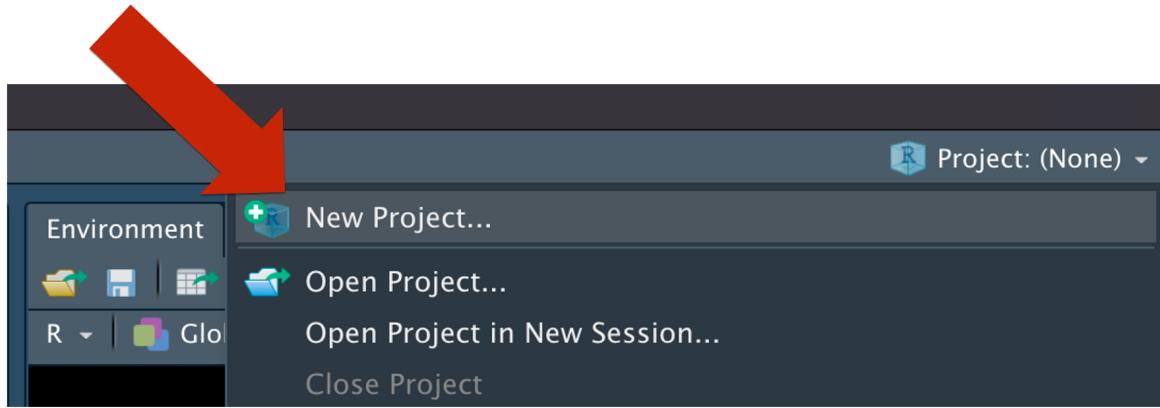
9 R Projects

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

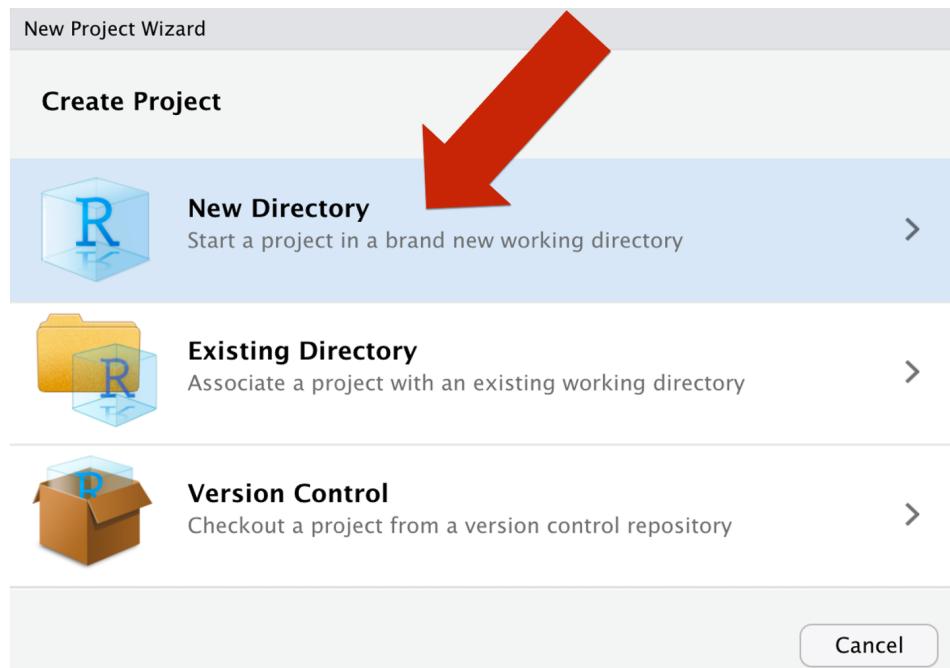
RStudio makes creating R projects really simple. For starters, let's take a look at the top right corner of our RStudio application window. Currently, we see an R project icon that looks like little blue 3-dimensional box with an "R" in the middle. To the right of the R project icon, we see words **Project: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



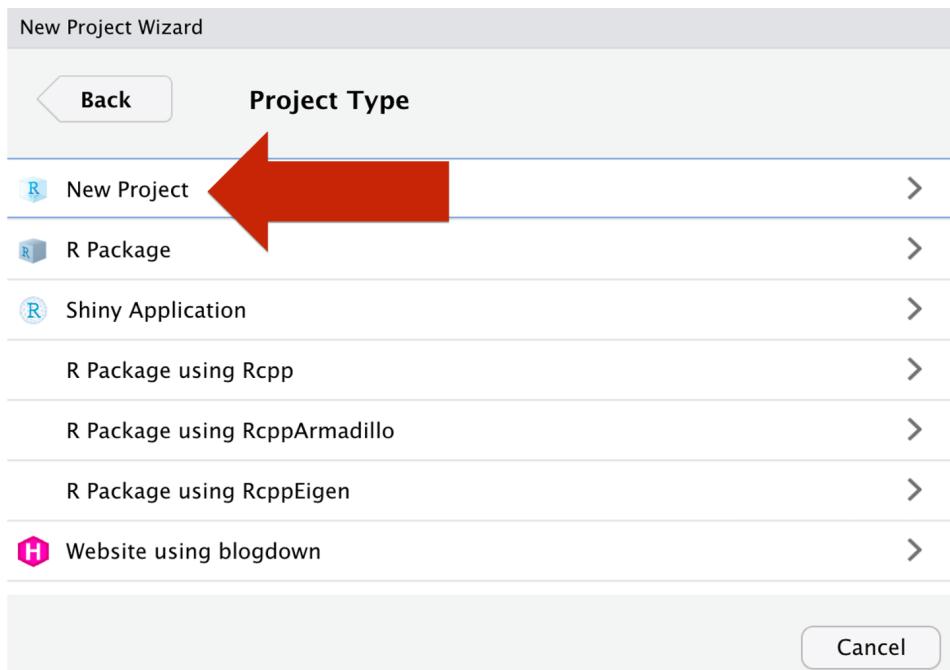
To create a new R project, we just need to click the drop-down arrow next to the words Project: (None) to open the projects menu. Then, we will click the New Project... option.



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



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

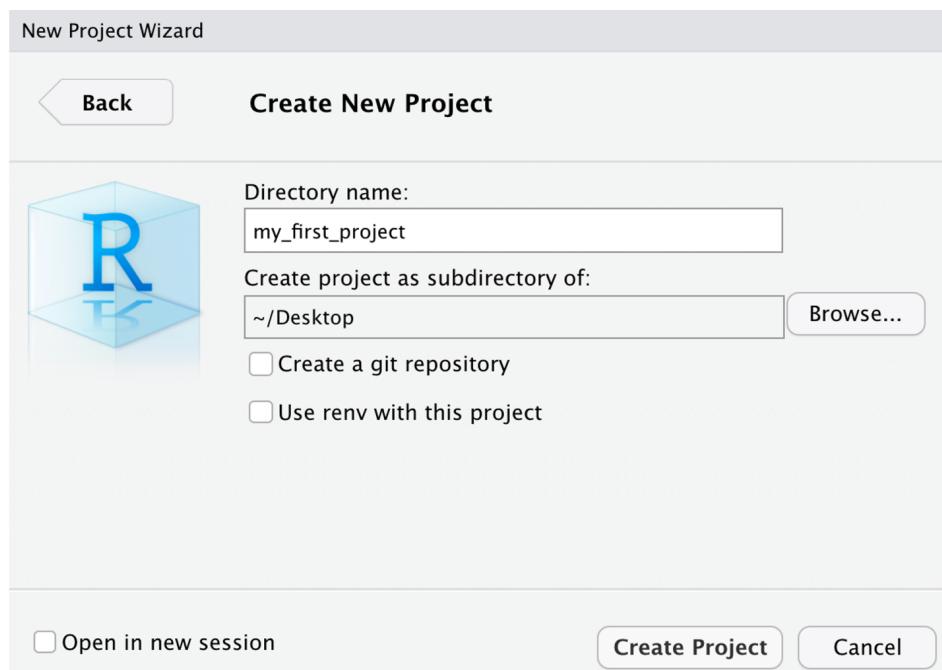


In the next window, we will have to make some choices and enter some information. The first thing we will have to do is name our project. We do so by entering a value in the **Directory name:** box. Often, we can name our R project directory to match the name of the larger project

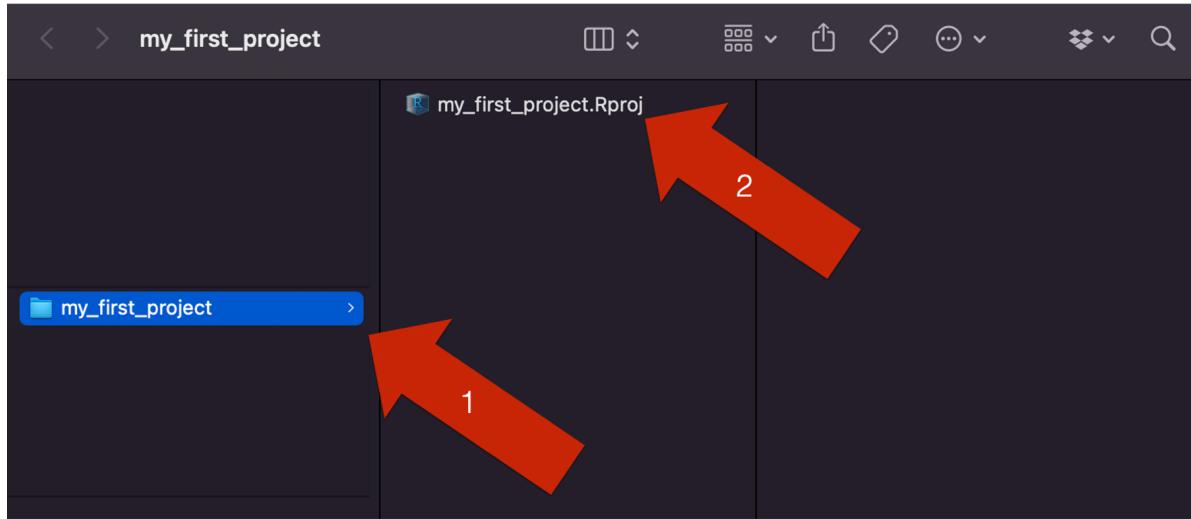
we are working on in a pretty natural way. If not, the name we choose for our project directory should essentially follow the same guidelines that we use for [object \(variable\) names](#), which we will learn about soon. In this example, we went with the very creative `my_first_project` project name.

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

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



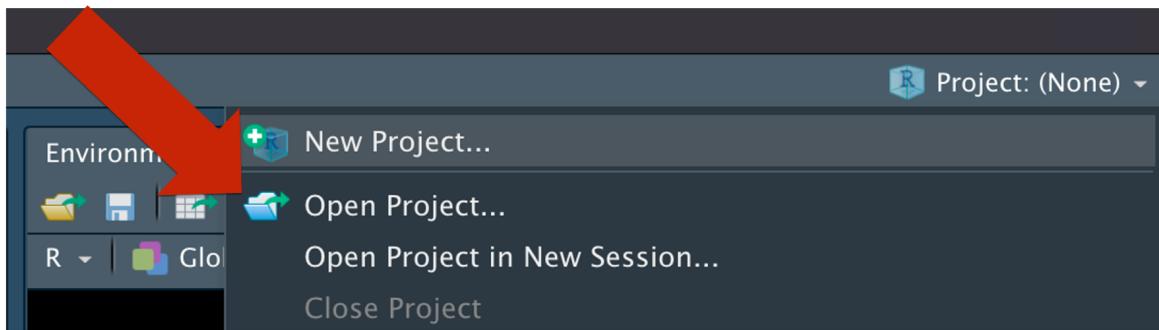
Doing so will create our new R project in the location we selected in the `Create project as subdirectory of:` text box in the new project wizard. In the screenshot below, we can see that a folder was created on our computer's desktop called `my_first_project`. Additionally, there is one file inside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



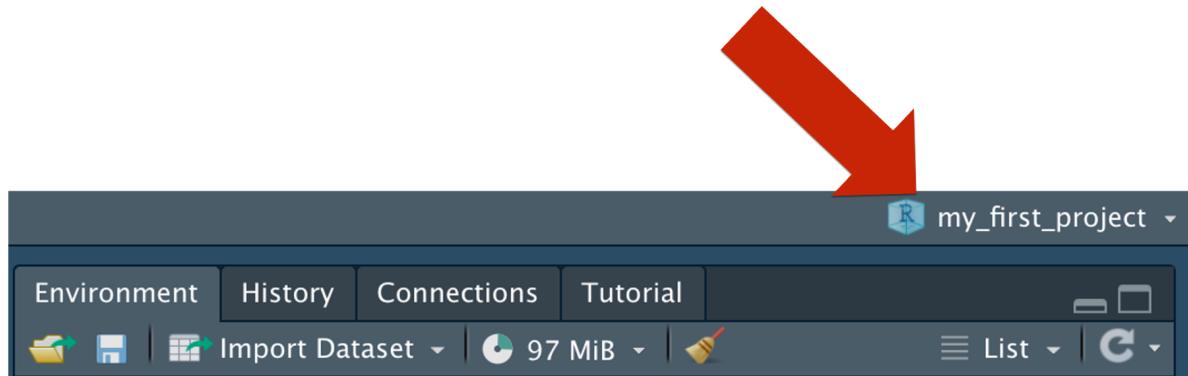
This file is called an R project file. Every time we create an R project, RStudio will create an R project file and add it to our project directory (i.e., the folder) for us. This file helps RStudio track and organize our R project.

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

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



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



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

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

10 Coding Best Practices

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

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

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

10.1 General principles

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

10.2 Code comments

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

10.2.1 Defining key variables

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

```
* **Case number definition**  
  - Case / investigation number.  
  
* **Intake stage definition**  
  - An ID number assigned to the Intake. Each Intake (Report) has its  
    own number. A case may have more than one intake. For example, case # 12345  
    has two intakes associated with it, 9 days apart, each with their own ID  
    number. Each of the two intakes associated with this case have multiple  
    allegations.  
  
* **Intake start definition**  
  - An intake is the submission or receipt of a report - a phone call or  
    web-based. The Intake Start Date refers to the date the staff member  
    opens a new record to begin recording the report.
```

10.2.2 What this code is trying to accomplish

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

```
## Standardize character strings

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

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

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

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

rm(vars)
```

10.2.3 Why we chose this particular strategy

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

```
### Create exact match dummy variables

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

10.3 Style guidelines

UsInG c_o_n_s_i_s_t_e_n_t STYLE i.s. import-ant!

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

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

10.3.1 Comments

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

10.3.2 Object (variable) names

In addition to the object naming guidance given in [the Tidyverse style guide](#), We suggest the following object naming conventions.

10.3.3 Use names that are informative

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

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

# Informative names
employed
married
education
```

10.3.3.1 Use names that are concise

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

```
# Write out entire name of the study the data comes from - Don't do this  
womens_health_initiative  
  
# Write out an acronym for the study the data comes from - assuming everyone  
# will be familiar with this acronym - Do this  
whi
```

10.3.3.2 Use all lowercase letters

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

```
# All upper case - so aggressive - Don't use  
MYSTUDYDATA  
  
# Camel case - Don't use  
myStudyData  
  
# All lowercase - Use  
my_study_data
```

10.3.3.3 Separate multiple words with underscores.

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

```
# Multiple words running together - Hard to read - Don't use  
mycancerdata  
  
# Camel case - easier to read, but more to remember and mess up - Don't use  
myCancerData  
  
# Separate with periods - easier to read, but doesn't translate well to many  
# other languages. For example, SAS won't accept variable names with
```

```
# periods - Don't use  
my.cancer.data  
  
# Separate with underscores - Use  
my_cancer_data
```

10.3.3.4 Prefix the names of similar variables

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

```
# Don't use  
first_name  
last_name  
middle_name  
  
# Use  
name_first  
name_last  
name_middle  
  
# Don't use  
street  
city  
state  
  
# Use  
address_street  
address_city  
address_state
```

10.3.4 File Names

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

10.3.4.1 Managing multiple files in projects

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

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

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

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

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

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

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



Note

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

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

11 Using Pipes

11.1 What are pipes?

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

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

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

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

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

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

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

```
[1] 3.662703
```

Here's what we did above:

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

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

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

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

Next, let’s try nesting these functions instead:

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

```
[1] 3.662703
```

Here's what we did above:

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

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

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

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

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

```
[1] 3.662703
```

Here's what we did above:

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

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

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

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

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

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

11.2 How do pipes work?

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

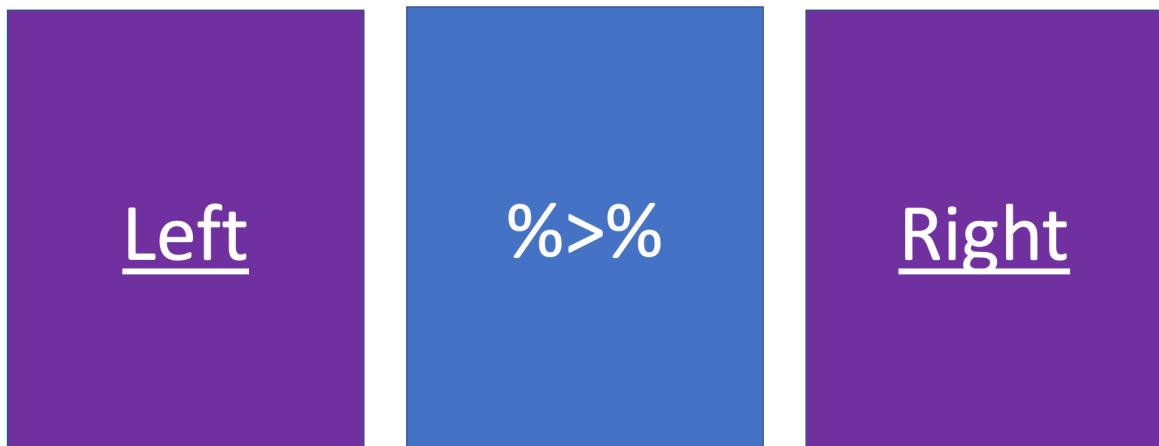


Figure 11.1: Pipes have a left side and a right side.

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

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

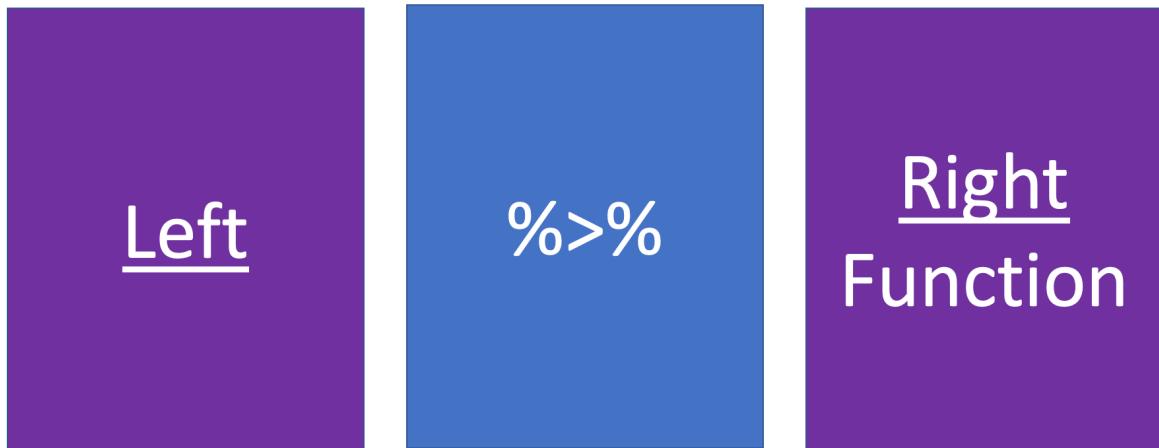


Figure 11.2: A function should always be to the right of the pipe operator.

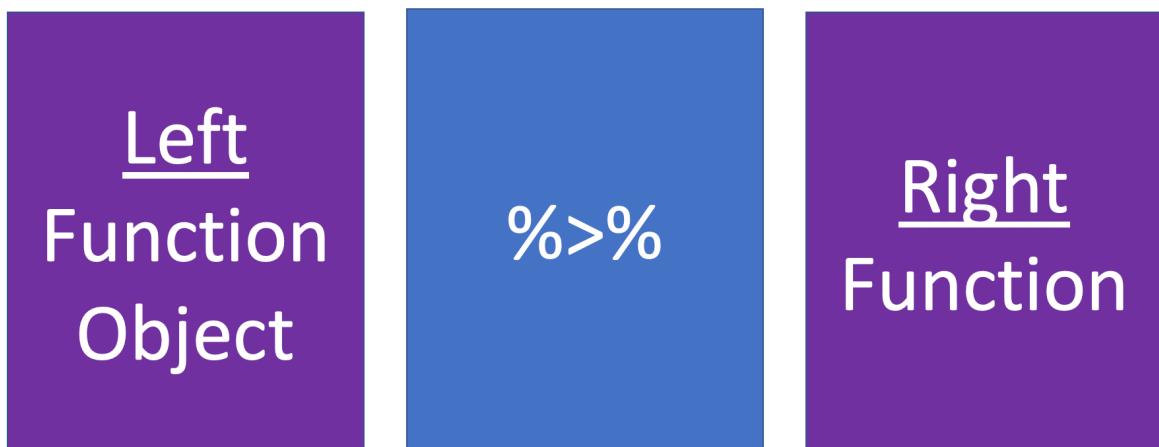


Figure 11.3: A function or an object can be to the left of the pipe operator.

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

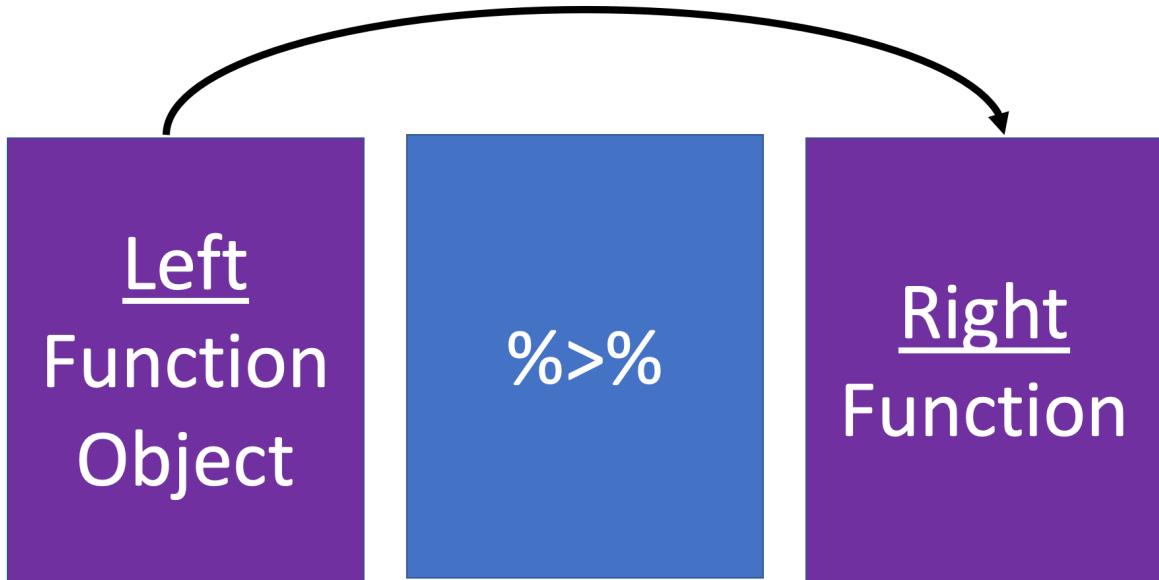


Figure 11.4: Pipe the left side to the first argument of the function on the right side.

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

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

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

```
[1] 5
```

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

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

```
[1] 5
```

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

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

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

height_and_weight
```



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

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

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



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

Here's what we did above:

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

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

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

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

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

 Note

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

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

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

11.2.1 Keyboard shortcut

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

On Mac type `shift + command + m`.

On Windows type `shift + control + m`

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

11.2.2 Pipe style

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

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

Each of these recommendations are demonstrated in the code below.

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

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

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

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

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

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

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

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

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

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

11.3 Final thought on pipes

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

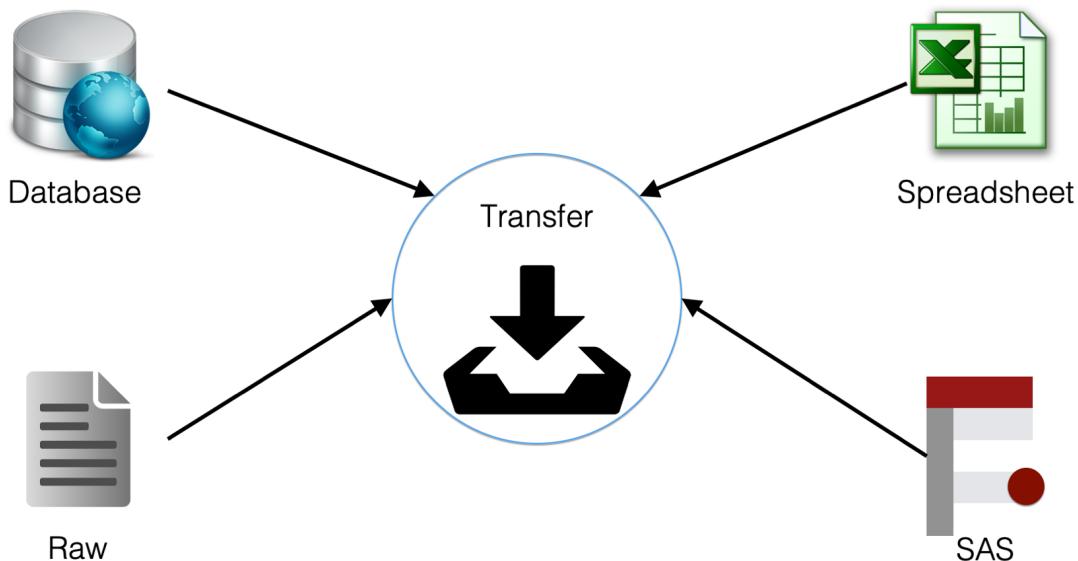
Part III

Data Transfer

12 Introduction to Data Transfer

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

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



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

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

13 File Paths

In this part of the book, we will need to work with **file paths**. File paths are nothing more than directions that tell R where to find, or place, data on our computer. In our experience, however, some students are a little bit confused about file paths at first. So, in this chapter we will briefly introduce what file paths are and how to find the path to a specific file on our computer.

Let's say that we want you to go to the store and buy a loaf of bread.



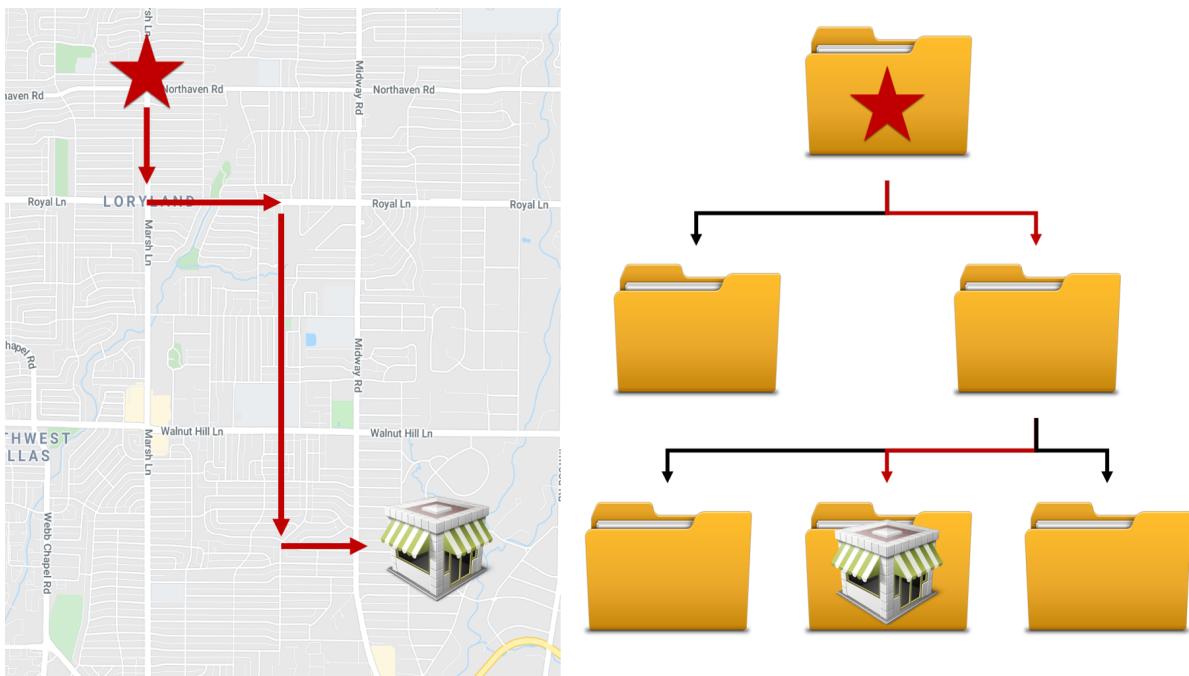
When we say, “go to the store”, this is really a shorthand way of telling you a much more detailed set of directions.



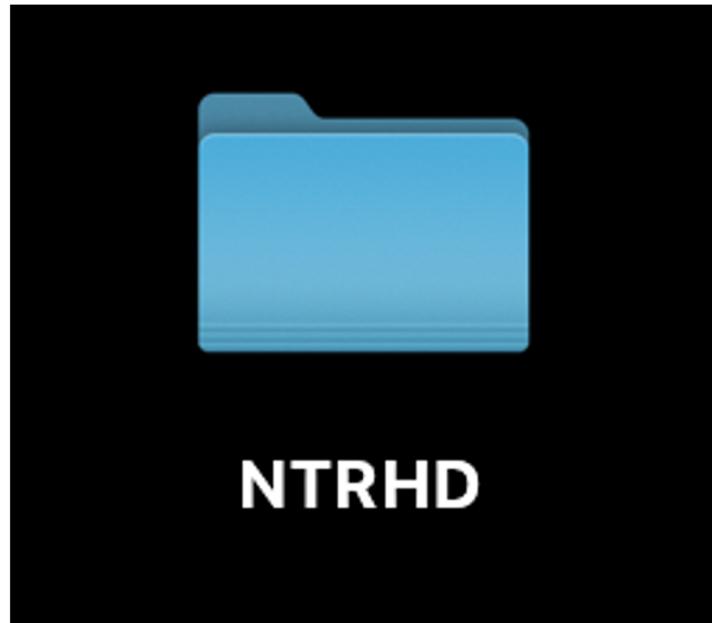
1. Start at home
2. Turn right on Camp Bowie Blvd.
3. Drive 1 mile
4. Turn left on Hulen St.
5. Drive .5 mile
6. Cross I-30
7. Turn right at second parking lot entrance

Not only do you need to do *all* of the steps in the directions above, but you also need to use the *exact sequence* above in order to arrive at the desired destination.

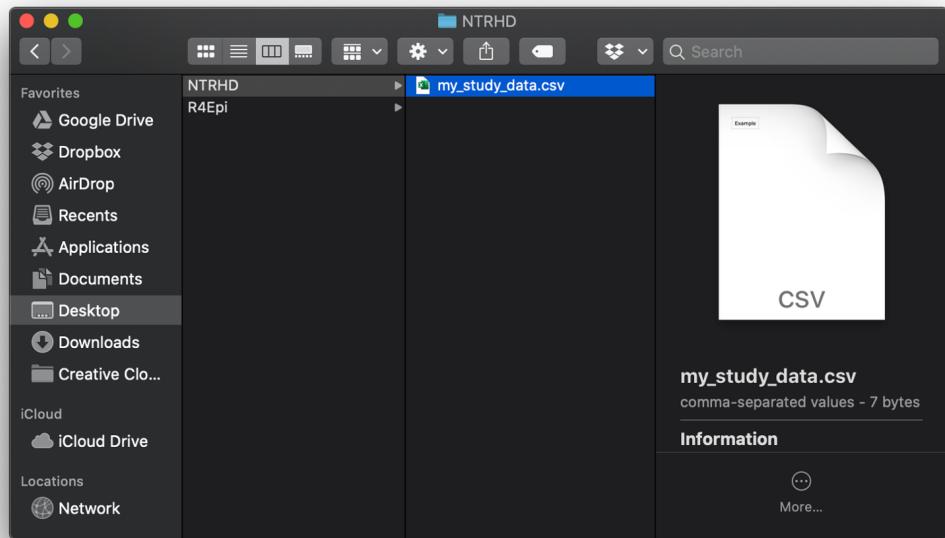
File paths aren't so different. If we want R to "go get" the file called `my_study_data.csv`, we have to give it directions to where that file is located. But the file's location is not a geographic location that involves making left and right turns. Rather, it is a location in your computer's file system that involves moving deeper into folders that are nested inside one another.



For example, let's say that we have a folder on our desktop called "NTRHD" for "North Texas Regional Health Department.



And, my_study_data.csv is inside the NTRHD folder.



We can give R directions to that data using the following path:

/Users/bradcannell/Desktop/NTRHD/my_study_data.csv (On Mac)

OR

C:/Users/bradcannell/Desktop/NTRHD/my_study_data.csv (On Windows)

! Warning

Mac and Linux use forward slashes (/) by default. Windows uses backslashes (\) in file paths by default. However, no matter which operating system we are using, we should still use forward slashes in the file paths we pass to import and export functions in RStudio. **In other words, use forward slashes even if you are using Windows.**

These directions may be read in a more human-like way by replacing the slashes with “and then”. For example, /Users/bradcannell/Desktop/NTRHD/my_study_data.csv can be read as “starting at the computer’s home directory, go into files that are accessible to the username bradcannell, and then go into the folder called Desktop, and then go into the folder called NTRHD, and then get the file called my_study_data.csv.”

Warning

You will need to change `bradcannell` to your username, unless your username also happens to be `bradcannell`

Warning

Notice that we typed `.csv` at the end immediately after the name of our file `my_study_data`. The `.csv` we typed is called a **file extension**. File extensions tell the computer the file's type and what programs can use it. In general, we MUST use the full file name and extension when importing and exporting data in R.

Self Quiz:

Let's say that we move `my_study_data.csv` to a different folder on our desktop called `research`. What file path would we need to give R to tell it how to find the data?

`/Users/bradcannell/Desktop/research/my_study_data.csv` (On Mac)

OR

`C:/Users/bradcannell/Desktop/research/my_study_data.csv` (On Windows)

Now let's say that we created a new folder inside of the `research` folder on our desktop called `my_studies`. Now what file path would we need to give R to tell it how to find the data?

`/Users/bradcannell/Desktop/research/my_studies/my_study_data.csv` (On Mac)

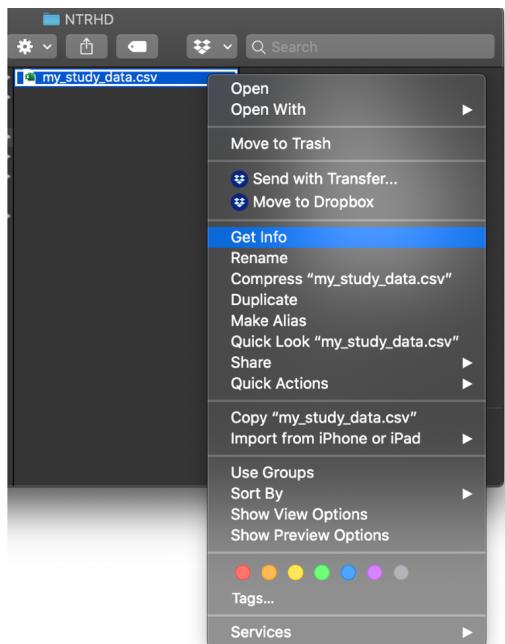
OR

`C:/Users/bradcannell/Desktop/research/my_studies/my_study_data.csv` (On Windows)

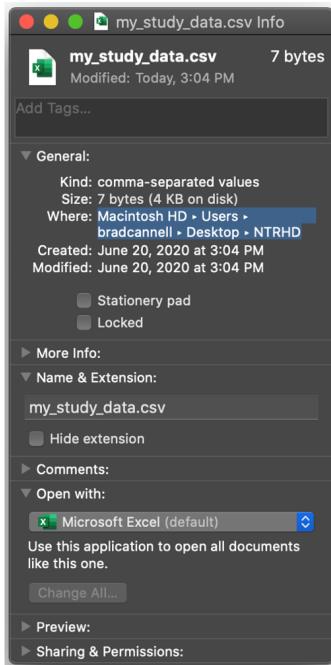
13.1 Finding file paths

Now that we know how file paths are constructed, we can always type them manually. However, typing file paths manually is tedious and error prone. Luckily, both Windows and MacOS have shortcuts that allow us to easily copy and paste file paths into R.

On a Mac, we right-click on the file we want the path for and a drop-down menu will appear. Then, click the `Get Info` menu option.



Now, we just copy the file path in the `Where` section of the get info window and paste it into our R code.



Alternatively, as shown below, we can right click on the file we want the path for to open the same drop-down menu shown above. But, if we hold down the alt/option key the `Copy`

menu option changes to `Copy ... as Pathname`. We can then left-click that option to copy the path and paste it into our R code.

A similar method exists in Windows as well. First, we *hold down the shift key* and right click on the file we want the path for. Then, we click `Copy as path` in the drop-down menu that appears and paste the file path into our R code.

13.2 Relative file paths

All of the file paths we've seen so far in this chapter are **absolute file paths** (as opposed to **relative file paths**). In this case, *absolute* just means that the file path begins with the computer's home directory. Remember, that the home directory in the examples above was `/Users/bradcannell`. When we are collaborating with other people, or sometimes even when we use more than one computer to work on our projects by ourselves, this can problematic. Pause here for a moment and think about why that might be...

Using absolute file paths can be problematic because the home directory can be different on every computer we use and is almost certainly different on one of our collaborator's computers. Let's take a look at an example. In the screenshot below, we are importing an Excel spreadsheet called `form_20.xlsx` into R as an R data frame named `df`. Don't worry about the import code itself. We will learn more about [importing Microsoft Excel spreadsheets](#) soon. For now, just look at the file path we are passing to the `read_excel()` function. By doing so, we are telling R where to go find the Excel file that we want to import. In this case, are we giving R an absolute or relative file path?

```
```{r}
library(dplyr, warn.conflicts = FALSE)
library(readxl)
```

Import using an **absolute** file path

```{r}
df <- read_excel("/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/my_first_project/data/form_20.xlsx")
```

```{r}
df
```

A tibble: 3 × 4
  date_received name_last name_first education
  <chr>          <chr>      <chr>        <dbl>
1 2013-08-22    Cooper     Samantha      4
2 2013-08-22    Rodriguez  Leslie       8
3 2013-08-22    Smith      Jane        5
3 rows
```

We are giving R an *absolute* file path. We know this because it starts with the home directory – `/Users/bradcannell`. Does our code work?

Yes! Our code does work. We can tell because there are no errors on the screen and the `df` object we created looks as we expect it to when we print it to the screen. Great!!

Now, let's say that our research assistant – Arthur Epi – is going to help us analyze this data as well. So, we share this code file with him. What do you think will happen when he runs the code on his computer?

The screenshot shows an RStudio interface with two code blocks and an error message.

```
```{r}
library(dplyr, warn.conflicts = FALSE)
library(readxl)
```

Import using an **absolute** file path
```

```
```{r}
df <- read_excel("/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/my_first_project/data/form_20.xlsx")
```

Error: `path` does not exist: '/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/my_first_project/data/form_20.xlsx'
```

The error message is displayed in red text, indicating that the specified file path does not exist.

When Arthur tries to import this file on his computer using our code, he gets an error. The error tells him that the path `/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/my_first_project/data/form_20.xlsx` doesn't exist. And on Arthur's computer it doesn't! The file `form_20.xlsx` exists, but not at the location `/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/my_first_project/data/`. This is because Arthur's home directory is `/Users/arthurrepi` not `/Users/bradcannell`. The directions are totally different!

To make this point clearer, let's return to our *directions to the store* example from earlier in the chapter. In that example, we only gave one list of directions to the store.



1. Start at home
2. Turn right on Camp Bowie Blvd.
3. Drive 1 mile
4. Turn left on Hulen St.
5. Drive .5 mile
6. Cross I-30
7. Turn right at second parking lot entrance

Notice that these directions assume that we are starting from our house. As long as we leave from our house, they work great! But what happens if we are at someone else's house and we ask you to go to the store and buy a loaf of bread? You'd walk out the front door and immediately discover that the directions don't make any sense! You'd think, "Camp Bowie Blvd.? Where is that? I don't see that street anywhere!"

Did the store disappear? No, of course not! The store is still there. It's just that our directions to the store assume that we are starting from our house. If these directions were a file path, they would be an *absolute* file path. They start all the way from our home and only work from our home.

So, could Arthur just change the absolute file path to work on his computer? Sure! He could do that, but then the file path wouldn't work on Brad's computer anymore. So, could there just be two code chunks in the file – one for Brad's computer and one for Arthur's computer? Sure! We could do that, but then one code chunk or the other will always throw an error on someone's computer. That will mean that we won't ever be able to just run our R code in its entirety. We'll have to run it chunk-by-chunk to make sure we skip the chunk that throws an error. And this problem would just be multiplied if we are working with 5, 10, or 15 other collaborators instead of just 1. So, is there a better solution?

Yes! A better solution is to use a **relative file path**. Returning to our *directions to the store* example, it would be like giving directions to the store from a common starting point that everyone knows.

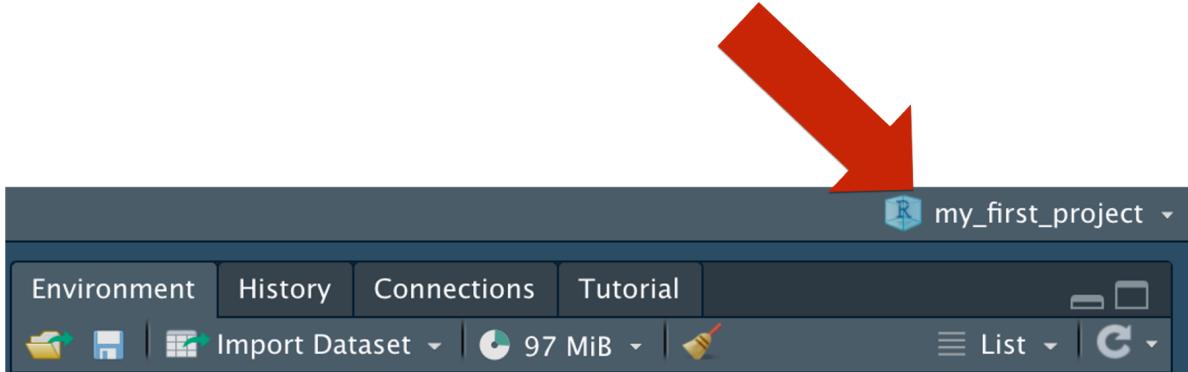


1. Start at the corner of Camp Bowie Blvd. and Hulen St.
2. Drive .5 mile
3. Cross I-30
4. Turn right at second parking lot entrance

Notice that the directions are now from a common location, which isn't somebody's "home". Instead, it's the corner of Camp Bowie Blvd. and Hulen St. You could even say that the directions are now *relative* to a common starting place. Now, we can give these directions to anyone and they can use them as long as they can find the corner of Camp Bowie and Hulen! Relative file paths work in much the same way. We tell RStudio to anchor itself at a common location that exists on everyone's computer and then all the directions are relative to that location. But, how can we do that? What location do all of our collaborators have on all of their computers?

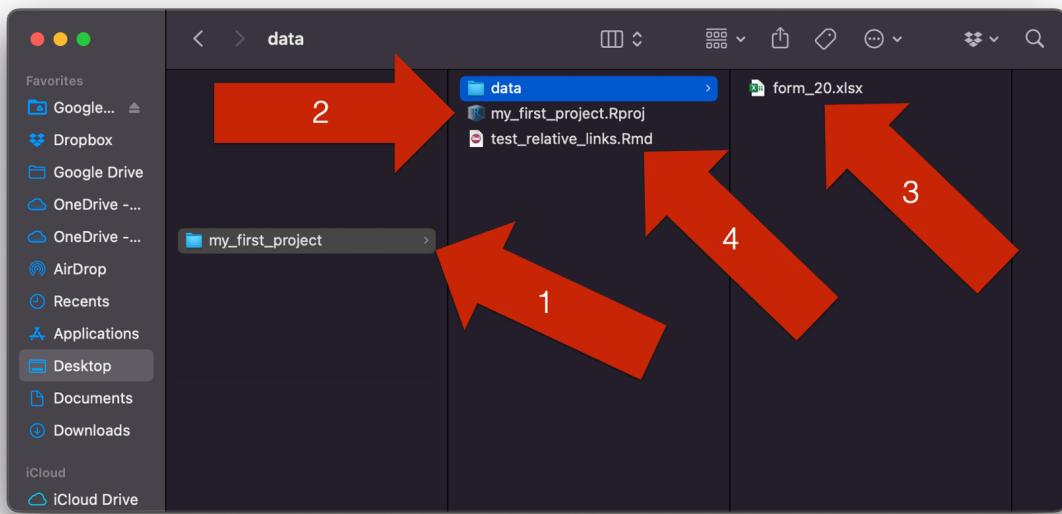
The answer is our R project's directory (i.e., folder)! In order to effectively use relative file paths in R, we start by creating an R project. If you don't remember how to create R projects, this would be a good time to go back and review the [R projects](#) chapter.

In the screenshot below, we can see that our RStudio session is open in the context of our R project called `my_first_project`.



In that context, R starts looking for files *in our R project folder* – no matter where we put the R project folder on our computer.

For example, in the next screenshot, we can see that the [R project](#) folder we previously created) (arrow 1), which is called `my_first_project`, is located on a computer's desktop. One way we can tell that it's an R project is because it contains an R project file (arrow 2). We can also see that our R project now contains a `folder`, which contains an Excel file called `form_20.xlsx` (arrow 3). Finally, we can see that we've added a new Quarto/ file called `test_relative_links.Rmd` (arrow 4). That file contains the code we wrote to import `form_20.xlsx` as an R data frame.



Because we are using an R project, we can tell R where to find `form_20.xlsx` using a *relative* file path. That is, we can give R directions that begin at the R project's directory. Remember, that just means the folder containing the R project file. In this case, `my_first_project`. Pause here for a minute. With that starting point in mind, how would you tell R to find `form_20.xlsx`?

Well, you would say, “go into the folder called `data`, and then get the file called `form_20.xlsx`.” Written as a file path, what would that look like?

It would look like `data/form_20.xlsx`. Let’s give it a try!

```

Import using a **relative** file path

```{r}
df <- read_excel("data/form_20.xlsx")
```

```{r}
df
```

```

A tibble: 3 × 4

| date_received | name_last | name_first | education |
|----------------------|------------------|-------------------|------------------|
| 2013-08-22 | Cooper | Samantha | 4 |
| 2013-08-22 | Rodriguez | Leslie | 8 |
| 2013-08-22 | Smith | Jane | 5 |

3 rows

It works! We can tell because there are no errors on the screen and the `df` object we created looks as we expect it to when we print it to the screen.

Now, let's try it on Arthur's computer and see what happens.

```

```{r}
library(dplyr, warn.conflicts = FALSE)
library(readxl)
```

Import using an **absolute** file path

```{r}
df <- read_excel("~/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/test_relative_links/data/form_20.xlsx")
```

Error in read_excel("~/Users/bradcannell/Dropbox/02 Teaching/R4Epi Textbook/test_relative_links/data/form_20.xlsx") :
  could not find function "read_excel"

Import using a **relative** file path

```{r}
df <- read_excel("data/form_20.xlsx")
```

```{r}
df
```

```

A tibble: 3 × 4

| date_received | name_last | name_first | education |
|----------------------|------------------|-------------------|------------------|
| 2013-08-22 | Cooper | Samantha | 4 |
| 2013-08-22 | Rodriguez | Leslie | 8 |
| 2013-08-22 | Smith | Jane | 5 |

3 rows

As you can see, the absolute path still doesn't work on Arthur's computer, but the relative path does! It may not be obvious to you now, but this makes collaborating so much easier!

Let's quickly recap what we needed to do to be able to use relative file paths.

1. We need to create an [R project](#).
2. We needed to save our R code and our data inside of the R project directory.
3. We needed to share the R project folder with our collaborators. This part wasn't shown, but it was implied. We could have shared our R project by email. We could have shared our R project by using a shared cloud-based file storage service like Dropbox, Google Drive, or OneDrive. Better yet, we could have shared our R project using a [GitHub repository](#), which we will discuss later in the book.
4. We replaced all absolute file paths in our code with relative file paths. In general, we should *always* use relative file paths if at all possible. It makes our code easier to read and maintain, and it makes life so much easier for us when we collaborate with others!

Now that we know what file paths are and how to find them, let's use them to import and export data to and from R.

14 Importing Plain Text Files

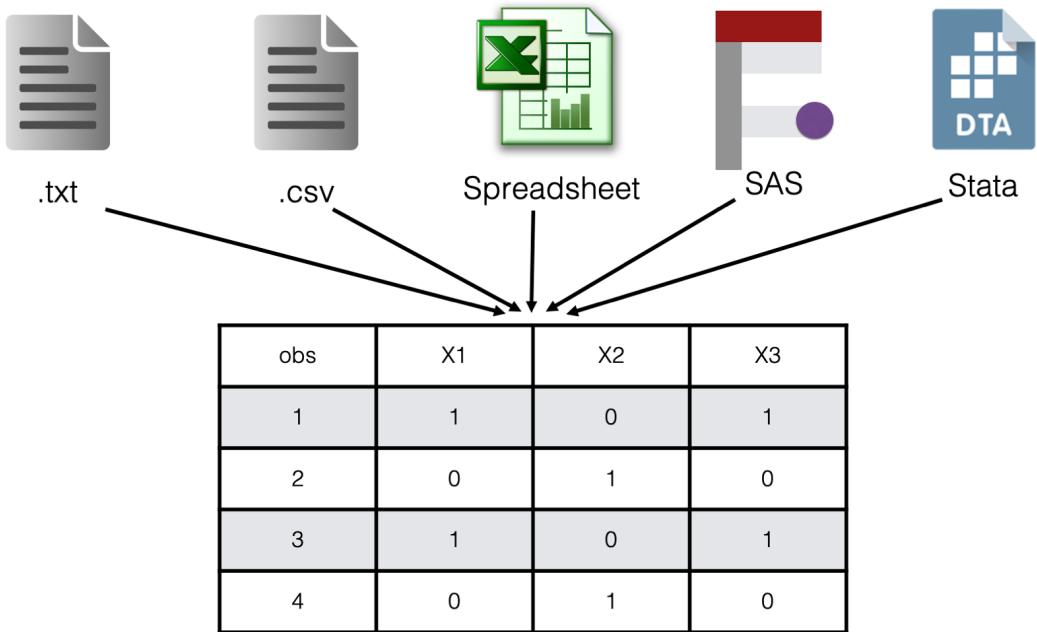
We previously learned how to manually create a data frame in RStudio with the `data.frame()` function, the `tibble()` function, or the `tribble()` function. This will get the job done, but it's not always very practical – particularly when you have larger data sets.

Additionally, others will usually share data with you that is already stored in a file of some sort. For our purposes, any file containing data that is not an R data frame is referred to as raw data. In my experience, raw data is most commonly shared as CSV (comma separated values) files or as Microsoft Excel files. CSV files will end with the `.csv` file extension and Excel files end with the `.xls` or `.xlsx` file extensions. But remember, generally speaking R can only manipulate and analyze data that has been imported into R's global environment. In this lesson, you will learn how to take data stored in several different common types of files import them into R for use.

There are many different file types that one can use to store data. In this book, we will divide those file types into two categories: [plain text files](#) and binary files. Plain text files are simple files that you (a human) can directly read using only your operating system's plain text editor (i.e., Notepad on Windows orTextEdit on Mac). These files usually end with the `.txt` file extension – one exception being the `.csv` extension. Specifically, in this chapter we will learn to import the following variations of plain text files:

- Plain text files with data delimited by a single space.
- Plain text files with data delimited by tabs.
- Plain text files stored in a fixed width format.
- Plain text files with data delimited by commas - csv files.

Later, we will discuss importing binary files. For now, you can think of binary files as more complex file types that can't generally be read by humans without the use of special software. Some examples include Microsoft Excel spreadsheets, SAS data sets, and Stata data sets.



14.1 Packages for importing data

Base R contains several functions that can be used to import plain text files; however, I'm going to use the `readr` package to import data in the examples that follow. Compared to base R functions for importing plain text files, `readr`:

- Is roughly 10 times faster.
- Doesn't convert character variables to factors by default.
- Behaves more consistently across operating systems and geographic locations.

If you would like to follow along, I suggest that you go ahead and install and load `readr` now.

```
library(readr)
```

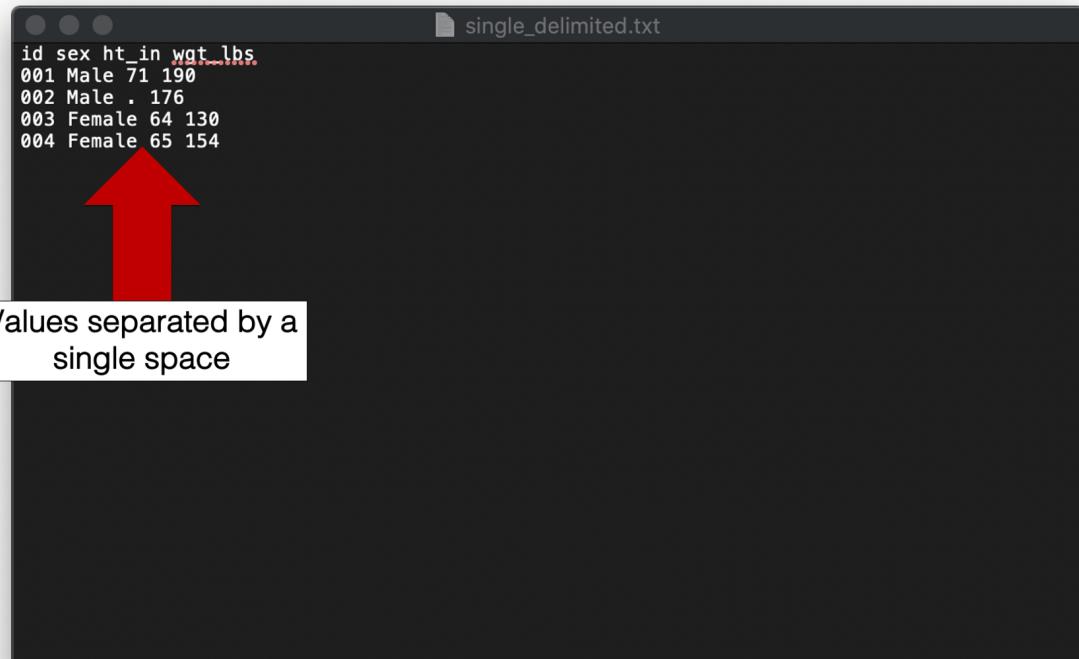
14.2 Importing space delimited files

We will start by importing data with values separated by a single space. Not necessarily because this is the most common format you will encounter; in my experience it is not. But

it's about as simple as it gets, and other types of data are often considered special cases of files separated with a single space. So, it seems like a good place to start.

 Tip

Side Note: In programming lingo, it is common to use the word **delimited** interchangeably with the word **separated**. For example, you might say “values separated by a single space” or you might say “a file with space delimited values.”



For our first example we will import a text file with values separated by a single space. The contents of the file are the now familiar height and weight data.

You may [click here](#) to download this file to your computer.

```
single_space <- read_delim(  
  file = "single_delimited.txt",  
  delim = " ")  
)
```

```
Rows: 4 Columns: 4  
-- Column specification -----  
Delimiter: " "  
chr (3): id, sex, ht_in
```

```
dbl (1): wgt_lbs

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
single_space
```

```
# A tibble: 4 x 4
  id    sex   ht_in wgt_lbs
  <chr> <chr>  <chr>   <dbl>
1 001  Male   71     190
2 002  Male   .      176
3 003  Female 64     130
4 004  Female 65     154
```

Here's what we did above:

- We used `readr`'s `read_delim()` function to import a data set with values that are delimited by a single space. Those values were imported as a data frame, and we assigned that data frame to the R object called `single_space`.
- You can type `?read_delim` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `read_delim()` function is the `file` argument. The value passed to the `file` argument should be a file path that tells R where to find the data set on your computer.
- The second argument to the `read_delim()` function is the `delim` argument. The value passed to the `delim` argument tells R what character separates each value in the data set. In this case, a single space separates the values. Note that we had to wrap the single space in quotation marks.
- The `readr` package imported the data and printed a message giving us some information about how it interpreted column names and column types. In programming lingo, deciding how to interpret the data that is being imported is called **parsing** the data.
 - By default, `readr` will assume that the first row of data contains variable names and will try to use them as column names in the data frame it creates. In this case, that was a good assumption. We want the columns to be named `id`, `sex`, `ht_in`, and `wgt_lbs`. Later, we will learn how to override this default behavior.

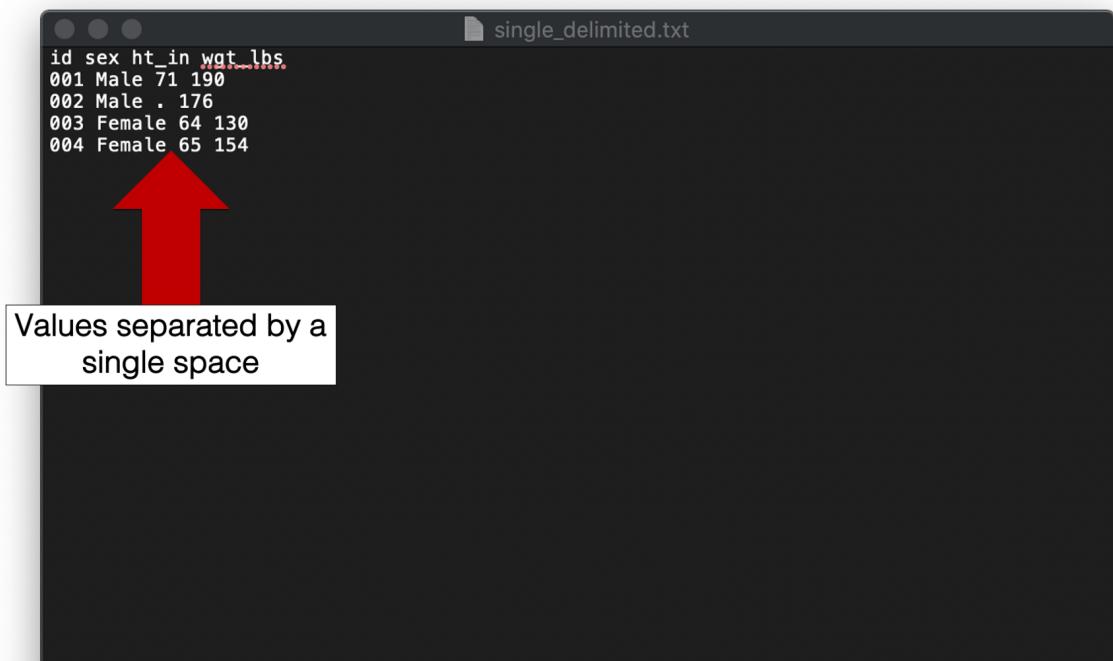
- By default, `readr` will try to guess what type of data (e.g., numbers, character strings, dates, etc.) each column contains. It will guess based on analyzing the contents of the first 1,000 rows of the data. In this case, `readr`'s guess was not entirely correct (or at least not what we wanted). `readr` correctly guessed that the variables `id` and `sex` should be character variables, but incorrectly guessed that `ht_in` should be a character variable as well. Below, we will learn how to fix this issue.

 Warning

Make sure to always include the file extension in your file paths. For example, using “/single_delimited” instead of “/single_delimited.txt” above (i.e., no .txt) would have resulted in an error telling you that the file does not exist.

14.2.1 Specifying missing data values

In the previous example, `readr` guessed that the variable `ht_in` was a character variable. Take another look at the data and see if you can figure out why?



```
id sex ht_in wgt_lbs
001 Male 71 190
002 Male . 176
003 Female 64 130
004 Female 65 154
```

Values separated by a single space

Did you see the period in the third value of the third row? The period is there because this value is missing, and a period is commonly used to represent missing data. However, R represents missing data with the special `NA` value – not a period. So, the period is just a regular character value to R. When R reads the values in the `ht_in` column, it decides that it

can easily turn the numbers into character values, but it doesn't know how to turn the period into a number. So, the column is parsed as a character vector.

But as we said, this is not what we want. So, how do we fix it? Well, in this case, we will simply need to tell R that missing values are represented with a period in the data we are importing. We do that by passing that information to the `na` argument of the `read_delim()` function:

```
single_space <- read_delim(  
  file = "single_delimited.txt",  
  delim = " ",  
  na = ".")  
)
```

```
Rows: 4 Columns: 4  
-- Column specification -----  
Delimiter: " "  
chr (2): id, sex  
dbl (2): ht_in, wgt_lbs  
  
i Use `spec()` to retrieve the full column specification for this data.  
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
single_space
```

```
# A tibble: 4 x 4  
  id     sex    ht_in wgt_lbs  
  <chr> <chr>   <dbl>   <dbl>  
1 001   Male     71     190  
2 002   Male     NA      176  
3 003   Female   64     130  
4 004   Female   65     154
```

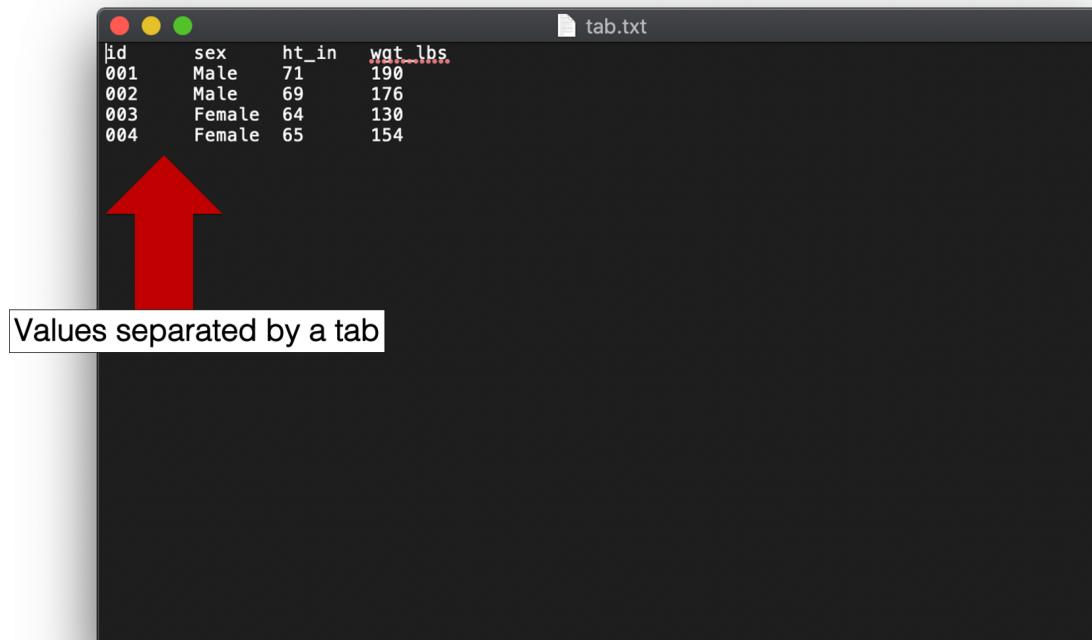
Here's what we did above:

- By default, the value passed to the `na` argument of the `read_delim()` function is `c("", "NA")`. This means that R looks for nothing (i.e., a value should be there but isn't - this really doesn't make sense when the delimiter is a single space) or an `NA`.
- We told R to look for a period to represent missing data instead of a nothing or an `NA` by passing the period character to the `na` argument.

- It's important to note that changing the value of the `na` argument does not change the way R represents missing data in the data frame that is created. It only tells R how to identify missing values in the raw data that we are importing. In the R data frame that is created, missing data will still be represented with the special NA value.

14.3 Importing tab delimited files

Sometimes you will encounter plain text files that contain values separated by tab characters instead of a single space. Files like these may be called **tab separated value** or **tsv** files, or they may be called **tab-delimited** files.



| <code>id</code> | <code>sex</code> | <code>ht_in</code> | <code>wt_lbs</code> |
|-----------------|------------------|--------------------|---------------------|
| 001 | Male | 71 | 190 |
| 002 | Male | 69 | 176 |
| 003 | Female | 64 | 130 |
| 004 | Female | 65 | 154 |

To import tab separated value files in R, we use a variation of the same program we just saw. We just need to tell R that now the values in the data will be delimited by tabs instead of a single space.

You may [click here](#) to download this file to your computer.

```
tab <- read_delim(
  file = "tab.txt",
  delim = "\t"
)
```

```
Rows: 4 Columns: 4
-- Column specification -----
Delimiter: "\t"
chr (2): id, sex
dbl (2): ht_in, wgt_lbs

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
tab
```

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

Here's what we did above:

- We used `readr`'s `read_delim()` function to import a data set with values that are delimited by tabs. Those values were imported as a data frame, and we assigned that data frame to the R object called `tab`.
- To tell R that the values are now separated by tabs, we changed the value we passed to the `delim` argument to "\t". This is a special symbol that means "tab" to R.

I don't personally receive tab separated values files very often. But, apparently, they are common enough to warrant a shortcut function in the `readr` package. That is, instead of using the `read_delim()` function with the value of the `delim` argument set to "\t", we can simply pass our file path to the `read_tsv()` function. Under the hood, the `read_tsv()` function does exactly the same thing as the `read_delim()` function with the value of the `delim` argument set to "\t".

```
tab <- read_tsv("tab.txt")
```

```
Rows: 4 Columns: 4
-- Column specification -----
Delimiter: "\t"
chr (2): id, sex
dbl (2): ht_in, wgt_lbs
```

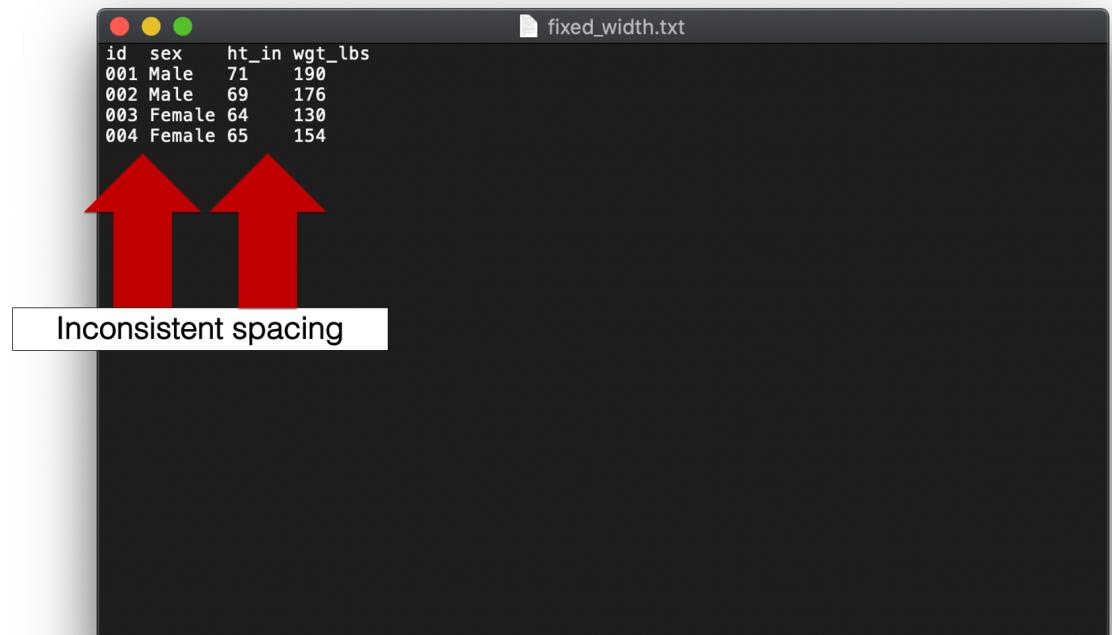
```
i Use `spec()` to retrieve the full column specification for this data.  
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
tab
```

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

14.4 Importing fixed width format files

Yet another type of plain text file we will discuss is called a **fixed width format** or **fwf** file. Again, these files aren't super common in my experience, but they can be sort of tricky when you do encounter them. Take a look at this example:



As you can see, a hallmark of fixed width format files is inconsistent spacing between values. For example, there is only one single space between the values 004 and Female in the fourth

row. But, there are multiple spaces between the values 65 and 154. Therefore, we can't tell R to look for a single space or tab to separate values. So, how do we tell R which characters (including spaces) go with which variable? Well, if you look closely you will notice that all variable values start in the same column. If you are wondering what I mean, try to imagine a number line along the top of the data:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|----|---|---|---|---|
| i | d | | | s | e | x | | | | h | t | _ | i | n | w | g | t | _ | l | b | s | | |
| 0 | 0 | 1 | M | A | L | E | | | | 7 | 1 | | | | 1 | 9 | 0 | | | | | | |
| 0 | 0 | 2 | M | A | L | E | | | | 6 | 9 | | | | 1 | 7 | 6 | | | | | | |
| 0 | 0 | 3 | F | E | M | A | L | E | | 6 | 4 | | | | 1 | 3 | 0 | | | | | | |
| 0 | 0 | 4 | F | E | M | A | L | E | | 6 | 5 | | | | 1 | 5 | 4 | | | | | | |

This number line creates a sequence of columns across your data, with each column being 1 character wide. Notice that spaces are also considered a character with width just like any other. We can use these columns to tell R exactly which columns contain the values for each variable.

You may [click here to download this file to your computer](#).

Now, in this case we can just use `readr`'s `read_table()` function to import this data:

```
fixed <- read_table("fixed_width.txt")

-- Column specification -----
cols(
  id = col_character(),
  sex = col_character(),
  ht_in = col_double(),
  wgt_lbs = col_double()
)
```

```
Warning: 1 parsing failure.  
row col  expected      actual          file  
1  -- 4 columns 5 columns 'fixed_width.txt'
```

```
fixed
```

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

Here's what we did above:

- We used `readr`'s `read_table()` function to import data from a fixed width format file. Those values were imported as a data frame, and we assigned that data frame to the R object called `fixed`.
- You can type `?read_table` into your R console to view the help documentation for this function and follow along with the explanation below.
- By default, the `read_table()` function looks for values to be separated by one or more columns of space.

However, how could you import this data if there weren't always spaces in between data values. For example:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|----|---|---|---|---|--|
| i | d | | s | e | x | | | h | t | _ | i | n | w | g | t | _ | l | b | s | | | | | |
| 0 | 0 | 1 | M | A | L | E | | 7 | 1 | | | | 1 | 9 | 0 | | | | | | | | | |
| 0 | 0 | 2 | M | A | L | E | | 6 | 9 | | | | 1 | 7 | 6 | | | | | | | | | |
| 0 | 0 | 3 | F | E | M | A | L | E | 6 | 4 | | | 1 | 3 | 0 | | | | | | | | | |
| 0 | 0 | 4 | F | E | M | A | L | E | 6 | 5 | | | 1 | 5 | 4 | | | | | | | | | |

In this case, the `read_table()` function does not give us the result we want.

```
fixed <- read_table("fixed_width_no_space.txt")

-- Column specification -----
cols(
  id = col_character(),
  sex = col_double(),
  ht_inwgt_lbs = col_double()
)

Warning: 3 parsing failures.
row col  expected      actual          file
 1   -- 3 columns 4 columns 'fixed_width_no_space.txt'
 3   -- 3 columns 2 columns 'fixed_width_no_space.txt'
 4   -- 3 columns 2 columns 'fixed_width_no_space.txt'

fixed

# A tibble: 4 x 3
  id       sex ht_inwgt_lbs
```

| | <chr> | <dbl> | <dbl> |
|---|-----------|-------|-------|
| 1 | 001Male | 71 | 190 |
| 2 | 002Male | 69 | 176 |
| 3 | 003Female | 64 | NA |
| 4 | 004Female | 65 | NA |

Instead, it parses the entire data set as a single character column. It does this because it can't tell where the values for one variable stop and the values for the next variable start. However, because all the variables start in the same column, we can tell R how to parse the data correctly. We can actually do this in a couple different ways:

[You may click here to download this file to your computer.](#)

14.4.1 Vector of column widths

One way to import this data is to tell R how many columns wide each variable is in the raw data. We do that like so:

```
fixed <- read_fwf(
  file = "fixed_width_no_space.txt",
  col_positions = fwf_widths(
    widths     = c(3, 6, 5, 3),
    col_names  = c("id", "sex", "ht_in", "wgt_lbs")
  ),
  skip = 1
)
```

```
Rows: 4 Columns: 4
-- Column specification ----

chr (2): id, sex
dbl (2): ht_in, wgt_lbs

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
fixed
```

```
# A tibble: 4 x 4
  id     sex     ht_in wgt_lbs
  <chr> <chr>   <dbl>   <dbl>
```

| | | | | |
|---|-----|--------|----|-----|
| 1 | 001 | Male | 71 | 190 |
| 2 | 002 | Male | 69 | 176 |
| 3 | 003 | Female | 64 | 130 |
| 4 | 004 | Female | 65 | 154 |

Here's what we did above:

- We used `readr`'s `read_fwf()` function to import data from a fixed width format file. Those values were imported as a data frame, and we assigned that data frame to the R object called `fixed`.
- You can type `?read_fwf` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `read_fwf()` function is the `file` argument. The value passed to the `file` argument should be file path that tells R where to find the data set on your computer.
- The second argument to the `read_fwf()` function is the the `col_positions` argument. The value passed to this argument tells R the width (i.e., number of columns) that belong to each variable in the raw data set. This information is actually passed to the `col_positions` argument directly from the `fwf_widths()` function. This is an example of nesting functions.
 - The first argument to the `fwf_widths()` function is the `widths` argument. The value passed to the `widths` argument should be a numeric vector of column widths. The column width of each variable should be calculated as the number of columns that contain the values for that variable. For example, take another look at the data with the imaginary number line:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|----|---|---|---|---|--|
| i | d | | s | e | x | | | | h | t | _ | i | n | w | g | t | _ | l | b | s | | | | |
| 0 | 0 | 1 | M | A | L | E | | | 7 | 1 | | | | 1 | 9 | 0 | | | | | | | | |
| 0 | 0 | 2 | M | A | L | E | | | 6 | 9 | | | | 1 | 7 | 6 | | | | | | | | |
| 0 | 0 | 3 | F | E | M | A | L | E | 6 | 4 | | | | 1 | 3 | 0 | | | | | | | | |
| 0 | 0 | 4 | F | E | M | A | L | E | 6 | 5 | | | | 1 | 5 | 4 | | | | | | | | |

All of the values for the variable `id` can be located within the first 3 columns of data. All of the values for the variable `sex` can be located within the next 6 columns of data. All of the values for the variable `ht_in` can be located within the next 5 columns of data. And, all of the values for the variable `wgt_lbs` can be located within the next 3 columns of data. Therefore, we pass the vector `c(3, 6, 5, 3)` to the `widths` argument.

The second argument to the `fwf_widths()` function is the `col_names` argument. The value passed to the `col_names` argument should be a character vector of column names.

- The third argument of the `read_fwf()` function that we passed a value to is the `skip` argument. The value passed to the `skip` argument tells R how many rows to ignore before looking for data values in the raw data. In this case, we passed a value of one, which told R to ignore the first row of the raw data. We did this because the first row of the raw data contained variable names instead of data values, and we already gave R variable names in the `col_names` argument to the `fwf_widths()` function.

14.4.2 Paired vector of start and end positions

Another way to import this data is to tell R how which columns each variable starts and stops at in the raw data. We do that like so:

```
fixed <- read_fwf(
  file = "fixed_width_no_space.txt",
```

```

col_positions = fwf_positions(
  start      = c(1, 4, 10, 15),
  end        = c(3, 9, 11, 17),
  col_names  = c("id", "sex", "ht_in", "wgt_lbs")
),
skip = 1
)

```

```

Rows: 4 Columns: 4
-- Column specification -----
chr (2): id, sex
dbl (2): ht_in, wgt_lbs

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

```

`fixed`

```

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

```

Here's what we did above:

- This time, we passed column positions to the `col_positions` argument of `read_fwf()` directly from the `fwf_positions()` function.
 - The first argument to the `fwf_positions()` function is the `start` argument. The value passed to the `start` argument should be a numeric vector containing the first column that contains a value for each variable. For example, take another look at the data with the imaginary number line:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|----|---|---|---|---|--|
| i | d | | s | e | x | | | h | t | _ | i | n | w | g | t | _ | l | b | s | | | | | |
| 0 | 0 | 1 | M | A | L | E | | 7 | 1 | | | | 1 | 9 | 0 | | | | | | | | | |
| 0 | 0 | 2 | M | A | L | E | | 6 | 9 | | | | 1 | 7 | 6 | | | | | | | | | |
| 0 | 0 | 3 | F | E | M | A | L | E | 6 | 4 | | | 1 | 3 | 0 | | | | | | | | | |
| 0 | 0 | 4 | F | E | M | A | L | E | 6 | 5 | | | 1 | 5 | 4 | | | | | | | | | |

The first column that contains part of the value for the variable `id` can be located in column 1 of data. The first column that contains part of the value for the variable `sex` can be located in column 4 of data. The first column that contains part of the value for the variable `ht_in` can be located in column 10 of data. And, the first column that contains part of the value for the variable `wgt_lbs` can be located in column 15 of data. Therefore, we pass the vector `c(1, 4, 10, 15)` to the `start` argument.

The second argument to the `fwf_positions()` function is the `end` argument. The value passed to the `end` argument should be a numeric vector containing the last column that contains a value for each variable. The last column that contains part of the value for the variable `id` can be located in column 3 of data. The last column that contains part of the value for the variable `sex` can be located in column 9 of data. The last column that contains part of the value for the variable `ht_in` can be located in column 11 of data. And, the last column that contains part of the value for the variable `wgt_lbs` can be located in column 17 of data. Therefore, we pass the vector `c(3, 9, 11, 17)` to the `end` argument.

The third argument to the `fwf_positions()` function is the `col_names` argument. The value passed to the `col_names` argument should be a character vector of column names.

14.4.3 Using named arguments

As a shortcut, either of the methods above can be written using named vectors. All this means is that we basically combine the `widths` and `col_names` arguments to pass a vector of column

widths, or we combine the `start`, `end`, and `col_names` arguments to pass a vector of start and end positions. For example:

Column widths:

```
read_fwf(  
  file = "fixed_width_no_space.txt",  
  col_positions = fwf_cols(  
    id      = 3,  
    sex     = 6,  
    ht_in   = 5,  
    wgt_lbs = 3  
)  
  skip = 1  
)
```

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

Column positions:

```
read_fwf(  
  file = "fixed_width_no_space.txt",  
  col_positions = fwf_cols(  
    id      = c(1, 3),  
    sex     = c(4, 9),  
    ht_in   = c(10, 11),  
    wgt_lbs = c(15, 17)  
)  
  skip = 1  
)
```

```
Rows: 4 Columns: 4  
-- Column specification -----  
chr (2): id, sex  
dbl (2): ht_in, wgt_lbs
```

```

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

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

```

14.5 Importing comma separated values files

The final type of plain text file that we will discuss is by far the most common type used in my experience. I'm talking about the **comma separated values** or **csv** file. Unlike space and tab separated values files, csv file names end with the **.csv** file extension. Although, csv files are plain text files that can be opened in plain text editors such as Notepad for Windows orTextEdit for Mac, many people view csv files in spreadsheet applications like Microsoft Excel, Numbers for Mac, or Google Sheets.

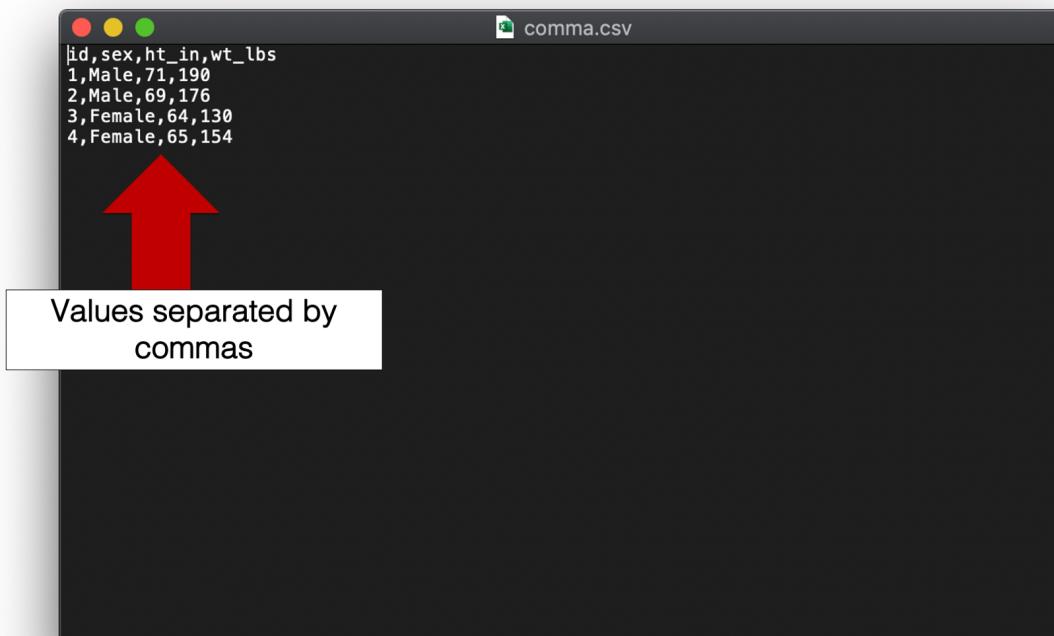


Figure 14.1: A csv file viewed in a plain text editor.

| | A | B | C | D | E |
|---|----|----------|-------|--------|---|
| 1 | id | sex | ht_in | wt_lbs | |
| 2 | | 1 Male | 71 | 190 | |
| 3 | | 2 Male | 69 | 176 | |
| 4 | | 3 Female | 64 | 130 | |
| 5 | | 4 Female | 65 | 154 | |
| 6 | | | | | |
| 7 | | | | | |

Figure 14.2: A csv file viewed in Microsoft Excel.

Importing standard csv files into R with the `readr` package is easy and uses a syntax that is very similar to `read_delim()` and `read_tsv()`. In fact, in many cases we only have to pass the path to the csv file to the `read_csv()` function like so:

You may [click here](#) to download this file to your computer.

```
csv <- read_csv("comma.csv")

Rows: 4 Columns: 4
-- Column specification ----
Delimiter: ","
chr (1): sex
dbl (3): id, ht_in, wt_lbs

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
csv
```

```
# A tibble: 4 x 4
```

```

  id sex    ht_in wt_lbs
<dbl> <chr>  <dbl>  <dbl>
1     1 Male      71    190
2     2 Male      69    176
3     3 Female    64    130
4     4 Female    65    154

```

Here's what we did above:

- We used `readr`'s `read_csv()` function to import a data set with values that are delimited by commas. Those values were imported as a data frame, and we assigned that data frame to the R object called `csv`.
- You can type `?read_csv` into your R console to view the help documentation for this function and follow along with the explanation below.
- Like `read_tsv()`, R is basically executing the `read_delim()` function with the value of the `delim` argument set to `,` under the hood. You could also use the `read_delim()` function with the value of the `delim` argument set to `,` if you wanted to.

14.6 Additional arguments

For the most part, the data we imported in all of the examples above was relatively well behaved. What I mean by that is that the data basically “looked” like each of the `read_` functions were expecting it to “look”. Therefore, we didn’t have to adjust many of the various `read_` functions’ default values. The exception was changing the default value of the `na` argument to the `read_delim()` function. However, all of the `read_` functions above have additional arguments that you may need to tweak on occasion. The two that I tend to adjust most often are the `col_names` and `col_types` arguments. It’s impossible for me to think of every scenario where you may need to do this, but I’ll walk through a basic example below, which should be sufficient for you to get the idea.

Take a look at this csv file for a few seconds. It started as the same exact height and weight data we’ve been using, but I made a few changes. See if you can spot them all.

| | A | B | C | D | E |
|---|----------|-----------------|------------------------|--------------------------|---------------------|
| 1 | Var1 | Var1 | Var3 | Var4 | Notes |
| 2 | | | | | |
| 3 | Study ID | Participant Sex | Paticipant Height (in) | Participant Weight (lbs) | |
| 4 | 1 | Male | 71 | 190 | |
| 5 | 2 | Male | | 176 | |
| 6 | 3 | Female | 64 | 130 | |
| 7 | 4 | Female | 65 | Missing | Call back on Monday |
| 8 | | | | | |

When people record data in Microsoft Excel, they do all kinds of crazy things. In the screenshot above, I've included just a few examples of things I see all the time. For example:

- Row one contains generic variable names that don't really serve much of a purpose.
- Row two is a blank line. I'm not sure why it's there. Maybe the study staff finds it aesthetically pleasing?
- Row three contains some variable descriptions. These are actually useful, but they aren't currently formatted in a way that makes for good variable names.
- Row 7, column D is a missing value. However, someone wrote the word "Missing" instead of leaving the cell blank.
- Column E also contains some notes for the data collection staff that aren't really part of the data.

All of the issues listed above are things we will have to deal with before we can analyze our data. Now, in this small data set we could just fix these issues directly in Microsoft Excel and then import the altered data into R with a simple call to `read_csv()` without adjusting any options. However, that this is generally a really bad idea.

 Warning

- I suggest that you don't **EVER** alter your raw data. All kinds of crazy things happen with data and data files. If you keep your raw data untouched and in a safe place, worst case scenario you can always come back to it and start over. If you start messing with the raw data, then you may lose the ability to recover what it looked like in its original form forever. If you import the data into R before altering it then your raw data stays preserved
- If you are going to make alterations in Excel prior to importing the data, I **strongly** suggest making a copy of the raw data first. Then, alter the copy before importing into R. But, even this can be a bad idea.
- If you make alterations to the data in Excel then there is generally no record of those alterations. For example, let's say you click in a cell and delete a value (maybe even by accident), and then send me the csv file. I will have no way of knowing that a value was deleted. When you alter the data directly in Excel (or any program that doesn't require writing code), it can be really difficult for others (including future you) to know what was done to the data. You may be able manually compare the altered data to the original data if you have access to both, but who wants to do that – especially if the file is large? However, if you import the data into R as-is and programmatically make alterations with R code, then your R code will, by definition, serve a record of all alterations that were made.
- Often data is updated. You could spend a significant amount of time altering your data in Excel only to be sent an updated file next week. Often, the manual alterations you made in one Excel file are not transferable to another. However, if all alterations are made in R, then you can often just run the exact same code again on the updated data.

So, let's walk through addressing these issues together. We'll start by taking a look at our results with all of `read_csv`'s arguments left at their default values.

You may [click here](#) to download this file to your computer.

```
csv <- read_csv("comma_complex.csv")
```

```
New names:
Rows: 6 Columns: 5
-- Column specification
----- Delimiter: ","
(5): Var1...1, Var1...2, Var3, Var4, Notes
i Use `spec()` to retrieve the full column specification for this data. i
```

```
Specify the column types or set `show_col_types = FALSE` to quiet this message.  
* `Var1` -> `Var1...1`  
* `Var1` -> `Var1...2`
```

```
csv
```

```
# A tibble: 6 x 5  
  Var1...1 Var1...2     Var3     Var4   Notes  
  <chr>    <chr>    <chr>    <chr>    <chr>  
1 <NA>      <NA>      <NA>      <NA>      <NA>  
2 Study ID Participant Sex Paticipant Height (in) Participant Weight (lbs) <NA>  
3 1          Male       71        190       <NA>  
4 2          Male       <NA>      176       <NA>  
5 3          Female     64        130       <NA>  
6 4          Female     65        Missing    Call~
```

That is obviously not what we wanted. So, let's start adjusting some of `read_csv()`'s defaults – starting with the column names.

```
csv <- read_csv(  
  file = "comma_complex.csv",  
  col_names = c("id", "sex", "ht_in", "wgt_lbs")  
)
```

```
Rows: 7 Columns: 5  
-- Column specification -----  
Delimiter: ","  
chr (5): id, sex, ht_in, wgt_lbs, X5  
  
i Use `spec()` to retrieve the full column specification for this data.  
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 7 x 5  
  id      sex       ht_in      wgt_lbs      X5  
  <chr>   <chr>    <chr>    <chr>    <chr>  
1 Var1    Var1     Var3      Var4      Notes  
2 <NA>    <NA>     <NA>      <NA>      <NA>  
3 Study ID Participant Sex Paticipant Height (in) Participant Weight (lbs) <NA>  
4 1          Male       71        190       <NA>  
5 2          Male       <NA>      176       <NA>  
6 3          Female     64        130       <NA>  
7 4          Female     65        Missing    Call~
```

Here's what we did above:

- We passed a character vector of variable names to the `col_names` argument. Doing so told R to use the words in the character vector as column names instead of the values in the first row of the raw data (the default).
- Because the character vector of names only contained 4 values, the last column was dropped from the data. R gives us a warning message to let us know. Specially, for each row it says that it was expecting 4 columns (because we gave it 4 column names), but actually found 5 columns. We'll get rid of this message next.

```
csv <- read_csv(  
  file = "comma_complex.csv",  
  col_names = c("id", "sex", "ht_in", "wgt_lbs"),  
  col_types = cols(  
    col_character(),  
    col_character(),  
    col_integer(),  
    col_integer(),  
    col_skip()  
)  
)
```

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,  
e.g.:  
  dat <- vroom(...)  
  problems(dat)
```

`CSV`

```
# A tibble: 7 x 4  
  id      sex      ht_in wgt_lbs  
  <chr>   <chr>     <int>   <int>  
1 Var1    Var1      NA      NA  
2 <NA>    <NA>      NA      NA  
3 Study ID Participant Sex      NA      NA  
4 1       Male       71      190  
5 2       Male       NA      176  
6 3       Female     64      130  
7 4       Female     65      NA
```

Here's what we did above:

- We told R explicitly what type of values we wanted each column to contain. We did so by nesting a `col_` function for each column type inside the `col()` function, which is passed directly to the `col-types` argument.
- You can type `?readr::cols` into your R console to view the help documentation for this function and follow along with the explanation below.
- Notice various column types (e.g., `col_character()`) are *functions*, and that they are nested inside of the `cols()` function. Because they are functions, you must include the parentheses. That's just how the `readr` package is designed.
- Notice that the last column type we passed to the `col_types` argument was `col_skip()`. This tells R to ignore the 5th column in the raw data (5th because it's the 5th column type we listed). Doing this will get rid of the warning we saw earlier.
- You can type `?readr::cols` into your R console to see all available column types.
- Because we told R explicitly what type of values we wanted each column to contain, R had to drop any values that couldn't be coerced to the type we requested. More specifically, they were coerced to missing (`NA`). For example, the value `Var3` that was previously in the first row of the `ht_in` column. It was coerced to `NA` because R does not know (nor do I) how to turn the character string “`Var3`” into an integer. R gives us a warning message about this.

Next, let's go ahead and tell R to ignore the first three rows of the csv file. They don't contain anything that is of use to us at this point.

```
csv <- read_csv(
  file = "comma_complex.csv",
  col_names = c("id", "sex", "ht_in", "wgt_lbs"),
  col_types = cols(
    col_character(),
    col_character(),
    col_integer(),
    col_integer(),
    col_skip()
  ),
  skip = 3
)
```

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,
e.g.:
dat <- vroom(...)
problems(dat)
```

```
csv
```

```
# A tibble: 4 x 4
  id    sex    ht_in wgt_lbs
  <chr> <chr>  <int>   <int>
1 1     Male     71     190
2 2     Male     NA      176
3 3     Female   64     130
4 4     Female   65     NA
```

Here's what we did above:

- We told R to ignore the first three rows of the csv file by passing the value 3 to the `skip` argument.
- The remaining warning above is R telling us that it still had to convert the word "Missing" to an NA in the 4th row of the `wgt_lbs` column because it didn't know how to turn the word "Missing" into an integer. This is actually exactly what we wanted to happen, but we can get rid of the warning by explicitly adding the word "Missing" to the list of values R looks for in the `na` argument.

```
csv <- read_csv(
  file = "comma_complex.csv",
  col_names = c("id", "sex", "ht_in", "wgt_lbs"),
  col_types = cols(
    col_character(),
    col_character(),
    col_integer(),
    col_integer(),
    col_skip()
  ),
  skip = 3,
  na = c("", "NA", "Missing")
)
```

```
csv
```

```
# A tibble: 4 x 4
  id    sex    ht_in wgt_lbs
  <chr> <chr>  <int>   <int>
1 1     Male     71     190
2 2     Male     NA      176
```

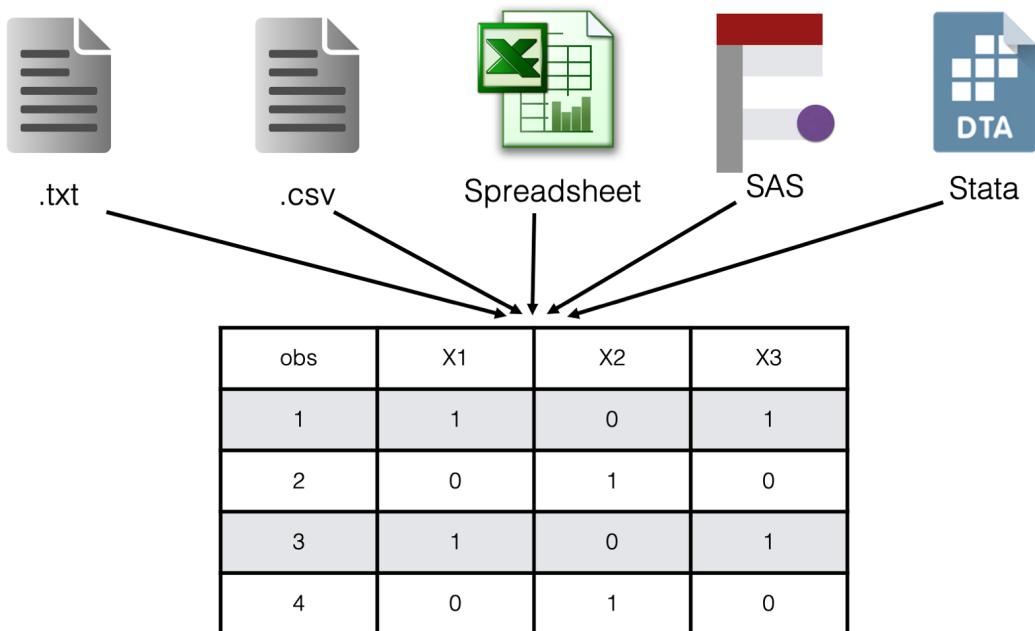
| | | | | |
|---|---|--------|----|-----|
| 3 | 3 | Female | 64 | 130 |
| 4 | 4 | Female | 65 | NA |

Wow! This was kind of a long chapter! But, you should now have the foundation you need to start importing data in R instead of creating data frames manually. At least as it pertains to data that is stored in plain text files. Next, we will learn how to import data that is stored in binary files. Most of the concepts we learned in this chapter will apply, but we will get to use a couple new packages .

15 Importing Binary Files

In the last chapter we learned that there are many different file types that one can use to store data. We also learned how to use the `readr` package to import several different variations of **plain text files** into R.

In this chapter, we will focus on data stored in **binary files**. Again, you can think of binary files as being more complex than plain text files and accessing the information in binary files requires the use of special software. Some examples of binary files that we have frequently seen used in epidemiology include Microsoft Excel spreadsheets, SAS data sets, and Stata data sets. Below, we will learn how to import all three file types into R.



15.1 Packages for importing data

Technically, base R does not contain any functions that can be used to import the binary file types discussed above. However, the `foreign` package contains functions that may be used to import SAS data sets and Stata data sets, and is installed by default when you install R on

your computer. Having said that, we aren't going to use the `foreign` package in this chapter. Instead, we're going to use the following packages to import data in the examples below. If you haven't done so already, we suggest that you go ahead and install these packages now.

- `readxl`. We will use the `readxl` package to import Microsoft Excel files.
- `haven`. We will use the `haven` package to import SAS and Stata data sets.

```
library(readxl)
library(haven)
```

15.2 Importing Microsoft Excel spreadsheets

We probably sent data in Microsoft Excel files more than any other file format. Fortunately, the `readxl` package makes it really easy to import Excel spreadsheets into R. And, because that package is maintained by the same people who create the `readr` package that you have already seen, we think it's likely that the `readxl` package will feel somewhat familiar right from the start.

We would be surprised if any of you had never seen an Excel spreadsheet before – they are pretty ubiquitous in the modern world – but we'll go ahead and show a screenshot of our height and weight data in Excel for the sake of completeness.

| | A | B | C | D | E |
|---|-----|--------|-------|---------|---|
| 1 | ID | sex | ht_in | wgt_lbs | |
| 2 | 001 | Male | 71 | 190 | |
| 3 | 002 | Male | 69 | 176 | |
| 4 | 003 | Female | 64 | 130 | |
| 5 | 004 | Female | 65 | 154 | |
| 6 | | | | | |

All we have to do to import this spreadsheet into R as a data frame is passing the path to the excel file to the `path` argument of the `read_excel()` function.

You may [click here](#) to download this file to your computer.

```
excel <- read_excel("excel.xlsx")
```

```
excel
```

```
# A tibble: 4 x 4
  ID     sex    ht_in wgt_lbs
  <chr> <chr>   <dbl>   <dbl>
1 001   Male     71     190
2 002   Male     69     176
3 003   Female   64     130
4 004   Female   65     154
```

Here's what we did above:

- We used `readxl`'s `read_excel()` function to import a Microsoft Excel spreadsheet. That spreadsheet was imported as a data frame and we assigned that data frame to the R object called `excel`.

 **Warning**

Make sure to always include the file extension in your file paths. For example, using “/excel” instead of “/excel.xlsx” above (i.e., no .xlsx) would have resulted in an error telling you that the file does not exist.

Fortunately for us, just passing the Excel file to the `read_excel()` function like this will usually “just work.” But, let’s go ahead and simulate another situation that is slightly more complex. Once again, we’ve received data from a team that is using Microsoft Excel to capture some study data.

| | A | B | C | D | E | F | G |
|----|-------------------------|-----------------------|-----------------|--------------|---------------|-------------------------|---------------------|
| 1 | Height and Weight Study | | | | | | |
| 2 | Study ID | Assigned Sex at Birth | Height (inches) | Weight (lbs) | Date of Birth | Annual Household Income | Notes |
| 3 | 001 | Male | 71 | 190 | 5/20/81 | \$46,000 | |
| 4 | 002 | Male | | 176 | 8/16/90 | \$67,000 | |
| 5 | 003 | Female | 64 | 130 | 2/21/80 | \$49,000 | |
| 6 | 004 | Female | 65 | Missing | 4/12/83 | \$89,000 | Call back on Monday |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |

◀ ▶
Data Dictionary
Study Phase 1
+

As you can see, this data looks very similar to the csv file we previously imported. However, it looks like the study team has done a little more formatting this time. Additionally, they've added a couple of columns we haven't seen before – date of birth and annual household income.

As a final little wrinkle, the data for this study is actually the second sheet in this Excel file (also called a workbook). The study team used the first sheet in the workbook as a data dictionary that looks like this:

| A | B | C | D |
|----|-------------------------|--|---------------------------|
| 1 | Height and Weight Study | | |
| 2 | Data Dictionary | | |
| 3 | Variable | Definition | Type |
| 4 | Study ID | Randomly assigned participant id | Continuous |
| 5 | Assigned Sex at Birth | Sex the participant was assigned at birth | Dichotomous (Female/Male) |
| 6 | Height (inches) | Participant's height in inches | Continuous |
| 7 | Weight (lbs) | Participant's weight in pounds | Continuous |
| 8 | Date of Birth | Participant's date of birth | Date |
| 9 | Annual Household Income | Participant's annual household income from all sources | Continuous (Currency) |
| 10 | | | |
| 11 | | | |
| 12 | | | |

◀ ▶
Data Dictionary
Study Phase 1
+

Once again, we will have to deal with some of the formatting that was done in Excel before we can analyze our data in R.

You may [click here to download this file to your computer](#).

We'll start by taking a look at the result we get when we try to pass this file to the `read_excel()` function without changing any of `read_excel()`'s default values.

```
excel <- read_excel("excel_complex.xlsx")
```

New names:

```
* ` ` -> `...2`  
* ` ` -> `...3`
```

```
excel
```

```
# A tibble: 8 x 3
`Height and Weight Study\r\nData Dictionary` ...2      ...3
<chr>                <chr>                <chr>
1 <NA>                <NA>                <NA>
2 Variable            Definition           Type
3 Study ID            Randomly assigned particip~ Cont~
4 Assigned Sex at Birth Sex the participant was as~ Dich~
```

| | |
|---------------------------|---------------------------------------|
| 5 Height (inches) | Participant's height in inches |
| 6 Weight (lbs) | Participant's weight in pounds |
| 7 Date of Birth | Participant's date of birth |
| 8 Annual Household Income | Participant's annual household income |

And, as we're sure you saw coming, this isn't the result we wanted. However, we can get the result we wanted by making a few tweaks to the default values of the `sheet`, `col_names`, `col_types`, `skip`, and `na` arguments of the `read_excel()` function.

```
excel <- read_excel(
  path = "excel_complex.xlsx",
  sheet = "Study Phase 1",
  col_names = c("id", "sex", "ht_in", "wgt_lbs", "dob", "income"),
  col_types = c(
    "text",
    "text",
    "numeric",
    "numeric",
    "date",
    "numeric",
    "skip"
  ),
  skip = 3,
  na = c("", "NA", "Missing")
)
```

excel

```
# A tibble: 4 x 6
  id     sex   ht_in wgt_lbs dob           income
  <chr> <chr>  <dbl>   <dbl> <dttm>        <dbl>
1 001   Male     71     190  1981-05-20 00:00:00  46000
2 002   Male     NA     176  1990-08-16 00:00:00  67000
3 003   Female   64     130  1980-02-21 00:00:00  49000
4 004   Female   65     NA   1983-04-12 00:00:00  89000
```

As we said, the `readr` package and `readxl` package were developed by the same people. So, the code above looks similar to the code we used to import the csv file in the previous chapter. Therefore, we're not going to walk through this code step-by-step. Rather, we're just going to highlight some of the slight differences.

- You can type `?read_excel` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `read_excel()` function is the `path` argument. It serves the same purpose as the `file` argument to `read_csv()` – it just has a different name.
- The `sheet` argument to the `read_excel()` function tells R which sheet of the Excel workbook contains the data you want to import. In this case, the study team named that sheet “Study Phase 1”. We could have also passed the value 2 to the `sheet` argument because “Study Phase 1” is the second sheet in the workbook. However, we suggest using the sheet name. That way, if the study team sends you a new Excel file next week with different ordering, you are less likely to accidentally import the wrong data.
- The value we pass to the `col_types` argument is now a vector of character strings instead of a list of functions nested in the `col()` function.
 - The values that the `col_types` function will accept are `"skip"` for telling R to ignore a column in the spreadsheet, `"guess"` for telling R to guess the variable type, `"logical"` for logical (TRUE/FALSE) variables, `"numeric"` for numeric variables, `"date"` for date variables, `"text"` for character variables, and `"list"` for everything else.
 - Notice that we told R to import income as a numeric variable. This caused the commas and dollar signs to be dropped. We did this because keeping the commas and dollar signs would have required us to make income a character variable (numeric variables can only include numbers). If we had imported income as a character variable, we would have lost the ability to perform mathematical operations on it. Remember, it makes no sense to “add” two words together. Later, we will show you how to add dollar signs and commas back to the numeric values if you want to display them in your final results.
- We used the `col_names`, `skip`, and `na` arguments in exactly the same way we used them in the `read_csv` function.

You should be able to import most of the data stored in Excel spreadsheets with just the few options that we discussed above. However, there may be times where importing spreadsheets is even more complicated. If you find yourself in that position, we suggest that you first check out [the readxl website here](#).

15.3 Importing data from other statistical analysis software

Many applications designed for statistical analysis allow you to save data in a binary format. One reason for this is that binary data formats allow you to save **metadata** alongside your data values. Metadata is data *about* the data. Using our running example, the data is about

the heights, weights, and other characteristics of our study participants. **Metadata** about this data might include information like when this data set was created, or value labels that make the data easier to read (e.g., the dollar signs in the income variable).

In our experience, you are slightly more likely to have problems importing binary files saved from other statistical analysis applications than plain text files. Perhaps because they are more complex, the data just seems to become corrupt and do other weird things more often than is the case with plain text files. However, in our experience, it is also the case that when we are able to import binary files created in other statistical analysis applications, doing so requires less adjusting of default values. In fact, we will usually only need to pass the file path to the correct `read_` function.

Below, we will see some examples of importing binary files saved in two popular statistical analysis applications – SAS and Stata. We will use the `haven` package to import both.

15.4 Importing SAS data sets

SAS actually allows users to save data in more than one type of binary format. Data can be saved as SAS data sets or as SAS Transport files. SAS data set file names end with the `.sas7bdat` file extension. SAS Transport file file names end with the `.xpt` file extension.

In order to import a SAS data set, we typically only need to pass the correct file path to `haven`'s `read_sas()` function.

You may [click here](#) to download this file to your computer.

```
sas <- read_sas("height_and_weight.sas7bdat")
```

```
sas
```

```
# A tibble: 4 x 4
  ID     sex   ht_in wgt_lbs
  <chr> <chr>  <dbl>   <dbl>
1 001   Male    71     190
2 002   Male    69     176
3 003   Female  64     130
4 004   Female  65     154
```

Here's what we did above:

- We used `haven`'s `read_sas()` function to import a SAS data set. That data was imported as a data frame and we assigned that data frame to the R object called `sas`.

In addition to SAS data sets, data that has been altered in SAS can also be saved as a SAS transport file. Some of the national, population-based public health surveys (e.g., BRFSS and NHANES) make their data publicly available in this format.

You can download the [2018 BRFSS data as a SAS Transport file here](#). About halfway down the webpage, there is a link that says, “2018 BRFSS Data (SAS Transport Format)”.

Data Files

There are 437,436 records for 2018. More information on participation is available in the [states conducting surveillance, by year table](#). The data files are provided in ASCII and SAS Transport formats. The November update includes the addition of E-Cigarettes optional module data from California and a correction for the Lung Cancer Screening optional module variable LCSLAST in two states (MD, TX).

[2018 BRFSS Data \(ASCII\)](#) [ZIP – 66.2 MB]
November, 2019
This file for the combined landline and cell phone data set is in ASCII format. It has a fixed record length of 2033 positions.

[2018 BRFSS Data \(SAS Transport Format\)](#) [ZIP – 101 MB] 
November, 2019
This file for the combined landline and cell phone data set was exported from SAS V9.3 in the XPT transport format. This file contains 275 variables. This format can be imported into SPSS or STATA. Please note: some of the variable labels get truncated in the process of converting to the XPT format so they may be slightly different from what is on the SASOUT18.SAS program.

[Variable Layout](#)
Format information on variable name by column position.

[The Combined Landline and Cellular Telephone Survey](#)
[Multiple Questionnaire Version Data–includes Optional Modules](#)
The combined landline and cellular telephone multiple

Clicking that link should download the data to your computer. Notice that the SAS Transport file is actually stored *inside* a zip file. You can unzip the file first if you would like, but you don't even have to do that. Amazingly, you can pass the path to the zipped .xpt file directly to the `read_xpt()` function like so:

```
brfss_2018 <- read_xpt("LLCP2018XPT.zip")
```

```
head(brfss_2018)
```

```
# A tibble: 6 x 275
#>   `_STATE`  FMONTH IDATE     IMONTH IDAY    IYEAR DISPCODE SEQNO      `_PSU` CTELENM1
#>   <dbl>    <dbl> <chr>     <chr>  <chr>    <chr>  <dbl> <chr>    <dbl>    <dbl>
#> 1       1      1 01052018 01      05    2018      1100 20180000~ 2.02e9     1
#> 2       1      1 01122018 01      12    2018      1100 20180000~ 2.02e9     1
#> 3       1      1 01082018 01      08    2018      1100 20180000~ 2.02e9     1
#> 4       1      1 01032018 01      03    2018      1100 20180000~ 2.02e9     1
```

```

5      1      1 01122018 01      12      2018      1100 20180000~ 2.02e9      1
6      1      1 01112018 01      11      2018      1100 20180000~ 2.02e9      1
# i 265 more variables: PVTRESID1 <dbl>, COLGHOUS <dbl>, STATERE1 <dbl>,
# CELLFON4 <dbl>, LADULT <dbl>, NUMADULT <dbl>, NUMMEN <dbl>, NUMWOMEN <dbl>,
# SAFETIME <dbl>, CTELNUM1 <dbl>, CELLFON5 <dbl>, CADULT <dbl>,
# PVTRESID3 <dbl>, CCLGHOUS <dbl>, CSTATE1 <dbl>, LANDLINE <dbl>,
# HHADULT <dbl>, GENHLTH <dbl>, PHYSHLTH <dbl>, MENTHLTH <dbl>,
# POORHLTH <dbl>, HLTHPLN1 <dbl>, PERSDOC2 <dbl>, MEDCOST <dbl>,
# CHECKUP1 <dbl>, EXERANY2 <dbl>, SLEPTIM1 <dbl>, CVDINFR4 <dbl>, ...

```

Here's what we did above:

- We used `haven`'s `read_xpt()` function to import a zipped SAS Transport File. That data was imported as a data frame and we assigned that data frame to the R object called `brfss_2018`.
- Because this is a large data frame (437,436 observations and 275 variables), we used the `head()` function to print only the first 6 rows of the data to the screen.

But, this demonstration actually gets even cooler. Instead of downloading the SAS Transport file to our computer before importing it, we can actually sometimes import files, including SAS Transport files, directly from the internet.

For example, you can download the [2017-2018 NHANES demographic data as a SAS Transport file here](#)

The screenshot shows the CDC National Center for Health Statistics website. The main navigation bar includes links for CDC, NCHS, National Health and Nutrition Examination Survey, Questionnaires, Datasets, and Related Documentation, and NHANES 2017-2018. The left sidebar has a menu with items like About NHANES, What's New, Questionnaires, Datasets, and Related Documentation (which is expanded to show Survey Methods and Analytic Guidelines, Search Variables, Frequently Asked Questions, All Continuous NHANES, NHANES 2019-2020, and NHANES 2017-2018). The right side features the NHANES logo and the title "National Health and Nutrition Examination Survey". Below the title is the heading "NHANES 2017-2018 Demographics Data". A table lists the data file information:

| Data File Name | Doc File | Data File | Date Published |
|--|----------------------------|--|----------------|
| Demographic Variables and Sample Weights | DEMO_J_Doc | DEMO_J_Data.XPT - 3.3 MB | February 2020 |

At the bottom right of the page, there is a note: "Page last reviewed: 2/21/2020" and "Content source: CDC/National Center for Health Statistics".

If you right-click on the link that says, “DEMO_I Data [XPT - 3.3 MB]”, you will see an option to copy the link address.

The screenshot shows a table of data files from the NHANES 2017-2018 Demographics Data page. One row is selected, showing "Demographic Variables and Sample Weights" as the Data File Name, "DEMO_J Doc" as the Doc File, and "DEMO_J Data [XPT - 3.3 MB]" as the Data File. A context menu is open over the link "DEMO_J Data [XPT - 3.3 MB]". The menu includes options like "Open Link in New Tab", "Send Link to SPH7729", "Save Link As...", "Copy Link Address" (which is highlighted with a red arrow), and "Copy". Below the main menu, there are additional options for "JSONView", "Paperpile", "Inspect", and "Speech Services".

| Data File Name | Doc File | Data File | Date Published |
|--|------------|----------------------------|----------------|
| Demographic Variables and Sample Weights | DEMO_J Doc | DEMO_J Data [XPT - 3.3 MB] | |

Click “Copy Link Address” and then navigate back to RStudio. Now, all you have to do is paste that link address where you would normally type a file path into the `read_xpt()` function. When you run the code chunk, the `read_xpt()` function will import the NHANES data directly from the internet (assuming you are connected to the internet).

```
nhanes_demo <- read_xpt("https://www.cdc.gov/Nchs/Nhanes/2017-2018/DEMO_J.XPT")
```

```
head(nhanes_demo)
```

```
# A tibble: 6 x 46
  SEQN SDDSRVYR RIDSTATR RIAGENDR RIDAGEYR RIDAGEMN RIDRETH1 RIDRETH3 RIDEXMON
  <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 93703      10       2       2       2      NA      5       6       2
2 93704      10       2       1       2      NA      3       3       1
3 93705      10       2       2      66      NA      4       4       2
4 93706      10       2       1      18      NA      5       6       2
5 93707      10       2       1      13      NA      5       7       2
6 93708      10       2       2      66      NA      5       6       2
# i 37 more variables: RIDEXAGM <dbl>, DMQMILIZ <dbl>, DMQADFC <dbl>,
#   DMDBORN4 <dbl>, DMDCITZN <dbl>, DMDYRSUS <dbl>, DMDEDUC3 <dbl>,
```

```
# DMDEDUC2 <dbl>, DMDMARTL <dbl>, RIDEXPRG <dbl>, SIALANG <dbl>,
# SIAPROXY <dbl>, SIAINTRP <dbl>, FIALANG <dbl>, FIAPROXY <dbl>,
# FIAINTRP <dbl>, MIALANG <dbl>, MIAPROXY <dbl>, MIAINTRP <dbl>,
# AIALANGA <dbl>, DMDHHSIZ <dbl>, DMDFMSIZ <dbl>, DMDHHSZA <dbl>,
# DMDHHSZB <dbl>, DMDHHSZE <dbl>, DMDHRGND <dbl>, DMDHRAGZ <dbl>, ...
```

Here's what we did above:

- We used `haven`'s `read_xpt()` function to import a SAS Transport File directly from the NHANES website. That data was imported as a data frame and we assigned that data frame to the R object called `nhanes_demo`.
- Because this is a large data frame (9,254 observations and 46 variables), we used the `head()` function to print only the first 6 rows of the data to the screen.

15.5 Importing Stata data sets

Finally, we will import a Stata data set (.dta) to round out our discussion of importing data from other statistical analysis software packages. There isn't much of anything new here – you could probably have even guessed how to do this without us showing you.

You may [click here to download this file to your computer](#).

```
stata <- read_stata("height_and_weight.dta")  
  
stata  
  
# A tibble: 4 x 4  
  ID     sex    ht_in wgt_lbs  
  <chr> <chr>   <dbl>   <dbl>  
1 001   Male     71     190  
2 002   Male     69     176  
3 003   Female   64     130  
4 004   Female   65     154
```

Here's what we did above:

- We used `haven`'s `read_stata()` function to import a Stata data set. That data was imported as a data frame and we assigned that data frame to the R object called `stata`.

You now know how to write code that will allow you to import data stored in all of the file formats that we will use in this book, and the vast majority of formats that you are likely to encounter in your real-world projects. In the next section, We will introduce you to a tool in RStudio that makes importing data even easier.

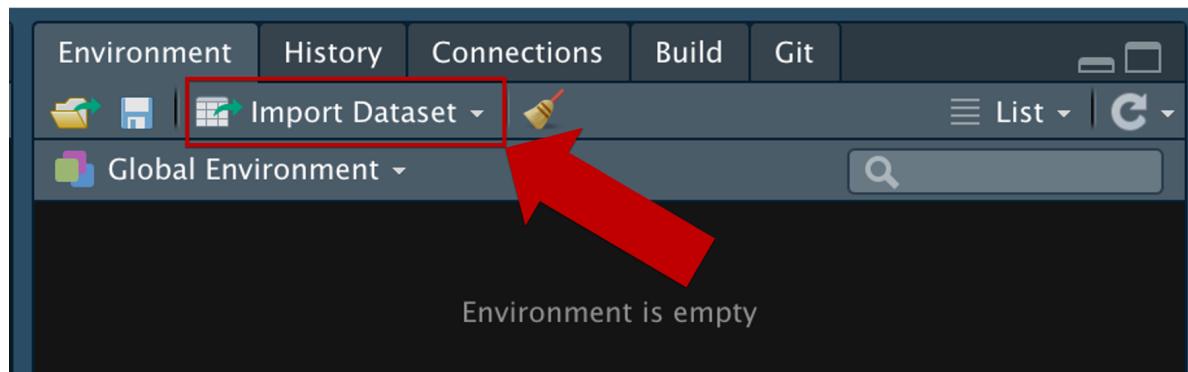
16 RStudio's Data Import Tool

In previous chapters, we learned how to programmatically import data into R. In this chapter, we will briefly introduce you to RStudio's data import tool. Conceptually, we won't be introducing anything you haven't already seen before. We just want to make you aware of this tool, which can be a welcomed convenience at times.

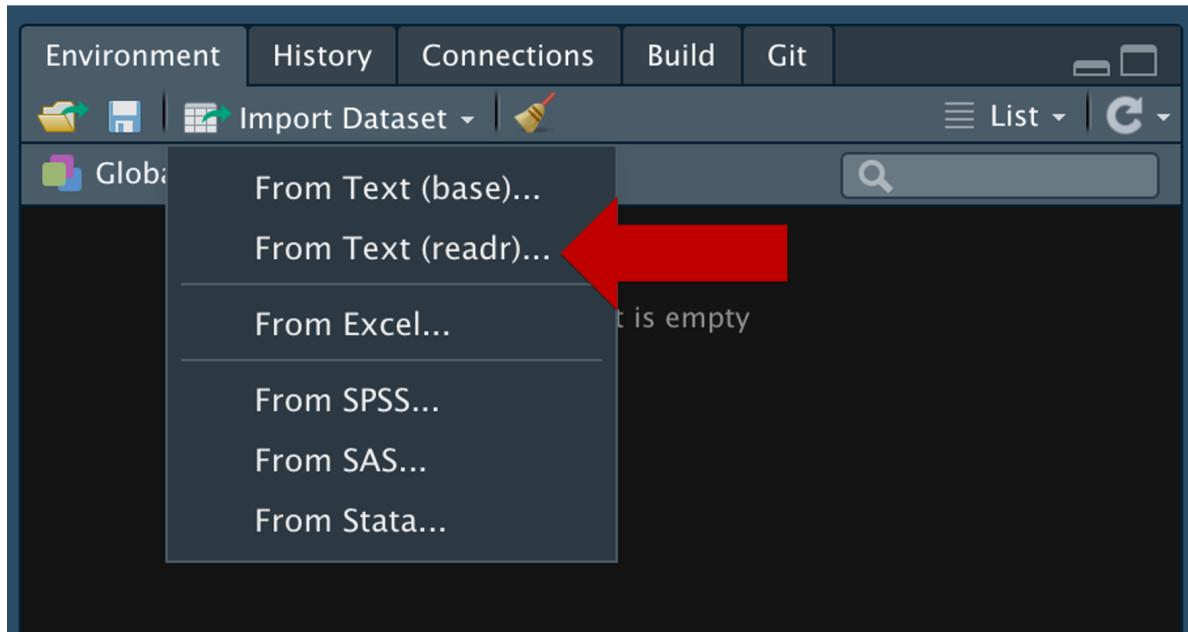
For this example, we will use the import tool to help us import the same height and weight csv file we imported in the [chapter on importing plain text files](#).

[You may click here to download this file to your computer.](#)

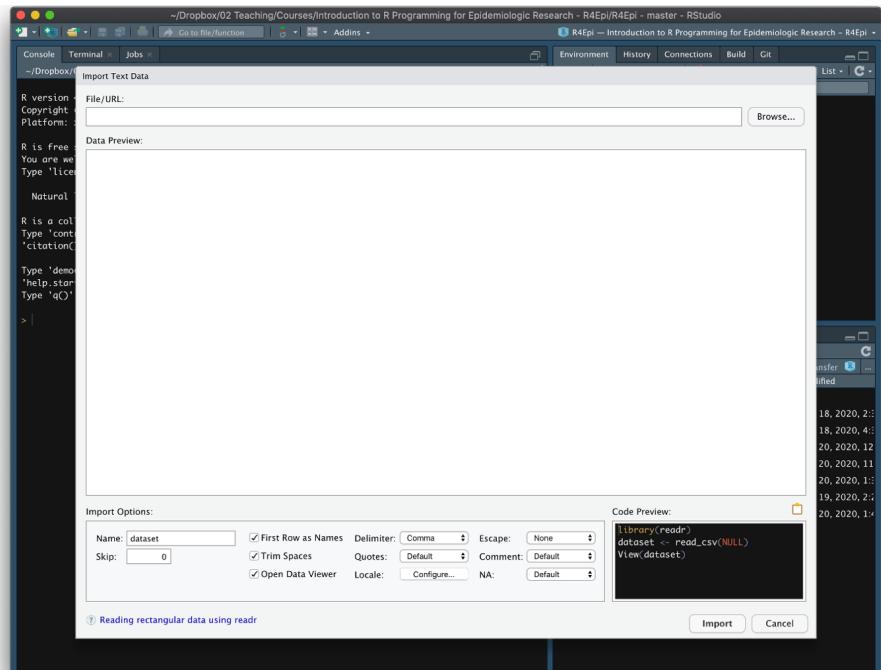
To open RStudio's data import tool, click the `Import Dataset` dropdown menu near the top of the environment pane.



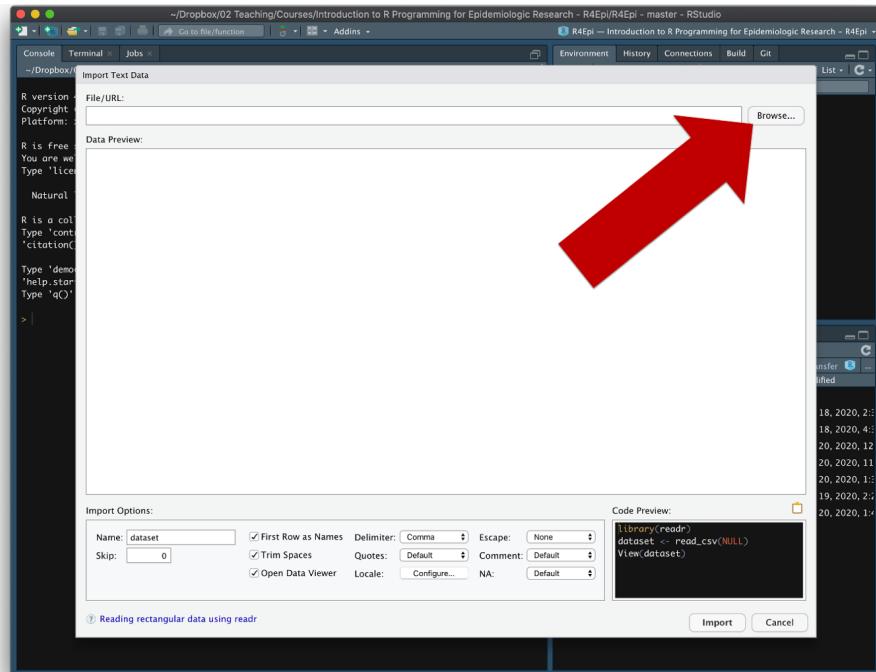
Next, because this is a csv file, we will choose the `From Text (readr)` option from the dropdown menu. The difference between `From Text (base)` and `From Text (readr)` is that `From Text (readr)` will use functions from the `readr` package to import the data and `From Text (base)` will use base R functions to import the data.



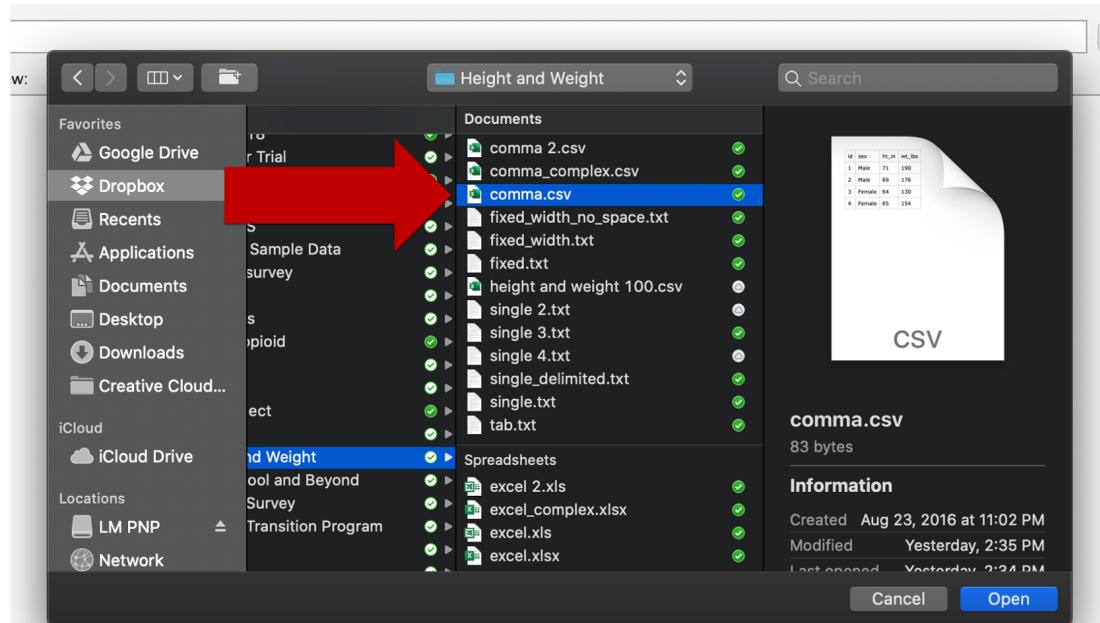
After you select a file type from the import tool dropdown menu, a separate data import window will open.



At this point, you should click the **browse** button to locate the file you want to import.

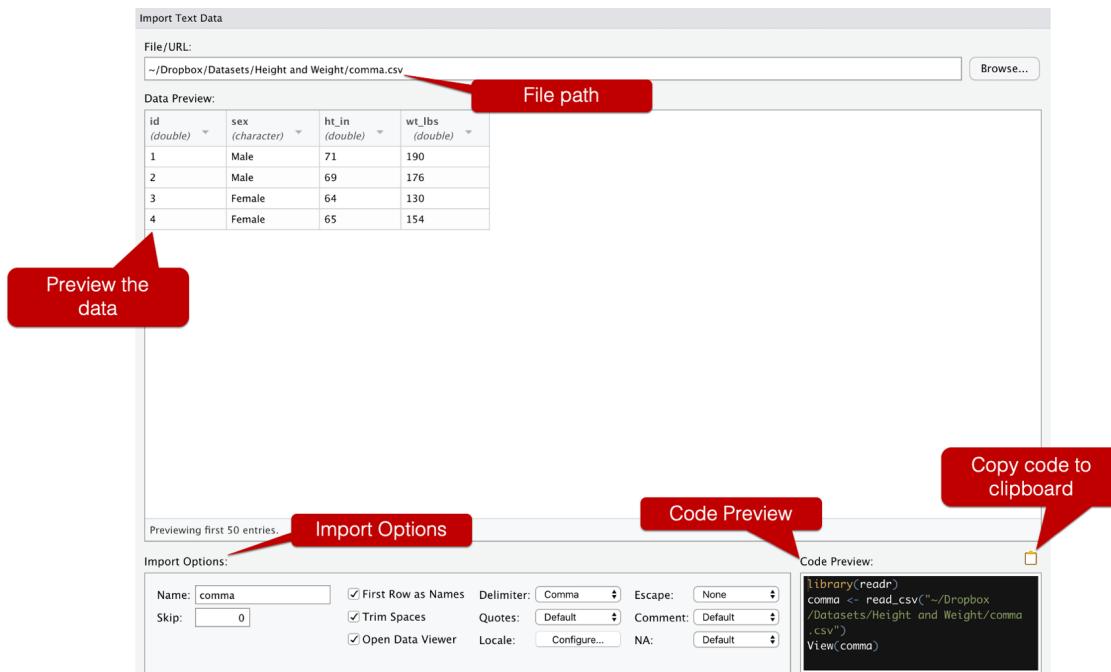


Doing so will open your operating system's file explorer window. Use that window to find and select the file you want to import. Again, we are using comma.csv for this demonstration.



After selecting your file, there will be some changes in the data import window. Specifically,

- The file path to the raw data you are importing will appear in the **File/URL** field.
- A preview of how R is currently parsing that data will appear in the **Data Preview** field.
- Some or all of the import options will become available for you to select or deselect.
- The underlying code that R is currently using to import this data is displayed in the **Code Preview** window.
- The copy to clipboard icon becomes clickable.



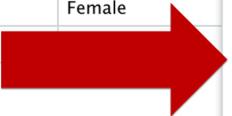
Importing this simple data set doesn't require us to alter many of the import options. However, we do want to point out that you can change the variable type by clicking in the column headers in the **Data Preview** field. After clicking, a dropdown menu will display that allows you to change variable types. This is equivalent to adjusting the default values passed to the `col_types` argument of the `read_csv()` function.

We will go ahead and change the `ht_in` and `wgt_lbs` variables from type double to type integer using the dropdown menu.

Data Preview:

| <code>id</code>
(<i>double</i>) | <code>sex</code>
(<i>character</i>) | <code>ht_in</code>
(<i>double</i>) | <code>wt_lbs</code>
(<i>double</i>) |
|--------------------------------------|--|---|--|
| 1 | Male | Guess | column 3: numeric
with range 64 - 72
170 |
| 2 | Male | Character | |
| 3 | Female | Double | 130 |
| 4 | | Integer | 154 |

Numeric
Logical
Date
Time
DateTime
Factor
Include
Skip
Only



At this point, our data is ready for import. You can simply press the **Import** button in the bottom-right corner of the data import window. However, we are going to suggest that you don't do that. Instead, we're going to suggest that you click the clipboard icon to copy the code displayed in the **Code Preview** window and then click the **Cancel** button.

Next, return to your R script or Quarto file and paste the code that was copied to your clipboard. At this point, you can run the code as though you wrote it. More importantly, this code is now a part of the record of how you conducted your data analysis. Further, if someone sends you an updated raw data set, you may only need to update the file path in your code instead of clicking around the data import tool again.

Code Preview:



```
library(readr)
comma <- read_csv("~/Dropbox
/Datasets/Height and Weight/comma
.csv",
  col_types = cols(ht_in =
col_integer(),
```

Cancel

That concludes the portion of the book devoted to importing data. In the next chapter, we will discuss strategies for exporting data so that you can store it in a more long-term way and/or share it with others.

17 Exporting Data

The data frames we've created so far don't currently live in our global environment from one programming session to the next because we haven't yet learned how to efficiently store our data long-term. This limitation makes it difficult to share our data with others or even to come back later to modify or analyze our data ourselves. In this chapter, you will learn to **export** data from R's memory to a file on your hard drive so that you may efficiently store it or share it with others. In the examples that follow, we're going to use this simulated data.

```
demo <- data.frame(  
  id  = c("001", "002", "003", "004"),  
  age = c(30, 67, 52, 56),  
  edu = c(3, 1, 4, 2)  
)
```

Here's what we did above:

- We created a data frame that is meant to simulate some demographic information about 4 hypothetical study participants.
- The first variable (`id`) is the participant's study id.
- The second variable (`age`) is the participant's age at enrollment in the study.
- The third variable (`edu`) is the highest level of formal education the participant completed. Where:
 - 1 = Less than high school
 - 2 = High school graduate
 - 3 = Some college
 - 4 = College graduate

17.1 Plain text files

Most of `readr`'s `read_` functions that were introduced in the [importing plain text files](#) chapter have a `write_` counterpart that allow you to export data from R into a plain text file.

Additionally, all of `haven`'s `read_` functions that were introduced in the [importing binary files](#) chapter have a `write_` counterpart that allow you to export data from R into SAS, Stata, and SPSS binary file formats.

Interestingly, `readxl` does not have a `write_excel()` function for exporting R data frames as .xls or .xlsx files. However, the importance of this is mitigated by the fact that Excel can open .csv files and `readr` contains a function (`write_csv()`) for exporting data frames in the .csv file format. If you absolutely have to export your data frame as a .xls or .xlsx file, there are other R packages capable of doing so (e.g., `xlsx`).

So, with all these options what format should you choose? our answer to this sort of depends on the answers to two questions. First, will this data be shared with anyone else? Second, will we need any of the metadata that would be lost if we export this data to a plain text file?

Unless you have a compelling reason to do otherwise, we're going to suggest that you always export your R data frames as csv files if you plan to share your data with others. The reason is simple. They just work. we can think of many times when someone sent me a SAS or Stata data set and we wasn't able to import it for some reason or the data didn't import in the way that we expected it to. we don't recall ever having that experience with a csv file. Further, every operating system and statistical analysis software application that we're aware of is able to accept csv files. Perhaps for that reason, they have become the closest thing to a standard for data sharing that exists – at least that we're aware of.

Exporting an R data frame to a csv file is really easy. The example below shows how to export our simulated demographic data to a csv file on our computer's desktop:

```
readr::write_csv(demo, "demo.csv")
```

Here's what we did above:

- We used `readr`'s `write_csv()` function to export a data frame called `demo` in our global environment to a csv file on our desktop called `demo.csv`.
- You can type `?write_csv` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `write_csv()` function is the `x` argument. The value passed to the `x` argument should be a data frame that is currently in our global environment.
- The second argument to the `write_csv()` function is the `path` argument. The value passed to the `path` should be a file path telling R where to create the new csv file.

- You name the csv file directly in the file path. Whatever name you write after the final slash in the file path is what the csv file will be named.
- As always, make sure you remember to include the file extension in the file path.

Even if you don't plan on sharing your data, there is another benefit to saving your data as a csv file. That is, it's easy to open the file and take a quick peek if you need to for some reason. You don't have to open R and load the file. You can just find the file on your computer, double-click it, and quickly view it in your text editor or spreadsheet application of choice.

However, there is a downside to saving your data frames to a csv file. In general, csv files don't store any metadata, which can sometimes be a problem (or at least a pain). For example, if you've coerced several variables to factors, that information would not be preserved in the csv file. Instead, the factors will be converted to character strings. If you need to preserve metadata, then you may want to save your data frames in a binary format.

17.2 R binary files

In the chapter on [importing binary files](#) we mentioned that most statistical analysis software allows you to save your data in a binary file format. The primary advantage to doing so is that potentially useful metadata is stored alongside your analysis data. We were first introduced to factor vectors in [Let's Get Programming](#) chapter. There, we saw how coercing some of your variables to factors can be useful. However, doing so requires R to store metadata along with the analysis data. That metadata would be lost if you were to export your data frame to a plain text file. This is an example of a time when we may want to consider exporting our data to a binary file format.

R actually allows you to save your data in multiple different binary file formats. The two most popular are the .Rdata format and the .Rds format. We're going to suggest that you use the .Rds format to save your R data frames. Exporting to this format is really easy with the `readr` package.

The example below shows how to export our simulated demographic data to an .Rds file on our computer's desktop:

```
readr::write_rds(demo, "demo.rds")
```

Here's what we did above:

- We used `readr`'s `write_rds()` function to export a data frame called `demo` in our global environment to an .Rds file on our desktop called `demo.rds`.
- You can type `?write_rds` into your R console to view the help documentation for this function and follow along with the explanation below.

- The first argument to the `write_rds()` function is the `x` argument. The value passed to the `x` argument should be a data frame that is currently in our global environment.
- The second argument to the `write_csv()` function is the `path` argument. The value passed to the `path` should be a file path telling R where to create the new .Rds file.
 - You name the .Rds file directly in the file path. Whatever name you write after the final slash in the file path is what the .Rds file will be named.
 - As always, make sure you remember to include the file extension in the file path.

To load the .Rds data back into your global environment, simply pass the path to the .Rds file to `readr::read_rds()` function:

```
demo <- readr::read_rds("demo.rds")
```

There is a final thought we want to share on exporting data frames. When we got to the end of this chapter, it occurred to me that the way we wrote it may give the impression that that you must choose to export data frames as plain text files *or* binary files, but not *both*. That isn't the case. we frequently export our data as a csv file that we can easily open and view and/or share with others, but *also* export it to an .Rds file that retains useful metadata we might need the next time we return to our analysis. we suppose there could be times that your files are so large that this is not an efficient strategy, but that is generally not the case in our projects.

Part IV

Descriptive Analysis

18 Introduction to Descriptive Analysis

18.1 What is descriptive analysis and why would we do it?

So, we have all this data that tells us all this information about different traits or characteristics of the people for whom the data was collected. For example, if we collected data about the students in this course, we may have information about how tall you are, about what kind of insurance you have, and about what your favorite color is.

But, unless you're a celebrity, or under investigation for some reason, it's unlikely that many people outside of your friends and family care to know any of this information about you, *per se*. Usually they want to know this information about the typical person in the population, or subpopulation, to which you belong. Or, they want to know more about the *relationship* between people who are like you in some way and some outcome that they are interested in.

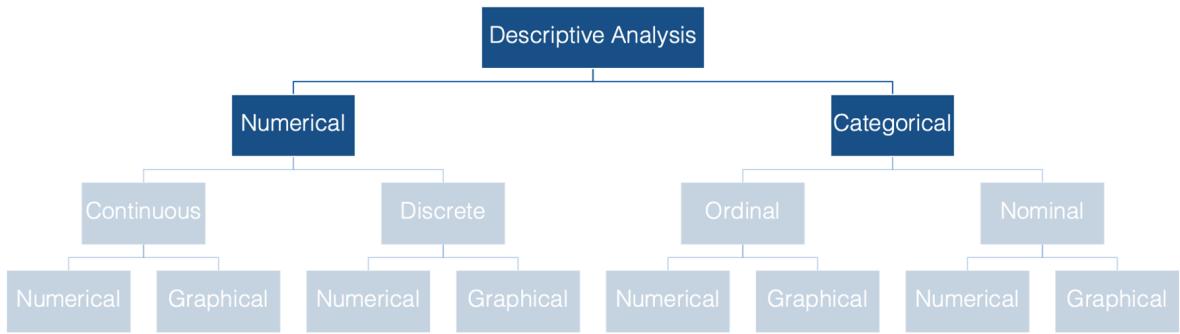
For example: We typically aren't interested in knowing that student 1002 (above) is 67.93 inches tall. We are typically more interested in knowing things like the average height of the class – [`r mean(height_in) |> round(2)`].

Before we can make any inferences or draw any conclusions, we must (or at least should) begin by conducting descriptive analysis of our data. This is also sometimes referred to as exploratory analysis. There are at least three reasons why we want to start with a descriptive analysis:

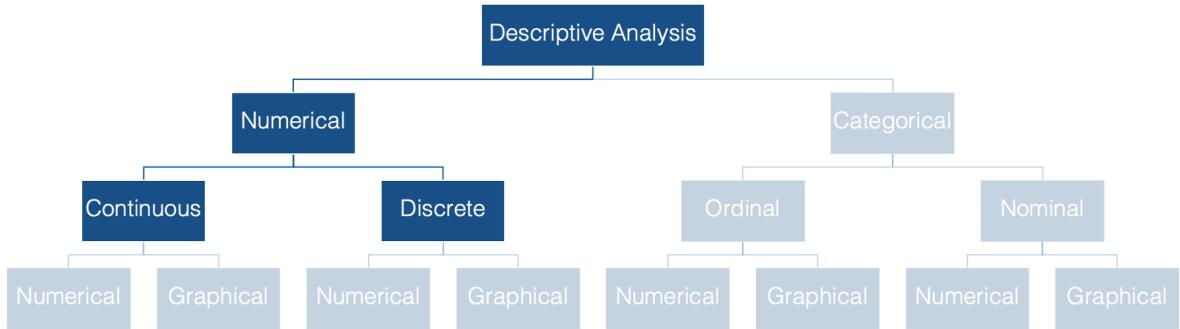
1. *We can use descriptive analysis to uncover errors in our data.*
2. *It helps us understand the distribution of values in our variables.*
3. *Descriptive analysis serve as a starting point for understanding relationships between our variables.*

18.2 What kind of descriptive analysis should we perform?

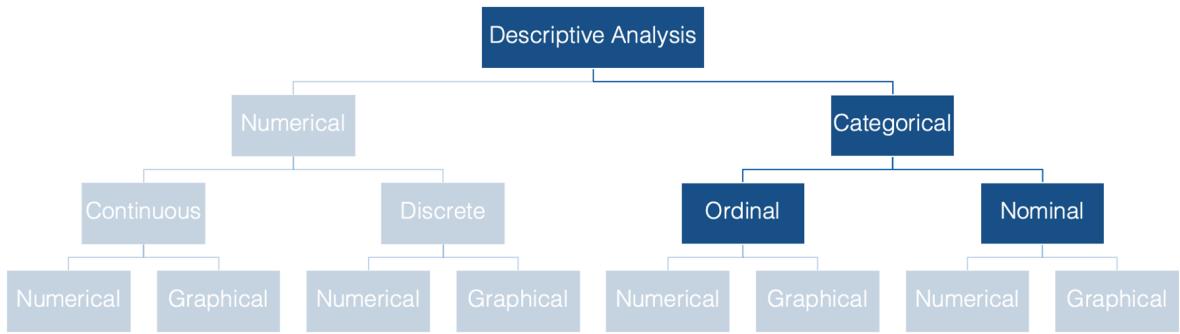
When conducting descriptive analysis, the method you choose will depend on the *type* of data you're analyzing. At the most basic level, variables can be described as numerical or categorical.



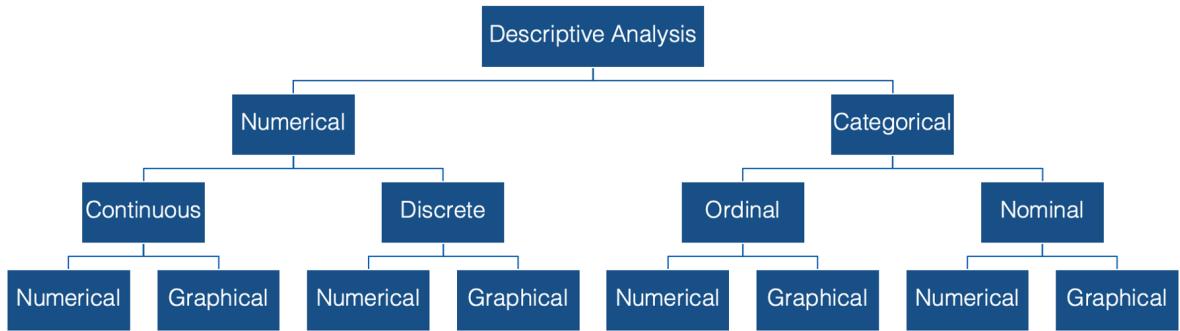
Numeric variables can then be further divided into continuous and discrete - the distinction being whether the variable can take on a continuum of values, or only set of certain values.



Categorical variables can be subdivided into ordinal or nominal variables - depending on whether or not the categories can logically be ordered in a meaningful way.



Finally, for all types, and subtypes, of variables there are both numerical and graphical methods we can use for descriptive analysis.

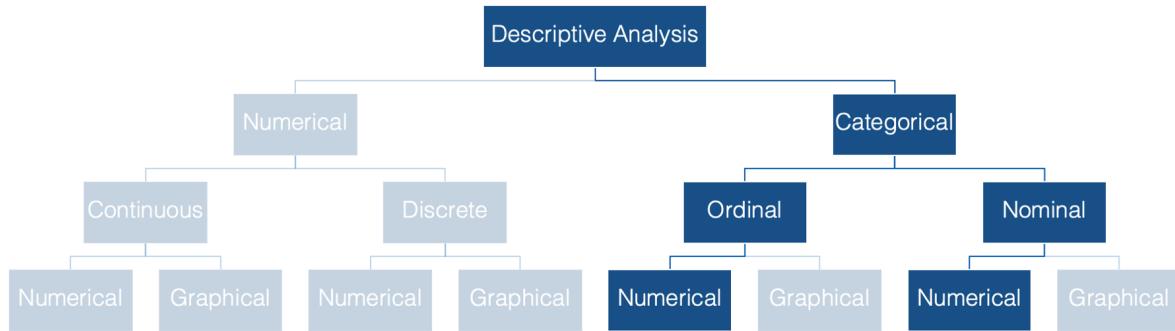


In the exercises that follow you will be introduced to measures of frequency, measures of central tendency, and measures of dispersion. Then, you'll learn various methods for estimating and

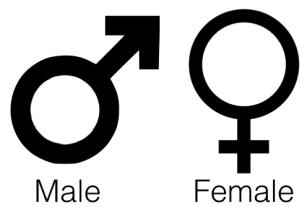
interpreting these measures using R.

19 Numerical Descriptions of Categorical Variables

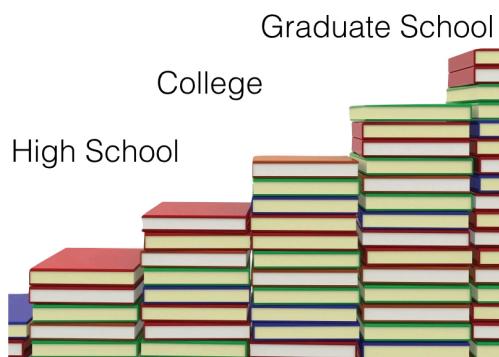
We'll begin our discussion of descriptive statistics in the categorical half of our flow chart. Specifically, we'll start by numerically describing categorical variables. As a reminder, categorical variables are variables whose values fit into categories.



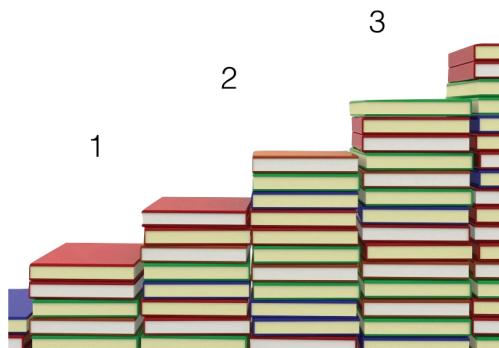
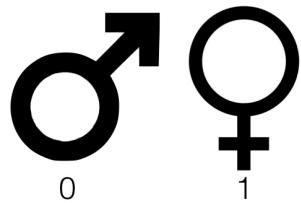
Some examples of categorical variables commonly seen in public health data are: sex, race or ethnicity, and level of educational attainment.



Asian African-American White



Notice that there is no inherent numeric value to any of these categories. Having said that, we can, and often will, assign a numeric value to each category using R.



The two most common numerical descriptions of categorical variables are probably the **frequency count** (you will often hear this referred to as simply the **frequency**, the **count**, or

the **n**) and the **proportion** or **percentage** (the percentage is just the proportion multiplied by 100).



| | Count / n | Proportion / Percent |
|------------------|-----------|----------------------|
| Asian | 2 | 0.4
40% |
| African-American | 1 | 0.2
20% |
| White | 2 | 0.4
40% |

The count is simply the number of observations, in this case people, which fall into each possible category.

The proportion is just the count divided by the total number of observations. In this example, 2 people out of 5 people (.40 or 40%) are in the Asian race category.

The remainder of this chapter is devoted to learning how to calculate frequency counts and percentages using R.

19.1 Factors

We first learned about factors in the [Let's Get Programming](#) chapter. Before moving on to calculating frequency counts and percentages, we will discuss factors in slightly greater depth here. As a reminder, factors can be useful for representing categorical data in R. To demonstrate, let's simulate a simple little data frame.

```
# Load dplyr for tibble()  
library(dplyr)
```

```
demo <- tibble(  
  id = c("001", "002", "003", "004"),  
  age = c(30, 67, 52, 56),  
  edu = c(3, 1, 4, 2)  
)
```

Here's what we did above:

- We created a data frame that is meant to simulate some demographic information about 4 hypothetical study participants.
- The first variable (`id`) is the participant's study id.
- The second variable (`age`) is the participant's age at enrollment in the study.
- The third variable (`edu`) is the highest level of formal education the participant completed. Where:
 - 1 = Less than high school
 - 2 = High school graduate
 - 3 = Some college
 - 4 = College graduate

Each participant in our data frame has a value for `edu` – 1, 2, 3, or 4. The value they have for that variable corresponds to the highest level of formal education they have completed, which is split up into categories that we defined. We can see which category each person is in by viewing the data.

```
demo
```

```
# A tibble: 4 x 3  
  id      age   edu  
  <chr> <dbl> <dbl>  
1 001     30     3  
2 002     67     1  
3 003     52     4  
4 004     56     2
```

We can see that person 001 is in category 3, person 002 is in category 1, and so on. This compact representation of the categories is convenient for data entry and data manipulation, but it also has an obvious limitation – what do these numbers mean? We defined what these values mean for you above, but if you didn't have that information, or some kind of prior

knowledge about the process that was used to gather this data, then you would likely have no idea what these numbers mean.

Now, we could have solved that problem by making education a character vector from the beginning. For example:

```
demo <- tibble(
  id      = c("001", "002", "003", "004"),
  age     = c(30, 67, 52, 56),
  edu     = c(3, 1, 4, 2),
  edu_char = c(
    "Some college", "Less than high school", "College graduate",
    "High school graduate"
  )
)

demo
```



```
# A tibble: 4 x 4
  id     age   edu edu_char
  <chr> <dbl> <dbl> <chr>
1 001     30     3 Some college
2 002     67     1 Less than high school
3 003     52     4 College graduate
4 004     56     2 High school graduate
```

But, this strategy also has a few limitations.

First, entering data this way requires more typing. Not such a big deal in this case because we only have 4 participants. But, imagine typing out the categories as character strings 10, 20, or 100 times.

Second, R summarizes character vectors alphabetically by default, which may not be the ideal way to order some categorical variables.

Third, creating categorical variables in our data frame as character vectors limits us to inputting only *observed* values for that variable. However, there are cases when other categories are possible and just didn't apply to anyone in our data. That information may be useful to know.

At this point, we're going to show you how to coerce a variable to a factor in your data frame. Then, we will return to showing you how using factors can overcome some of the limitations outlined above.

19.1.1 Coerce a numeric variable

The code below shows one method for coercing a numeric vector into a factor.

```
# Load dplyr for pipes and mutate()
library(dplyr)

demo <- demo |>
  mutate(
    edu_f = factor(
      x       = edu,
      levels = 1:4,
      labels = c(
        "Less than high school", "High school graduate", "Some college",
        "College graduate"
      )
    )
  )

# A tibble: 4 x 5
# id     age   edu edu_char          edu_f
# <chr> <dbl> <dbl> <chr>           <fct>
1 001     30     3 Some college      Some college
2 002     67     1 Less than high school Less than high school
3 003     52     4 College graduate   College graduate
4 004     56     2 High school graduate High school graduate
```

Here's what we did above:

- We used `dplyr`'s `mutate()` function to create a new variable (`edu_f`) in the data frame called `demo`. The purpose of the `mutate()` function is to add new variables to data frames. We will discuss `mutate()` in greater detail in the [later in the book][creating and modifying columns].
 - You can type `?mutate` into your R console to view the help documentation for this function and follow along with the explanation below.
 - We assigned this new data frame the name `demo` using the assignment operator (`<-`).

- Because we assigned it the name `demo`, our previous data frame named `demo` (i.e., the one that didn’t include `edu_f`) no longer exists in our global environment. If we had wanted to keep that data frame in our global environment, we would have needed to assign our new data frame a different name (e.g., `demo_w_factor`).
- The first argument to the `mutate()` function is the `.data` argument. The value passed to the `.data` argument should be a data frame that is currently in our global environment. We passed the data frame `demo` to the `.data` argument using the pipe operator (`|>`), which is why `demo` isn’t written inside `mutate`’s parentheses.
- The second argument to the `mutate()` function is the `...` argument. The value passed to the `...` argument should be a name value pair. That means, a variable name, followed by an equal sign, followed by the values to be assigned to that variable name (`name = value`).
 - The name we passed to the `...` argument was `edu_f`. This value tells R what to name the new variable we are creating.
 - * If we had used the name `edu` instead, then the previous values in the `edu` variable would have been replaced with the new values. That is sometimes what you want to happen. However, when it comes to creating factors, we typically keep the numeric version of the variable in our data frame (e.g., `edu`) and *add a new factor variable*. 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 variable we name it the same thing the original variable was named with the addition of `_f` (for factor) at the end.
 - In this case, the value that will be assigned to the name `edu_f` will be the values returned by the `factor()` function. This is an example of nesting functions.
- We used the `factor()` function to create a factor vector.
 - 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 `edu` 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 `edu_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.

When we printed the data frame above, the values in `edu_f` *looked* the same as the character strings displayed in `edu_char`. Notice, however, that the variable type displayed below `edu_char` in the data frame above is `<chr>` for character. Alternatively, the variable type displayed below `edu_f` is `<fctr>`. Although, labels are used to make factors *look* like character vectors, they are still integer vectors under the hood. For example:

```
as.numeric(demo$edu_char)
```

`Warning: NAs introduced by coercion`

```
[1] NA NA NA NA
```

```
as.numeric(demo$edu_f)
```

```
[1] 3 1 4 2
```

There are two main reasons that you may want to use factors instead of character vectors at times:

First, R summarizes character vectors alphabetically by default, which may not be the ideal way to order some categorical variables. However, we can explicitly set the order of factor levels. This will be useful to us later when we analyze categorical variables. Here is a glimpse of things to come:

```
table(demo$edu_char)
```

| | College graduate | High school graduate | Less than high school |
|--------------|------------------|----------------------|-----------------------|
| | 1 | 1 | 1 |
| Some college | | | |
| | 1 | | |

```
table(demo$edu_f)
```

| | | |
|-----------------------|----------------------|--------------|
| Less than high school | High school graduate | Some college |
| 1 | 1 | 1 |
| College graduate | | |
| | 1 | |

Here's what we did above:

- You can type `?base::table` into your R console to view the help documentation for this function and follow along with the explanation below.
- We used the `table()` function to get a count of the number of times each unique value of `edu_char` appears in our data frame. In this case, each value appears one time. Notice that the results are returned to us in alphabetical order.
- Next, we used the `table()` function to get a count of the number of times each unique value of `edu_f` appears in our data frame. Again, each value appears one time. Notice, however, that this time the results are returned to us in the order that we passed to the `levels` argument of the `factor()` function above.

Second, creating categorical variables in our data frame as character vectors limits us to inputting only *observed* values for that variable. However, there are cases when other categories are possible and just didn't apply to anyone in our data. That information may be useful to know. Factors allow us to tell R that other values are possible, even when they are *unobserved* in our data. For example, let's add a fifth possible category to our education variable – graduate school.

```
demo <- demo |>
  mutate(
    edu_5cat_f = factor(
      x       = edu,
      levels = 1:5,
      labels = c(
        "Less than high school", "High school graduate", "Some college",
        "College graduate", "Graduate school"
      )
    )
  )

demo
```

```
# A tibble: 4 x 6
  id     age   edu edu_char      edu_f      edu_5cat_f
  <chr> <dbl> <dbl> <chr>        <fct>        <fct>
1 001     30     3 Some college Some college Some college
2 002     67     1 Less than high school Less than high school Less than high ~
3 003     52     4 College graduate College graduate College graduate
4 004     56     2 High school graduate High school graduate High school gra~
```

Now, let's use the `table()` function once again to count the number of times each unique level of `edu_char` appears in the data frame and the number of times each unique level of `edu_5cat_f` appears in the data frame:

```
table(demo$edu_char)
```

| | | |
|------------------|----------------------|-----------------------|
| College graduate | High school graduate | Less than high school |
| 1 | 1 | 1 |
| Some college | | |
| 1 | | |

```
table(demo$edu_5cat_f)
```

| | | |
|-----------------------|----------------------|--------------|
| Less than high school | High school graduate | Some college |
| 1 | 1 | 1 |
| College graduate | Graduate school | |
| 1 | 0 | |

Notice that R now tells us that the value `Graduate school` was possible but was observed zero times in the data.

19.1.2 Coerce a character variable

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

```

demo <- demo |>
  mutate(
    edu_f_from_char = factor(
      x = edu_char,
      levels = c(
        "Less than high school", "High school graduate", "Some college",
        "College graduate", "Graduate school"
      )
    )
  )

demo

# A tibble: 4 x 7
#> # ... with 7 variables:
#> #   id     <chr> > age     <dbl> > edu     <dbl> > edu_char <chr> > edu_f     <fct> > edu_5cat_f <fct> > edu_f_from_char <fct>
#> 1 001     30     3 Some college     Some colle~ Some coll~ Some college
#> 2 002     67     1 Less than high school Less than ~ Less than~ Less than high~
#> 3 003     52     4 College graduate College gr~ College g~ College graduat~
#> 4 004     56     2 High school graduate High schoo~ High scho~ High school gr~

```

| edu_f_from_char | Count |
|-----------------------|-------|
| Less than high school | 1 |
| High school graduate | 1 |
| Some college | 1 |
| College graduate | 1 |
| Graduate school | 0 |

| Less than high school | High school graduate | Some college |
|-----------------------|----------------------|--------------|
| 1 | 1 | 1 |
| College graduate | Graduate school | |
| 1 | 0 | |

Here's what we did above:

- We coerced a character vector (`edu_char`) 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 your analyses.

Now that we know how to use factors, let's return to our discussion of describing categorical variables.

19.2 Height and Weight Data

Below, we're going to learn to do descriptive analysis in R by experimenting with some simulated data that contains several people's sex, height, and weight. You can follow along with this lesson by copying and pasting the code chunks below in your R session.

```
# Load the dplyr package. We will need several of dplyr's functions in the
# code below.
library(dplyr)

# Simulate some data
height_and_weight_20 <- tibble(
  id = c(
    "001", "002", "003", "004", "005", "006", "007", "008", "009", "010", "011",
    "012", "013", "014", "015", "016", "017", "018", "019", "020"
  ),
  sex = c(1, 1, 2, 2, 1, 1, 2, 1, 2, 1, 1, 2, 2, 2, 1, 2, 2, 2, 2, 2),
  sex_f = factor(sex, 1:2, c("Male", "Female")),
  ht_in = c(
    71, 69, 64, 65, 73, 69, 68, 73, 71, 66, 71, 69, 66, 68, 75, 69, 66, 65, 65,
    65
  ),
  wt_lbs = c(
    190, 176, 130, 154, 173, 182, 140, 185, 157, 155, 213, 151, 147, 196, 212,
    190, 194, 176, 176, 102
  )
)
```

19.2.1 View the data

Let's start our analysis by taking a quick look at our data...

```
height_and_weight_20
```

```
# A tibble: 20 x 5
  id      sex sex_f  ht_in wt_lbs
  <chr>   <dbl> <fct>   <dbl>   <dbl>
1 001      1 Male     71     190
2 002      1 Male     69     176
3 003      2 Female   64     130
4 004      2 Female   65     154
```

| | | | | | |
|----|-----|---|--------|----|-----|
| 5 | 005 | 1 | Male | 73 | 173 |
| 6 | 006 | 1 | Male | 69 | 182 |
| 7 | 007 | 2 | Female | 68 | 140 |
| 8 | 008 | 1 | Male | 73 | 185 |
| 9 | 009 | 2 | Female | 71 | 157 |
| 10 | 010 | 1 | Male | 66 | 155 |
| 11 | 011 | 1 | Male | 71 | 213 |
| 12 | 012 | 2 | Female | 69 | 151 |
| 13 | 013 | 2 | Female | 66 | 147 |
| 14 | 014 | 2 | Female | 68 | 196 |
| 15 | 015 | 1 | Male | 75 | 212 |
| 16 | 016 | 2 | Female | 69 | 190 |
| 17 | 017 | 2 | Female | 66 | 194 |
| 18 | 018 | 2 | Female | 65 | 176 |
| 19 | 019 | 2 | Female | 65 | 176 |
| 20 | 020 | 2 | Female | 65 | 102 |

Here's what we did above:

- Simulated some data that we can use to practice categorical data analysis.
- We viewed the data and found that it has 5 variables (columns) and 20 observations (rows).
- Also notice that you can use the “Next” button at the bottom right corner of the printed data frame to view rows 11 through 20 if you are viewing this data in RStudio.

```{r}  
height\_and\_weight\_20  
```

Description: df[4] [20 x 4]

id <chr>	sex <chr>	ht_in <dbl>	wt_lbs <dbl>
001	Male	71	190
002	Male	69	176
003	Female	64	130
004	Female	65	154
005	Male	73	173
006	Male	69	182
007	Female	68	
008	Male	73	185
009	Female	71	157
010	Male	66	155

1-10 of 20 rows

Previous 1 2 Next

Click "Next" to see additional rows

19.3 Calculating frequencies

Now that we're able to easily view our data, let's return to the original purpose of this demonstration – calculating frequencies and proportions. At this point, we suspect that few of you would have any trouble telling me that the frequency of females in this data is 12 and the frequency of males in this data is 8. It's pretty easy to just count the number of females and males in this small data set with only 20 rows. Further, if we asked you what proportion of this sample is female, most of you would still be able to easily tell me $12/20 = 0.6$, or 60%. But, what if we had 100 observations or 1,000,000 observations? You'd get sick of counting pretty quickly. Fortunately, you don't have to! Let R do it for you! As is almost always the case with R, there are multiple ways we can calculate the statistics that we're interested in.

19.3.1 The base R table function

As we already saw above, we can use the base R `table()` function like this:

```
table(height_and_weight_20$sex)
```

```
1 2  
8 12
```

Additionally, we can use the `CrossTable()` function from the `gmodels` package, which gives us a little more information by default.

19.3.2 The gmodels CrossTable function

```
# Like all packages, you will have to install gmodels (install.packages("gmodels")) before you run this code.  
gmodels::CrossTable(height_and_weight_20$sex)
```

Cell Contents	
	N
N / Table Total	

Total Observations in Table: 20

	1	2
	8	12
	0.400	0.600

19.3.3 The tidyverse way

The final way we're going to discuss here is the `tidyverse` way, which is our preference. We will have to write a little additional code, but the end result will be more flexible, more readable, and will return our statistics to us in a data frame that we can save and use for further analysis. Let's walk through this step by step...

Note

Side Note: You should already be familiar with the pipe operator (`|>`), but if it doesn't look familiar to you, you can learn more about it in [Using pipes](#). Don't forget, if you are using RStudio, you can use the keyboard shortcut `shift + command + m` (Mac) or `shift + control + m` (Windows) to insert the pipe operator.

First, we don't want to view the individual values in our data frame. Instead, we want to condense those values into summary statistics. This is a job for the `summarise()` function.

```
height_and_weight_20 |>  
  summarise()
```

```
# A tibble: 1 x 0
```

As you can see, `summarise()` doesn't do anything interesting on its own. We need to tell it what kind of summary information we want. We can use the `n()` function to count rows. By default, it will count all the rows in the data frame. For example:

```
height_and_weight_20 |>
  summarise(n())
```

```
# A tibble: 1 x 1
`n()`
<int>
1     20
```

Here's what we did above:

- We passed our entire data frame to the `summarise()` function and asked it to count the number of rows in the data frame.
- The result we get is a new data frame with 1 column (named `n()`) and one row with the value 20 (the number of rows in the original data frame).

This is a great start. However, we really want to count the number of rows that have the value “Female” for `sex_f`, and then separately count the number of rows that have the value “Male” for `sex_f`. Said another way, we want to break our data frame up into smaller data frames – one for each value of `sex_f` – and then count the rows. This is exactly what `dplyr`'s `group_by()` function does.

```
height_and_weight_20 |>
  group_by(sex_f) |>
  summarise(n())
```

```
# A tibble: 2 x 2
  sex_f   `n()`
  <fct>   <int>
1 Male      8
2 Female    12
```

And, that's what we want.

Note

Side Note: `dplyr`'s `group_by()` function operationalizes the **Split - Apply - Combine** strategy for data analysis. That sounds sort of fancy, but all it really means is that we split our data frame up into smaller data frames, apply our calculation separately to each smaller data frame, and then combine those individual results back together as a single result. So, in the example above, the `height_and_weight_20` data frame was split into two separate little data frames (i.e., one for females and one for males), then the

`summarise()` and `n()` functions counted the number of rows in each of the two smaller data frames (i.e., 12 and 8 respectively), and finally combined those individual results into a single data frame, which was printed to the screen for us to view.

However, it will be awkward to work with a variable named `n()` (i.e., with parentheses) in the future. Let's go ahead and assign it a different name. We can assign it any valid name we want. Some names that might make sense are `n`, `frequency`, or `count`. We're going to go ahead and just name it `n` without the parentheses.

```
height_and_weight_20 |>
  group_by(sex_f) |>
  summarise(n = n())
```

```
# A tibble: 2 x 2
  sex_f     n
  <fct>   <int>
1 Male      8
2 Female    12
```

Here's what we did above:

- We added `n =` to our `summarise` function (`summarise(n = n())`) so that our count column in the resulting data frame would be named `n` instead of `n()`.

Finally, estimating categorical frequencies like this is such a common operation that `dplyr` has a shortcut for it – `count()`. We can use the `count()` function to get the same result that we got above.

```
height_and_weight_20 |>
  count(sex_f)
```

```
# A tibble: 2 x 2
  sex_f     n
  <fct>   <int>
1 Male      8
2 Female    12
```

19.4 Calculating percentages

In addition to frequencies, we will often be interested in calculating percentages for categorical variables. As always, there are many ways to accomplish this task in R. From here on out, we're going to primarily use `tidyverse` functions.

In this case, the proportion of people in our data who are female can be calculated as the number who are female (12) divided by the total number of people in the data (20). Because we already know that there are 20 people in the data, we could calculate proportions like this:

```
height_and_weight_20 |>
  count(sex_f) |>
  mutate(prop = n / 20)
```

```
# A tibble: 2 x 3
  sex_f     n   prop
  <fct> <int> <dbl>
1 Male      8    0.4
2 Female    12    0.6
```

Here's what we did above:

- Because the `count()` function returns a data frame just like any other data frame, we can manipulate it in the same ways we can manipulate any other data frame.
- So, we used `dplyr`'s `mutate()` function to create a new variable in the data frame named `prop`. Again, we could have given it any valid name.
- Then we set the value of `prop` to be equal to the value of `n` divided by 20.

This works, but it would be better to have R calculate the total number of observations for the denominator (20) than for us to manually type it in. In this case, we can do that with the `sum()` function.

```
height_and_weight_20 |>
  count(sex_f) |>
  mutate(prop = n / sum(n))
```

```
# A tibble: 2 x 3
  sex_f     n   prop
  <fct> <int> <dbl>
1 Male      8    0.4
2 Female    12    0.6
```

Here's what we did above:

- Instead of manually typing in the total count for our denominator (20), we had R calculate it for us using the `sum()` function. The `sum()` function added together all the values of the variable `n` (i.e., $12 + 8 = 20$).

Finally, we just need to multiply our proportion by 100 to convert it to a percentage.

```
height_and_weight_20 |>
  count(sex_f) |>
  mutate(percent = n / sum(n) * 100)
```

```
# A tibble: 2 x 3
  sex_f     n percent
  <fct> <int>   <dbl>
1 Male      8     40
2 Female    12     60
```

Here's what we did above:

- Changed the name of the variable we are creating from `prop` to `percent`. But, we could have given it any valid name.
- Multiplied the proportion by 100 to convert it to a percentage.

19.5 Missing data

In the real world, you will frequently encounter data that has missing values. Let's quickly take a look at an example by adding some missing values to our data frame.

```
height_and_weight_20 <- height_and_weight_20 |>
  mutate(sex_f = replace(sex, c(2, 9), NA)) |>
  print()
```

```
# A tibble: 20 x 5
  id      sex sex_f ht_in wt_lbs
  <chr> <dbl> <dbl> <dbl>   <dbl>
1 001      1     1     71     190
2 002      1     NA     69     176
3 003      2     2     64     130
4 004      2     2     65     154
```

5	005	1	1	73	173
6	006	1	1	69	182
7	007	2	2	68	140
8	008	1	1	73	185
9	009	2	NA	71	157
10	010	1	1	66	155
11	011	1	1	71	213
12	012	2	2	69	151
13	013	2	2	66	147
14	014	2	2	68	196
15	015	1	1	75	212
16	016	2	2	69	190
17	017	2	2	66	194
18	018	2	2	65	176
19	019	2	2	65	176
20	020	2	2	65	102

Here's what we did above:

- Replaced the 2nd and 9th value of `sex_f` with NA (missing) using the `replace()` function.

Now let's see how our code from above handles this

```
height_and_weight_20 |>
  count(sex_f) |>
  mutate(percent = n / sum(n) * 100)
```

```
# A tibble: 3 x 3
  sex_f     n percent
  <dbl> <int>   <dbl>
1     1     7     35
2     2    11     55
3    NA     2     10
```

As you can see, we are now treating missing as if it were a category of `sex_f`. Sometimes this will be the result you want. However, often you will want the `n` and `percent` of *non-missing* values for your categorical variable. This is sometimes referred to as a **complete case analysis**. There's a couple of different ways we can handle this. We will simply filter out rows with a missing value for `sex_f` with `dplyr`'s `filter()` function.

```

height_and_weight_20 |>
  filter(!is.na(sex_f)) |>
  count(sex_f) |>
  mutate(percent = n / sum(n) * 100)

```

```

# A tibble: 2 x 3
  sex_f     n percent
  <dbl> <int>   <dbl>
1     1      7     38.9
2     2     11     61.1

```

Here's what we did above:

- We used `filter()` to keep only the rows that have a *non-missing* value for `sex_f`.
 - In the R language, we use the `is.na()` function to tell the R interpreter to identify NA (missing) values in a vector. We *cannot* use something like `sex_f == NA` to identify NA values, which is sometimes confusing for people who are coming to R from other statistical languages.
 - In the R language, `!` is the NOT operator. It sort of means “do the opposite.”
 - So, `filter()` tells R which rows of a data frame to *keep*, and `is.na(sex_f)` tells R to find rows with an NA value for the variable `sex_f`. Together, `filter(is.na(sex_f))` would tell R to *keep* rows with an NA value for the variable `sex_f`. Adding the NOT operator `!` tells R to do the opposite – *keep* rows that do *NOT* have an NA value for the variable `sex_f`.
- We used our code from above to calculate the `n` and `percent` of non-missing values of `sex_f`.

19.6 Formatting results

Notice that now our percentages are being displayed with 5 digits to the right of the decimal. If we wanted to present our findings somewhere (e.g., a journal article or a report for our employer) we would almost never want to display this many digits. Let's get R to round these numbers for us.

```

height_and_weight_20 |>
  filter(!is.na(sex_f)) |>
  count(sex_f) |>
  mutate(percent = (n / sum(n) * 100) |> round(2))

```

```
# A tibble: 2 x 3
  sex_f     n percent
  <dbl> <int>   <dbl>
1     1      7    38.9
2     2     11    61.1
```

Here's what we did above:

- We passed the calculated percentage values (`n / sum(n) * 100`) to the `round()` function to round our percentages to 2 decimal places.
 - Notice that we had to wrap `n / sum(n) * 100` in parentheses in order to pass it to the `round()` function with a pipe.
 - We could have alternatively written our R code this way: `mutate(percent = round(n / sum(n) * 100, 2))`.

19.7 Using freqtables

In the sections above, we learned how to use `dplyr` functions to calculate the frequency and percentage of observations that take on each value of a categorical variable. However, there can be a fair amount of code writing involved when using those methods. The more we have to repeatedly type code, the more tedious and error-prone it becomes. This is an idea we will return to many times in this book. Luckily, the R programming language allows us to write our own functions, which solves both of those problems.

Later in this book, we will show you [how to write your own functions][writing functions]. For the time being, We're going to suggest that you install and use a package we created called `freqtables`. The `freqtables` package is basically an enhanced version of the code we wrote in the sections above. We designed it to help us quickly make tables of descriptive statistics (i.e., counts, percentages, confidence intervals) for categorical variables, and it's specifically designed to work in a `dplyr` pipeline.

Like all packages, you need to first install it...

```
# You may be asked if you want to update other packages on your computer that
# freqtables uses. Go ahead and do so.
install.packages("freqtables")
```

And then load it...

```
# After installing freqtables on your computer, you can load it just like you
# would any other package.
library(freqtables)
```

Now, let's use the `freq_table()` function from `freqtables` package to rerun our analysis from above.

```
height_and_weight_20 |>
  filter(!is.na(sex_f)) |>
  freq_table(sex_f)
```

```
# A tibble: 2 x 9
  var   cat      n n_total percent    se t_crit   lcl   ucl
  <chr> <chr> <int>   <int>    <dbl> <dbl> <dbl> <dbl> <dbl>
1 sex_f  1        7     18    38.9  11.8  2.11  18.2  64.5
2 sex_f  2       11     18    61.1  11.8  2.11  35.5  81.8
```

Here's what we did above:

- We used `filter()` to keep only the rows that have a *non-missing* value for sex and passed the data frame on to the `freq_table()` function using a pipe.
- We told the `freq_table()` function to create a univariate frequency table for the variable `sex_f`. A “univariate frequency table” just means a table (data frame) of useful statistics about a single categorical variable.
- The univariate frequency table above includes:
 - `var`: The name of the categorical variable (column) we are analyzing.
 - `cat`: Each of the different categories the variable `var` contains – in this case “Male” and “Female”.
 - `n`: The number of rows where `var` equals the value in `cat`. In this case, there are 7 rows where the value of `sex_f` is Male, and 11 rows where the value of `sex_f` is Female.
 - `n_total`: The sum of all the `n` values. This is also to total number of rows in the data frame currently being analyzed.
 - `percent`: The percent of rows where `var` equals the value in `cat`.
 - `se`: The standard error of the percent. This value is not terribly useful on its own; however, it's necessary for calculating the 95% confidence intervals.

- `t_crit`: The critical value from the t distribution. This value is not terribly useful on its own; however, it’s necessary for calculating the 95% confidence intervals.
- `lcl`: The lower (95%, by default) confidence limit for the percentage `percent`.
- `ucl`: The upper (95%, by default) confidence limit for the percentage `percent`.

We will continue using the `freqtables` package at various points throughout the book. We will also show you some other cool things we can do with `freqtables`. For now, all you need to know how to do is use the `freq_table()` function to calculate frequencies and percentages for single categorical variables.

Congratulations! You now know how to use R to do some basic descriptive analysis of individual categorical variables.

Part V

Collaboration

20 Introduction to git and GitHub



If you read this book's introductory material, specifically the section on [Contributing to R4Epi](#), then you have already been briefly exposed to GitHub. If not, taking a quick look at that section may be useful. [GitHub](#) is a website specifically designed to facilitate collaboratively creating programming code. In many ways, GitHub is a cloud-based file storage service like Dropbox, Google Drive, and OneDrive, but with special tools built-in for collaborative coding. [Git](#) is the name of the **versioning** software that powers many of GitHub's special tools. We will talk about what versioning means shortly.

The goal of this, and the next few, chapters isn't to teach you everything you need to know about git and GitHub. Not even close! That would fill up its own book. The goal here is just to expose you to git and GitHub, show you a brief example of how they may be useful to you, and provide you with some resources you can use to learn more if you're interested.

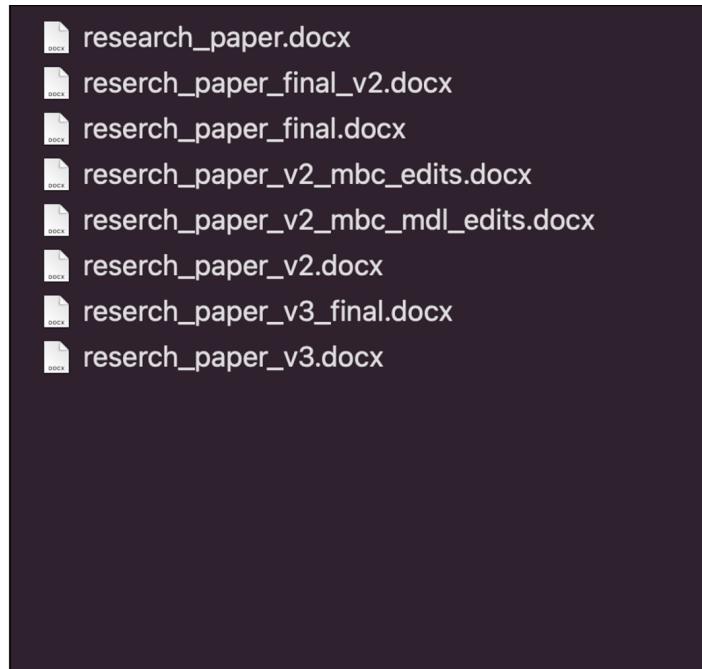
But, why should you be interested in the first place? Well, there are at least four overarching reasons why you should consider learning to use git and GitHub as part of your workflow when your projects include data and/or coding:

1. Versioning
2. Preservation
3. Reproducibility
4. Collaboration

We'll elaborate on what each of these means to us below. Then, we will introduce you to git and GitHub, and explain why they are some of the best tools currently available to help you with versioning and collaborating. We'll go ahead and warn you now — git and GitHub can be hard to wrap your mind around at first. In fact, using git and GitHub still frequently causes us confusion and frustration at times. However, we still believe that the payoff is ultimately worth the upfront investment in time and frustration. Additionally, we will do our best to make this introduction as gentle, comprehensible, and practically applicable as possible.

20.1 Versioning

Have you ever worked on a paper or report and had a folder on your computer that looked something like this?



Saving a bunch of different versions of a file like this is a real mess. It becomes even worse when you are trying to work with multiple people. What is contained in each document again?

What order were the documents created in? What are the differences between the documents? Versioning helps us get around all of these problems.

Instead of jumping straight into learning versioning with git and GitHub, we will start our discussion about versioning using a simple example in Google Docs. Not because Google Docs are especially relevant to anything else in this course, but because there are a lot of parallels between the Google Docs versioning system and the git versioning system when it is paired with GitHub. However, the Google Docs versioning system is a little bit more basic, easy to understand, and easy to experiment with. Later, we will refer back to some of these Google Docs examples when we are trying to explain how to use git and GitHub. If you'd like to do some experimenting of your own, feel free to navigate to <https://docs.google.com/> now and follow along with the following demonstration.

First, we will type a little bit of text in our Google Doc. It doesn't really matter what we type — this is purely for demonstration purposes. In the example below, we type "Here is some text."

Now, let's say that we decide to make a change to our text. Specifically, we decide to replace "some" with "just a little."

Now, let's say that we changed our mind again and we want to go back to using the original text. In this case, it would be really easy to go back to using the original text even without versioning. We could just use "undo" or even retype the previous text. But, let's pretend for a minute that we changed a lot of text, and that we made those changes several weeks ago. Under those circumstances, how might we view the original version of the document? We can use the Google Docs versioning system. To do so, we can click **File** then **Version history** then **See version history**. This will bring up a new view that shows us all the changes we've made to this document, and when we made them.

This is great! We don't have to save a bunch of different files like we saw in the "messy" folder at the beginning of this section. Instead, there is only one document, and we can see all the versions of that document, who created the various versions of that document, when all the various versions of that document were created, and exactly what changed from one version to the next. In other words, we have a complete record of the evolution of this document in the **version history** — how we got from the blank document we started with to the current version of the document we are working with today.

Further, if we want to turn back the clock to a previous version of the document, we need only select that version and click the **Restore this version** button like this.

But, you can probably imagine how difficult it can be to find a previous version of a document by searching through a list of dates. In the example above, there were only three dates to look through, but in a real work document, there may be hundreds of versions saved. The dates, by themselves, aren't very informative. Luckily, when we hit key milestones in the development of our document, Google Docs allows us to name them. That way, it will be easy to find that version in the future if we ever need to refer to it (assuming we give it an [informative name](#)).

For example, let's say that we just added a table to our document that includes the mean values of the variables X and Y for two groups of people - Group 1 and Group 2. Completing this table is a key milestone in the evolution of our document and this is a great time to name the current version of the document just in case we ever need to refer back to it. To do so, we can click **File** then **Version history** then **Name current version**.

Notice that in the example above I used the word **commit** instead of the word **save**. In this case, they essentially mean the same thing, but soon you will see that git also uses the word **commit** to refer to taking a snapshot of the state of our project — similar to the way we just took a snapshot of the state of our document.

Now let's say that we decide to use medians in our table instead of means. After making that change, our document now looks like this.

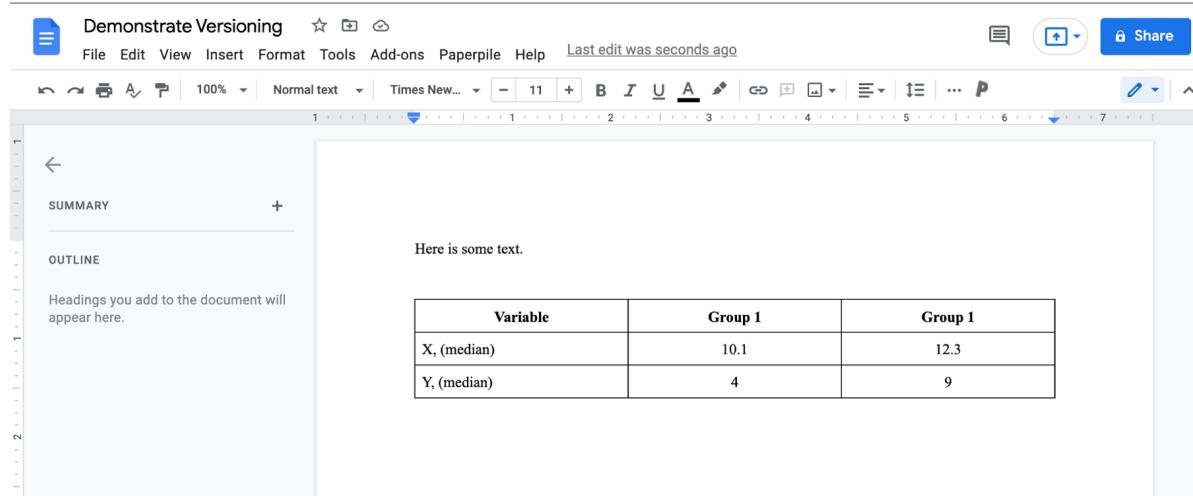


Figure 20.1: A gif about switching back to an old version in Google Docs.

Can you guess what we are about to do next? That's right! We changed our minds again and decided to switch back to using the mean values in the table. No problem! We can easily search for the version of the document that we committed, which includes the table of mean values. We can then restore that version as we did above.

20.2 Preservation

In addition to versioning, the ability to preserve all of your code and related project files in the cloud is another great reason to consider using GitHub. In other words, you don't have to worry about losing your code if your computer is lost, damaged, or replaced. All of your project files can easily be retrieved and restored from GitHub. Although the same is true for other cloud-based file storage services like Dropbox, Google Drive, and OneDrive, remember that GitHub has special built-in tools that those services do not provide.

20.3 Reproducibility

Reproducibility, or more precisely, **reproducible research**, is a term that may be unfamiliar to many of you. Peng and Hichs (2021) give a nice introduction to reproducible research:⁷

Scientific progress has long depended on the ability of scientists to communicate to others the details of their investigations... In the past, it might have sufficed to describe the data collection and analysis using a few key words and high-level language. However, with today's computing-intensive research, the lack of details about the data analysis in particular can make it impossible to recreate any of the results presented in a paper. Compounding these difficulties is the impracticality of describing these myriad details in traditional journal publications using natural language. To address this communication problem, a concept has emerged known as reproducible research, which aims to provide for others far more precise descriptions of an investigator's work. As such, reproducible research is an extension of the usual communications practices of scientists, adapted to the modern era.

They go on to define reproducible research in the following way:^{7 8}

A published data analysis is reproducible if the analytic data sets and the computer code used to create the data analysis are made available to others for independent study and analysis.

We will not delve deeper into the general importance and challenges of reproducible research in this book; however, we encourage readers who are interested in learning more about reproducible research to take a look at both of the articles cited above. Additionally, we believe it's important to highlight that GitHub is a great tool for making our research more reproducible. Specifically, it provides a platform where others can easily download the data (when we are allowed to make it available), computer code, and documentation needed to recreate our research results. This is a great asset for scientific progress, but only if researchers like us use it effectively.

20.4 Collaboration

In the sections above, we discussed the ways in which git and GitHub are tools we can use for versioning, preserving our code in the cloud, and making our research more reproducible. All of these are important benefits of using git and GitHub even if we don't routinely collaborate with others to complete our projects. However, the power of GitHub is even greater when we think about using it as a tool for collaboration — including collaboration with our future selves.

For example, one research project that we (the authors) both work on is the Detection of Elder abuse Through Emergency Care Technicians (DETECT) project. Let's say that we would like to start collaborating with you on DETECT. Perhaps we need your help preprocessing some of the DETECT data and conducting an analysis. So, how do we get started?

Because we created a **repository** on GitHub for the DETECT project, all of the files and documentation you need to get started are easily accessible to you. In fact, you don't even have to reach out to us first for access. They are freely available to anyone who is interested. Please go ahead and use the following URL to view the DETECT repository now: https://github.com/brad-cannell/detect_pilot_test_5w. GitHub repositories may look a little confusing at first, but you will get used to them with practice.

 Note

Side Note: Repository is a git term that can seem a little confusing or intimidating at first. However, it's really no big deal. You can think of a git repository as a folder that holds all of the files related to your project. On GitHub, each repository has its own separate website where people from anywhere in the world can access the files and documents related to your project. They can also communicate with you through your GitHub repository, post issues to your GitHub repository if they encounter a problem, and contribute code to your project.

We could have emailed the files back and forth, but what if we accidentally forget to send you one? What if one of the files is too large to email? What if two people are working on the same file at the same time and send out their revisions via email? Which version should we use? In the chapters that follow, we will show you how using GitHub to share project files gets around these, and other, collaboration issues.

20.5 Summary

In summary, git and GitHub are awesome tools to use when our projects involve research and/or data analysis. They allow us to store all of our files in the cloud with the added benefit of versioning and many other collaboration tools. The primary disadvantage of using GitHub

instead of just emailing code files or using general-purpose cloud storage services is its learning curve. But, in the following chapters, we hope to give you enough knowledge to make GitHub immediately useful to you. Over time, you can continue to hone your GitHub skills and really take advantage of everything it has to offer. We think if you make this initial investment, it is unlikely that you will ever look back.

21 Using git and GitHub



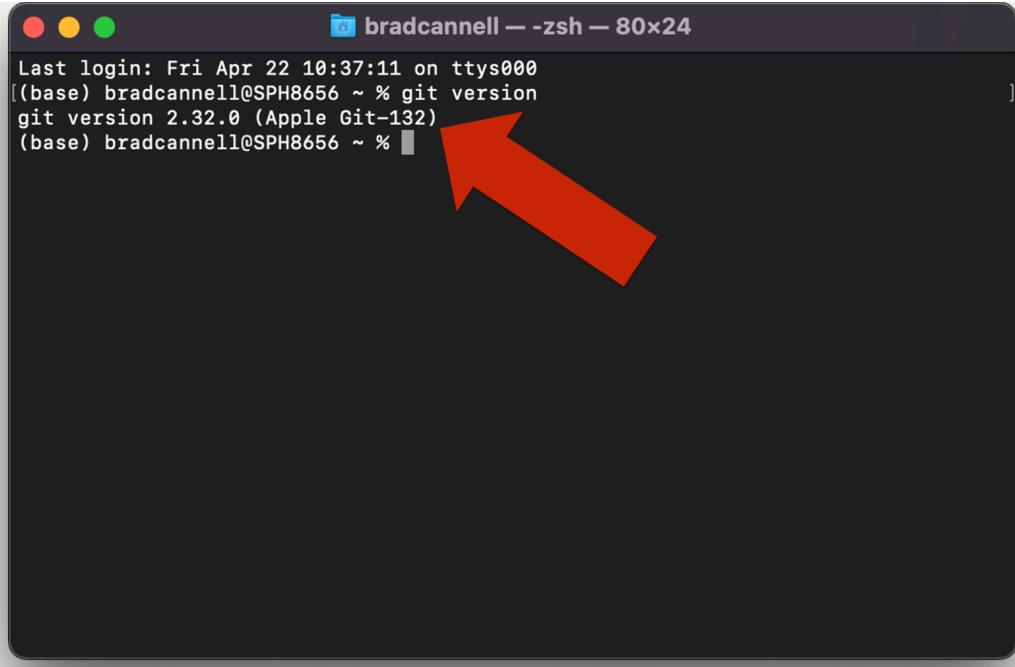
In the previous chapter, we discussed *why* we should consider learning to use git and GitHub as part of our workflow when our projects include data and/or coding. In this chapter, we will begin to talk about *how* to use git and GitHub. We will also introduce a third tool, GitKraken, that makes it easier for us to use git and GitHub.

21.1 Install git

Before we can use git, we will need to install it on our computer. The following chapter of Pro Git provides instructions for installing git on Linux, Windows, and MacOS operating systems: [Get Started Installing Git](#).

If you are using a Mac, it's likely that you already have git — most Macs ship with git installed. To check, open your Terminal app. The Terminal app is located in the Utilities folder, which is located in the Applications folder. In the terminal app, type “git version”. If you see a version

number, then it is already installed. If not, then please follow the installation instructions given in the link to Pro Git above.

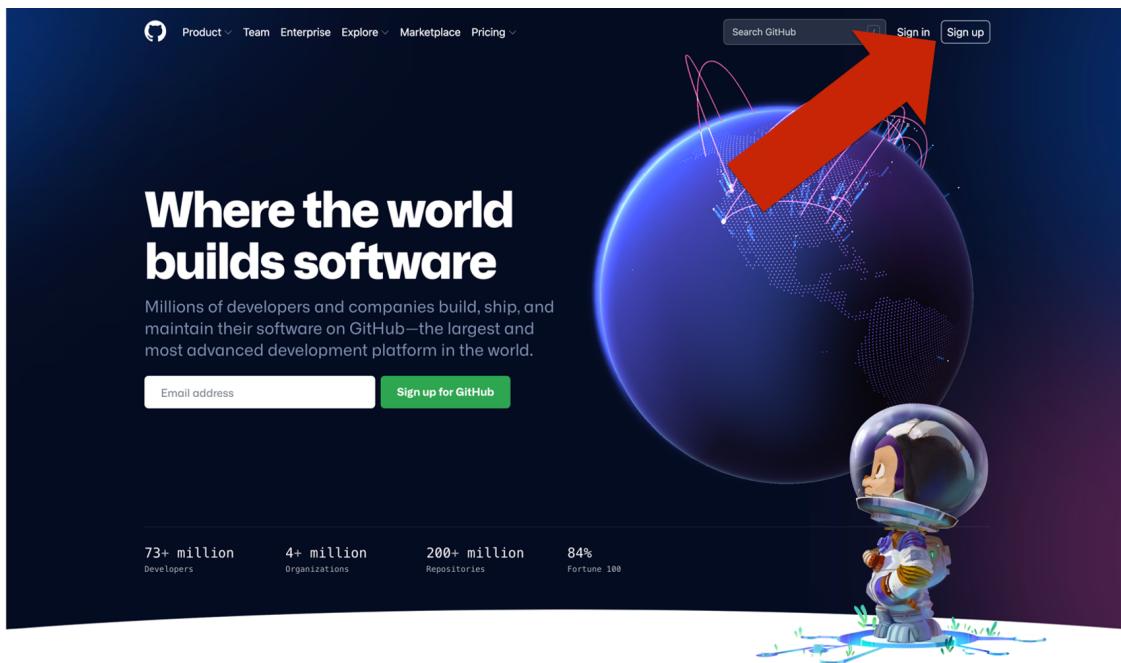


```
Last login: Fri Apr 22 10:37:11 on ttys000
[(base) bradcannell@SPH8656 ~ % git version
git version 2.32.0 (Apple Git-132)
(base) bradcannell@SPH8656 ~ % ]
```

Figure 21.1: Checking git version in the MacOS terminal.

21.2 Sign up for a GitHub account

We have already alluded to the fact that git and GitHub are not the same thing. You can use git locally on your computer without ever using GitHub. Conversely, you can browse GitHub, and even do some limited contributing to code, without ever installing git on your computer (e.g., see [Contributing to R4Epi](#)). However, git and GitHub work best when used together. You don't need to download anything to start using GitHub, but you will need to sign up for a free GitHub account. To do so, just navigate to <https://github.com/>



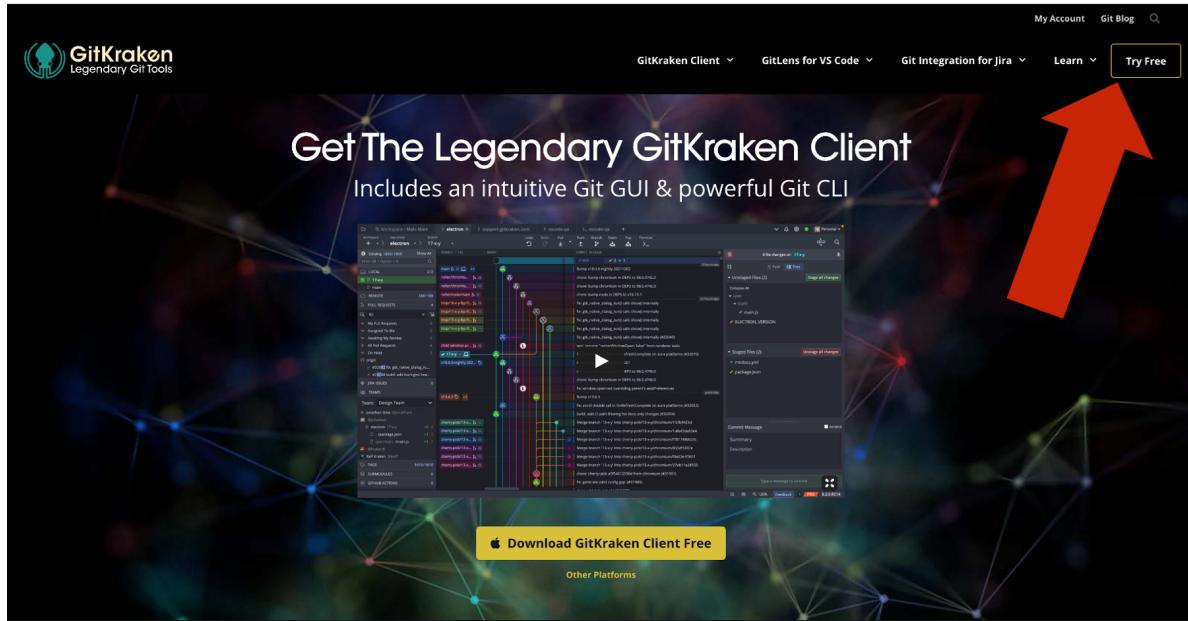
21.3 Install GitKraken

Git is software for our computer. However, unlike most of the software we are used to using, git does not have a [graphical user interface](#) (GUI - pronounced “gooey”). In other words, there is no git application that we can open and start clicking around in. Instead, by default, we interact with git by typing commands into the computer’s terminal – also called “command line” in GitHub’s documentation – like we saw in Figure 21.1. The commands we type to use git kind of look like their own programming language. In our experience, interacting with git in the terminal is awkward, inefficient, and unnecessary for most new git users. And learning to use git in this way is a barrier to getting started in the first place.

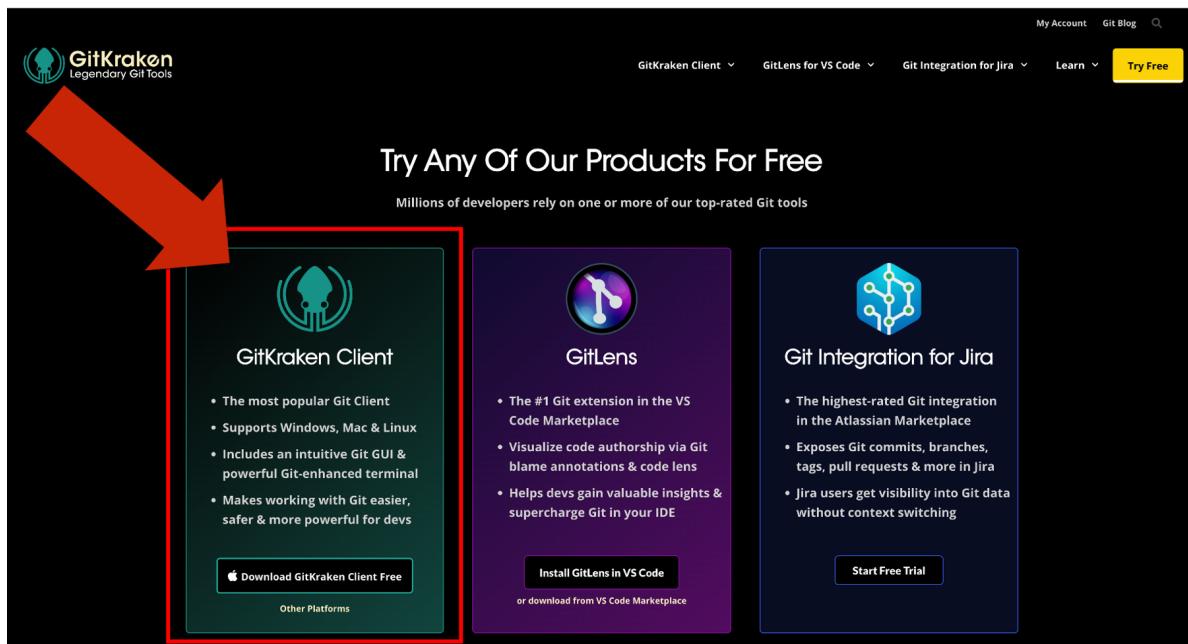
Thankfully, other third-party vendors have made excellent GUI’s for git that we can download and use for free. Our current favorite is called **GitKraken**. To use GitKraken, you will first need to navigate to the GitKraken website (<https://www.gitkraken.com/>). If it helps, you can think of git and GitKraken as having a relationship that is very similar to the relationship between R and RStudio. R is the language. RStudio is the application that makes it easier for us to use the R language to work with data. Similarly, git is the language and GitKraken is the application that makes it easier for us to use git to track versions of our project files.

Before you use the GitKraken client, you will need to sign up for an account. It may say that you need to sign up for a free trial. Go ahead and do it. The free trial is just for the “Pro” version. At the end of the free trial, you will automatically be downgraded to the “Free”

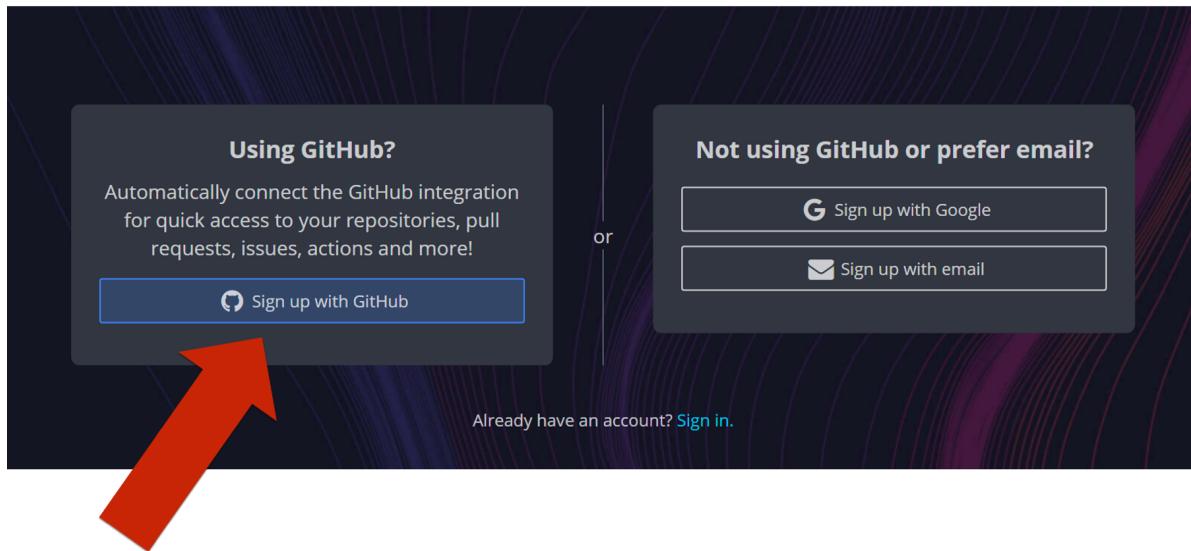
version, which is... free. And, the free version will do everything you need to do to follow along with this book.



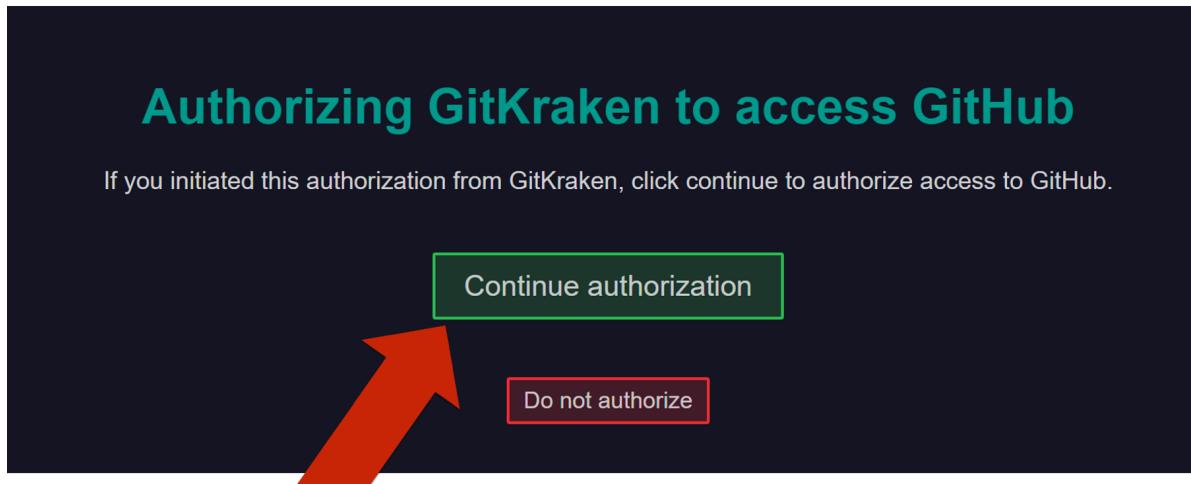
Next, you will need to click on the “Try Free” button. Then, download and install the GitKraken Client to your computer.



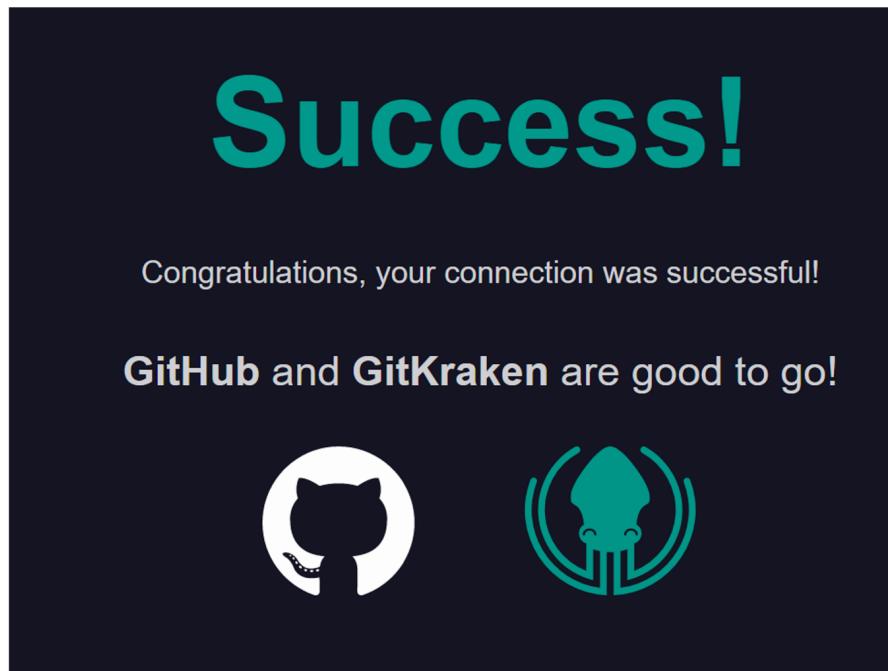
As you are installing GitKraken, it should ask you if you want to sign up with your GitHub account. Yes, you do! It will make your life much easier down the road. If you didn't sign up for a GitHub account in the previous step, please go back and do so.



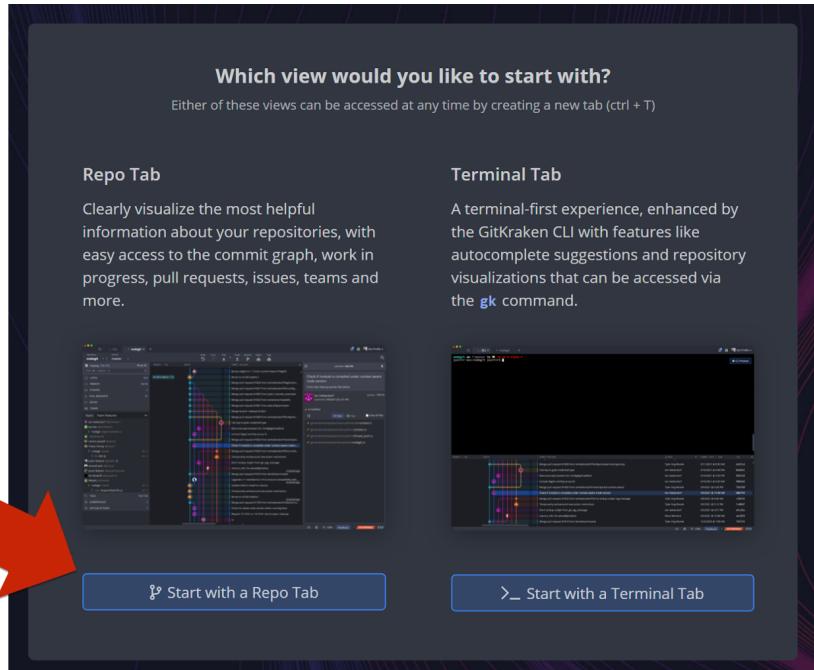
Then click the green Continue authorization button.



Then, you will be asked to sign into your GitHub account – possibly using your two-factor authentication. When you see the success screen, you can close your browser and return to GitKraken.



The next thing you will do is create a profile. After you create a profile, you will be asked if you want the Repo Tab first or the Terminal Tab first. We recommend that you select the Repo Tab option.



Once you have installed Git and GitKraken, and you've created your GitHub account, you will have all the tools you need to follow along with all of the examples in this book. Speaking of examples, let's go ahead and take a look at a couple now.

21.4 Example 1: Contribute to R4Epi

If you haven't already done so, please read the [contributing to R4Epi portion of the book's welcome page](#). This will give you a gentle introduction to using GitHub, for a very practical purpose, without even needing to use git or GitKraken.

21.5 Example 2: Create a repository for a research project

In this example, we will learn how to create our very own git and GitHub repositories from scratch. We can immediately begin using the lessons from this example for our research projects – even if we aren't collaborating with others on them. Remember, [there are at least four overarching reasons](#) why you should consider learning to use git and GitHub as part of your workflow for your projects, and collaboration is only one of them. Not to mention the fact that it is often useful to think of our future selves as other collaborators, which we have mentioned and/or alluded to many times in this book.

There are many possible ways we could set up our project to take advantage of all that git and GitHub have to offer. We're going to show you one possible sequence of steps in this example, but you may decide that you prefer a different sequence as you get more experience, and that's totally fine!

This example is long! So, we created a brief outline that you can quickly reference in the future. Details are below.

- [Step 1: Create a repository on GitHub](#)
- [Step 2: Clone the repository to your computer](#)
- [Step 3: Add an R project file to the repository](#)
- [Step 4: Update and commit gitignore](#)
- [Step 5: Keep adding and committing files](#)

Step 1: Create a repository on GitHub

The first thing we will do is create a repository on *GitHub*. **Repositories** are the fundamental organizational units of your GitHub account. Other cloud storage services like Dropbox are organized into file folders at every level. Meaning, you have your main Dropbox folder, which has other folders nested inside of it — many of which may have their own nested folders. Your GitHub account also stores all your files in file folders; however, the level one folders — those that aren't nested inside of another folder — are called repositories (represented by the book icon in the image below and on the GitHub website). Typically, each repository is an entire, self-contained project. Like a file folder, each repository can contain other folders, code files, media files, data sets, and any other type of file needed to reproduce your research project.

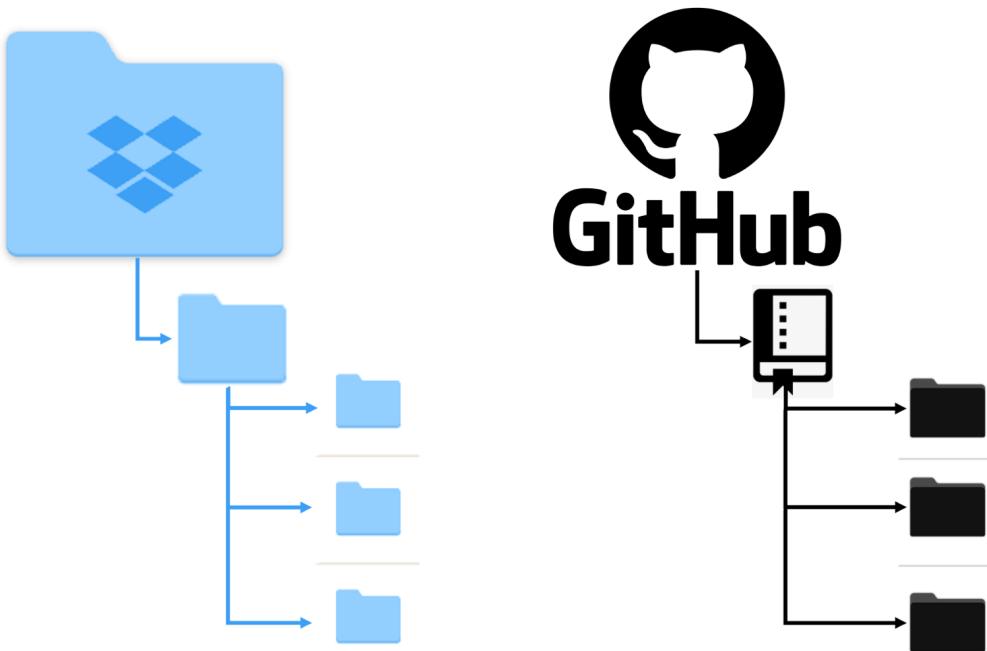
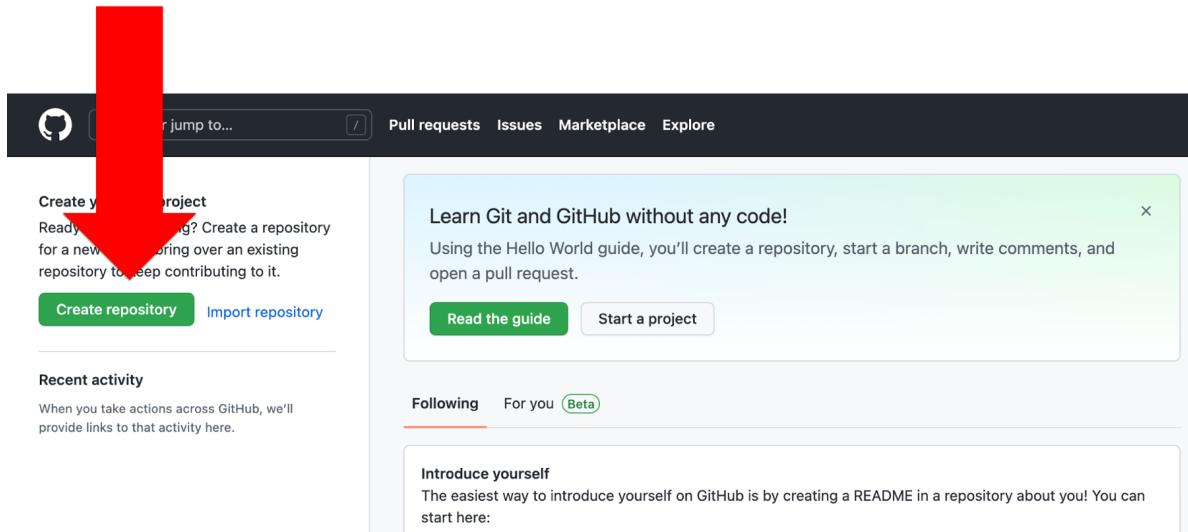


Figure 21.2: GitHub repositories compared to Dropbox.

⚠️ Warning

Just because we *can* upload data to GitHub doesn't mean we *should* upload data to GitHub. Often, the data we use in epidemiology contains [protected health information \(PHI\)](#) that we must go to great lengths to keep secure. In general, GitHub is **NOT** considered a secure place to store our data and should not be used for this purpose. Below, we will demonstrate how to make sure our data isn't uploaded to GitHub with the rest of the files in our repository.

To create a new repository in GitHub, we will simply click the green `Create repository` button. This button will look slightly different depending on where we are at in the GitHub website. The screenshot below was taken from Arthur Epi's (our fictitious research assistant) main landing page (i.e., <https://github.com/>).



After clicking the green `Create repository` button, the next page Arthur will see is the setup page for his repository. For the purposes of this example, he will use the following information to set it up.

- **Repository name:** As the on-screen prompt says, great repository names are short and memorable. Further, the repository name must be unique to his account (i.e., he can't have two repositories with the same name), and it can only include letters, numbers, dashes (-), underscores (_), and periods (.). We recommend using underscores to separate words to be consistent with the object naming guidelines from [coding-best-practices](#). For this example, he will name the repository `r4epi_example_project`.
- **Description:** The description is optional, but we like to fill it in. Arthur's description should also be brief. Ideally it will allow others scanning our repository to quickly determine what it's all about. For this example, the description will say, "An example repository that accompanies the git and GitHub chapters in the R4Epi book."
- **Public/Private:** We can choose to make our repositories public or private. If we make them public, they can be *viewed* by anyone on the internet. If we make them private, we can control who is able to view them. At first, you may be tempted to make your repositories private. It can feel vulnerable to put your project/code out there for the entire internet to view. However, we are going to recommend that you make all of your repositories public and be thoughtful about the files/documents/information you choose to upload to them. For example, we **NEVER** want to upload data containing information with PHI or individual identifiers in it. So, we will often need to figure out a different way to share our data with others who legitimately need access to it, but we can

often use GitHub to share all other files related to the project. Making our repository public makes it easier for others to locate our work and potentially collaborate with us.

- **Add a README file:** A **README** file has a special place in GitHub. Under the hood, it is just a markdown file. No different than the Quarto files we learned about in the chapter on [Quarto files](#). However, naming it **README** gives it a special status. When we include a **README** file in our repository, GitHub will automatically add it to our repository’s homepage. We should use it to give others more information about our project, what our repository does, how to use the files in our repository, and/or how to contribute. So, we will definitely want a **README** file. Arthur may as well go ahead and check the box to create it along with his repository (although, we can always add it later).
- **Add `.gitignore`:** We will discuss `.gitignore` later. Briefly, you can think of it as a list of files we are telling GitHub to ignore (i.e., not to track). This gets back to versioning, which we discussed in the [Versioning](#) section of the introduction to git and GitHub chapter. For now, Arthur will just leave it as is.
- **License:** The GitHub documentation states that, “Public repositories on GitHub are often used to share open-source software. For your repository to truly be open source, you’ll need to license it so that others are free to use, change, and distribute the software.”⁹ Because we aren’t currently using our repository to create and distribute open-source software (like R!!), we don’t need to worry about adding a license. That isn’t to say that you won’t *ever* need to worry about a license. For more on choosing a license, we can consult the [GitHub documentation](#) or potentially consult with our employer or study sponsor. For example, our universities have officials that help us determine if our repositories need a license.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner *



arthur-epi ▾

Repository name *

r4epi_example_project



Great repository names are short and memorable. Need inspiration? How about [effective-octo-rotary-phone?](#)

Description (optional)

An example repository that accompanies the git and GitHub chapters in the R4Epi book.

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more](#).

Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

.gitignore template: None ▾

Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

License: None ▾

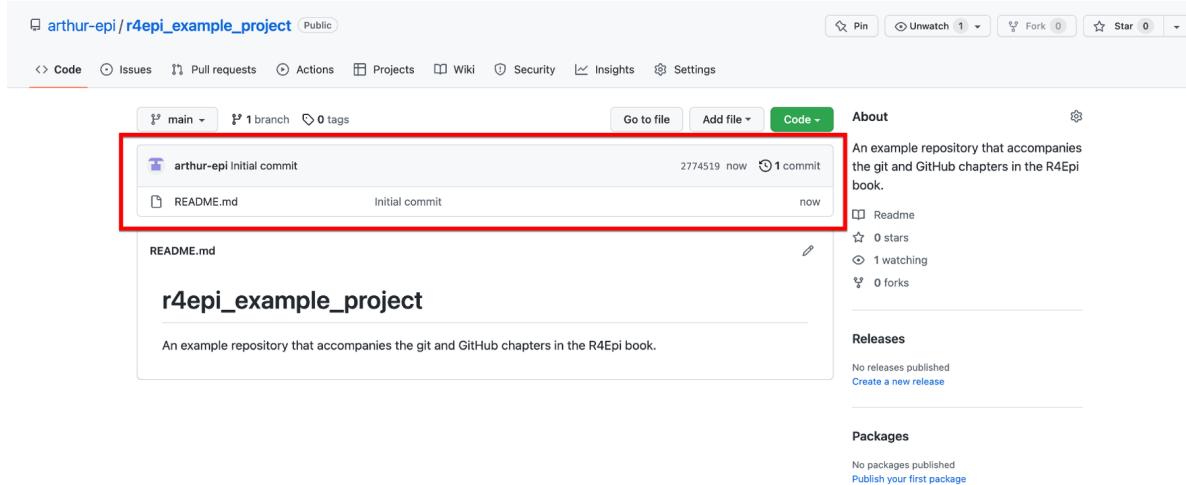
This will set main as the default branch. Change the default name in your [settings](#).

You are creating a public repository in your personal account.

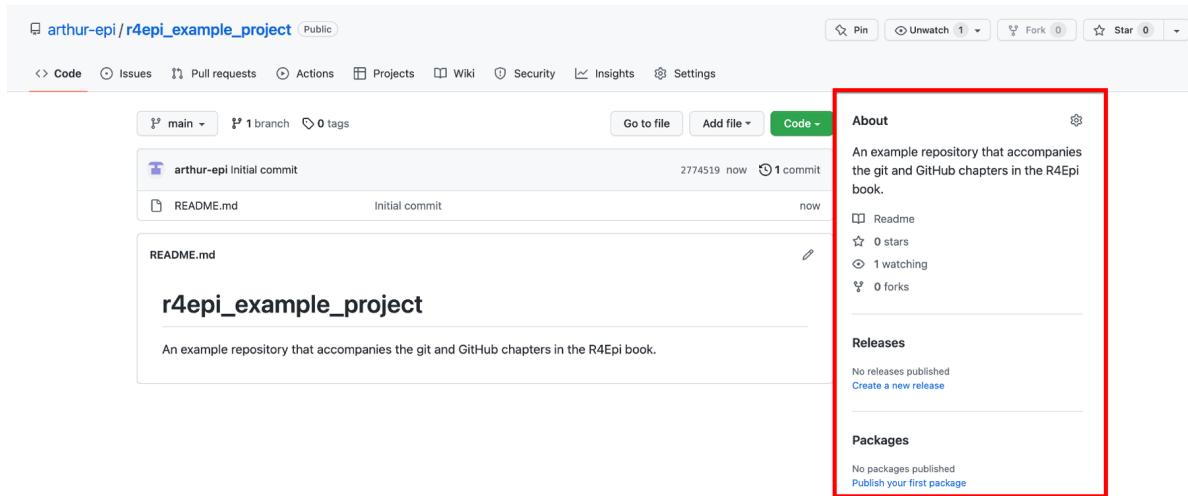
Create repository

Now, that he has completed all the setup steps, Arthur can click the green **Create repository** button. This will create his repository and take him to its homepage on GitHub. As you can

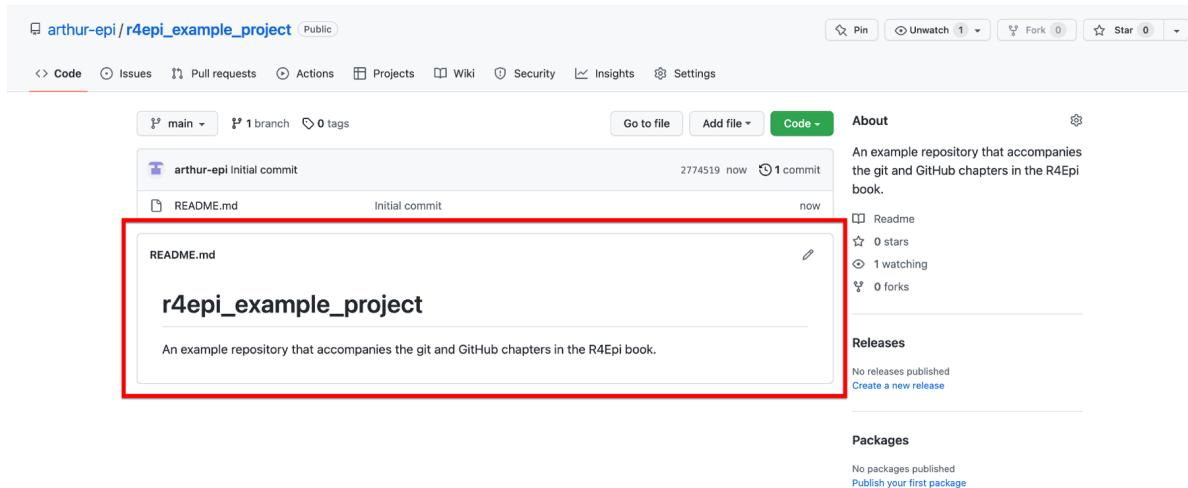
see in the screenshot below ([you can also navigate to the website yourself](#)), GitHub creates a basic little website for the repository. The top middle portion of the page (outlined in red below) displays all of the files and folders in the repository. Currently, the repository only contains one file – `README.md` – but Arthur will add others soon.



To the right of files and folders section of the homepage is the **About** section of the page. This section (outlined in red below) contains the repository's description, tags, and other information that we will ignore for now.



Below the files and folders section of the page is where the **README** file is displayed. Notice that by default, GitHub added the repository's name and description to the README file. Not a bad start, but we can add all kinds of cool stuff to README – including tables, figures, images, links, and other media. In fact, you can add almost anything to a README file that you can add to any other website. This is a great place to get creative and really make your project stand out!



Now, Arthur has a working GitHub repository up and running. Let's pause for a moment to and celebrate!

Okay, celebration complete. Now, what does he do with this new GitHub repository? Well, he does the four things covered in [Introduction to git and GitHub](#)

1. He will start adding files to his repository and document their purpose and evolution with **versioning**.
2. In the process, he will **preserve** his files, and by extension, his project.
3. Doing so will help to make his research more **reproducible**.
4. And make it easier for him to **collaborate** with others – including his future self.

Let's start by taking a look at versioning in GitHub. As we discussed in the [Versioning](#) section of the [Introduction to git and GitHub](#) chapter, GitHub uses the word **commit** to refer to taking a snapshot of the state of our project, similar to how we might typically think about saving a version of a document we are working on. We saw how we could view the version history of our Google Doc by clicking **File** then **Version history** then **See version history**. In GitHub, we can similarly view the version history (also called the commit history) of our repository. To do so, we navigate to our repository's homepage, and click on the word **commit** in the top right corner of the files section (outlined in red below).

The screenshot shows a GitHub repository page for 'arthur-epi/r4epi_example_project'. The 'Code' tab is selected. At the top, there are buttons for 'Pin', 'Unwatch', 'Fork', and 'Star'. Below the header, there are links for 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. A dropdown menu shows 'main' branch, '1 branch', and '0 tags'. On the right, there are buttons for 'Go to file', 'Add file', and 'Code'. The main content area shows a single commit: 'arthur-epi Initial commit' made at '2774519 now'. To the right of the commit message, there are three small buttons: a blue square with a white icon, a red square with a white number '1', and a green square with a white icon. A red box highlights this row. Below the commit, there is a file list: 'README.md' and 'Initial commit'. On the far right, there is an 'About' section with a brief description of the repository, followed by 'Readme', '0 stars', '1 watching', and '0 forks'. There are also sections for 'Releases' (no releases published) and 'Packages' (no packages published).

This will take us to our repository's version history page. Currently, this repository only has one commit – the “Initial commit”. This name is used by convention in the GitHub community to refer to the first commit in the repository. The history also tells us when the commit was made and who made it. On the right side of the commit, there are three buttons.

The screenshot shows the commit history for the same repository. The 'Code' tab is selected. The commit history shows a single entry: 'Commits on May 20, 2022' followed by 'Initial commit' made by 'arthur-epi' at '2774519' 'committed 2 minutes ago'. To the right of the commit message, there are three small buttons: a blue square with a white icon, a red square with a white number '1', and a green square with a white icon. A red box highlights this row. Below the commit, there are 'Newer' and 'Older' buttons.

1. The first button on the left that looks like two partially overlapping boxes will copy the commit's ID so that we can paste it elsewhere if we want. In GitHub, every commit is assigned a unique ID, which is also called an "SHA" or "hash". The commit ID is a string of 40 characters that can be used to refer to a specific commit. The 274519 displayed on the middle button is the first 7 characters of this commit's ID.
2. As noted above, the middle button is labeled with the first 7 characters of this commit's ID - 274519. Clicking on it will take us to a new screen with the details of what this commit does to the files in the repository (i.e., additions, edits, and deletions). Arthur will click it so we take a look momentarily.
3. The button on the far right, which is labeled with two angle brackets (< >) will take us back to the repository's homepage. However, the files in the repository will be set back to the state they were in when the commit was made. In this case, there is only one commit. So, there's no difference between the current state of the repository and the state it would be in if Arthur clicked this button. However, this button can be useful. If Arthur makes some changes to a file and then later wants to see what the file looked like before he made those changes, he can use this button to take a look.

Now, Arthur will click the middle button labeled with the short version of the commit ID.

On the page he is taken to, we can see more details about what commit 274519 does to the files in the repository. The top section of the page (outlined in red below) contains pretty much the same information we saw on the previous page. The little symbol on the left that looks kind of like a backwards 4 with open circles at the ends of the lines tells us which branch we are operating on. Branches are a more advanced topic that we will discuss later. Currently, our repository only has one branch – the default `main` branch – and the symbol followed by the word “main” is telling us that this commit is on the main branch. To the far right of this section, there is a button that says `Browse files`. Clicking this button does the exact same thing as the button on the previous page that was labeled with two angle brackets (< >). Below the `Browse files` button, are the words `0 parents` and `commit 277451996a7e9a0a6e583124d762db2a9cd439a2`. This tells us that this commit doesn't have any parent commits and that the full commit ID is `277451996a7e9a0a6e583124d762db2a9cd439a2`. We discussed commit ID's above. The parent commit is the commit or commits that this commit is based on. In other words, what were the other things that happened to get us to this point? Because this is the initial commit, there are no parent commits.

Initial commit

arthur-epi committed 3 minutes ago (Verified)

0 parents commit 277451996a7e9a0a6e583124d762db2a9cd439a2

Showing 1 changed file with 2 additions and 0 deletions.

README.md

```

@@ -0,0 +1,2 @@
1 + # r4epi_example_project
2 + An example repository that accompanies the git and GitHub chapters in the R4Epi book.

```

0 comments on commit 277451996a7e9a0a6e583124d762db2a9cd439a2

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Comment on this commit

The middle section of the commit details page tells us that applying this commit to the repository changes 1 file. In that file, there are two additions and no deletions. Below this text we can see which file was changed - `README.md`. This is also called the **diff view** because we can see the differences between this version of the file and previous versions of the file. In this case, because there wasn't a previous version of the file, we just see the two additions that were made to the file. They are the level one header that was added to the first line of the file (i.e., `# r4epi_example_project`) and our project's description was added to the second line of the file. These additions were made automatically by GitHub. We know they are additions because the background color is green and there is a little plus sign immediately to their left. We know which lines of the file were changed because GitHub shows us the line number immediately to the left of the plus signs.

The screenshot shows a GitHub commit details page for a repository named "arthur-epi/r4epi_example_project". The commit is titled "Initial commit" and was made by "arthur-epi" 3 minutes ago. It has 0 parents and a commit hash of 277451996a7e9a0a6e583124d762db2a9cd439a2. The commit message is as follows:

```
@@ -0,0 +1,2 @@
1 + # r4epi_example_project
2 + An example repository that accompanies the git and GitHub chapters in the R4Epi book.
```

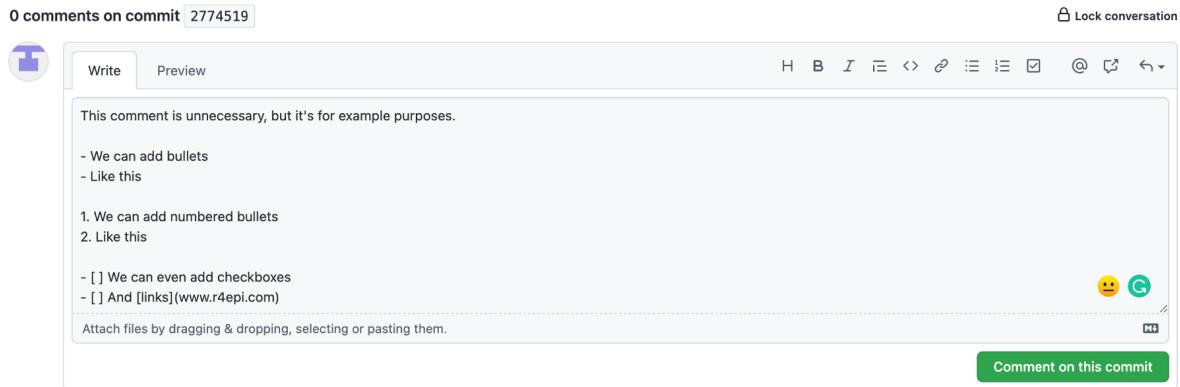
Below the commit message, there is a section titled "0 comments on commit 277451996a7e9a0a6e583124d762db2a9cd439a2". This section contains a text input field labeled "Leave a comment" and a green button labeled "Comment on this commit".

The final section of the commit details page shows us any existing comments that Arthur, or others, made about this commit. It also allows us, or others to create a new comment, using the text box.

This screenshot is identical to the one above, showing the same commit details for the "arthur-epi/r4epi_example_project" repository. The commit message and the "Comment on this commit" section are the same. A red box highlights the "Comment on this commit" button and the surrounding comment input area.

In the screenshot below, we can see an example comment. Note all the cool things features

GitHub comments allow us to use. We can format the text, add bullets, add links, and even add clickable checkboxes.



Finally, clicking the green `Comment on this commit` button adds our comment to the commit details page.

The screenshot shows a GitHub repository page for an 'Initial commit' to a 'main' branch. A recent commit by 'arthur-epi' is displayed, showing changes to 'README.md'. The commit message includes a link to 'R4Epi_example_project' and describes it as an example repository. Below the commit, there is a single comment from 'arthur-epi' stating: 'This comment is unnecessary, but it's for example purposes.' This comment includes a bulleted list and checkboxes for further examples. At the bottom, there is a comment input field with a 'Comment on this commit' button.

Let's pause here for a moment and try to appreciate how powerful GitHub already is compared to other cloud-based file storage services like Dropbox, Google Drive, or OneDrive. Like those file storage services, all of our files are backed up and preserved in the cloud and can easily be shared with others. However, unlike Dropbox, Google Drive, and OneDrive, we can turn our repository's homepage into a little website describing our project, we can view all the changes that have been made to our project over time, we can see which specific lines of each file have changed and how, and we can gather all comments, questions, and concerns about the files in one place. Oh, and it's **Free!**

Step 2: Clone the repository to your computer

At this point, Arthur's repository, which is just a fancy file folder, and the one file in his repository (README.md), only exist on the GitHub cloud.

Note

Side Note: What is “the GitHub cloud”? For our purposes, the cloud just refers to a specific type of computer – called a server – that physically exists somewhere else in the world, which we can connect to over the internet. GitHub owns many servers, and our files are stored on one of them. After we connect to the GitHub server, we can pass files back and forth between our computer and GitHub’s computer (i.e., the server).

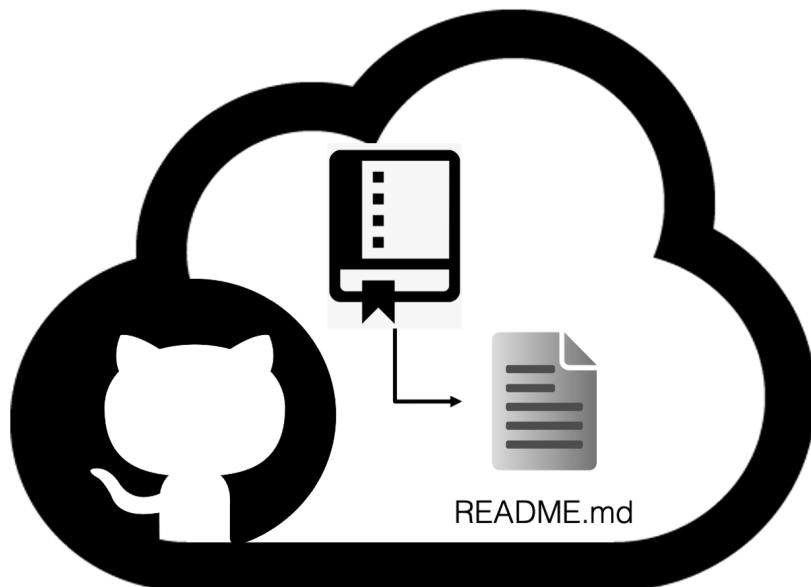


Figure 21.3: GitHub Cloud.

So, how does he get the repository from the GitHub cloud to his computer so that he can start making changes to it?

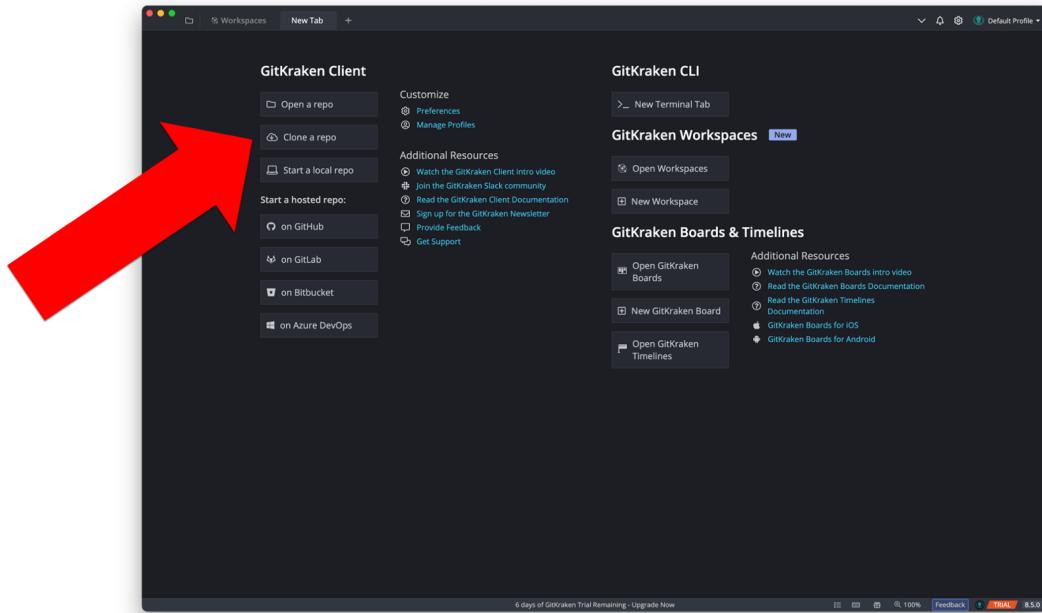
He will [clone](#) the repository to his computer. Don’t get thrown off by the funny name. You can simply think “make a copy of” whenever you see the word “clone” for now. So, he will “make a copy of” the repository on his computer. However, cloning the repository actually does two very useful things at once:

1. It creates a copy of our repository, and all of the files and folders in it, on our computer.
2. It creates a connection between our computer and the GitHub cloud that allows us to pass files back and forth.

There are multiple possible ways we could clone our repository, but we’re going to use

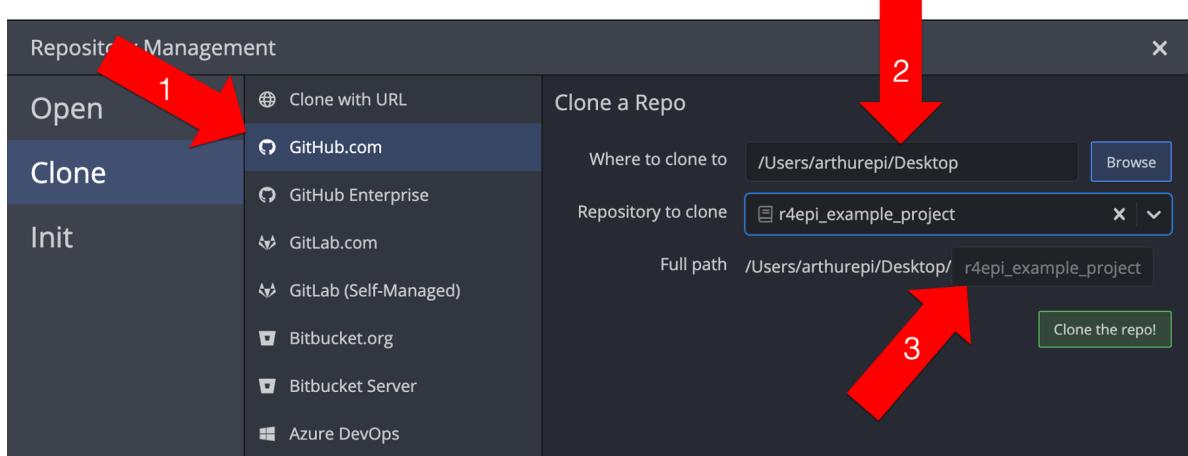
GitKraken in this book. If you did not already download GitKraken and connect it with your GitHub account as demonstrated at the beginning of the chapter, please do so now.

When we open GitKraken, we should see something similar to the screenshot below. Arthur will start the cloning process by clicking the **Clone a repo** button.



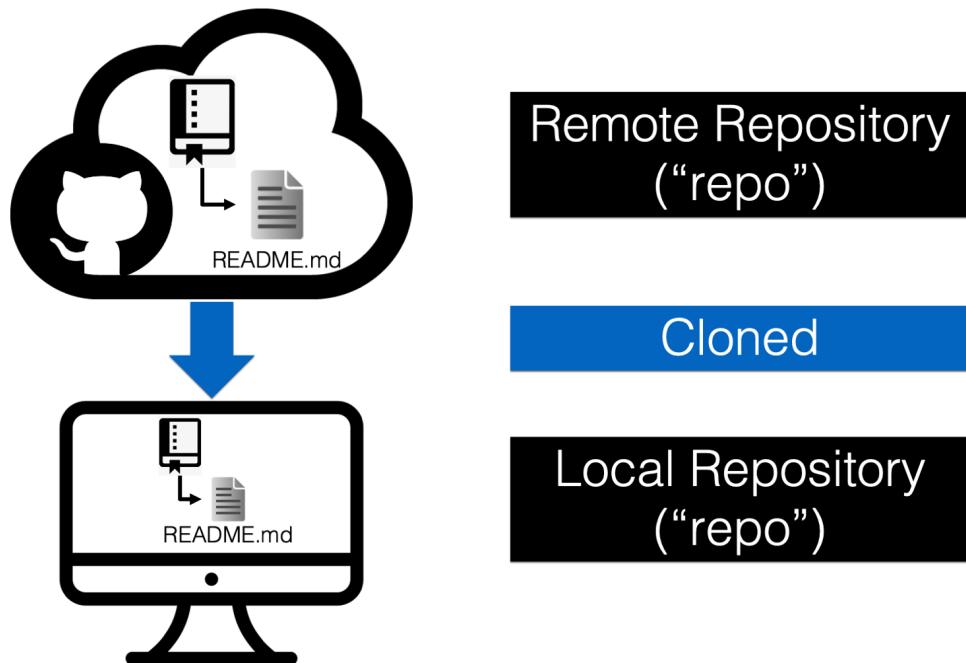
When the Repository Management dialogue box opens, he will need to make 3 changes.

1. Click **GitHub.com** in the clone menu. This tells GitKraken that the repository he wants to clone currently lives on his GitHub account. Note that it has to be on **his** account in order for it to show up on this list – not someone else’s account. We will learn how to get files from someone else’s account later.
2. Set the path where he wants the repository to be cloned to. Remember, the repository is just a folder with some files in it. When we clone the repository to our computer, those files and folders will live on our computer somewhere. We need to tell GitKraken where we want them to live. In the screenshot below, Arthur is just cloning the repository to his computer’s desktop.
3. Tell GitKraken which repository on his GitHub account he wants to clone. We can use the drop-down arrow to search a list of all of our repositories. In the screenshot below, Arthur selected the **r4epi_example_project** repository.



Finally, he will click the green `Clone the repo!` button. Now, he has successfully cloned his repository to his computer!

Before moving on, let's pause and review what just happened.



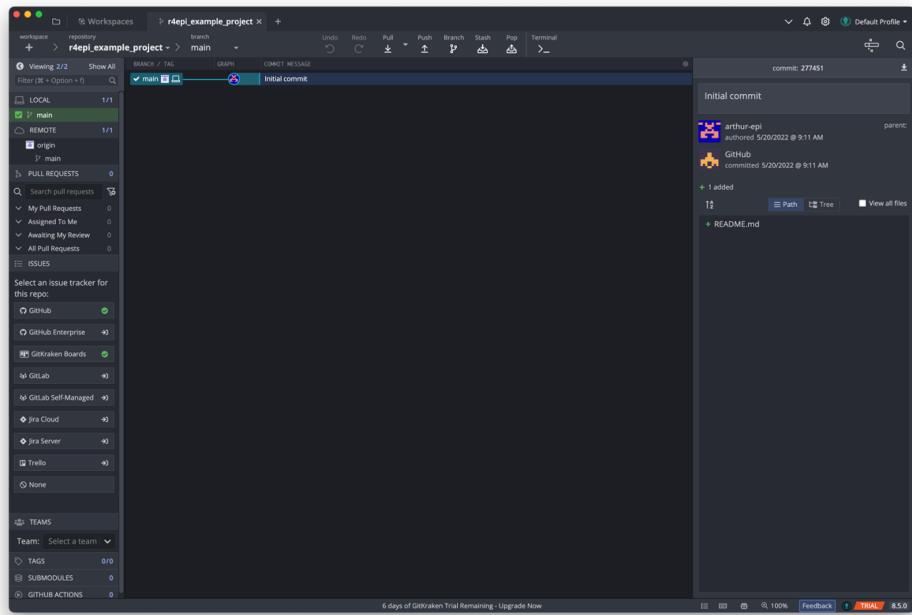
As we discussed above, Arthur's repository already existed on the GitHub cloud see Figure 21.3. In git terminology, the GitHub cloud called a **remote repository**, or “repo” for short. Remote repositories are just copies of our repository that live on the internet or some other network. Arthur then **cloned** his remote repository to his computer. That means, he made a copy of all of the files and folders on his computer. In git terminology, the repository on our computer is called a **local** repository.

Now that he has successfully cloned his repository, he should be able to view it in two different ways.

First, he should be able to see his repository's file folder on his desktop (because that's the location he chose above).



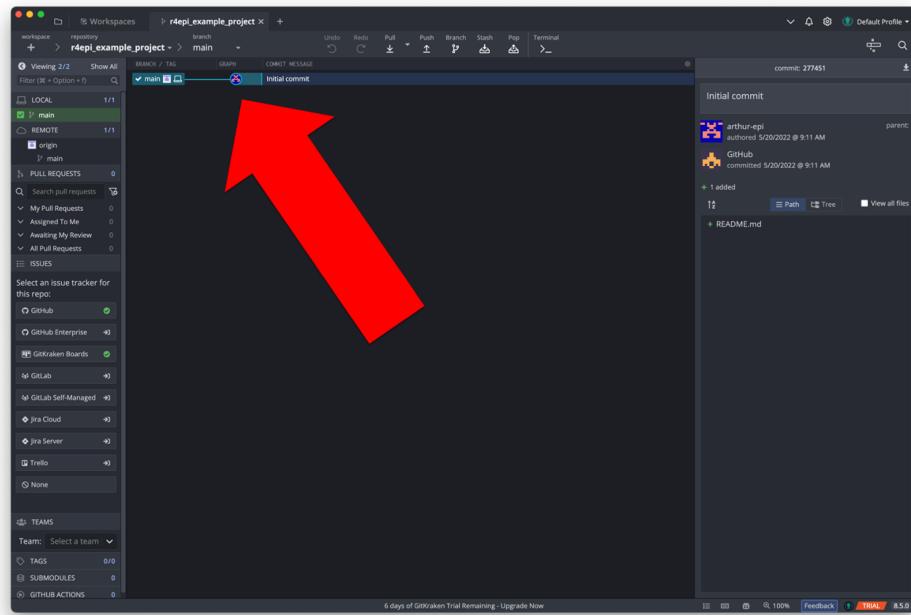
Second, he should be able to open a tab in GitKraken with all the versioning information about his repository.



Let's pause here and **watch a brief video** from GitKraken that orients us to the GitKraken user interface. For now, the first three minutes of the video is all we need. There may be some unfamiliar terms in the video. Don't stress about it! We will cover the most important parts after the video and learn some of the other terms in future examples.

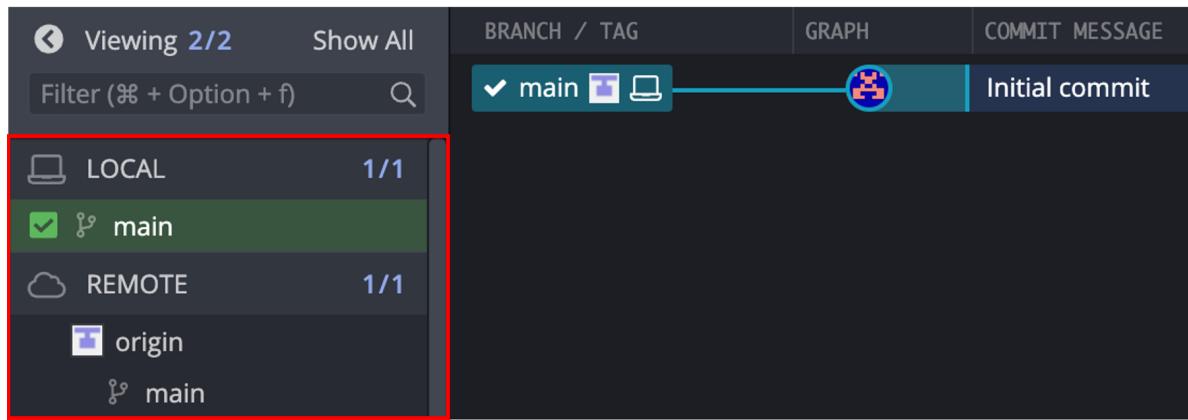
<https://www.youtube.com/embed/RiAeNSFjjLc>

Moving back to Arthur's repository, we can see that the repository graph in the middle section of the user interface has only on commit – the initial commit. This matches what we saw on GitHub.

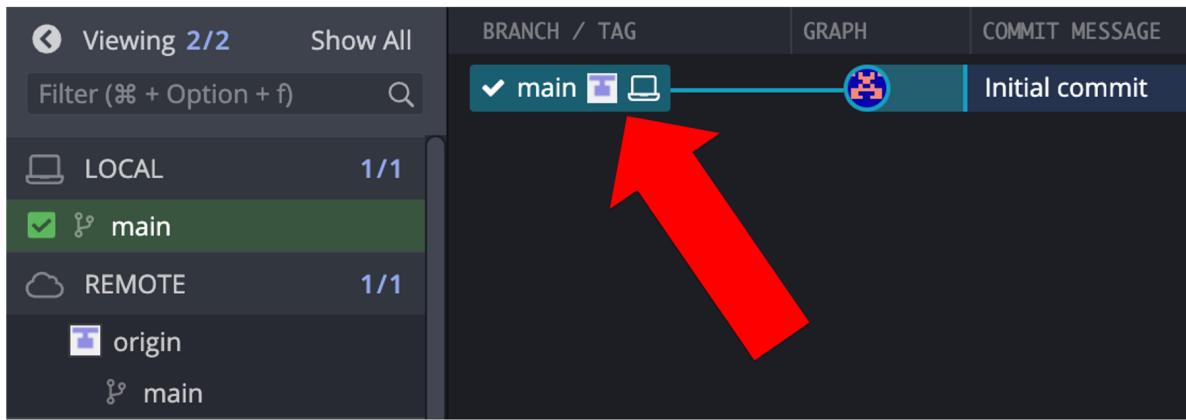


If we zoom in on the upper left corner of the left sidebar menu (outlined in red below), we can see that GitKraken is aware of two different places where the repository lives. First, it tells us that Arthur has a local repository on his computer with one branch – the main branch. Next, it tells us that there is one remote location for the repository – called “origin” – with one branch – the main branch.

The term “origin” is used by convention in the git language to refer to the remote repository that we originally cloned from. It uses the nickname “origin” instead of using the remote repository’s full URL (i.e., web address). Arthur could change this name if he wanted, but there’s really no need.

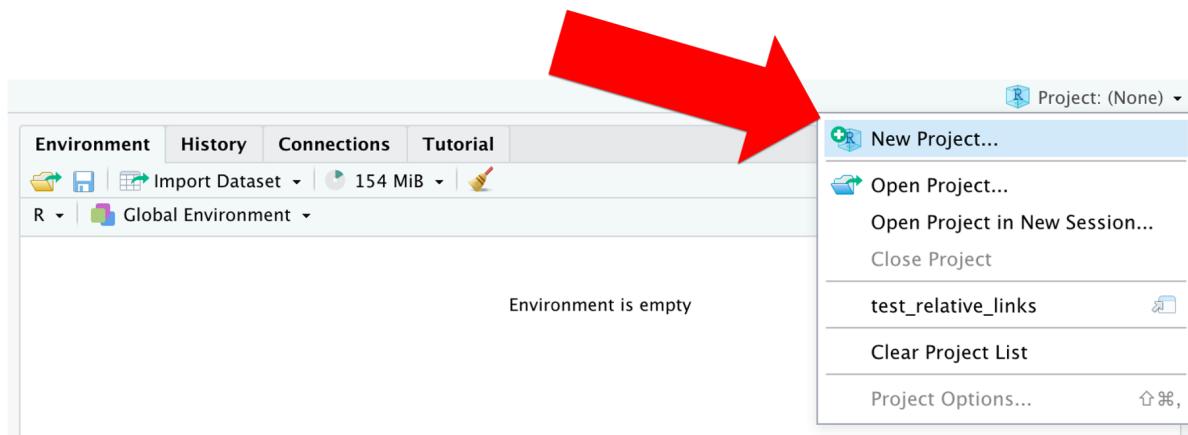


Another useful thing we can see in the current view, is that the local repository and the remote repository on GitHub are in sync. Meaning, the files and folders in the repository on Arthur's computer are identical to the files and folders in the repository on the GitHub cloud. We know this because the little white and gray picture that represents the remote repository and the little picture of the laptop that represents the local repository are located side-by-side on the repository graph (see red arrow below). When we have made changes in one location or another, but haven't synced those changes to the other location, the two icons will be in different rows of the repository graph. We will see an example of this soon.

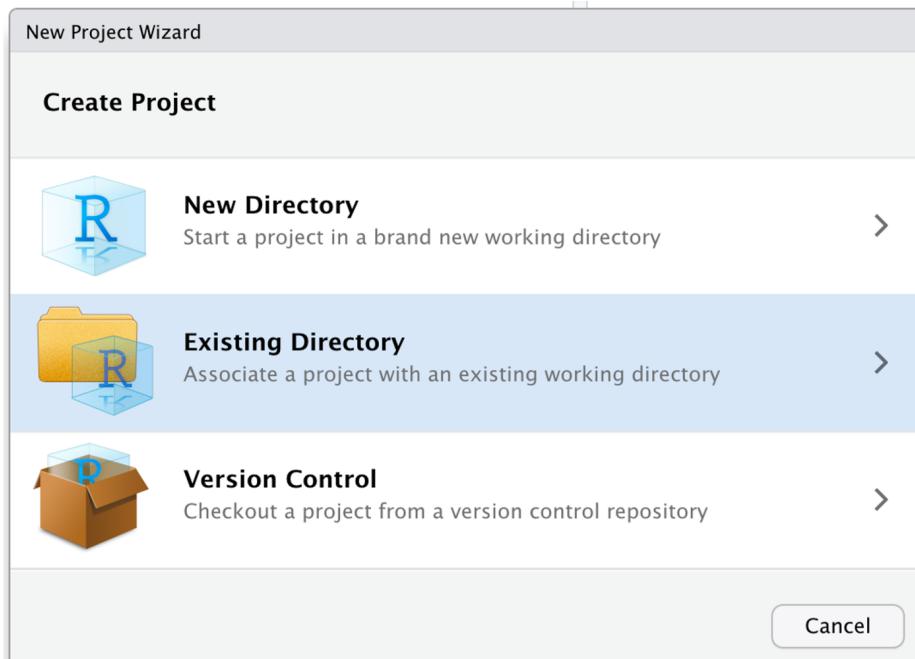


Step 3: Add an R project file to the repository

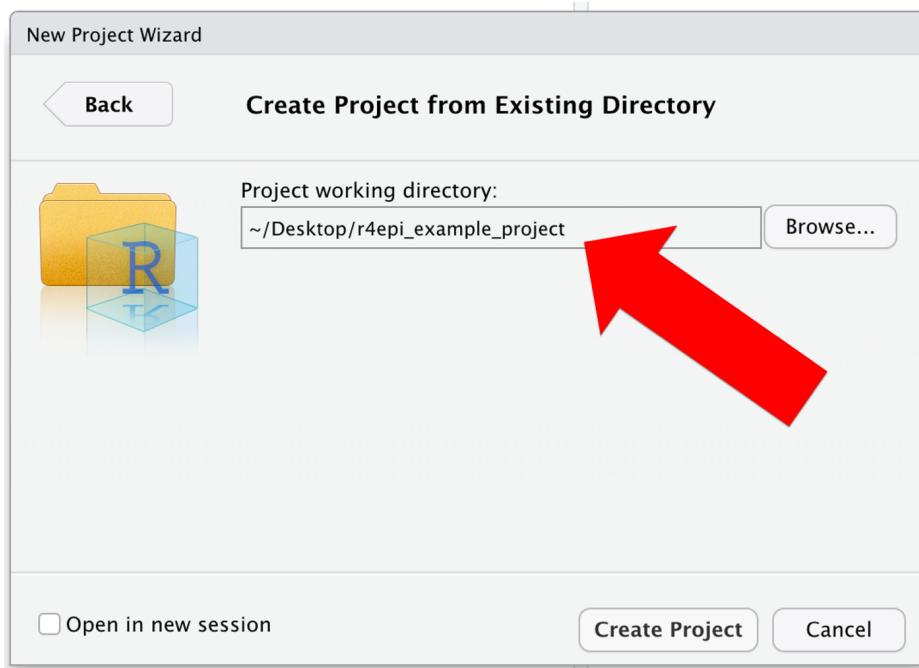
This step is technically optional, but we highly recommend it! We introduced [R projects](#) earlier in the book. Arthur will go ahead and add an R project file to his repository now. This will make his life easier later. To create a new R project, he just needs to click the drop-down arrow next to the words **Project: (None)** to open the projects menu. Then, he will click the **New Project...** option.



That will open the new project dialogue box. This time, he will click the **Existing Directory** option instead of clicking the **New Directory** option. Why? Because the directory (i.e., folder) he wants to contain his R project already exists on his computer. Arthur cloned it to his desktop in [step 2][Step 2: Clone the repository] above.



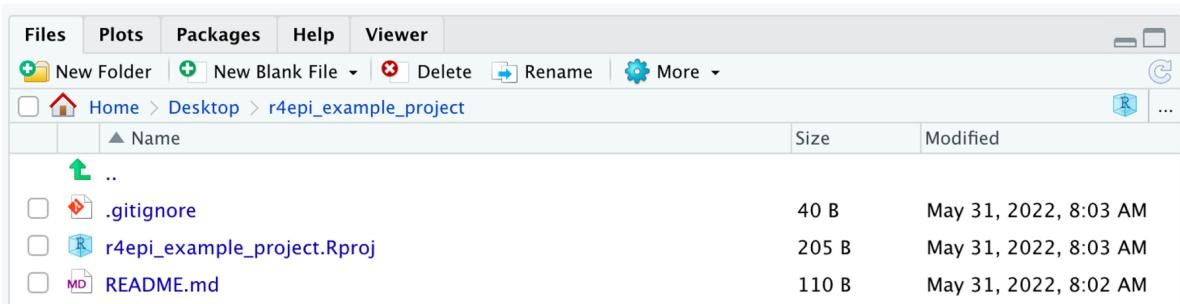
All Arthur has to do now, is tell RStudio where to find the `r4epi_example_project` directory on his computer using the **Browse...** button. In this case, on his desktop. Finally, he will click the **Create Project** button.



Step 4: Update and commit `gitignore`

Let's take a look at Arthur's RStudio files pane. Notice that there are now three files in the project directory. There is the `README` file, the `.Rproj` file, and a file called `.gitignore`. RStudio created this file automatically when Arthur designated the directory as an R project.

Outside of the name – `.gitignore` – there is nothing special about this file. It's just a plain text file. But naming it `.gitignore` tells the git software that it contains a list of files that git should ignore. By ignore, we mean, “pretend they don't exist.”



Arthur will now open the `.gitignore` file and see what's there.



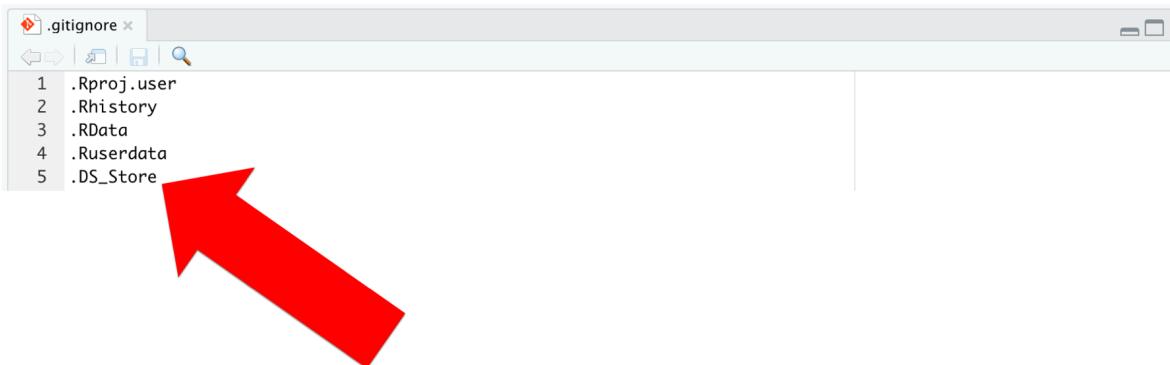
Currently, there are four files on the `.gitignore` list. These files were added automatically by RStudio to try to help him out. Tracking versions of these files typically isn't useful. Because

these files are on the `.gitignore` list, git and GitHub won't even notice if Arthur creates, edits, or deletes any of them. This means that they also won't ever be uploaded to GitHub.

At this point, Arthur is going to go ahead and add one more file to the `.gitignore` list. He will add `.DS_Store` to the list. `.DS_Store` is a file that the MacOS operating system creates automatically when a Mac user navigates to a file or folder using Finder. None of that really matters for our purposes, though. What does matter is that there is no need to track versions of this file and it will be a constant annoyance if Arthur doesn't ignore it.

If Arthur were using a Windows PC instead of a Mac, the `.DS_Store` file should not be an issue. However, adding `.DS_Store` to `.gitignore` isn't a bad idea even when using a Windows PC for at least two reasons. First, there is no harm in doing so. Second, if Arthur ever collaborates with someone else on this project who is using a Mac, then the `.DS_Store` file could find its way into the repository and become an annoyance. Therefore, we recommend always adding `.DS_Store` to the `.gitignore` list regardless of the operating system you personally use.

Adding `.DS_Store` (or any other file name) to the `.gitignore` list is as simple as typing `.DS_Store` on its own line of the `.gitignore` file and clicking **Save**.

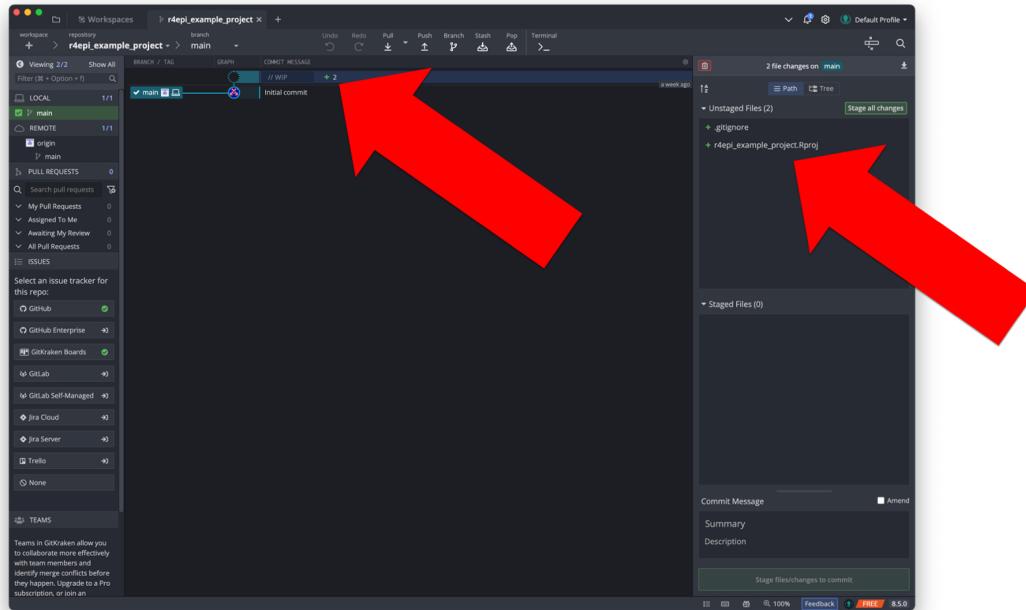


Typically, the next thing we would do after creating our repository is to start creating and adding the files we need to complete our analyses.

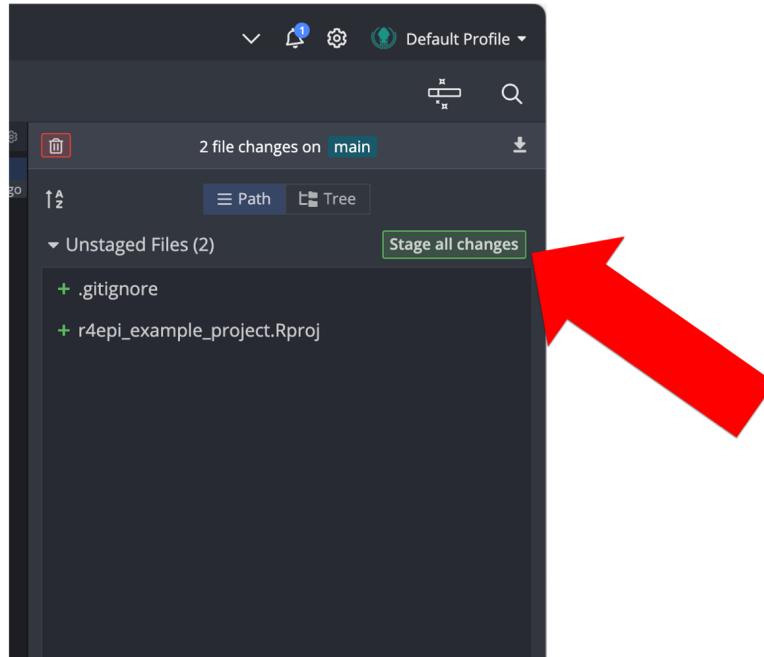
Now, Arthur will open GitKraken so we can take a look. Notice that Arthur's GitKraken looks different than it did the last time we viewed it. That's because we've been making changes to the repository. Specifically, we've added two files since the last commit was made. There are at least two ways we can tell that is the case.

First, the repository graph in the middle section of the user interface has now has two rows. The bottom row is still the initial commit, but now there is a row above it that says // WIP and has a + 2 symbol. WIP stands for work in progress and the + 2 indicates that there are two files that have changed (in this case, they were added) since the last commit. So, Arthur has been working on two files since his last commit.

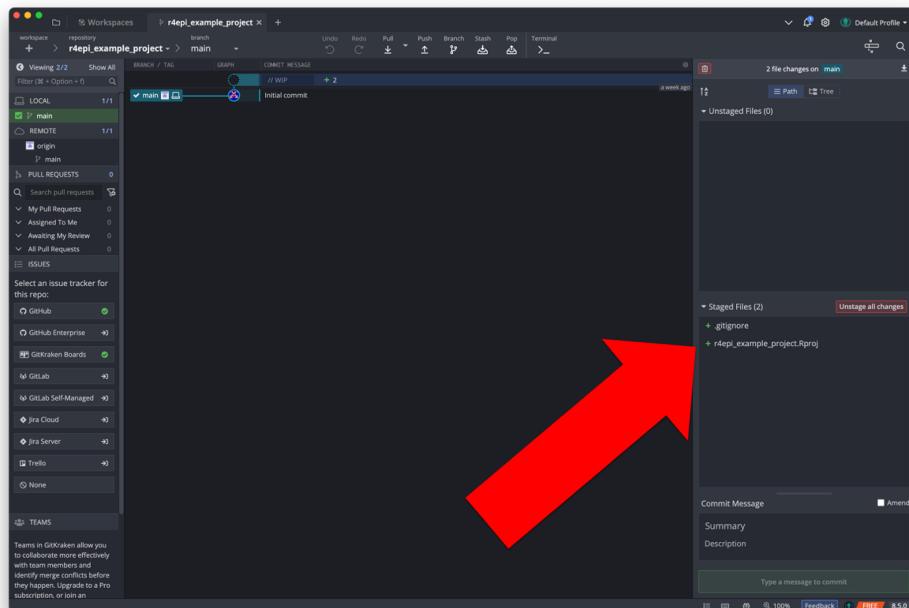
Additionally, the commit panel on the right side of the screen shows that there are two new uncommitted and unstaged files in the directory. They are `.gitignore` and `r4epi_example_project.Rproj`.



At this point, Arthur wants to take a snapshot of the state of his repository. Meaning, he wants to save a version of his repository as it currently exists. To do that, he first needs to **stage** the changes since the previous commit that he wants to be included in this commit. In this case, he wants to include all changes. So, he will click the green **Stage all changes** button located in the commit panel.



After clicking the `Stage all changes` button, the two new files are moved down to the `Staged Files` window of the commit panel.



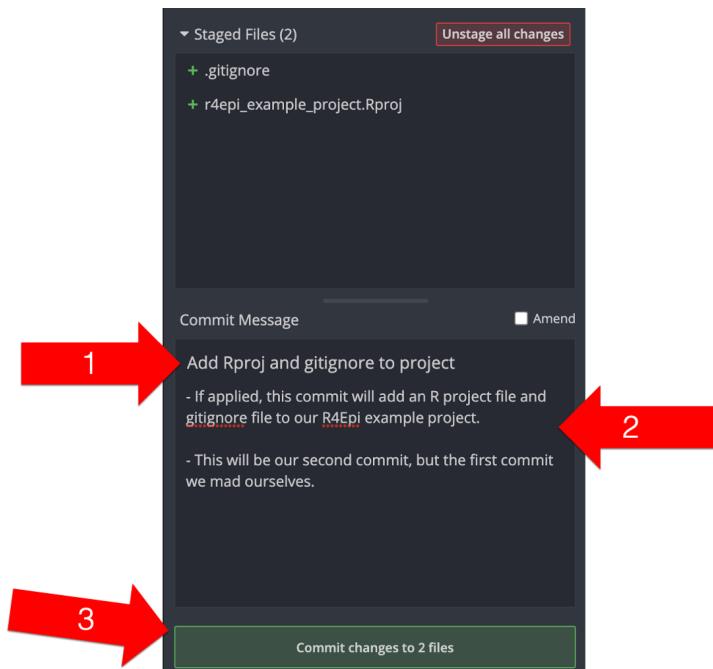
Next, Arthur will write a commit message. Just like there are [best practices for writing R code](#), there are also best practices for writing commit messages. Here is a link to a blog post that

we think does a good job of explaining these best practices: <https://cbea.ms/git-commit>.

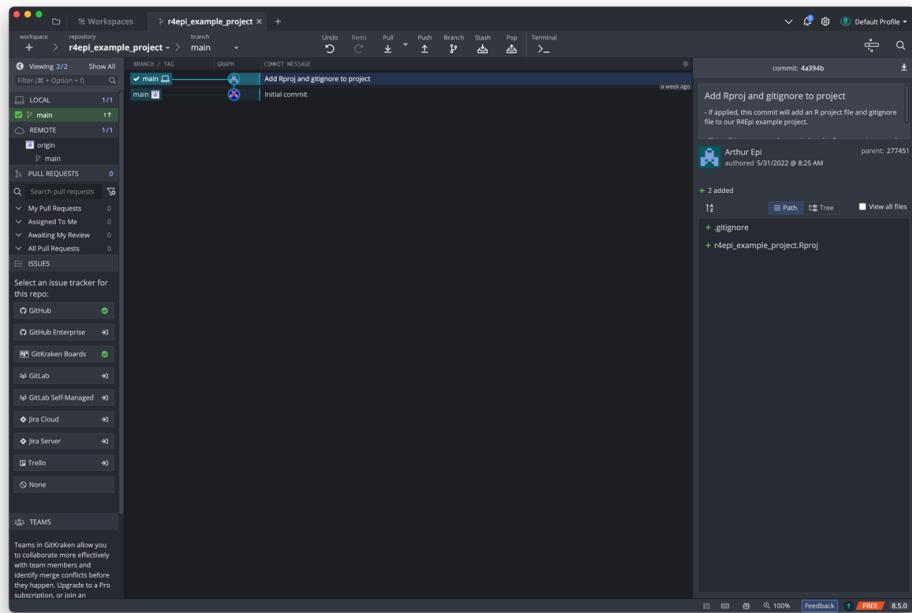
The first line is called the **commit message**. You can think of the commit message as a brief summary of what this commit does to the repository. This message will help Arthur and his collaborators find key commits later in the future. In this context, “brief” means 72 characters or less. GitKraken tries to help us out by telling us how many characters we’ve typed in our commit message. Additionally, the commit message should be written in the imperative voice – like a command. Another way to think about it is that the commit message should typically complete the phrase, “If applied, this commit will...”. The screenshot below shows that Arthur wrote **Add Rproj and gitignore to project** (red arrow 1).

In addition to the commit message, there is also a description box we can use to add more details about the commit. Sometimes, this is unnecessary. However, when we do choose to add a description, it is best practice to use it to explain *what* the commit does or *why* we chose to do it rather than *how* it does whatever it does. That’s in the code. In the screenshot below, you can see that Arthur added some bulleted notes to the description (red arrow 2).

Finally, Arthur will click the green commit button at the bottom of the commit panel (red arrow 3). This will commit (save) a version of our repository that includes the changes to any of the files in the **Staged Files** window.



And here is what his GitKraken screen looks like after committing.



Let's pay special attention to what is being displayed in a couple of different areas. We'll start by zooming in on the commit panel.

At the top of the commit panel, we can see the short version of the commit ID – 4a394b. Below that, we can see the commit message and description. Below that, we can see who created the commit and when. This tends to be more useful when we are collaborating with others. To the right of that information, GitKraken also shows us the commit ID for this commit's parent commit – 277451. Finally, it shows us the file changes that this commit applies to our repository. More specifically, it shows us the changes that commit 4a394b makes to commit 277451.

commit: 4a394b

Add Rproj and gitignore to project

- If applied, this commit will add an R project file and gitignore file to our R4Epi example project.

Arthur Epi authored 5/31/2022 @ 8:25 AM parent: 277451

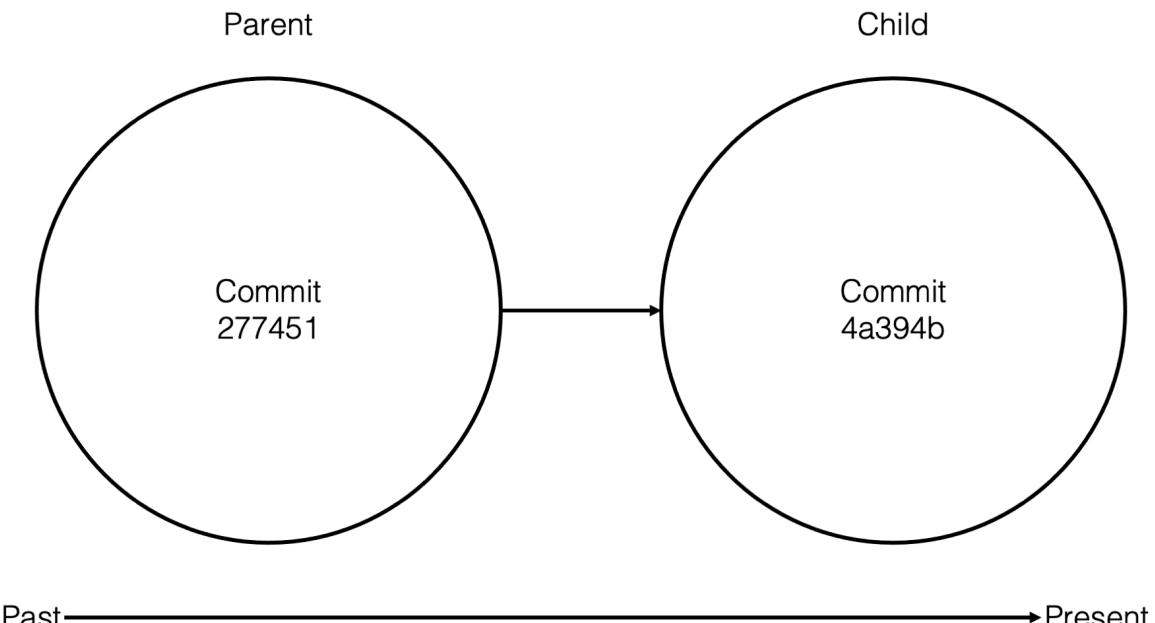
+ 2 added

Path Tree View all files

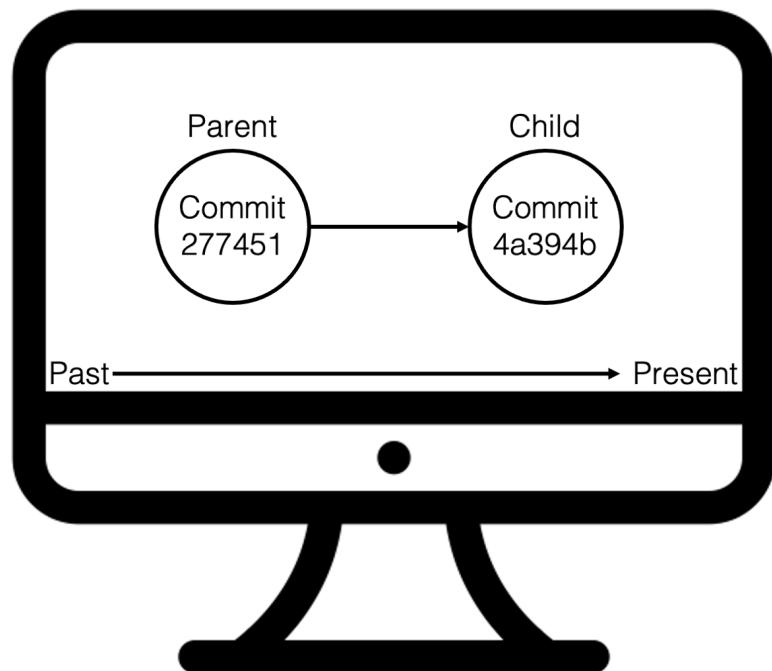
+ .gitignore

+ r4epi_example_project.Rproj

At this point, you may be wondering what this whole parent-child thing is and why we keep talking about it. The diagram below is a very simple graphical representation of how git views our repository. It views it as a series of commits that chronologically build our repository when they are applied to each other in sequence. Familial terms are often used in the git community to describe the relationship between commits. For example, in the diagram below commit 4a394b is a child of commit 288451. Child commits are always more recent than parent commits. This knowledge is not incredibly useful to us at this point, but it can be helpful when we start to learn about more advanced topics like merging commits. For now, just be aware of the terminology.



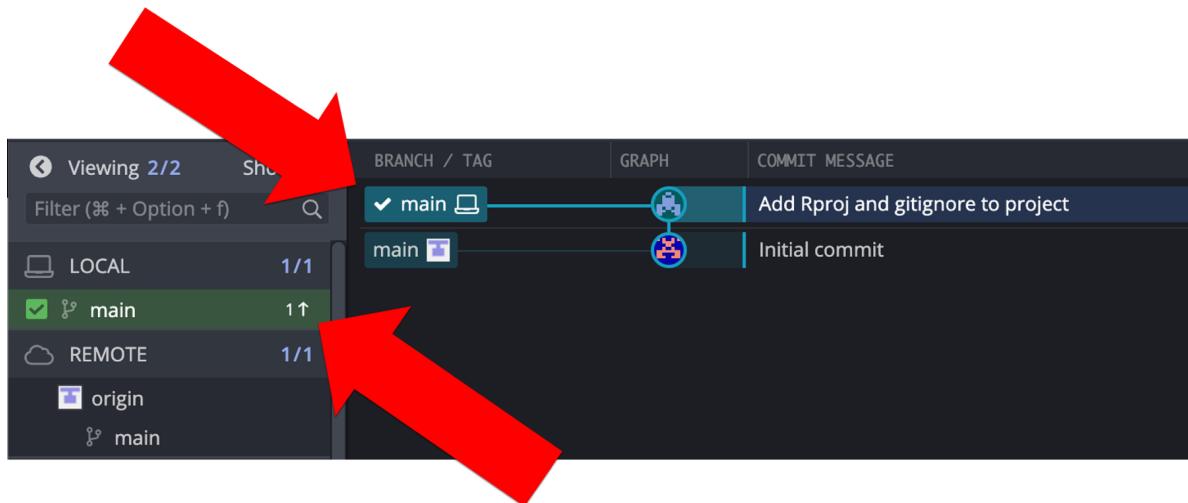
It is also important to point out that Arthur's most recent commit (4a394b) **only exists in his local repository**. That is, the repository on his computer. He has not yet shared the commit – or the new files associated with the commit – to the remote repository on GitHub.



How do we know? Well, one way we can tell is by looking at Arthur's GitKraken window. In

the repository graph, the local repository (i.e., the little laptop icon) and the remote repository (i.e., the little gray and white icon) are on different rows. Additionally, there is a little 1 next to an up arrow displayed to the left of the main branch of our local repository in the left panel of GitKraken. Both of these indicate that the most recent commits contained in each repository are different. Specifically, that the local repository is one commit ahead of the remote repository.

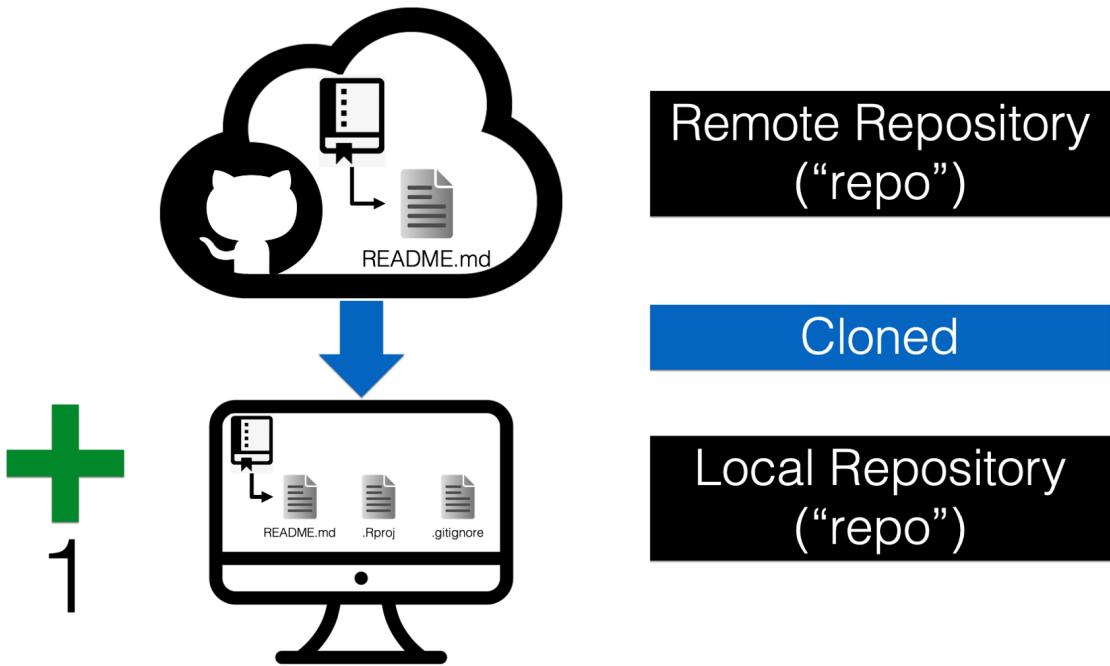
This concept is important to understand. In Google Docs, when we made a change to our document locally, that change was automatically synced to Google's servers. We didn't have to *do* anything to save/create a version of the document. We had to put in a little effort if we wanted to name a particular version, but the version itself was already saved – identified using a date-time stamp. Conversely, git does not automatically make commits (i.e., save snapshots of the state of the files in our repository), nor does our local repository automatically sync up with our remote repository (in this case, GitHub). We have to do both of these things manually. This will create a little extra work for us, but it will also give us a lot more control.



As one additional check, Arthur can go look at the repository's commit history on GitHub. As shown in the screenshot below, the commit history still only shows one commit – the initial commit.



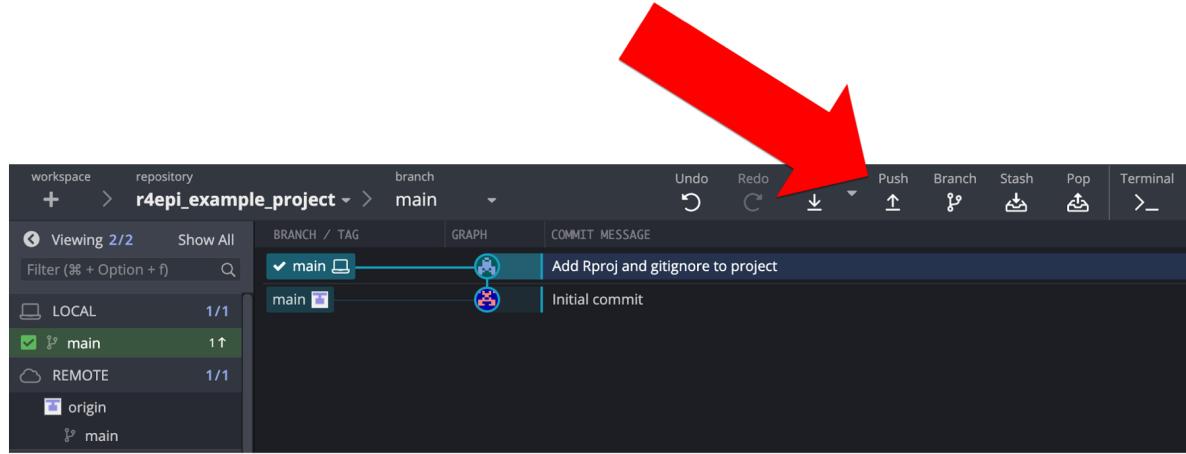
Let's quickly pause and recap what Arthur has done so far.



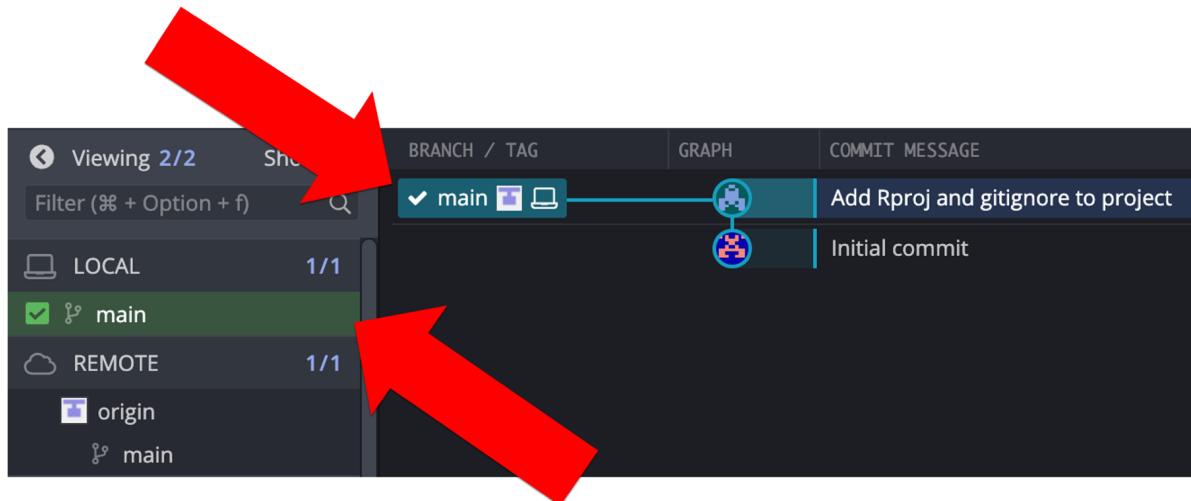
First, Arthur created a repository on GitHub. It was a remote repository because he accesses it over the internet. Then, he cloned (i.e., made a copy of) the remote repository to his computer. This copy is referred to as a local repository. Next, Arthur made some changes to

the repository locally and committed them. At this point, the local repository is 1 commit ahead of the remote repository, and the changes that Arthur made locally are not currently reflected on GitHub.

So, how does Arthur sync the changes he made locally with GitHub? He will **push** them to GitHub, which GitKraken makes incredibly easy. All he needs to do is click the Push button at the top of his GitKraken window (see below).



After doing so, we will once again see some changes. What changes do you notice in the screenshot below?



In the repository graph, the local repository (i.e., the little laptop icon) and the remote repository (i.e., the little gray and white icon) are back on the same row. Additionally, the little 1 next to an up arrow is no longer displayed in the left panel. Both of these changes indicate that the most recent commits contained in each repository are the same.

And if Arthur once again checks GitHub...

A screenshot of a GitHub repository page for 'r4epi_example_project'. The page shows a commit history with four entries:

- arthur-epi Add Rproj and gitignore to project ... (commit 4a394b7, 32 minutes ago)
- .gitignore Add Rproj and gitignore to project (32 minutes ago)
- README.md Initial commit (11 days ago)
- r4epi_example_project.Rproj Add Rproj and gitignore to project (32 minutes ago)

A large red arrow points from the text 'He will now see that the GitHub repository also has two commits.' to the commit history section.

README.md

r4epi_example_project

An example repository that accompanies the git and GitHub chapters in the R4Epi book.

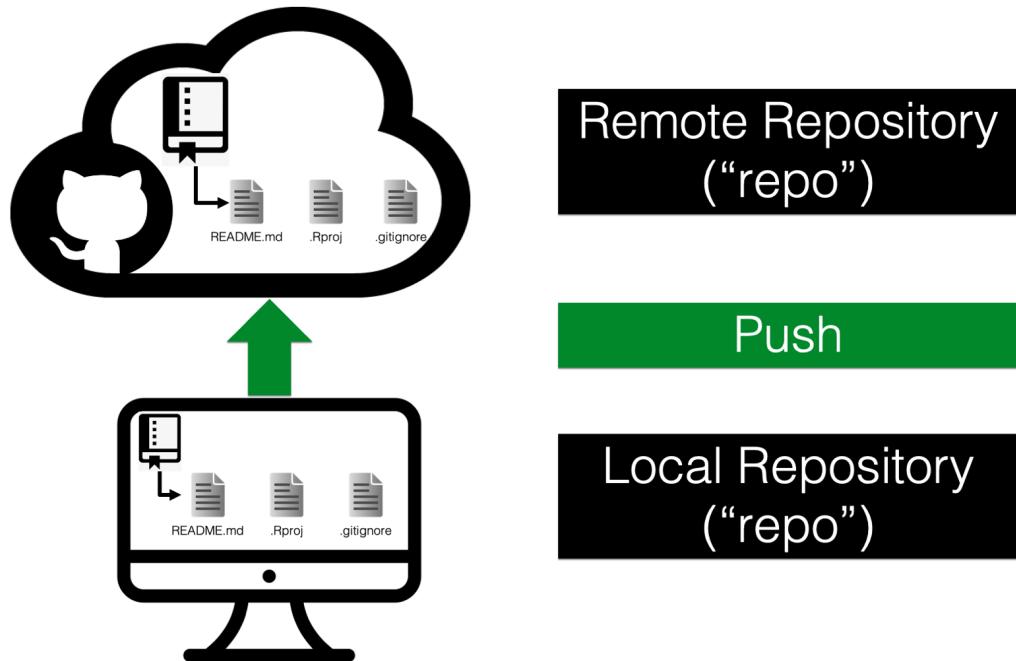
He will now see that the GitHub repository also has two commits. He can click on the text that says **2 commits** to view each commit in the commit history.

A screenshot of the commit history for the 'r4epi_example_project' repository. It shows two main commit groups:

- Commits on May 31, 2022**:
 - Add Rproj and gitignore to project ... (commit 4a394b7, arthur-epi committed 33 minutes ago)
- Commits on May 20, 2022**:
 - Initial commit (Verified, commit 2774519, arthur-epi committed 11 days ago)

A large red arrow points from the text 'In the commit history, he can now see commit 4a394b7.' to the commit history section.

In the commit history, he can now see commit 4a394b7. Let's take another pause here and recap.

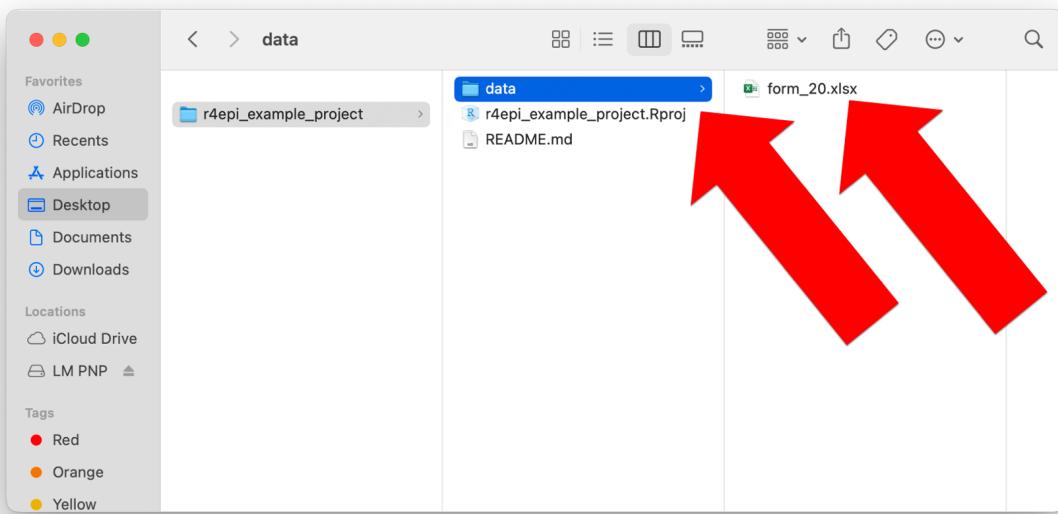


First, Arthur created a repository on GitHub. Then, he **cloned** the remote (i.e., GitHub) repository to his computer. Next, Arthur made some changes to the repository locally and **committed** them locally. Finally, he **pushed** the local commit up to GitHub. Now, his GitHub repository and local repository are in sync with each other.

We realize that it probably seems like it took a lot of work for Arthur to get everything set up. But in reality, all of the steps up to this point will only take a couple of minutes once you've gone through them a few times.

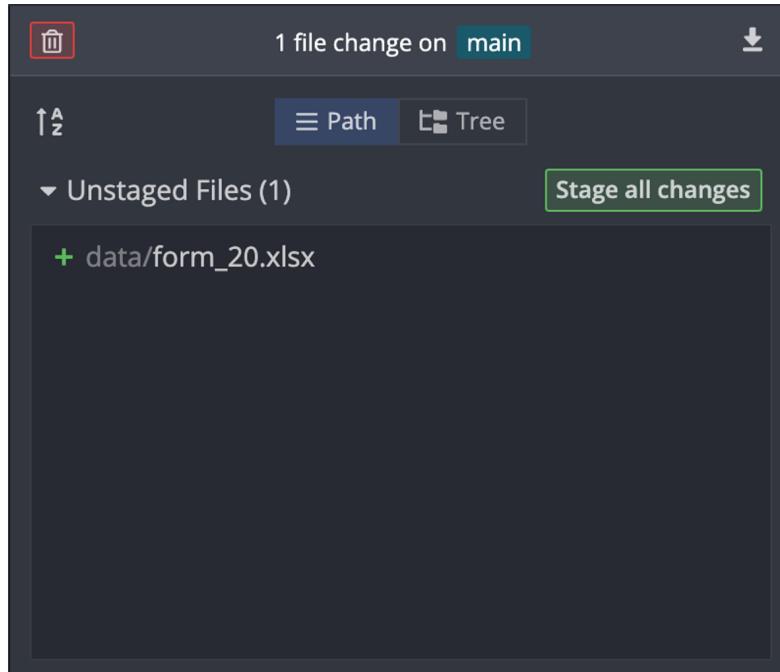
Step 5: Keep adding and committing files

At this point, Arthur has his repositories all set up and is ready to start rocking and rolling on his actual data analysis. To round out this example, Arthur will add some data to his repository that he will eventually analyze using R.



The screenshot above shows that Arthur created a new folder inside the R project directory called `data`. He created it in the same way he would create any other new folder in his computer's operating system. Then, he added a data set to the data folder he created. This particular data set happens to be stored in an Excel file named `form_20.xlsx`.

Now, when Arthur checks GitKraken, this is what he sees in the commit panel.

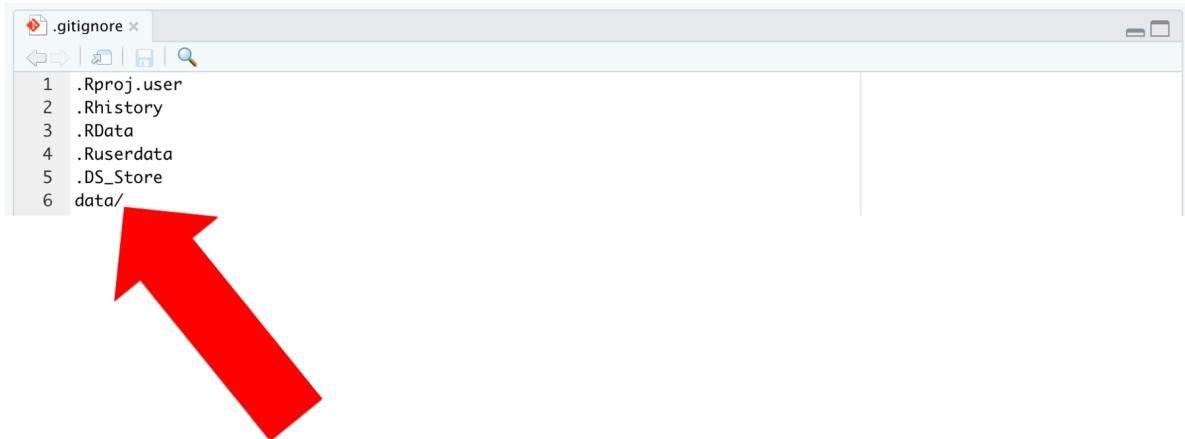


Just like before, GitHub is telling Arthur that he has a new unstaged file in the repository. Stop for a moment and think. What should Arthur do next?

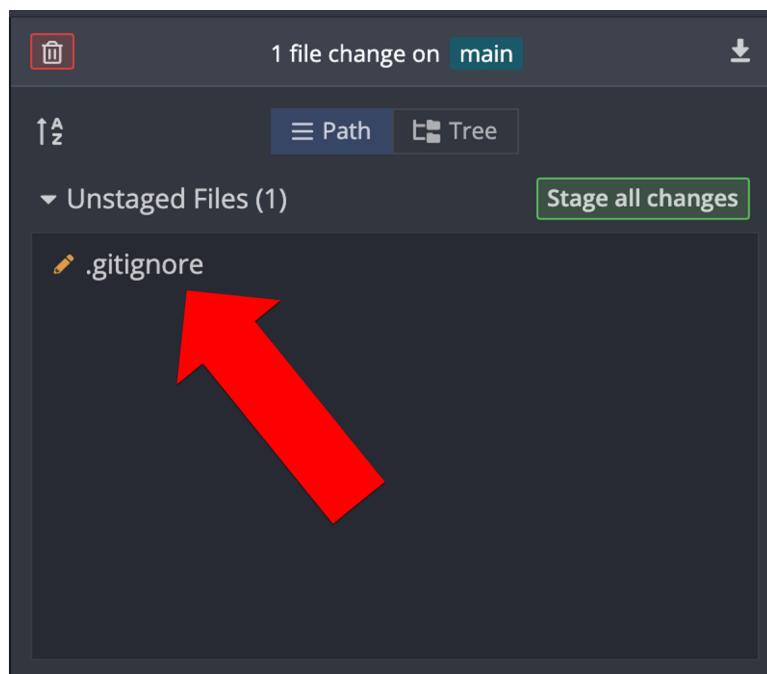
Was your answer, “stage and commit the new file”? If so, slow down and think again. Remember, in general, we don’t *ever* want to commit our research data to our GitHub repository. GitHub is not typically considered secure or private. So, how can Arthur keep the data in his local repository so that he can work with it, keep his local repository synced with GitHub, but make sure the data doesn’t get pushed up to GitHub?

Do you remember earlier when Arthur told git and GitHub to ignore the `.DS_Store` file? In exactly the same way, Arthur can tell git and GitHub to ignore this data set. And once it’s ignored, it won’t ever be pushed to GitHub. Remember, our local git repository only includes files it’s **tracking** in commits, and it only pushes commits (and the files included in them) up to GitHub.

In the screenshot below, Arthur added `data/` to line 6 of the `.gitignore` file. He could have added `form_20.xlsx` instead. That would have told git to ignore the `form_20.xlsx` data set specifically. However, Arthur doesn’t want to push *any* data to GitHub – including any data sets that he may add in the future. By adding `data/` to the `.gitignore` file, he is telling git to ignore the entire folder named `data` and all of the files it contains – now and in the future.



After saving the updated .gitignore file, the commit pane in GitKraken changes once again.



The new file `data/form_20.xlsx` is no longer showing up as an unstaged change. Instead, the only unstaged change showing up is the edited `.gitignore` file. We can tell that the changes to the `.gitignore` file are edits – as opposed to adding the file for the first time – because

there is a little pencil icon to the left of the file name instead of a little green plus icon. Now what should Arthur do next?

Was your answer, “stage and commit the edited file”? If so, you are correct! Now it is safe for Arthur to go ahead and commit these changes.

After doing so, he can see that the GitHub repository contains 3 commits. Additionally, as shown the red box below, the data folder is nowhere to be found among the files contained in the GitHub repository.

The screenshot shows a GitHub repository interface. At the top, there are buttons for 'main' (with a dropdown arrow), '1 branch', '0 tags', 'Go to file', 'Add file', and a green 'Code' button. Below this, a list of commits is displayed:

File	Commit Message	Time
.gitignore	Add the data folder to gitignore	now
README.md	Initial commit	11 days ago
r4epi_example_project.Rproj	Add Rproj and gitignore to project	1 hour ago

Arthur will now add one final file to the `r4epi_example_project` as part of this example. He will add an Quartofile with a little bit of R code in it. The code will import `form_20.xlsx` into the global environment as a data frame.

```

1 ---  

2 title: "Import Form 20 data for the R4Epi Example Project"  

3 ---  

4  

5 # ★ Overview  

6  

7 In this file, we import the mtcars data. This file is unrealistically simple, but we are using it for demonstration purposes only.  

8  

9  

10 # 📦 Load packages  

11  

12 ```{r message=FALSE}  

13 library(dplyr, warn.conflicts = FALSE)  

14 library(readxl)  

15 ````  

16  

17  

18 # 📈 Import data  

19  

20 This data is packaged with base R.  

21  

22 ```{r}  

23 form_20 <- read_excel("data/form_20.xlsx")  

24 ````  

25  

26 ```{r}  

27 glimpse(form_20)  

28 ````
```

Rows: 3
Columns: 4
\$ date_received <chr> "2013-08-22", "2013-08-22", "2013-08-22"
\$ name_last <chr> "Cooper", "Rodriguez", "Smith"
\$ name_first <chr> "Samantha", "Leslie", "Jane"
\$ education <dbl> 4, 8, 5

An then he will commit and push the `data_01_import.Rmd` to GitHub in the same way he committed and pushed previous files to Github.

Arthur can continue adding files to his local repository and then pushing them to GitHub in this fashion for the remainder of the time he is working on this project, and the [introduction to git and GitHub chapter](#) discusses *why* he should consider doing so.

After going through this example, many students have three lingering questions:

1. How often should we commit?
2. How often should we push our commits to GitHub?
3. If we can't use GitHub to share our data, how *should* we share data?

We will answer questions 1 & 2 immediately below. We will answer the third question in the [next example](#).

21.6 Committing and pushing

As we are learning to use git and GitHub, it is reasonable to ask how often we should commit our work as we go along. For better or worse, there is no hard-and-fast rule we can give you here. In Happy Git and GitHub for the useR, Dr. Jennifer (Jenny) Bryan writes that we should commit “every time you finish a valuable chunk of work, probably many times a day.”¹⁰ This seems like a pretty good starting place to us.

Of course, a natural follow-up question is to ask how often we should push our commits to GitHub. We could automatically push every commit we make to GitHub as soon as we make it. However, this isn’t always a good idea. It is much easier to edit or rollback commits that we have only made locally than it is to edit or rollback commits that we’ve pushed to our remote repository. For example, if we accidentally include a data set in a commit and push it to GitHub, this is a much bigger problem than if we accidentally include a data set in a commit and catch it before we push to GitHub. For this reason, we don’t suggest that you automatically push every commit you make to GitHub. So, how often *should* you push? Well, once again, there is no hard-and-fast rule. And once again, we think Dr. Bryan’s advice is a good starting point. She writes, “Do this [push] a few times a day, but possibly less often than you commit.”¹⁰ It is also worth noting that how often you commit and push will also be dictated, at least partially, by the dynamics of the group of people who are contributing to the repository. So far, we have really only seen a repository with a single contributor (i.e., Arthur Epi). That will change in the next example.

The advice above about committing and pushing may seem a little vague to you right now. It *is* a little vague. We apologize for that. However, we believe it’s also the best we can do. On the bright side, as you practice with git and GitHub, you will eventually fall into a rhythm that works well for you. Just give it a little time!

21.7 Example 3: Contribute to a research project

When our research assistants begin helping us with data management and analysis projects, we often have them start by going to the project’s GitHub repository to read the existing documentation and [clone](#) all the existing code to their computer. This example is going to walk through that process step-by-step. For demonstration purposes, we will work with the example repository that our fictitious research assistant named Arthur Epi created in [Example 2 above](#).

 Note

Side Note: It’s probably worth noting that in most real-world scenarios the roles here would be reversed. That is, we (Brad or Doug) would have created the original repository

and Arthur would be working off of it. However, the example repository above was already created using Arthur’s GitHub account, and we will continue to work off of it in this example. If you are a research assistant working with us (i.e., Brad or Doug) in real life, and using this example to walk yourself through getting started on a real project, you should insert yourself (and your GitHub account) into Brad’s role (and GitHub account) in the example below.

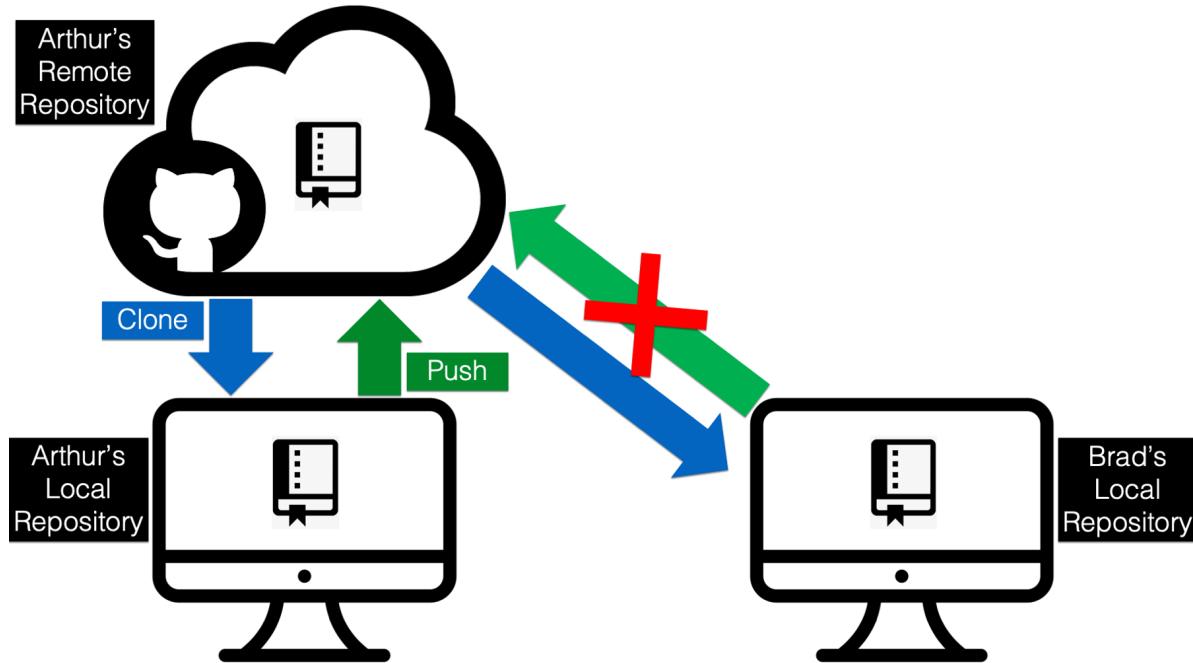
In this example, we’re going to work collaboratively with Arthur on the `r4epi_example_project`. Arthur could have just emailed us all of the project files, but sometimes that might be *many* files, some of them may be very large, and he runs the risk of forgetting to send some of them by accident. Further, every time any of the contributors adds or updates a file, they will have to email all the other contributors the new file(s) and an explanation of the updates they’ve made. This process is typically inefficient and error prone. Conversely, Arthur could set up a shared folder on a cloud-based file storage service like Dropbox, Google Drive, or OneDrive. Doing so would circumvent the issues caused by emailing files that we just mentioned (i.e., many files, large files, forgetting files, and manually sending updates). However, Dropbox, Google Drive, and OneDrive aren’t designed to take advantage of all that git and GitHub have to offer (e.g., project documentation, versioning and version history, viewing differences between code versions, issue tracking, creating static websites for research dissemination, and more). Because Arthur created his repository on GitHub, all of the files and documentation we need to get started assisting him are easily accessible to us. All, he has to do is send us the repository’s web address, which is https://github.com/arthur-epi/r4epi_example_project.

After navigating to a GitHub repository, the first thing we typically want to do is read the README. It should have some useful information for us about what the repository does, how it is organized, and how to use it. Because this is a fictitious, minimal example for the book, the current README in the `r4epi_example_project` project isn’t that useful, impressive, or informative. Matias Singers maintains a list of great READMEs at the following link that you may want to check out: <https://github.com/matiassingers/awesome-readme>. If you want to see an example README from a real research project that we worked on, you can check out this link: https://github.com/brad-cannell/detect_pilot_test_5w. After we read over the README file, we are ready to start making edits and additions to the project. But how do we do that?

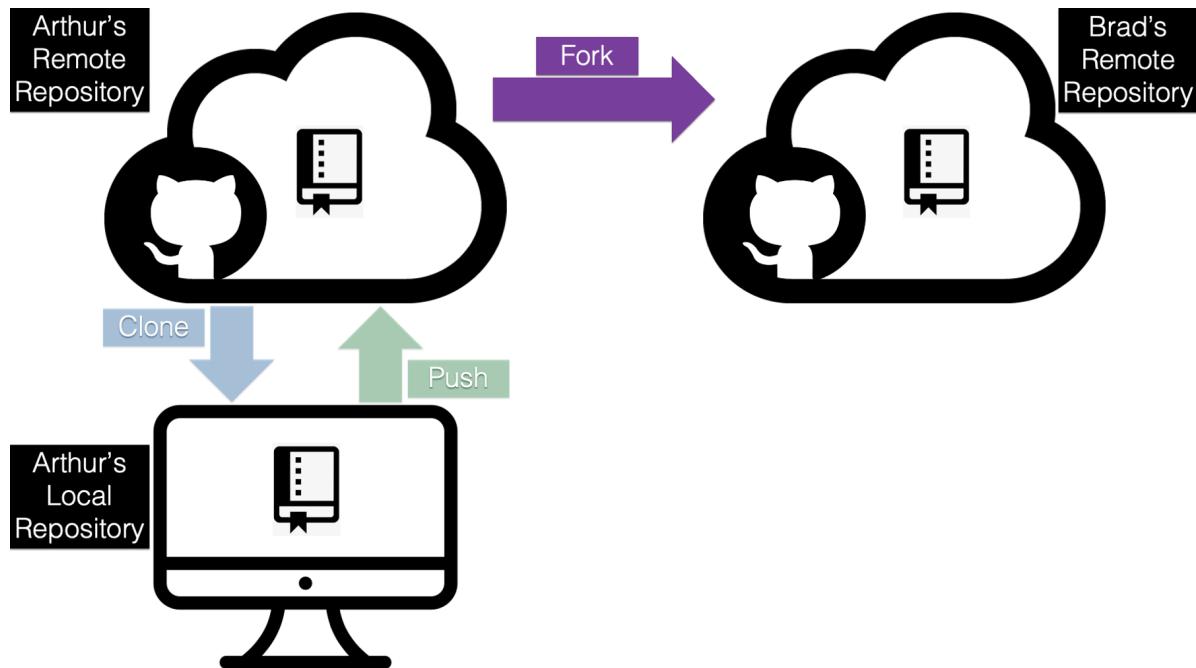
While it is technically possible for us to edit code files directly on GitHub (see [Contributing to R4Epi]), this is typically *only* a good idea for *extremely minor edits* (e.g., a typo in the documentation). Typically, we will want to make a copy of all the code files on our computer so that we can experiment with the edits we are making. Said another way, we can suggest *edits* to R code files directly on GitHub, but we can’t *run* those files in R directly on GitHub to make sure they do what we intend for them to do. To test our changes in R, we will need all of the repository’s files on our local computer. And how do we do that?

21.7.1 Forking a repository

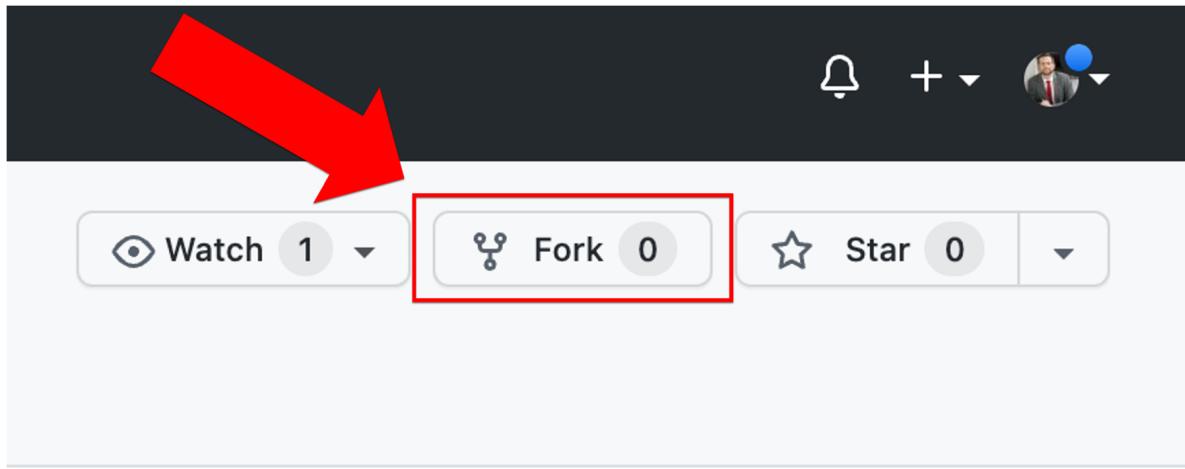
If your answer the question above was, “we **clone** the `r4epi_example_project` repository to our computer” you were close, but that isn’t our best option here. While we technically *can* clone public repositories that aren’t on our account, we *can’t* push any changes to them. And this is a **good thing!** Think about it, do we really want any person out there on the internet to be able to make changes to our repository anytime they want without any oversight from us? No way!



In this case, **forking** the repository is going to be the better option. This is another funny name, but we are once again just talking about making a copy of the repository. However, this time we are copying the repository from the original *GitHub account* (i.e., Arthur’s) to our *GitHub account*. With cloning, we were copying the repository from the original *GitHub account* to our *computer*. Do you see the difference? Let’s try to visualize it.



The purple arrow above indicates that we are forking (i.e., making a copy of) the original `r4epi_example_project` repository on Arthur's GitHub account to Brad's GitHub account. And doing so is really easy. All Brad has to do is log in to GitHub and navigate to *Arthur's r4epi_example_project* repository located at https://github.com/arthur-epi/r4epi_example_project. Then, he needs to click on the `Fork` button located near the top-right corner of the screen.



Then Brad will click the green `Create fork` button on the next page.

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Owner * Repository name *

 brad-cannell / r4epi_example_project ✓

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

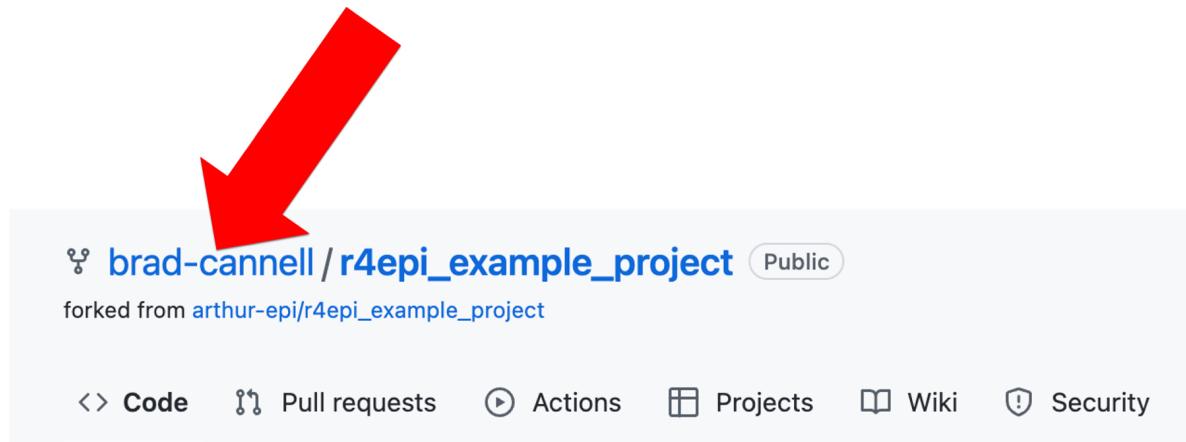
An example repository that accompanies the git and GitHub chapters in the R4Epi book.

(i) You are creating a fork of the brad-cannell organization.

Create fork

And after a few moments, this will create an entirely new repository on Brad's GitHub account. It will contain an exact copy of all the files that were on the repository in Arthur's GitHub

account, but *Brad* is the owner of *this* repository on his account (shown in the screenshot below).

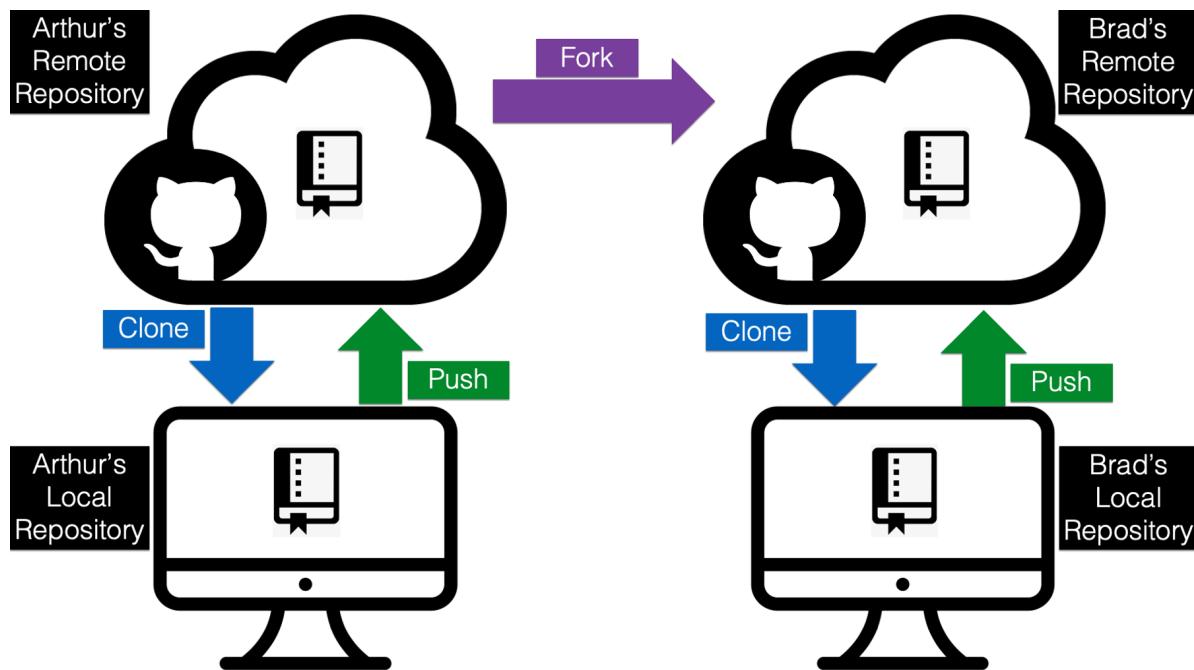


Because Brad is the owner of this repository, he can clone it to his local computer, work on it, and push changes up to GitHub in exactly the same way that Arthur did in the [example above](#). Just to be clear, the changes that Brad pushes to *his* GitHub repository will have no effect on Arthur's GitHub repository.

Note

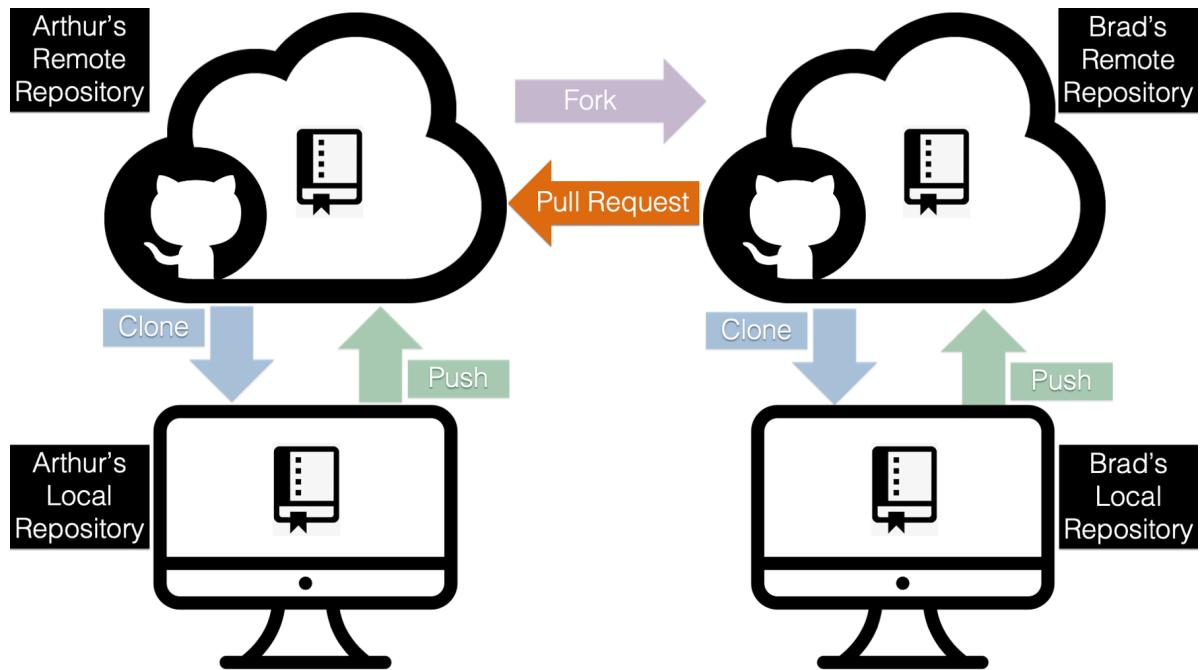
Side Note: As we've pointed out multiple times in this chapter, we generally do not want to upload research data to GitHub. Why? Because it isn't typically considered private or secure. However, in order for Brad to do work on this project, he will need to access the data somehow. This will require Arthur to share data with Brad through some means other than GitHub. Different organizations have different rules about what is considered secure. For example, it may be an encrypted email or it may be a link to a shared drive on a secure server. However the data is shared, it is important for Brad to **create the same file structure on his computer** that Arthur has on his computer. Otherwise, the R code will not work on both computers. Remember from the example above that Arthur created a `data/` folder in his local repository and he moved the `form_20.xlsx` data to that folder. Then, in the `data_01_import` Quartofile, he imports the data using the relative path `data/form_20.xlsx`. In the chapter on [file paths](#) we discussed the advantages of using relative file paths when working collaboratively. Just

remember, in order for this relative file path to work identically on Arthur's computer and Brad's computer, the folder structure and file names must also be *identical*. So, if Brad put the `form_20.xlsx` data in a folder in his local repository called `data sets/` instead of `data/`, then the code in the `data_01_import` Quartofile would throw an error.



Notice that in the diagram above, Arthur's original repository is totally unaffected by any changes that Brad is pushing from his local computer to the repository on his GitHub account. There is no arrow from Brad's remote repository going *into* Arthur's remote repository. Again, this is a good thing. Literally anyone else in the world with a GitHub account could just as easily fork the repository and start making changes. If they also had the ability to make changes to the original repository at will, they could potentially do a lot of damage!

However, in this case, Arthur and Brad *do* know each other and they *are* working collaboratively on this project. And at some point, the work that Brad is doing needs to be synced up with the work that Arthur is doing. In order to make that happen, Brad will need to send Arthur a *request* to *pull* the changes from Brad's remote repository into Arthur's remote repository. This is called a **pull request**.



21.7.2 Creating a pull request

To make this section slightly more realistic, let's say that Brad adds some code to `data_01_import.Qmd`. Specifically, he adds some code that will coerce the `date_received` column from character strings to dates (code below).

```
# Data management

Convert date_received from character strings to dates.

```{r}
form_20 <- form_20 %>%
 mutate(date_received = as.Date(date_received))
```

```{r}
glimpse(form_20)
```

Rows: 3
Columns: 4
$ date_received <date> 2013-08-22, 2013-08-22, 2013-08-22
$ name_last     <chr> "Cooper", "Rodriguez", "Smith"
$ name_first    <chr> "Samantha", "Leslie", "Jane"
$ education     <dbl> 4, 8, 5
```

Then, Brad commits the changes and pushes them up to his GitHub account. Now, when he checks his GitHub account he can see that his remote repository is 1 commit ahead of Arthur's remote repository. And that makes sense, right? Brad just updated the code in `data_01_import.Qmd`, committed that change, and pushed the commit to his GitHub account, but nothing has changed in the repository on Arthur's GitHub account.

This screenshot shows a GitHub repository interface. At the top, there are three buttons: 'main' (selected), '1 branch' (disabled), and 'Code'. Below these are two links: 'Go to file', 'Add file', and a green 'Code' button. A large red arrow points from the bottom of the previous image to the status message 'This branch is 1 commit ahead of arthur-epi:main.' in the center of the screen. To the right of this message are 'Contribute' and 'Fetch upstream' buttons. The main content area displays a list of commits by user 'mbcann01' with details like commit hash, time, and message.

| Commit | Message | Time |
|---------|---|---------------|
| fcbdaae | Convert date_received from character strings to dates | 2 minutes ago |
| | Add the data folder to gitignore | 7 days ago |
| | Initial commit | 18 days ago |
| | Convert date_received from character strings to dates | 2 minutes ago |
| | Add Rproj and gitignore to project | 7 days ago |

Now, Brad needs to create a pull request. This pull request will let Arthur know that Brad has made some changes to the code that he wants to share with Arthur. To do so, Brad will click **Contribute** and then click the green **Open pull request** button as shown below.

This screenshot shows the same GitHub repository interface as the previous one, but with a modal dialog box overlaid. The dialog box contains the message 'This branch is 1 commit ahead of arthur-epi:main.' and instructions to 'Open a pull request to contribute your changes upstream.' It features a prominent green 'Open pull request' button. A large red arrow points from the bottom of the previous image to this button. The rest of the interface remains the same, including the commit list and navigation buttons.

| Commit | Message | Time |
|---------|---|---------------|
| fcbdaae | Convert date_received from character strings to dates | 2 minutes ago |
| | Add the data folder to gitignore | 7 days ago |
| | Initial commit | 18 days ago |
| | Convert date_received from character strings to dates | 2 minutes ago |
| | Add Rproj and gitignore to project | 7 days ago |

The top section of the next screen, which is outlined in red below, allows Brad to select the

repository and branch on his GitHub account that he wants to share with Arthur (to the right of the arrow). More specifically, he is sending a request to Arthur asking him to merge his code into Arthur's code. In this case, the code he wants to ask Arthur to merge is on the main branch of the `brad-cannell/r4epi_example_project repository` (Brad's repository only has one branch – the main branch – at this point). To the left of the arrow, Brad can select the repository and branch on Arthur's GitHub account that he wants to ask Arthur to merge the code into. In this case, the main branch of the `arthur-epi/r4epi_example_project repository` (Arthur's repository only has the main branch at this point as well).

Below the red box, GitHub is telling Brad about the commits that will be sent in this pull request and the changes that will be made to Arthur's files if he merges the pull request into his repository. In this case, only one file in Arthur's repository would be altered – `data_01_import.Rmd`. Below that, Brad can see that the exact differences between his version of `data_01_import.Rmd` and the version that currently exists in Arthur's repository. How cool is that that Brad and Arthur can actually see exactly how this pull request changes the file state down to individual lines of code?

Because Brad is satisfied with what he sees here, he clicks the green `Create pull request` button shown in the middle right of the screenshot below.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

The screenshot shows a GitHub comparison interface between two repositories. At the top, it says "base repository: arthur-epi/r4epi_example_pr..." and "head repository: brad-cannell/r4epi_example...". It indicates "base: main" and "head: main". A red box highlights the message "✓ Able to merge. These branches can be automatically merged." Below this, there are buttons for "Create pull request" and "Compare".

Below the header, it says "Discuss and review the changes in this comparison with others. [Learn about pull requests](#)".

Summary statistics: "-o 1 commit", "1 file changed", and "1 contributor".

A collapsible section titled "Commits on Jun 7, 2022" shows one commit:

Convert date_received from character strings to dates
mbcann01 committed 6 minutes ago

Commit details: "Showing 1 changed file with 16 additions and 1 deletion." with "Split" and "Unified" options.

The diff view shows the following code changes in `data_01_import.Rmd`:

```
@@ -25,4 +25,19 @@ form_20 <- read_excel("data/form_20.xlsx")
25 25
26 26   ``{r}
27 27   glimpse(form_20)
28 28 - ``
29 29 +
30 30 + # Data management
31 31 +
32 32 + Convert date_received from character strings to dates.
33 33 +
34 34 + ``{r}
35 35 + form_20 <- form_20 %>%
36 36 +   mutate(date_received = as.Date(date_received))
37 37 +
38 38 +
39 39 + ``{r}
40 40 + glimpse(form_20)
41 41 +
42 42 +
43 43 +
```

Let's pause here and get explicit about two things.

1. As we've tried to really drive home above, this pull request will **not** automatically make any changes to Arthur's repository. Rather, it will only send Arthur Brad's code, ask him to review it, and then allow him to *choose* whether to incorporate it into his repository or not.
2. Pull requests are sent at the **branch level** not at the **file level**. Meaning, if Arthur accepts Brad's pull request, it will make *all* of the files on his main branch identical to *all* of the files on Brad's main branch (the main branch because that is the branch Brad chose in the screenshot above – and currently the only branch in either repository). In this case, that means that the only file that would change as a result of copying over the entire branch is `data_01_import.Rmd`. However, if Brad had made changes to `data_01_import.Rmd` and another file, Arthur would only have the option to merge *both* files or *neither* file. He would not have the option of merging `data_01_import.Rmd` *only*. Pull requests merge the entire branch, not specific files. We are emphasizing this because

this may affect how you commit, push, and create pull requests when you are working collaboratively. More specifically, you may want to commit, push, and send pull requests more frequently than you would if you were working on a project independently.

On the next screen, Brad is given an opportunity to give the pull request a title and add a message for Arthur that give him some additional details. In general, it's a good idea to fill this part out using similar conventions to those described above for commit messages.

After filling out the commit message, Brad will click the green **Create pull request** button on last time, and he is done. This will send Arthur the pull request.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

The screenshot shows the GitHub interface for creating a pull request. At the top, there are dropdown menus for 'base repository' (arthur-epi/r4epi_example_pr...) and 'head repository' (brad-cannell/r4epi_example...), both set to 'main'. A green checkmark indicates 'Able to merge'. The main area contains a message from Brad to Arthur:

Convert date_received from character strings to dates

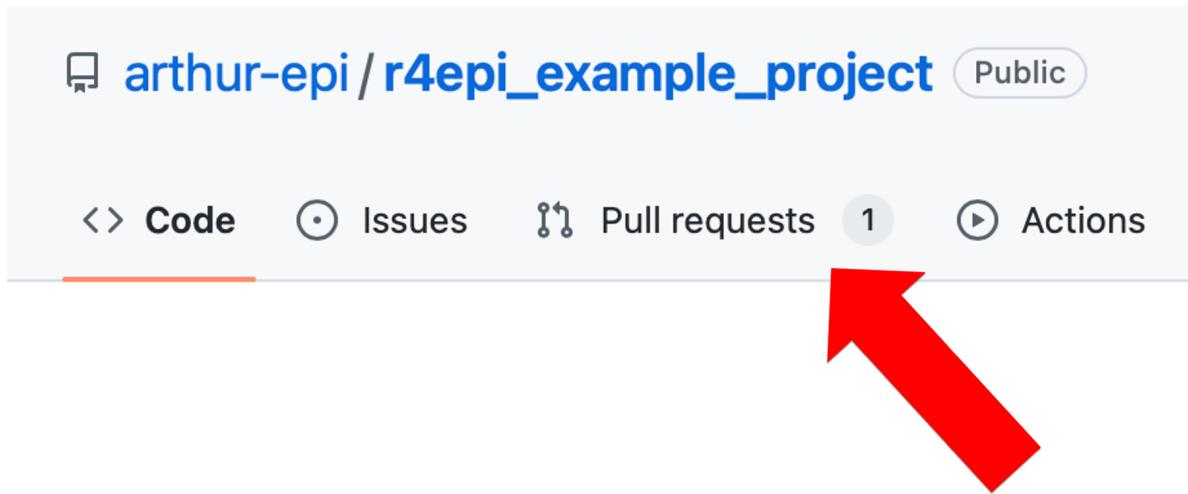
Hi Arthur,

As we discussed yesterday, I added some code to data_01_import.Rmd that converts the date_received column to dates that we can do those calculations.

Thanks!
Brad

At the bottom right of the message area, there are two small circular icons: one with a smiley face and another with a question mark containing the number '2'. Below the message area, there is a note: "Attach files by dragging & dropping, selecting or pasting them." At the very bottom is a large green button labeled "Create pull request".

The next time Arthur checks the `r4epi_example_project` on GitHub, he will see that he has a new pull request.



If he clicks on the text **Pull requests** text, he will be taken to his pull requests page. It will show him all pending pull requests. In this case, there is just the one pull request that Brad sent.

A screenshot of the GitHub pull requests page. At the top, a modal dialog says 'Label issues and pull requests for new contributors' with a 'Dismiss' button. Below it, a search bar contains 'is:pr is:open'. The main area shows a single open pull request: '#1 Convert date_received from character strings to dates' by mbcann01, which was opened 1 minute ago. The pull request has 9 labels and 0 milestones. A green 'New pull request' button is at the bottom right.

When he clicks on it, he will see a screen like the one in the screenshot below. Scanning from

top to bottom, it will tell him which branch Brad is requesting to merge the code into, show him the message Brad wrote, tell him that he can merge this branch without any conflicts if he so chooses, and give him an opportunity to write a message back to Brad before deciding whether to merge this pull request or close it.

Convert date_received from character strings to dates #1

The screenshot shows a GitHub pull request page for a repository. At the top, there's a green 'Open' button and a message from mbcann01 wanting to merge 1 commit into the `arthur-epi:main` branch from `brad-cannell:main`. Below this, the main interface includes tabs for Conversation (0), Commits (1), Checks (0), and Files changed (1). The pull request has 16 changes, 1 conflict, and 1 file changed.

In the Conversation tab, a comment from `mbcann01` is shown:

```
Hi Arthur,  
As we discussed yesterday, I added some code to data_01_import.Rmd that converts the date_received column to dates that we can do those calculations.  
Thanks!  
Brad
```

The pull request details on the right side include:

- Reviewers:** arthur-epi (Request)
- Suggestions:** None
- Assignees:** None—assign yourself
- Labels:** None yet
- Projects:** None yet
- Milestone:** None
- Development:** None yet
- Notifications:** You're receiving notifications because you're watching this repository. (Unsubscribe)
- Participants:** 1 participant (Arthur)

At the bottom, there's a 'Merge pull request' button and a note about GitHub Actions and other apps for CI setup. A 'Comment' section is also visible.

He also has the option to view some additional details by clicking the `Commits` tab, `Checks` tab, and/or `Files changed` tab towards the top of the screen. Let's say he decides to click on the `Files changed` tab.

On the `Files changed` tab, Arthur can see each of the files that the pull request would change if he were to merge it into his repository (in this case, only one file). For each file, he can see (and even comment on) each specific line of code that would change. In this case, Arthur is pleased with the changes and navigates back to the `Conversation` tab by clicking on it.

Convert date_received from character strings to dates #1

mbcann01 wants to merge 1 commit into [arthur-epi:main](#) from [brad-cannell:main](#)

Conversation 0 Commits 1 Checks 0 Files changed 1

+16 -1

```

25 25
26 26  ``{r}
27 27 glimpse(form_20)
28 - ``
29 + ``
30 + # Data management
31 +
32 + Convert date_received from character strings to dates.
33 +
34 + ``{r}
35 + form_20 <- form_20 %>%
36 +   mutate(date_received = as.Date(date_received))
37 + ``
38 +
39 + ``{r}
40 + glimpse(form_20)
41 +
42 +
43 +

```

Changes from all commits ▾ File filter ▾ Conversations ▾ Jump to ▾ Review changes ▾

Back on the Conversation tab (see screenshot below), Arthur has some options. If he wants more clarification about the pull request, he can send leave a comment for Brad using the comment box near the bottom of the screen. If he knows that he does **NOT** want to merge this pull request into his code, he can click the **Close pull request** button at the bottom of the screen. This will close the pull request and his code will remain unchanged. In this case, Arthur wants to incorporate the changes that Brad sent over, so he clicks the green **Merge pull request** button in the middle of the screen.

Convert date_received from character strings to dates #1

The screenshot shows a GitHub pull request page for a repository. At the top, there's a green 'Open' button and a message from user 'mbcann01' wanting to merge their changes into the 'arthur-epi:main' branch from the 'brad-cannell:main' branch. Below the message, there are tabs for 'Conversation' (0), 'Commits' (1), 'Checks' (0), and 'Files changed' (1). On the right side, there are sections for 'Reviewers' (with 'arthur-epi' listed), 'Suggestions' (empty), and a 'Request' button. Below these are sections for 'Assignees' (empty), 'Labels' (empty), 'Projects' (empty), 'Milestone' (empty), 'Development' (empty), and 'Notifications' (with an 'Unsubscribe' button). The main area shows a commit message: 'Convert date_received from character strings to dates' with hash 'fcbdaae'. A large red arrow points to a green 'Merge pull request' button. Below the button is a note: 'This branch has no conflicts with the base branch. Merging can be performed automatically.' At the bottom of the pull request interface, there's a 'Write' tab, a preview section, and a comment input field with a 'Comment' button.

Then, he is given an opportunity to add some details about the changes this merge will make to the repository once it is committed. You can once again think of this message as having a very similar purpose to commit messages, which were discussed above. In fact, it will appear as a commit in the repository's commit history.

Finally, he clicks the green **Confirm merge** button.

Convert date_received from character strings to dates #1

The screenshot shows a GitHub pull request page for a repository. At the top, there's a green 'Open' button and a message from user 'mbcann01' wanting to merge their changes into the 'arthur-epi:main' branch from the 'brad-cannell:main' branch. The pull request has 1 commit, 0 checks, and 1 file changed. The commit message is 'Convert date_received from character strings to dates'. On the right side, there are sections for Reviewers, Suggestions, Assignees, Labels, Projects, Milestone, Development, Notifications, and Participants. A 'Request' button is also present. Below the main commit, there's a merge dialog with 'Merge pull request #1 from brad-cannell/main' and a 'Convert date_received from character strings to dates' summary. At the bottom, there's a comment input field with 'Leave a comment' placeholder, a file attachment area, and buttons for 'Close pull request' and 'Comment'. A note at the bottom left says 'Remember, contributions to this repository should follow our GitHub Community Guidelines.' and a pro tip about adding '.patch' or '.diff' to URLs.

And if Arthur navigates back to his commit history page, he can see two new commits. Brad's commit with the updated `data_01_import.Qmd` file, and the commit that was automatically created when Arthur merged the branches together.

The screenshot shows a GitHub commit history for the 'main' branch. The commits are organized by date:

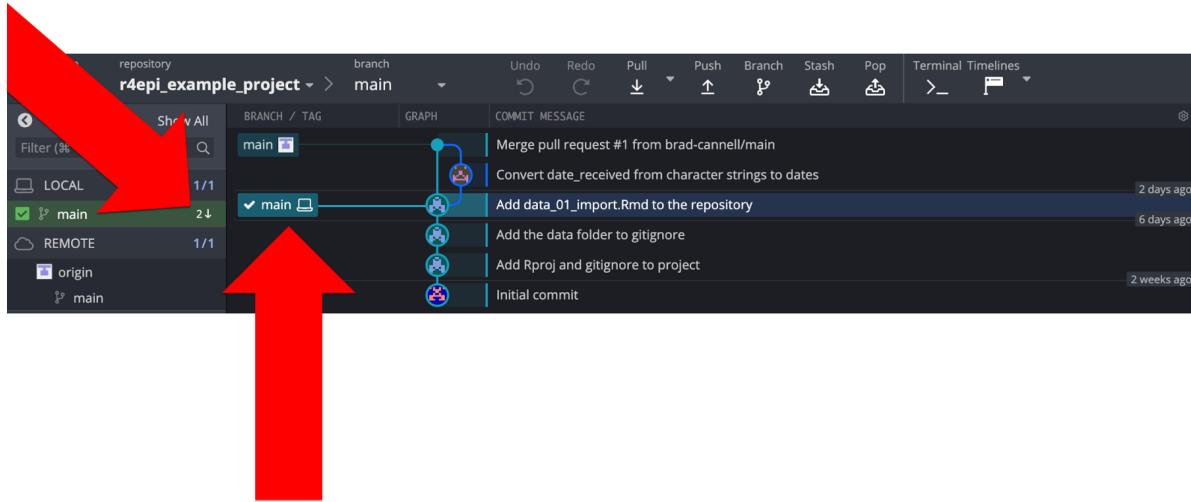
- Commits on Jun 7, 2022:
 - Merge pull request #1 from brad-cannell/main
 - Convert date_received from character strings to dates
- Commits on Jun 4, 2022:
 - Add data_01_import.Rmd to the repository
- Commits on May 31, 2022:
 - Add the data folder to gitignore
 - Add Rproj and gitignore to project
- Commits on May 20, 2022:
 - Initial commit

Each commit card includes the author (arthur-epi), the commit message, the date (e.g., committed 38 seconds ago, 3 days ago, etc.), and a 'Verified' badge. There are also buttons for copy, diff, and view.

Now, Arthur takes a look at `data_01_import.Qmd` on his computer. To his surprise, the code to coerce `date_received` into dates isn't there. Why not?

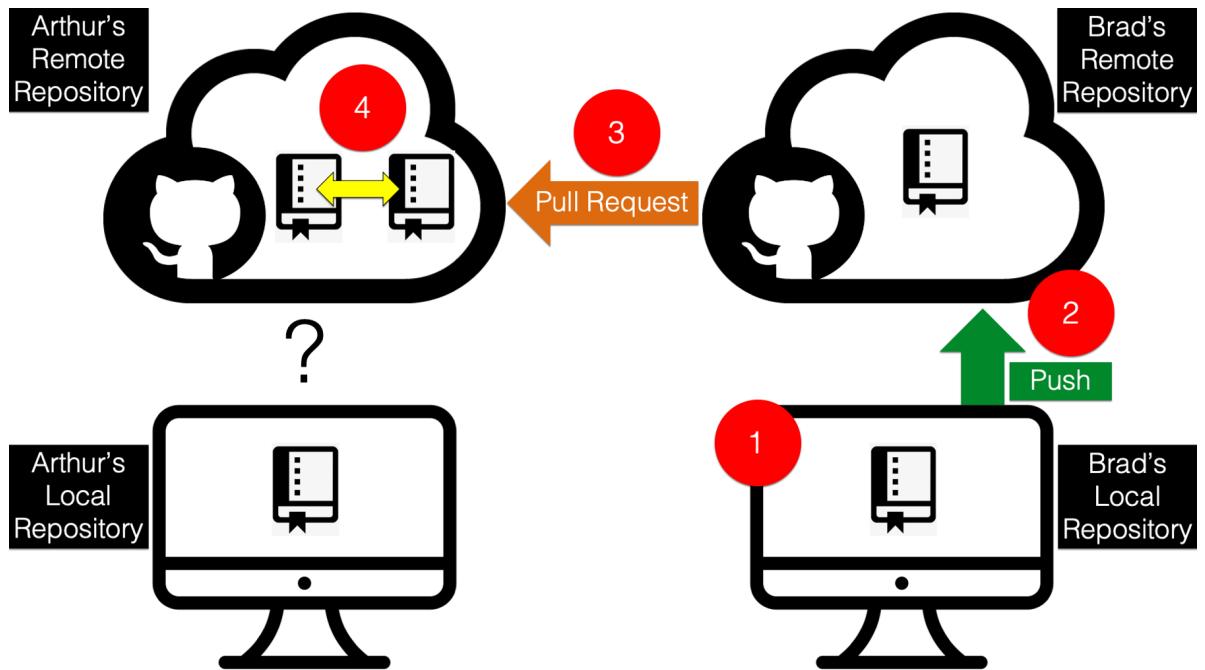
```
1 ---  
2 title: "Import Form 20 data for the R4Epi Example Project"  
3 ---  
4  
5 # ★ Overview  
6  
7 In this file, we import the mtcars data. This file is unrealistically simple, but we are using it for demonstration purposes only.  
8  
9  
10 # 📦 Load packages  
11  
12 ```{r message=FALSE}  
13 library(dplyr, warn.conflicts = FALSE)  
14 library(readxl)  
15 ```  
16  
17  
18 # 📈 Import data  
19  
20 This data is packaged with base R.  
21  
22 ```{r}  
23 form_20 <- read_excel("data/form_20.xlsx")  
24```  
25  
26 ```{r}  
27 glimpse(form_20)  
28```
```

Well, let's open GitKraken on Arthur's computer and see if we can help him figure it out. In the repository graph, Arthur's local repository (i.e., the little laptop icon) and the remote repository (i.e., the little gray and white icon) are on different rows. Additionally, there is a little 2 next to a down arrow displayed to the left of the main branch of our local repository in the left panel of GitKraken. Both of these indicate that the most recent commits contained in each repository are different. Specifically, that the local repository is two commits *behind* the remote repository.



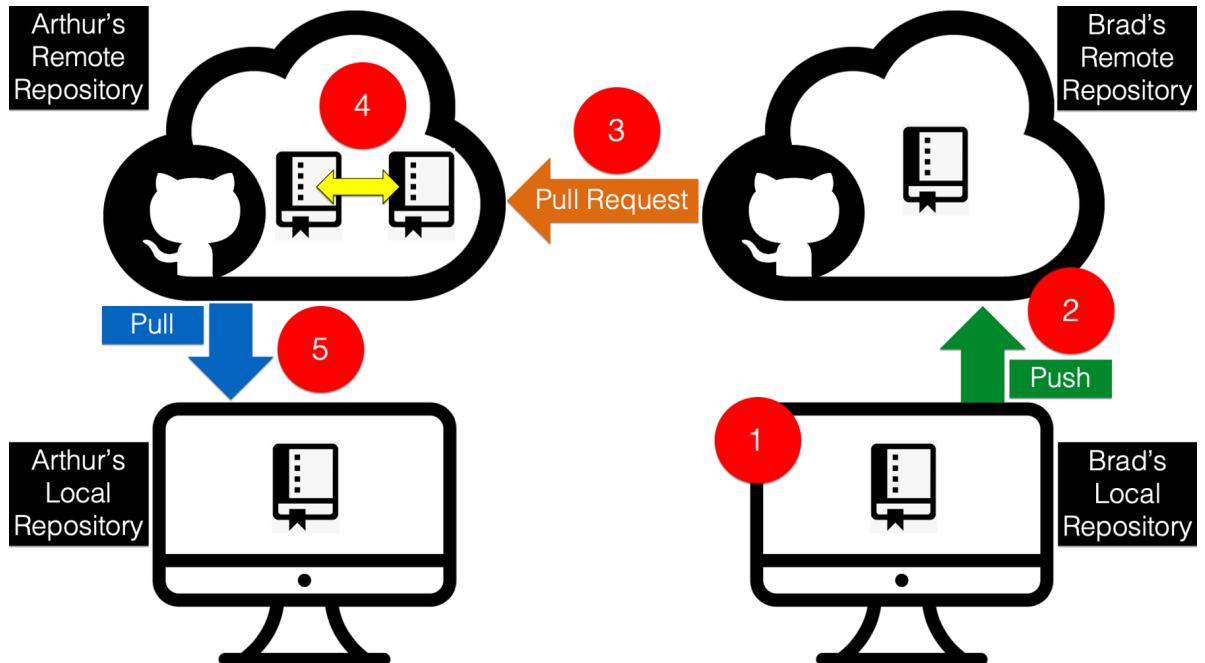
So, let's pause here for a second and review what we've done so far. As shown in the figure below:

1. Brad made some updates to the code on his computer and then committed those changes to his local repository. At this point, his local repository is out of sync with his remote repository, Arthur's remote repository, and Arthur's local repository.
2. Next, Brad pushed that commit from his local repository up to his remote repository on GitHub. After doing so, his local repository and remote repository are synced with each other, but they are still out of sync with Arthur's remote repository and Arthur's local repository.
3. Then, Brad created a pull request for Arthur. The request was for Arthur to pull the latest commit from Brad's remote repository into Arthur's remote repository.
4. Arthur accepted and merged Brad's pull request. After doing so, his remote repository, Brad's remote repository, and Brad's local repository are all contain the updated `data_01_import.Qmd` file, but Arthur's local repository still does not.

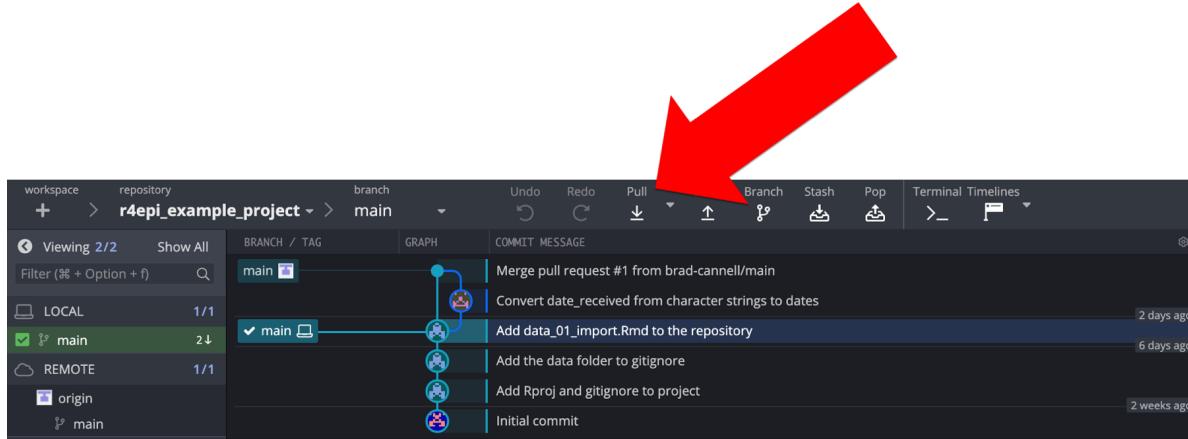


So, how does Arthur get his local repository in sync with his remote repository?

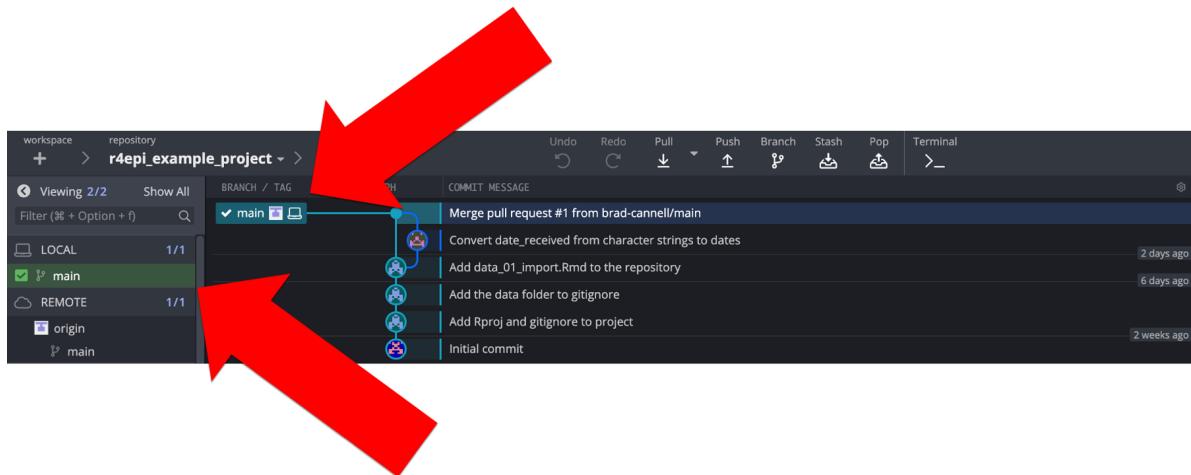
Arthur just needs to use the **pull** command to download the files from his updated remote repository and merge them into his local repository (step 5 below).



And GitKraken makes pulling the files from his remote repository really easy. All Arthur needs to do is click the pull button shown in the screenshot below. GitKraken will download (also called **fetch**) the updated repository and merge the changes into his local repository.



And as shown in the screenshot below, Arthur can now see that his local repository is now in sync with his remote repository once again!



But, what about Brad's repository? Well, as you can see in the screenshot below, Brad's remote repository is now 1 commit *behind* Arthur's. Why?

This one is kind of weird/tricky. Although the code in Brad's repository is now identical to the code in Arthur's repository, the *commit history* is not. Remember, Arthur's commit history from above? When he merged Brad's code into his own, that automatically created an additional commit. And that additional commit does not currently exist in Brad's commit history. It's an easy fix though!

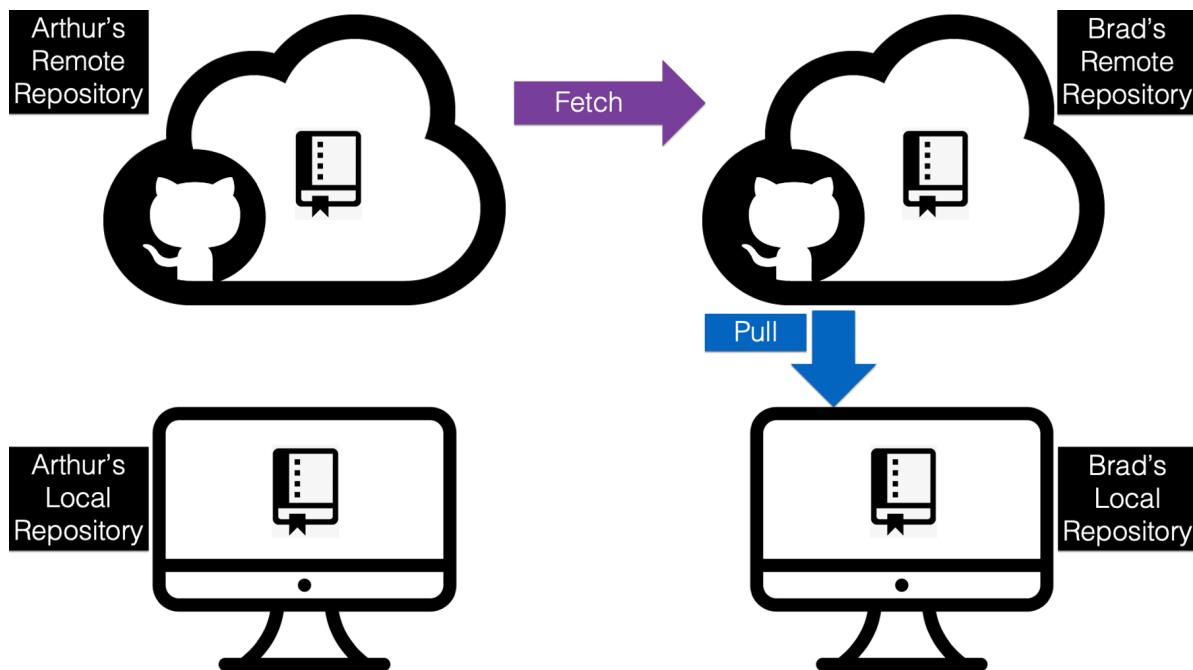
```
1 ---  
2 title: "Import Form 20 data for the R4Epi Example Project"  
3 ---  
4  
5 # ★ Overview  
6  
7 In this file, we import the mtcars data. This file is unrealistically simple, but we are using it for demonstration purposes only.  
8  
9  
10 # 📦 Load packages  
11  
12 ```{r message=FALSE}  
13 library(dplyr, warn.conflicts = FALSE)  
14 library(readxl)  
15 ```  
16  
17  
18 # 📈 Import data  
19  
20 This data is packaged with base R.  
21  
22 ```{r}  
23 form_20 <- read_excel("data/form_20.xlsx")  
24```  
25  
26 ```{r}  
27 glimpse(form_20)  
28```  
29  
30 # 📄 Data management  
31  
32 Convert date_received from character strings to dates.  
33  
34 ```{r}  
35 form_20 <- form_20 %>%  
36 mutate(date_received = as.Date(date_received))  
37```  
38  
39 ```{r}  
40 glimpse(form_20)  
41```
```

All Brad needs to do is a quick **fetch** from Arthur's remote repository to merge that last commit into his commit history, and then **pull** it down to his local repository.

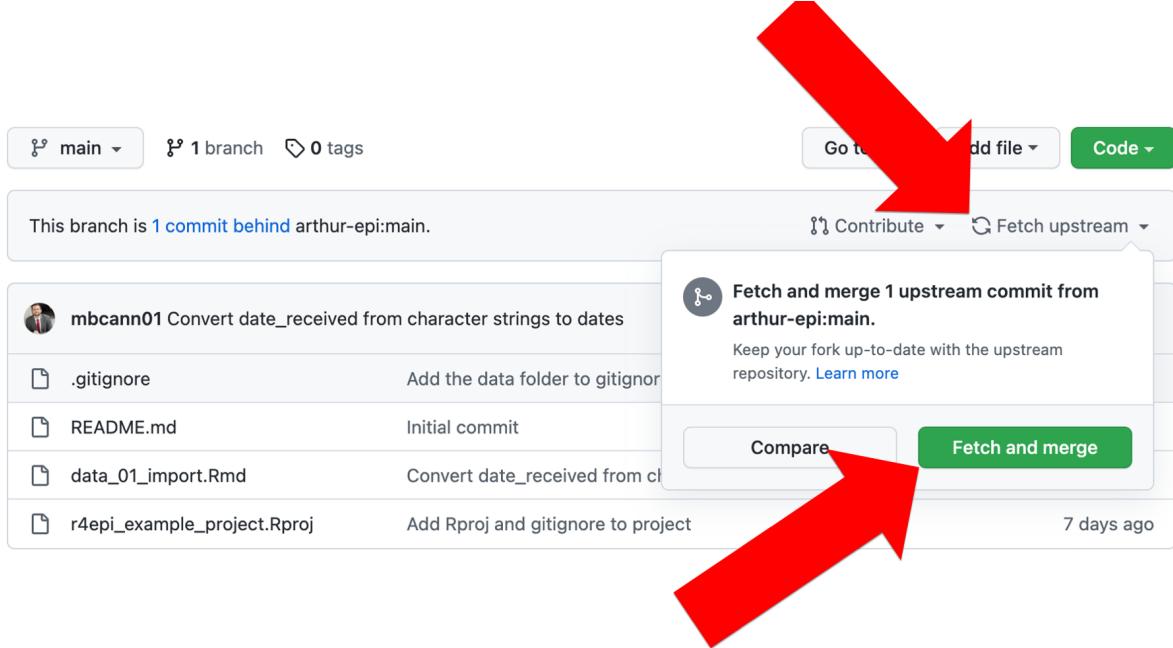
This branch is [1 commit behind](#) arthur-epi:main.

| Author | Commit Message | Date |
|----------|---|----------------|
| mbcann01 | Convert date_received from character strings to dates | 13 minutes ago |
| | .gitignore | 7 days ago |
| | README.md | 18 days ago |
| | data_01_import.Rmd | 13 minutes ago |
| | r4epi_example_project.Rproj | 7 days ago |

To do so, Brad will first click **Fetch upstream** followed by the green **Fetch** and **merge** button.



After a few seconds, GitHub will show him that his remote repository is now synced up with Arthur's remote repository. All he has to do now is a quick pull in GitHub.



And now we have seen the basic process for collaboratively coding with git and GitHub. Don't feel bad if you are still feeling a little bit confused. Git and GitHub are confusing at times even for experienced programmers. But that doesn't mean that they aren't still valuable tools! They are!

We also recognize that it might seem like that was a ton of steps above. Again, we went through this process slowly and methodically because we are all trying to learn here. In a real-life project with two experienced collaborators, the steps in this example would typically be completed in a matter of minutes. No big deal.

21.8 Summary

There is so much more to learn about git and GitHub, but that's not what this book is about. So, we will stop here. We hope the examples above demonstrate some of the potential value of using git and GitHub in your project workflow. We also hope they give you enough information to get you started.

Here are some free resources we recommend if you want to learn even more:

1. Chacon S, Straub B. Pro Git. Second. Apress; 2014. Accessed June 13, 2022. <https://git-scm.com/book/en/v2>
2. GitHub. Getting started with GitHub. GitHub Docs. Accessed June 13, 2022. <https://ghdocs-prod.azurewebsites.net/en/get-started>

3. Bryan J. Happy Git and GitHub for the useR.; 2016. Accessed June 2, 2022. <https://happygitwithr.com/index.html>
4. Keyes D. How to Use Git/GitHub with R. R for the Rest of Us. Published February 13, 2021. Accessed June 13, 2022. <https://rfortherestofus.com/2021/02/how-to-use-git-github-with-r/>
5. Wickham H, Bryan J. Chapter 18 Git and GitHub. In: R Packages. Accessed June 13, 2022. <https://r-pkgs.org/git.html>

Part VI

References

22 References

1. Ismay C, Kim AY. Chapter 1 getting started with data in R. Published online November 2019.
2. Stack Overflow. What are tags, and how should I use them? Published online January 2022.
3. Stack Overflow. How do I ask a good question? Published online January 2022.
4. RStudio. FAQ: Tips for writing r-related questions. Published online September 2021.
5. Wickham H. Style guide. In: *Advanced R.*; 2019.
6. Wickham H, Çetinkaya-Rundel M, Grolemund G. Workflow: Code style. In: *R for Data Science*. second.; 2023.
7. Peng RD, Hicks SC. Reproducible research: A retrospective. *Annu Rev Public Health*. 2021;42:79-93.
8. Peng RD. Reproducible research in computational science. *Science*. 2011;334(6060):1226-1227.
9. GitHub. Licensing a repository. Published online May 2022.
10. Bryan J. *Happy Git and GitHub for the userR.*; 2016.
11. GitHub. *About Issues*. Github; 2024.
12. R Core Team. *What Is r?* R Foundation for Statistical Computing; 2024.
13. GitHub. About repositories. Published online December 2023.
14. RStudio. RStudio. Published online 2020.

A Glossary

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

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

Functions. Coming soon.

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

Global environment. Coming soon.

Issue (GitHub) GitHub’s documentation says issues are “items you can create in a repository to plan, discuss and track work. Issues are simple to create and flexible to suit a variety of scenarios. You can use issues to track work, give or receive feedback, collaborate on ideas or tasks, and efficiently communicate with others.”¹¹

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.”¹² 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.”¹³ 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).¹⁴