R 4 Epidemiology

2025-07-01

Table of contents

Welcome

Welcome to R for Epidemiology!

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

Acknowledgements

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

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

Introduction

Goals

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

The goals of this book are:

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

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

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

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

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

Text conventions used in this book

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

Other reading

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

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

Contributing

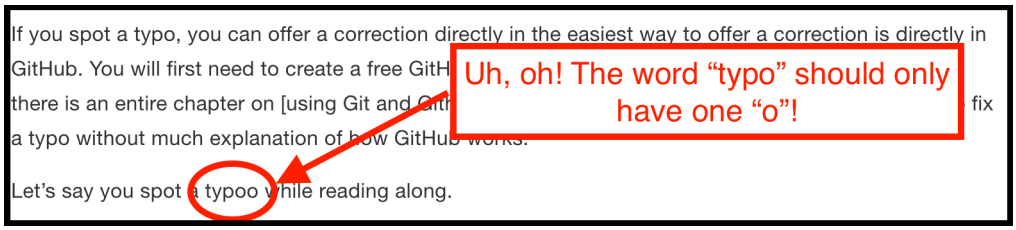
Over the years, we have learned so much from our students and colleagues, and we anticipate that there is much more we can learn from you – our readers. Therefore, we welcome and appreciate all constructive contributions to R4Epi!

Typos

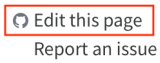
The easiest way for you to contribute is to help us clean up the little typos and grammatical errors that inevitably sneak into the text.

If you spot a typo, you can offer a correction directly in GitHub. You will first need to create a free GitHub account: sign-up at github.com. Later in the book, we will cover using GitHub in greater depth 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.

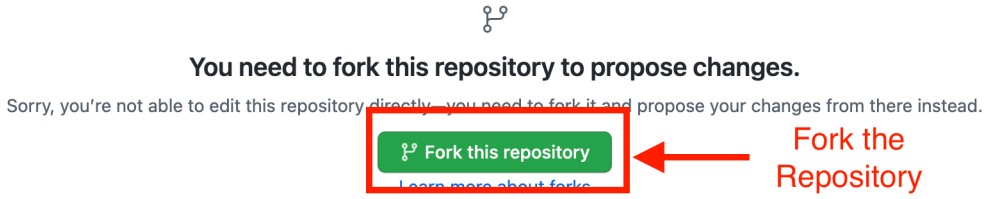
Let's say you spot a typo while reading along.



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



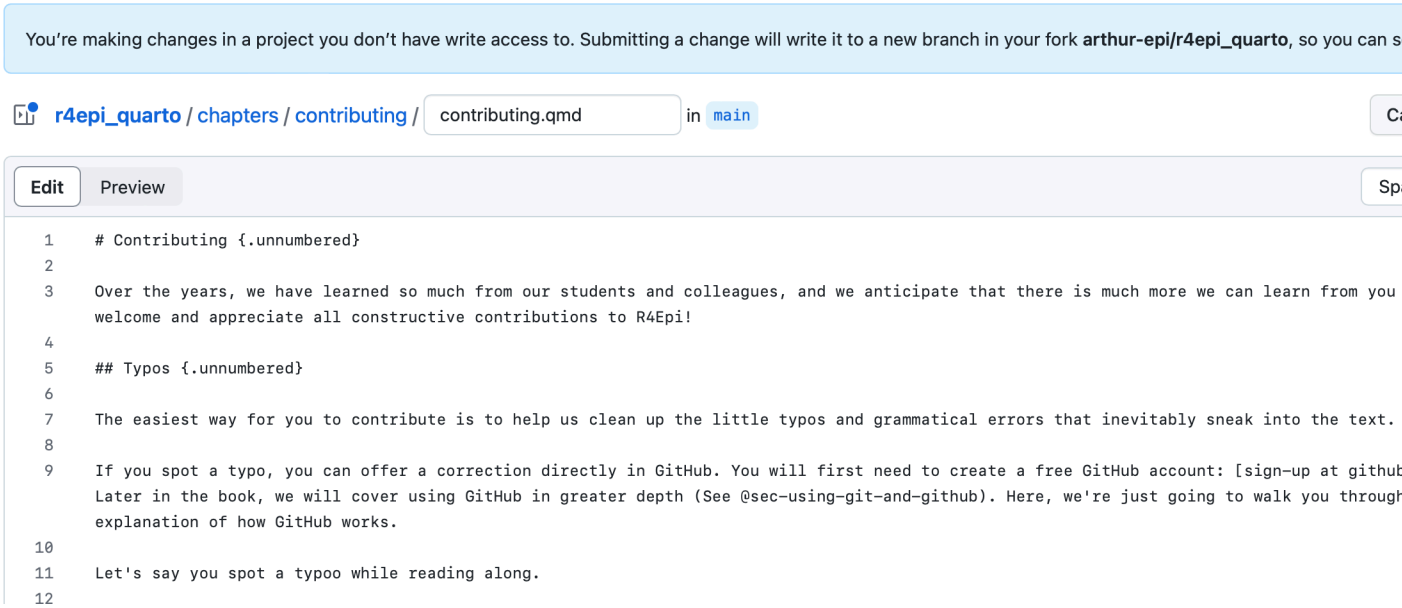
The first time you click the icon, you will be taken to the R4Epi repository on GitHub and asked to fork it. For our purposes, you can think of a GitHub repository as being similar to a shared folder on Dropbox or Google Drive.



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

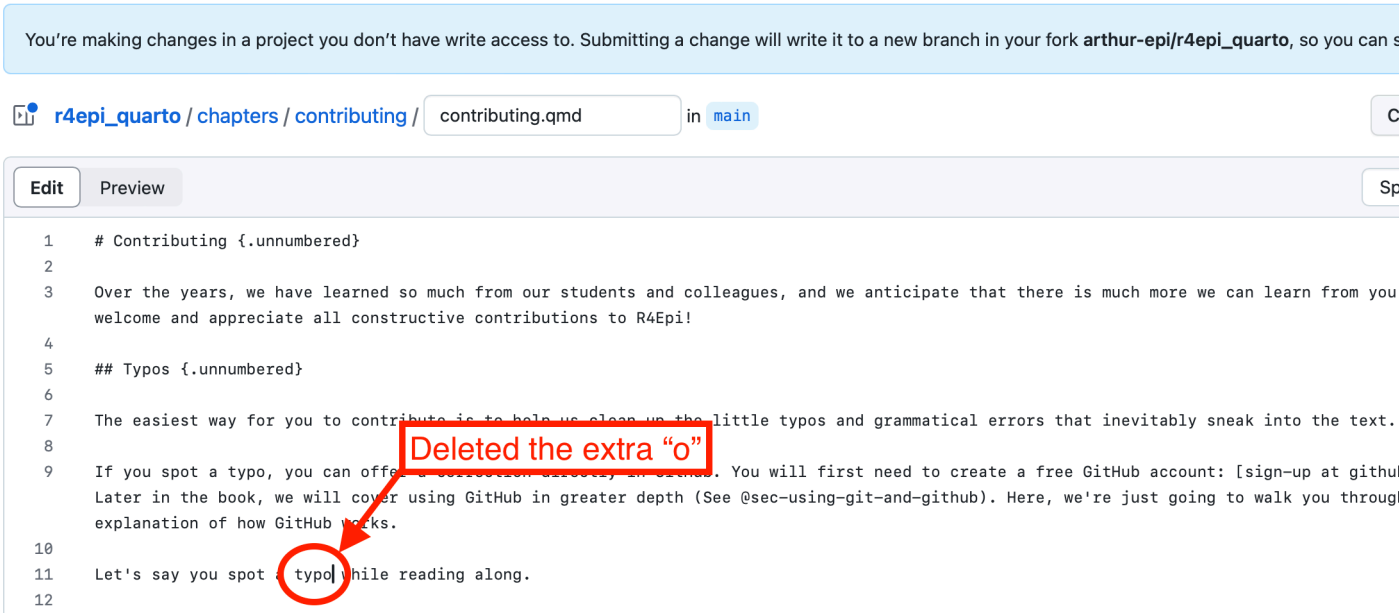
> **Note**
>
> Forking the R4Epi repository does not cost any money or add any files to your computer.

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

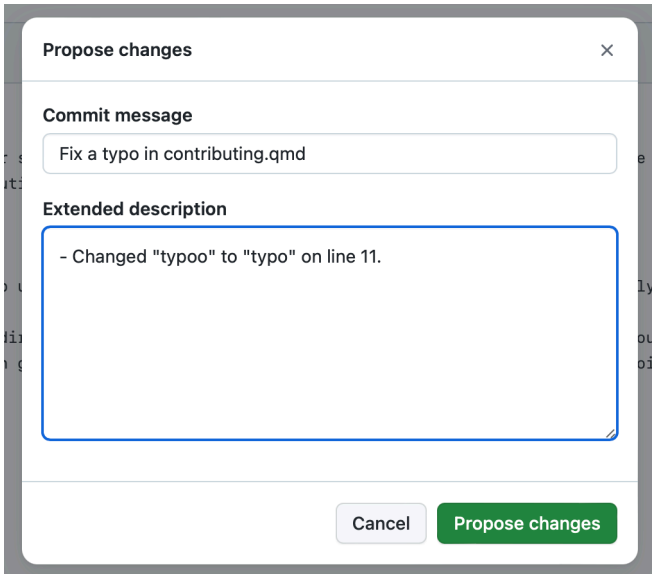


The text editor will display the contents of the file used to make the chapter you were looking at when you clicked the `edit` button. In this example, it was a file named `contributing.qmd`. The `.qmd` file extension means that the file is a Quarto/file. We will learn more about Quarto files, but for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

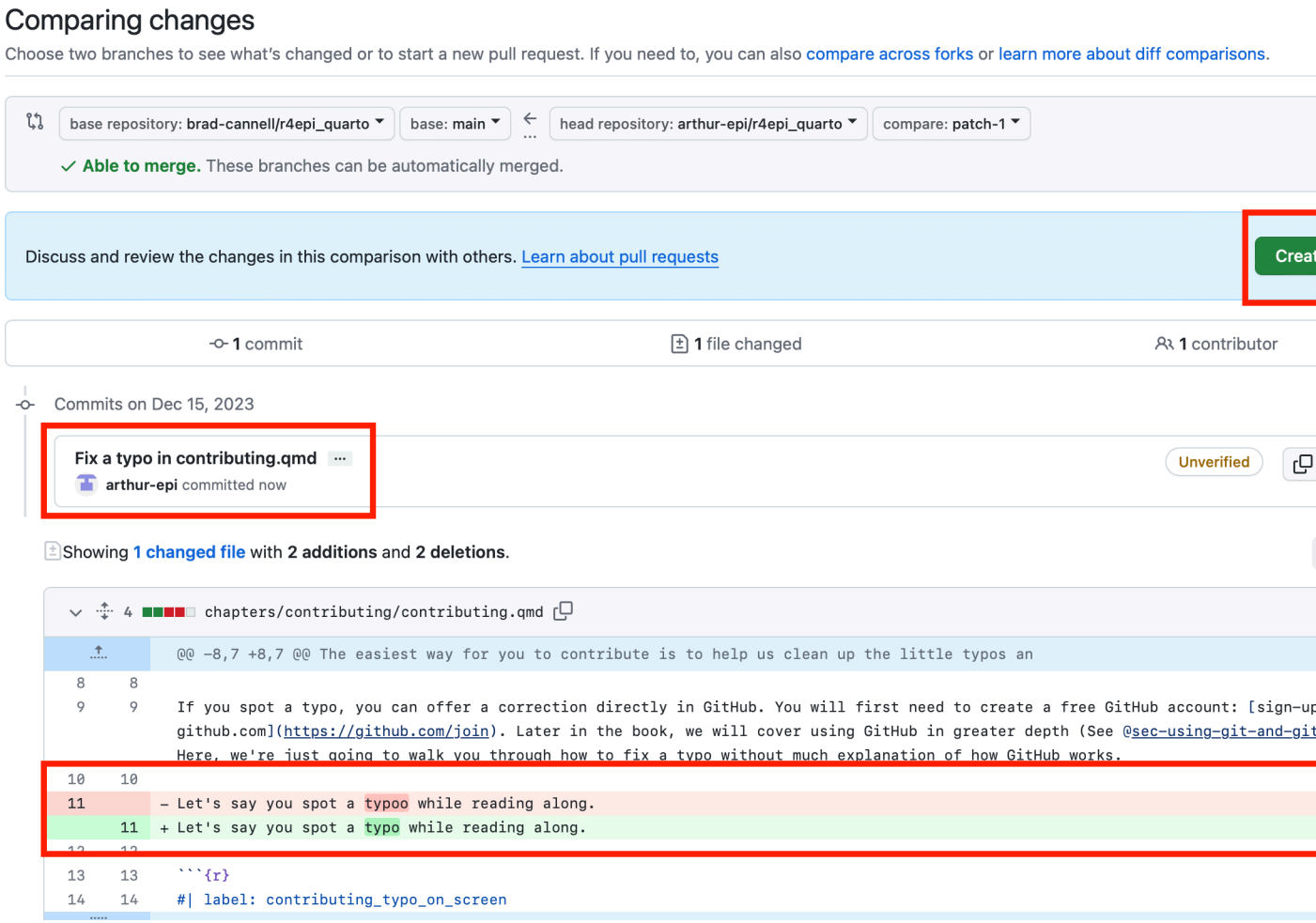
Next, scroll down through the text until you find the typo and fix it. In this case, line 11 contains the word "typoo". To fix it, you just need to click in the editor window and begin typing. In this case, you would click next to the word "typoo" and delete the second "o".



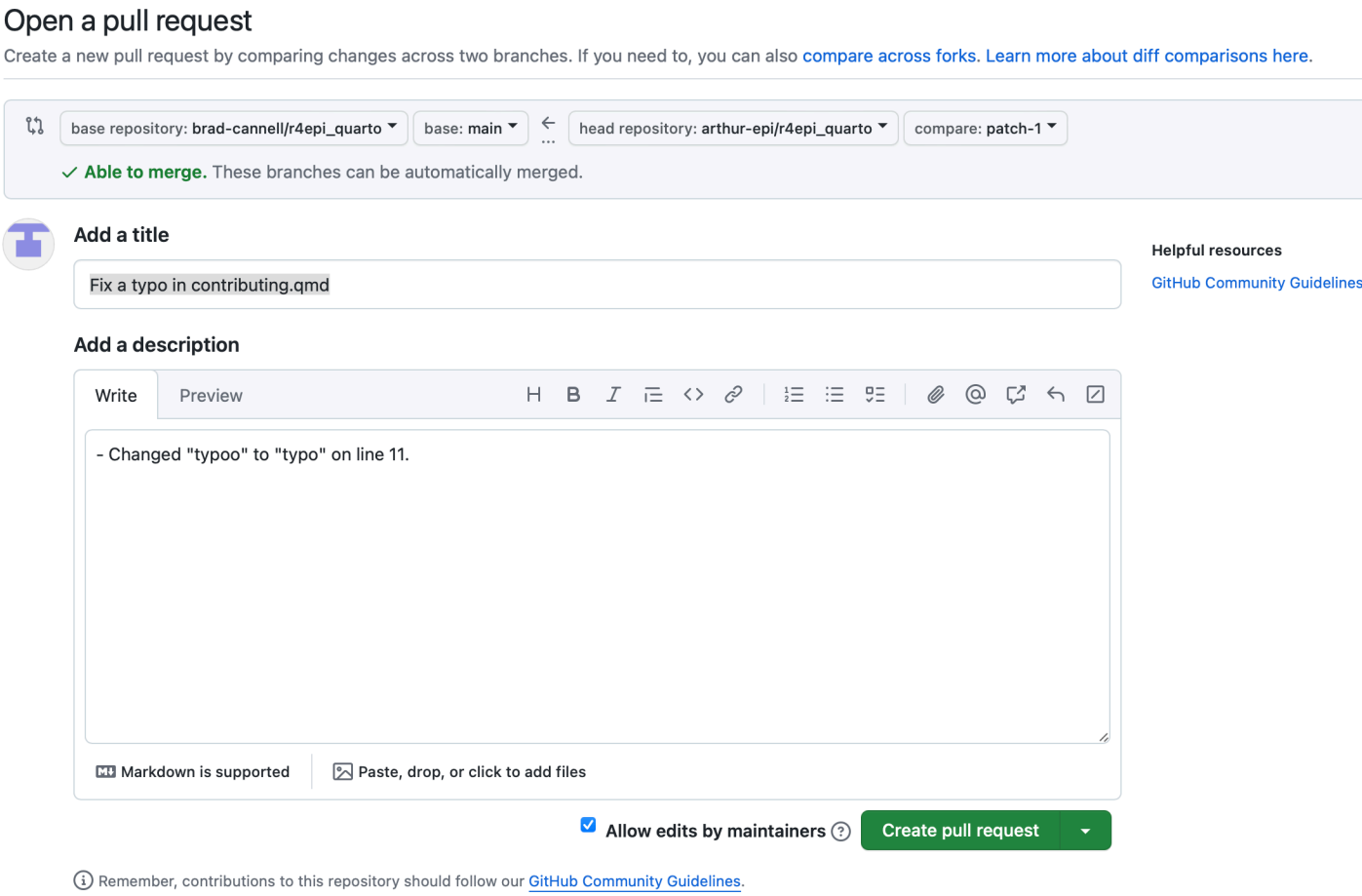
Now, the only thing left to do is propose your typo fix to the authors. To do so, click the green `Commit changes...` button on the right side of the screen above the text editor (surrounded with a red box in the screenshot above). When you click it, a new `Propose changes` box will appear on your screen.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



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

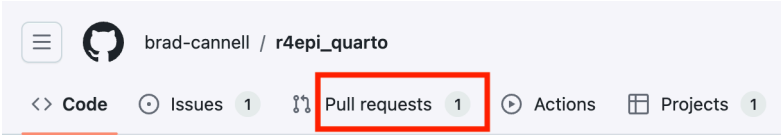


For now, we will focus on the three different sections of the screen that are highlighted with a red outline. We will start at the bottom and work our way up. The red box that is closest to the bottom of the screenshot shows us that the change that made was on line 11. The word "typoo" (highlighted in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



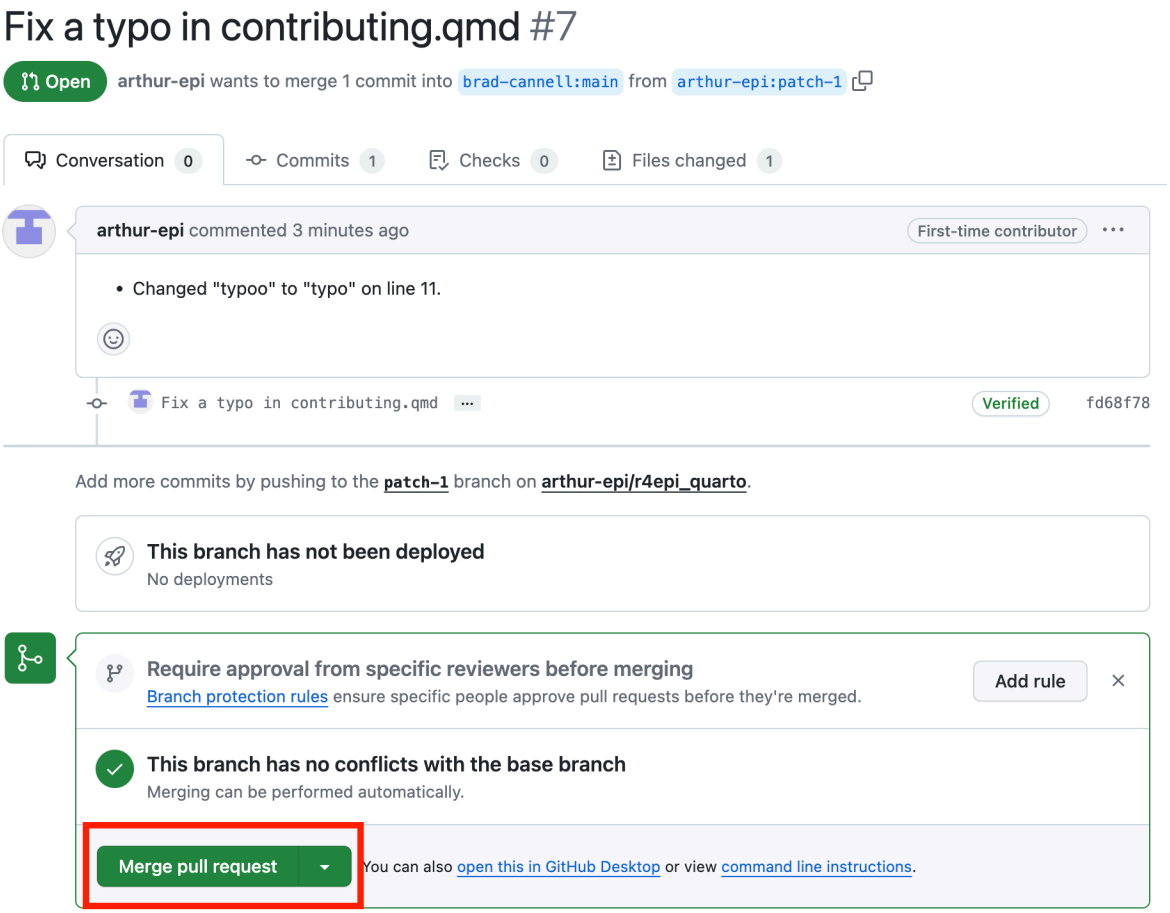
After doing so, you will get one final chance to amend the description of your proposed changes. If you are happy with the commit message and description, then click the `Create pull request` button one more time. At this point, your job is done! It is now up to the authors to review the changes you've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

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



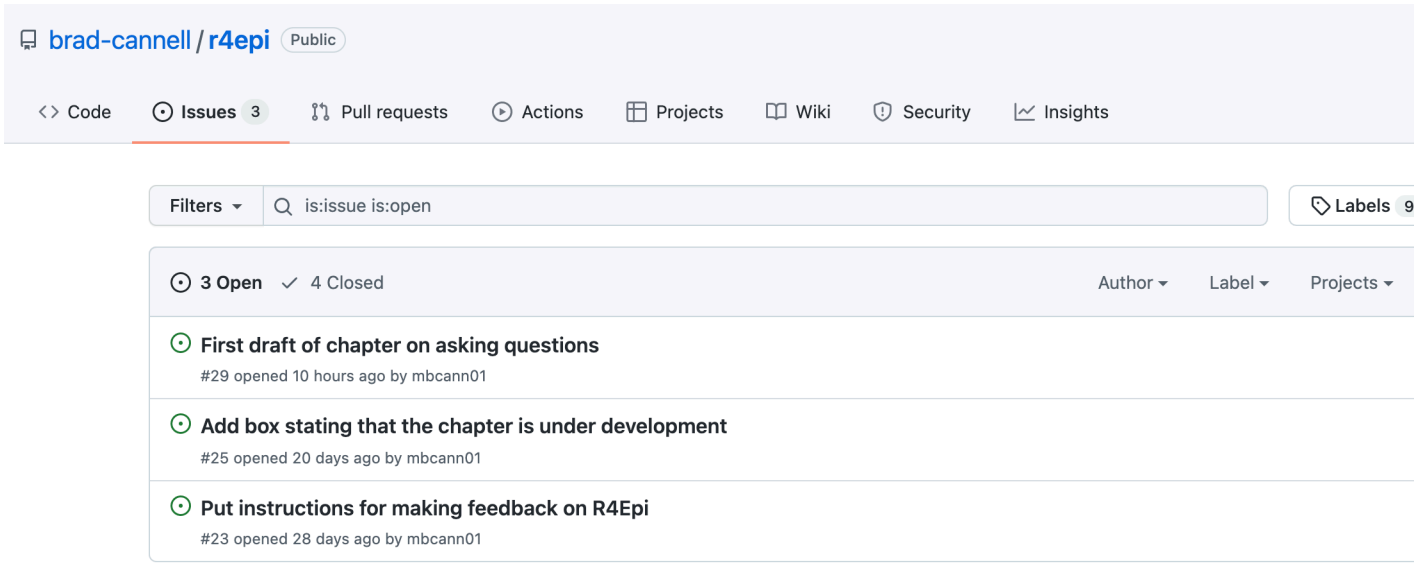
When we open the pull request, we can see the proposed changes to the file.

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



Issues

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



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

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

License Information

About the Authors

Brad Cannell

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

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

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

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

Contact
Connect with Dr. Cannell and follow his work.

Melvin Livingston

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

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

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

Contact
Connect with Dr. Livingston and follow his work.

Part I

Getting Started

1 Installing R and RStudio

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

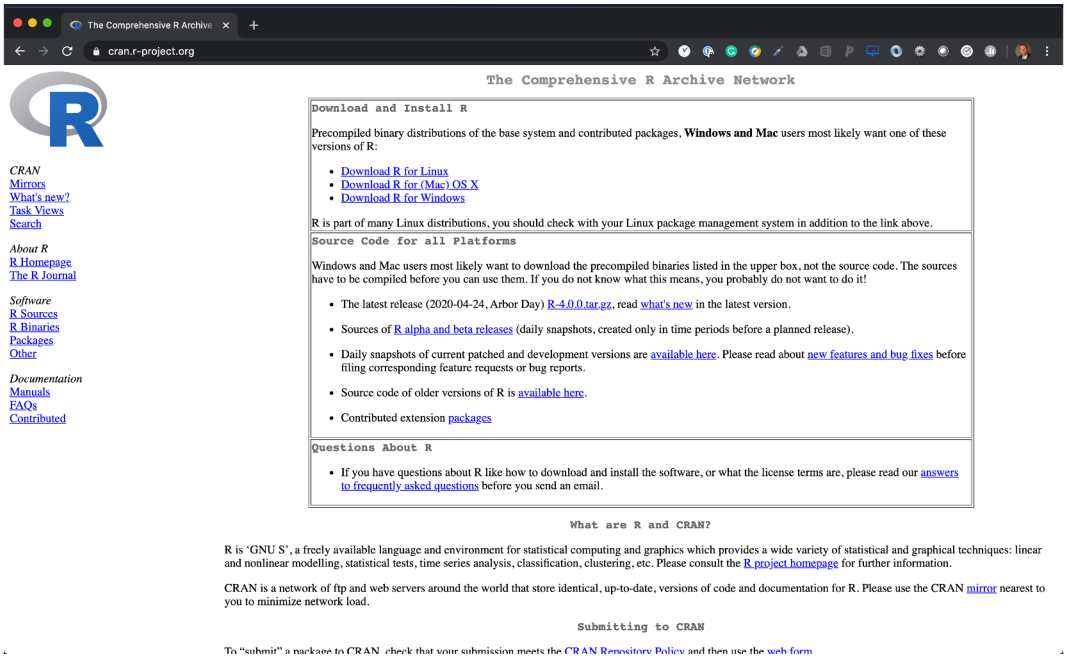
At this point, you may be wondering what R is, what RStudio is, and how they are related. We will answer those questions in the near future. However, in the interest of keeping things brief and simple, We're not going to get into them right now. Instead, all you have to worry about is getting the R programming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

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

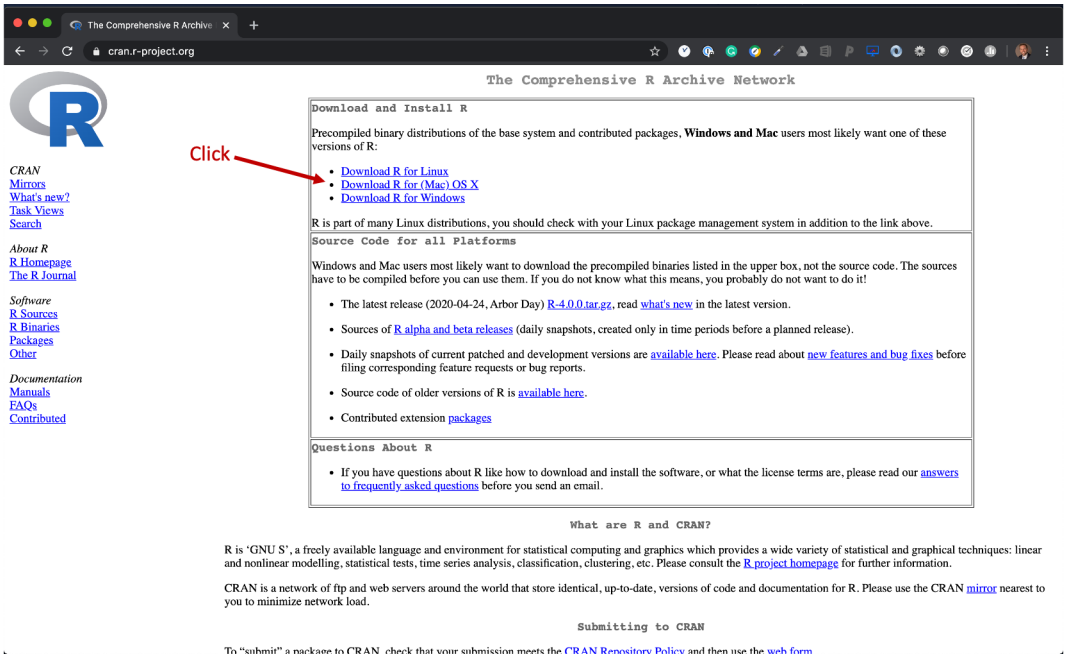
Step 1: Regardless of which operating system you are using, please make sure your computer is on, properly functioning, connected to the internet, and has enough space on your hard drive to save R and RStudio.

1.1 Download and install on a Mac

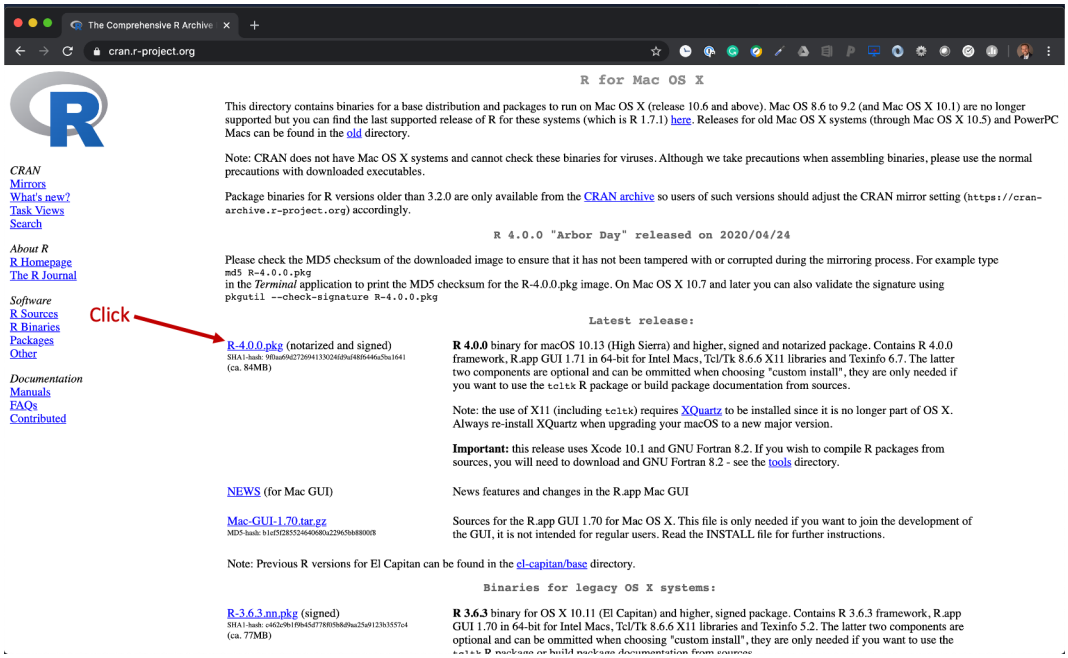
Step 2: Navigate to the Comprehensive R Archive Network (CRAN), which is located at https://cran.r-project.org/.



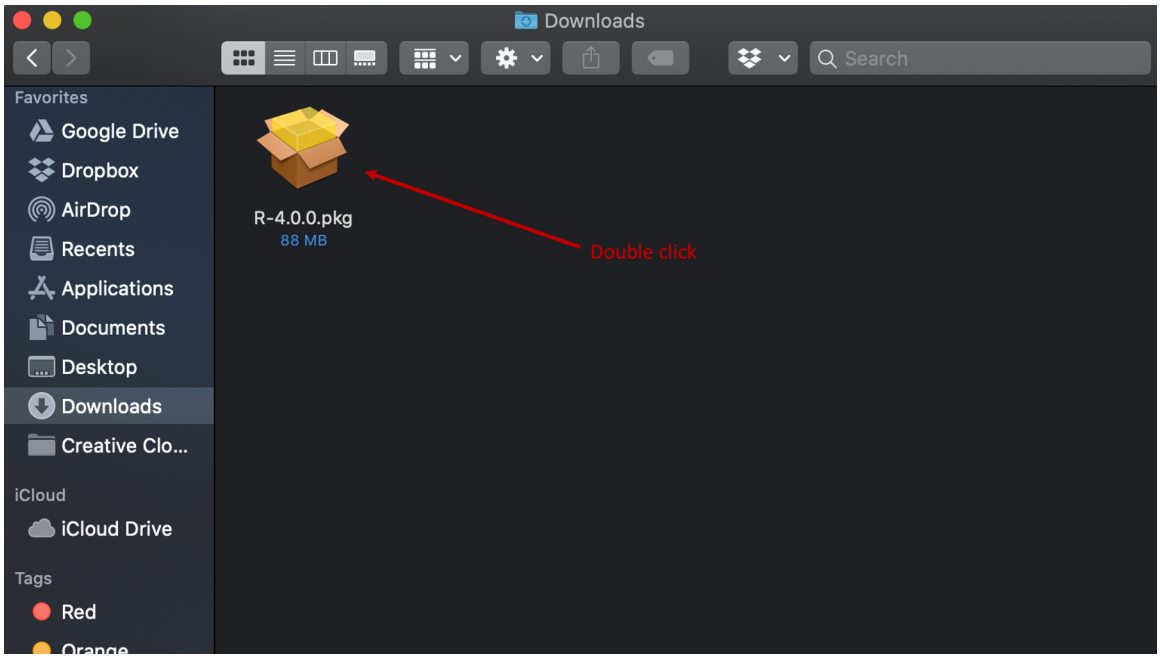
Step 3: Click on Download R for macOS.



Step 4: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version should be roughly in the same place – the middle of the screen under "Latest release:". After clicking the link, R should start to download to your computer automatically.

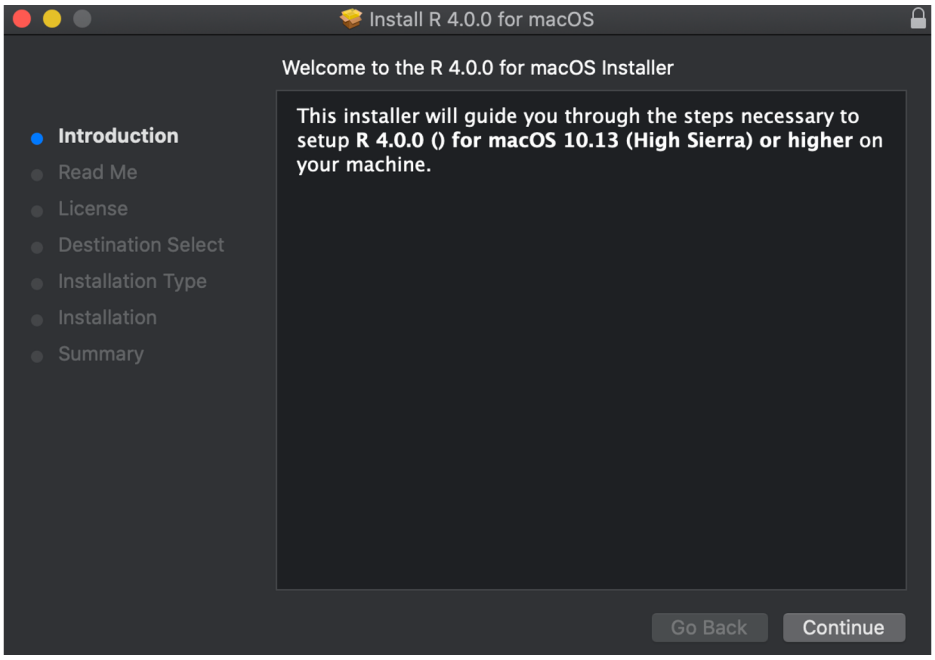


Step 5: Locate the package file you just downloaded and double click it. Unless you've changed your download settings, this file will probably be in your "downloads" folder. That is the default location for most web browsers. After you locate the file, just double click it.



Step 6: A dialogue box will open and ask you to make some decisions about how and where you want to install R on your computer. We typically just click "continue" at every step

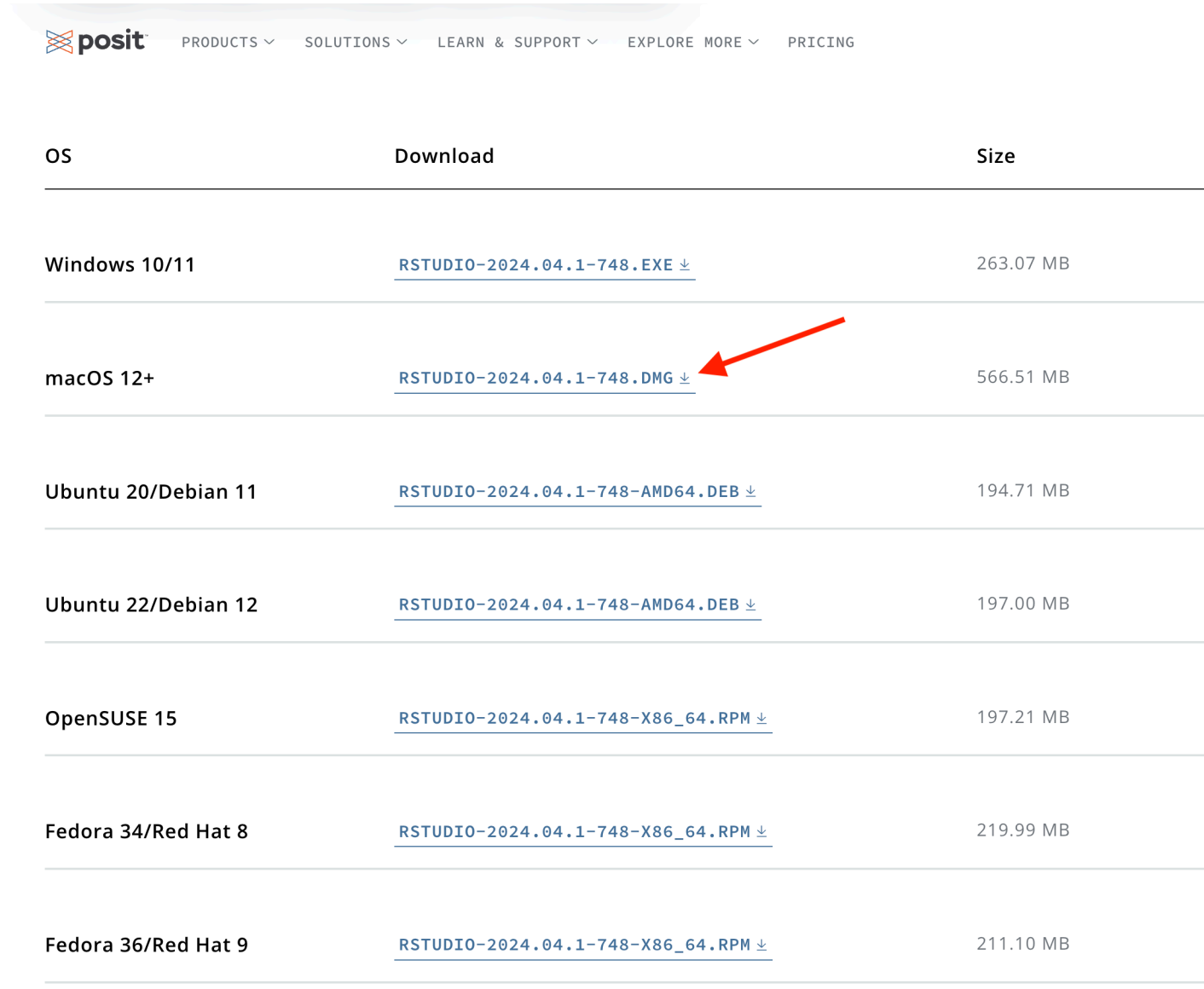
without changing any of the default options.



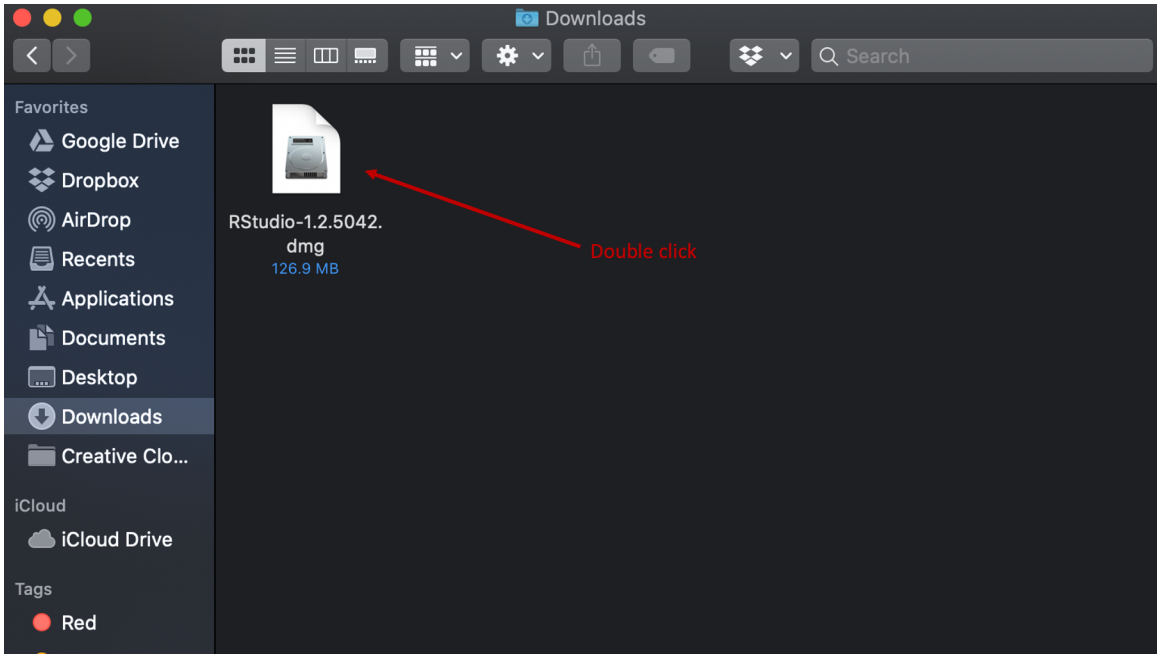
If R installed properly, you should now see it in your applications folder.

Step 7: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at https://posit.co/download/rstudio-desktop/. On that

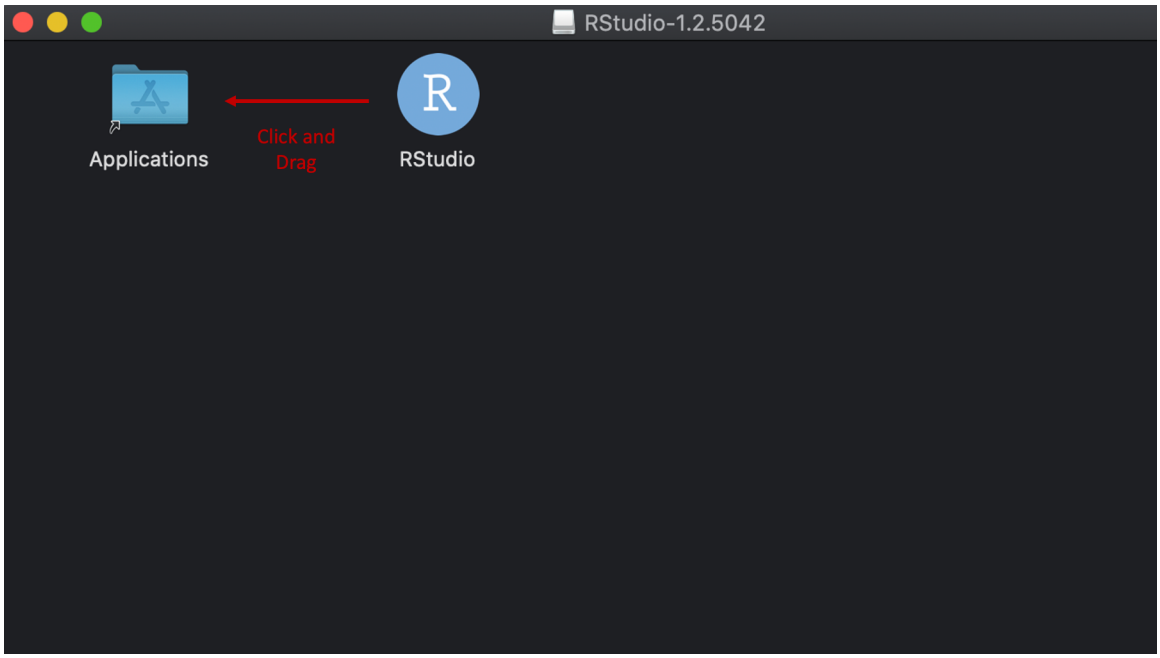
page, click the button to download the latest version of RStudio for your computer. Note that the website may look different that what you see in the screenshot below because websites change over time.



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

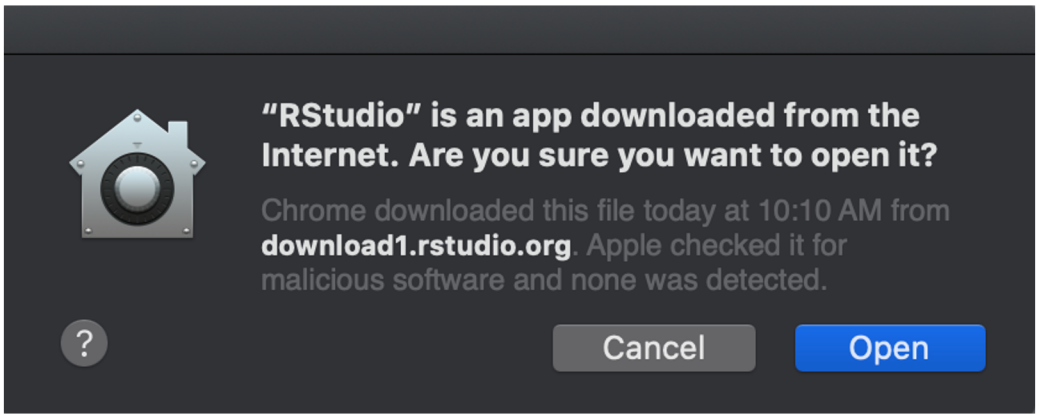


Step 9: A new finder window should automatically pop up that looks like the one you see below. Click on the RStudio icon and drag it into the Applications folder.

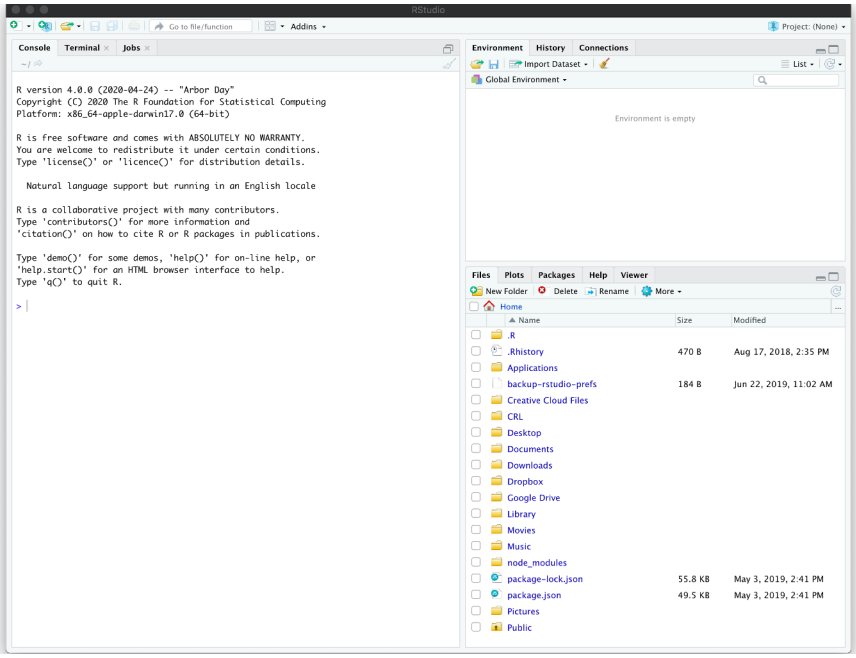


You should now see RStudio in your Applications folder. Double click the icon to open RStudio.

If this warning pops up, just click Open.

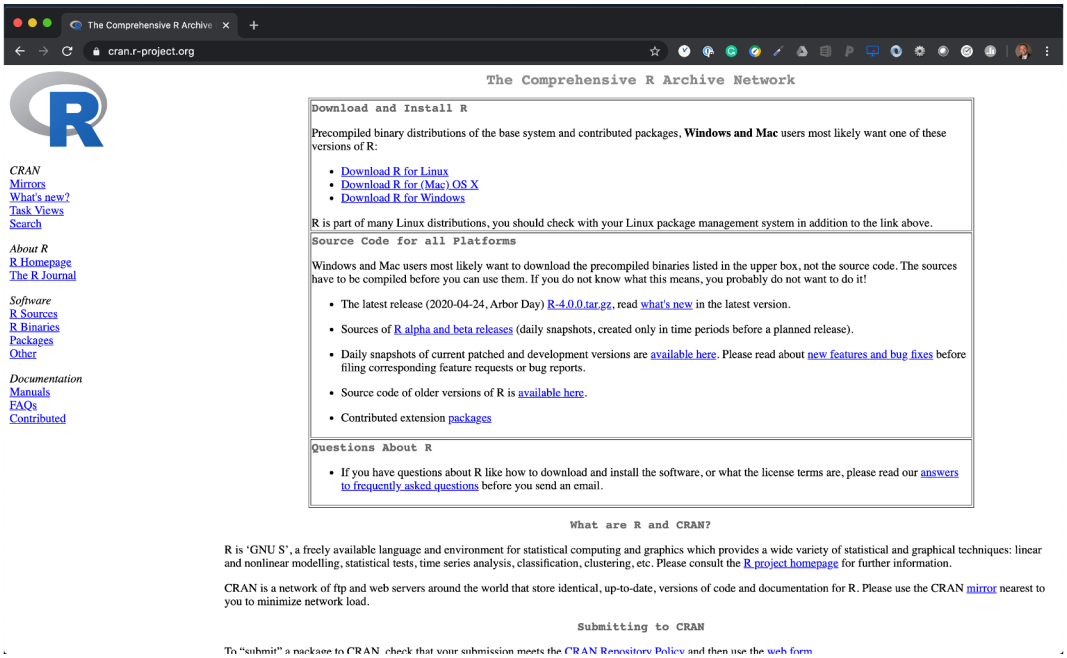


The RStudio IDE should open and look something like the window you see here. If so, you are good to go!

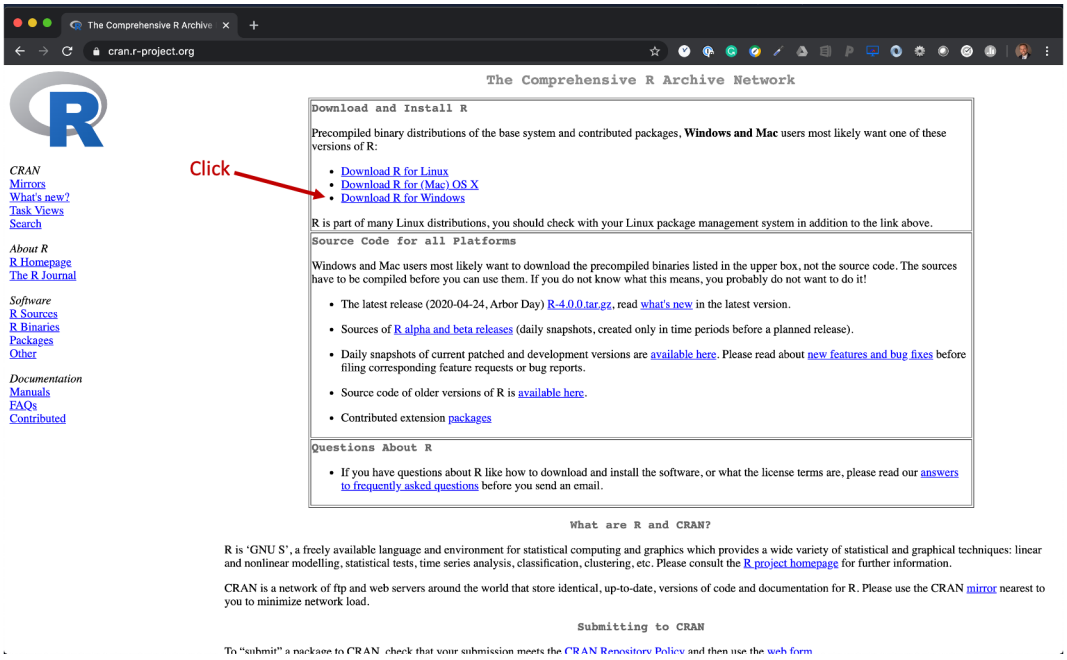


1.2 Download and install on a PC

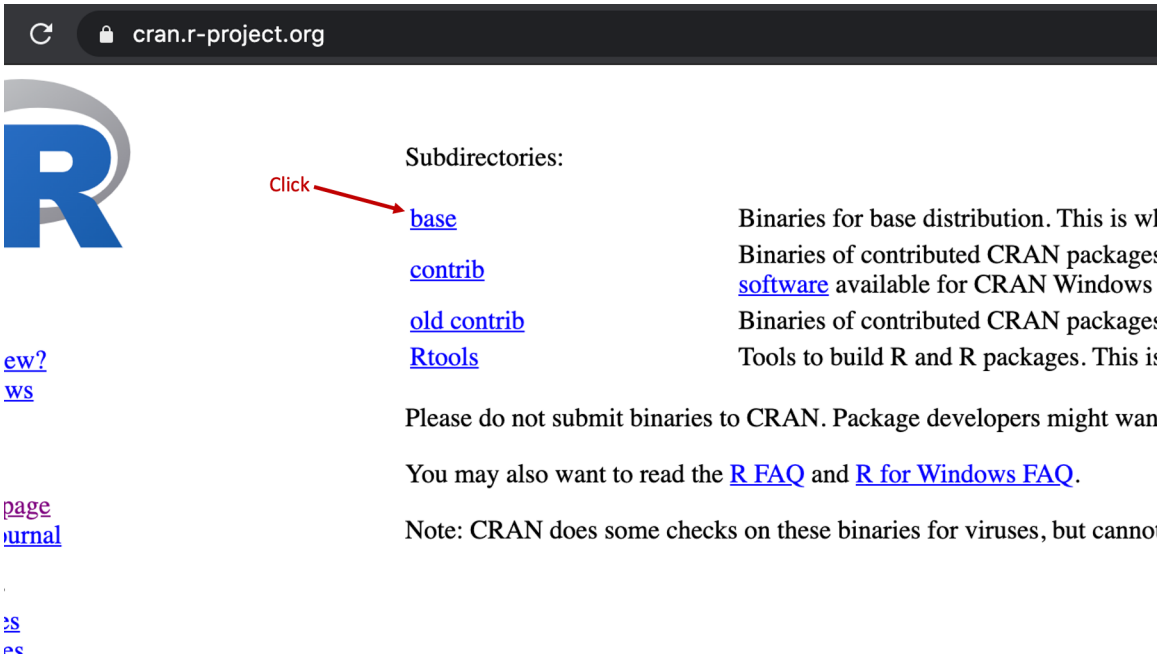
Step 2: Navigate to the Comprehensive R Archive Network (CRAN), which is located at https://cran.r-project.org/.



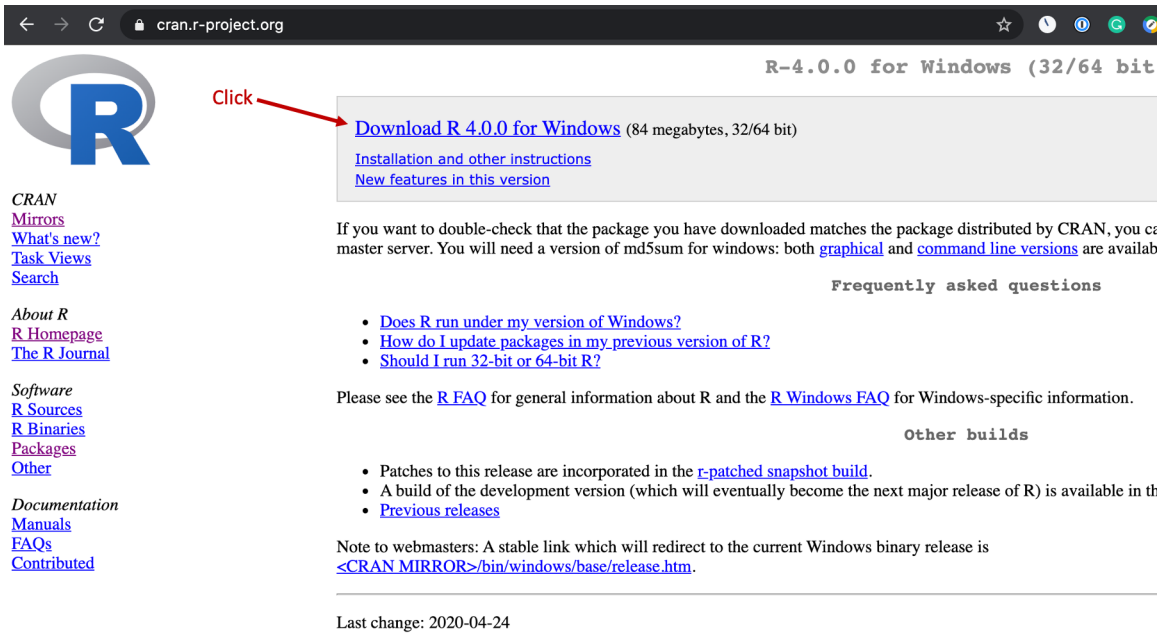
Step 3: Click on Download R for Windows.



Step 4: Click on the base link.

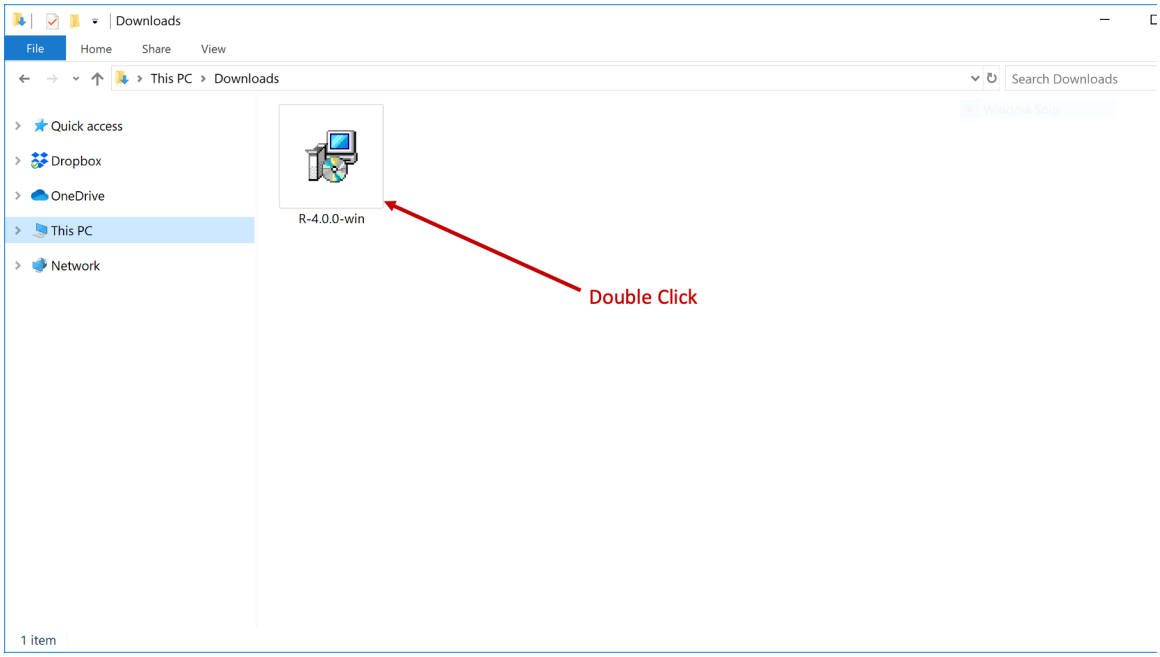


Step 5: Click on the link for the latest version of R. As you are reading this, the newest version may be different than the version you see in this picture, but the location of the newest version should be roughly the same. After clicking, R should start to download to your computer.

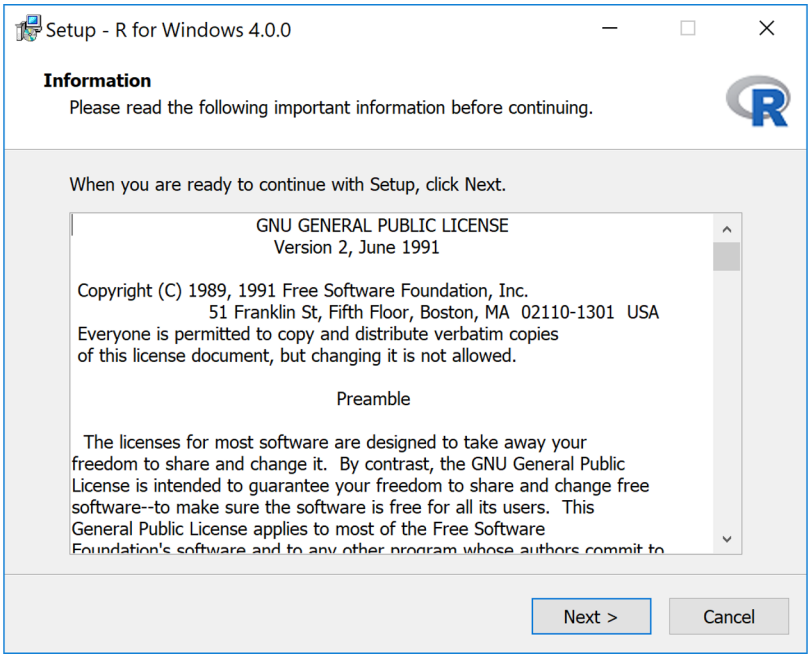


Step 6: Locate the installation file you just downloaded and double click it. Unless you've changed your download settings, this file will probably be in your downloads folder. That is

the default location for most web browsers.

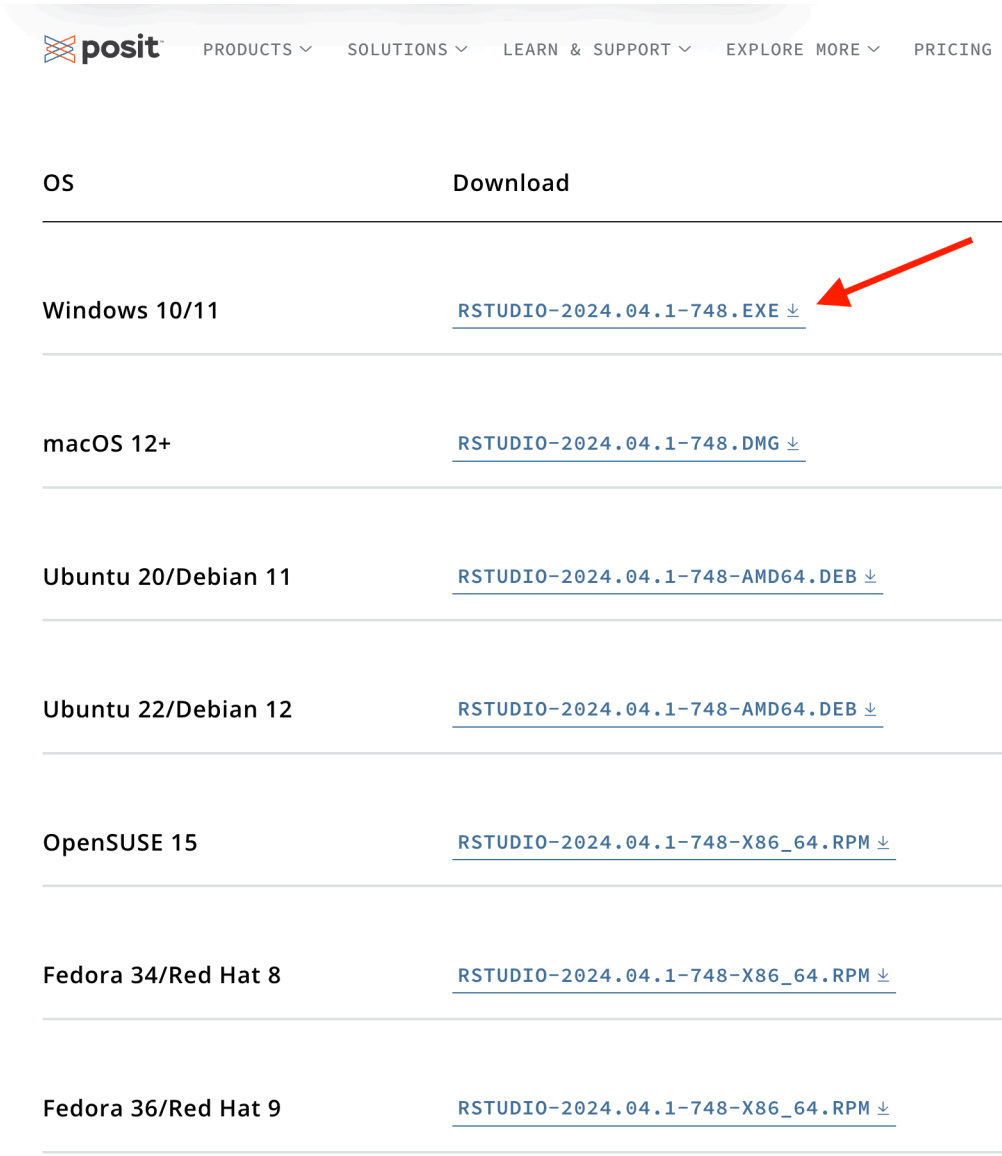


Step 7: A dialogue box will open that asks you to make some decisions about how and where you want to install R on your computer. We typically just click "Next" at every step without changing any of the default options.

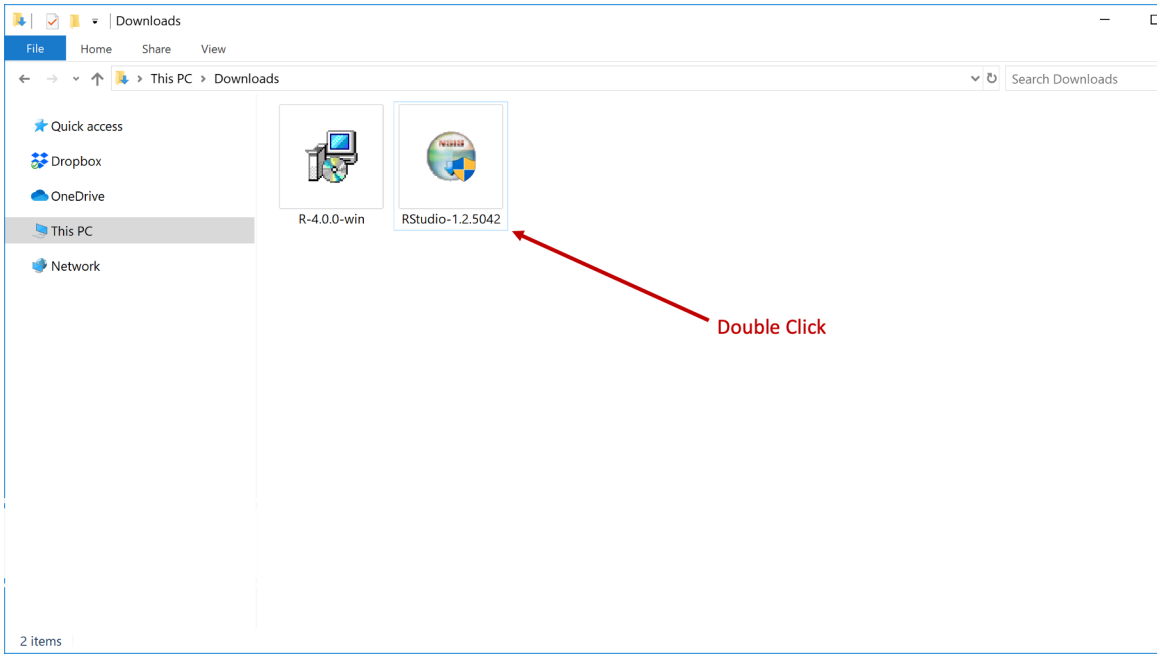


If R installed properly, you should now see it in the Windows start menu.

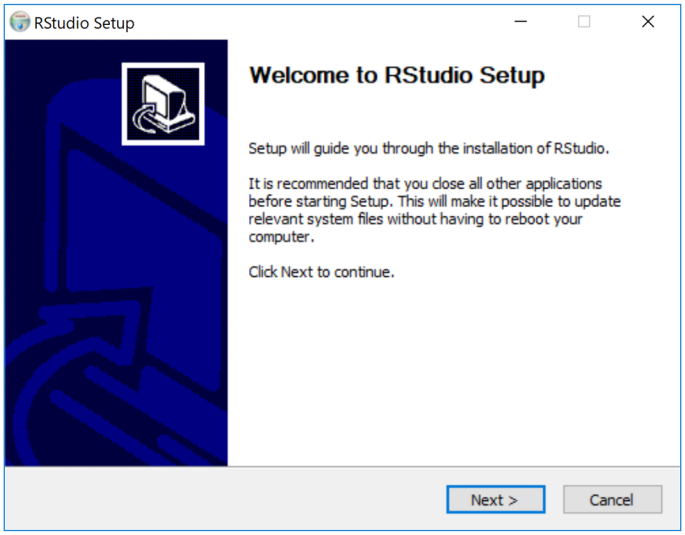
Step 8: Now, we need to install the RStudio IDE. To do this, navigate to the RStudio desktop download website, which is located at https://posit.co/download/rstudio-desktop/. On that page, click the button to download the latest version of RStudio for your computer. Note that the website may look different that what you see in the screenshot below because websites change over time.



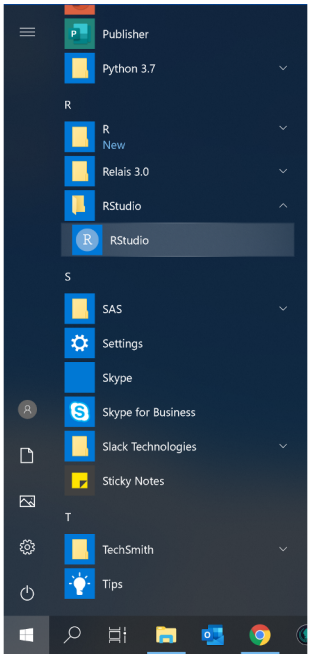
Step 9: Again, locate the installation file you just downloaded and double click it. Unless you've changed your download settings, this file should be in the same location as the R installation file you already downloaded.



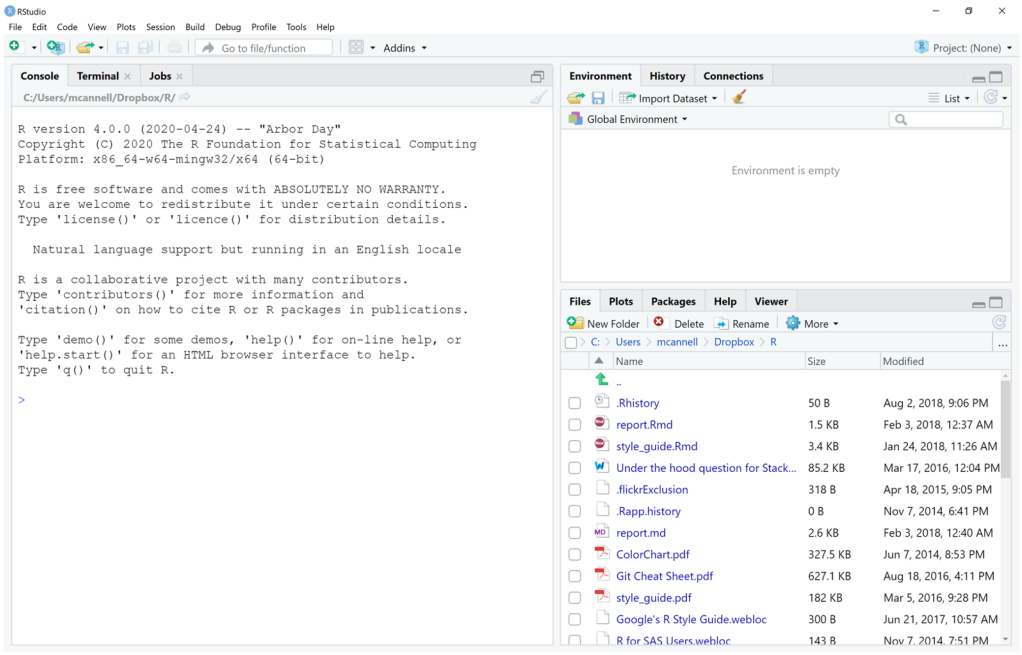
Step 10: Another dialogue box will open and ask you to make some decisions about how and where you want to install RStudio on your computer. We typically just click "Next" at every step without changing any of the default options.



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



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



2 What is R?

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

2.1 What is data?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

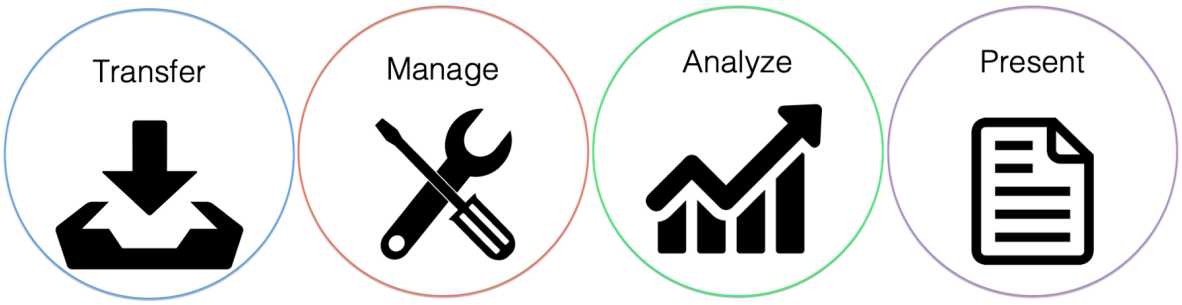
Finally, the contents of each cell are called **values**.

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

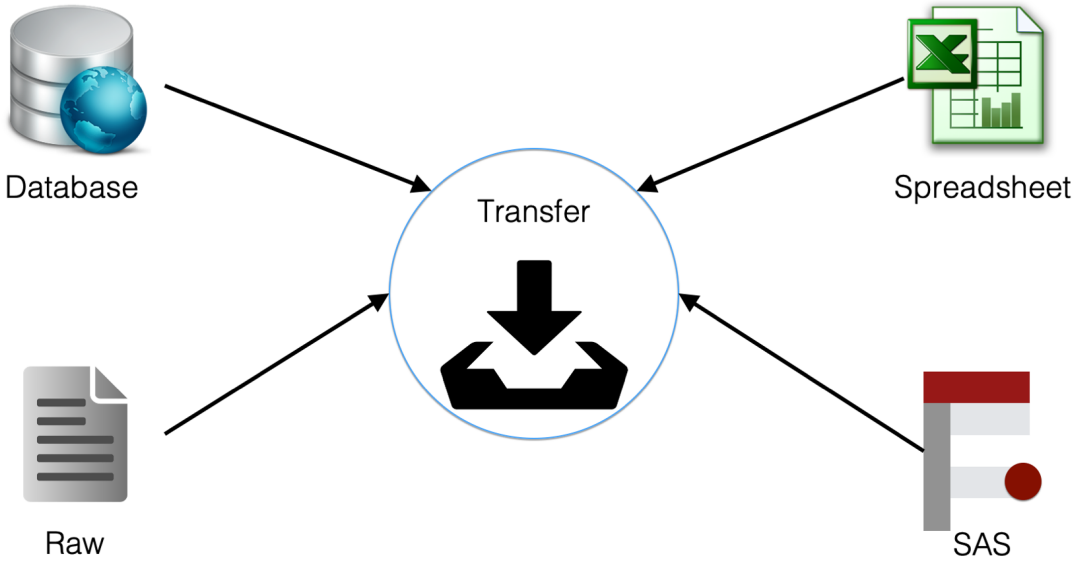
We should now be up to speed on some basic terminology used by R, as well as other analytic, database, and spreadsheet programs. These terms will be used repeatedly throughout the book.

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 our computer to do math and statistics for us. More specifically, by the end of this book we will be able to independently use R to transfer data, manage data, analyze data, and present the results of our 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 we may come across in epidemiology are database files, spreadsheets, raw data files, and SAS data sets. No matter how the data is stored, we can't do anything with it until we can get it into R, in a form that R can use, and in a location that R can access.

2.2.2 Managing data

Validating and Cleaning

Subsetting
(Using parts of the data)

New Variables

Combining

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

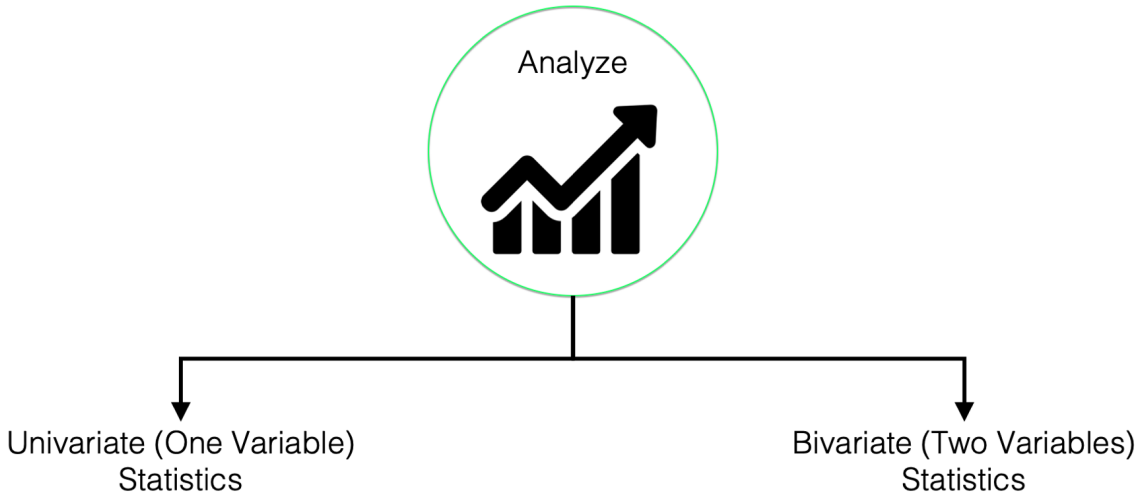
- Validating and cleaning data. In other words, dealing with potential errors in the data.
- Subsetting data – using only some of the columns or some of the rows.
- Creating new variables. For example, we might want to create a new `BMI` variable from existing `height` and `weight` variables.
- Combining data frames. For example, we might want to combine a data frame containing sociodemographic data about study participants with a data frame containing intervention outcomes data about those same participants.

We may sometimes hear people refer to the 80/20 rule about data management. This "rule" says that in a typical data analysis project, roughly 80% of our time will be spent on data management, while 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 courses 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 book will start by heavily emphasizing the latter.

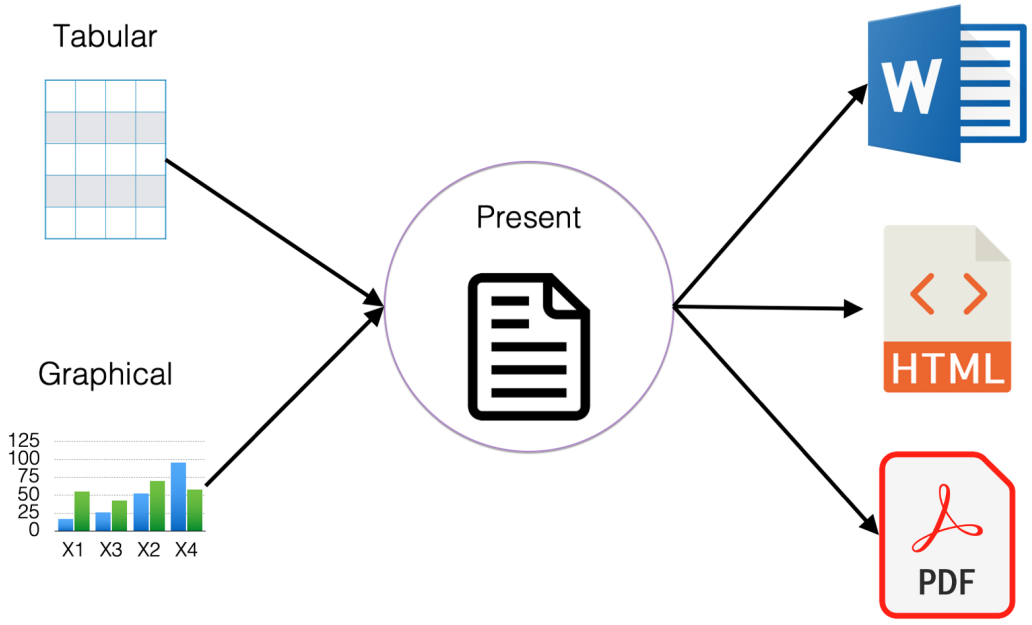
2.2.3 Analyzing data



As discussed above, this is probably the capability most people 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 our findings in some form or another. For example, a report, a website, or a journal article. With R we can present our results in many different formats with relative ease. In fact, this is one of our favorite things about R and RStudio. In this book we will learn how to publish our text, tabular, or graphical results in many different formats including Microsoft Word documents, 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 Word document created with R and the `officedown` package.
2. **PDF documents**: Click here to view a gallery of documents, including PDF.

3. **HTML files**: HTML (HyperText Markup Language) is the standard format for web pages. R can create HTML files that can be shared via email or published online for others to view in their browser. Click here to browse a gallery of interactive dashboards built with R.

4. **Web applications**: R can even be used to build full-featured web applications. Click here to explore examples created with the Shiny package.

Now that we've explored what R is and how it can be used in public health and the health sciences, it's time to start learning how to actually use it. In the next chapter, Navigating the RStudio Interface, we'll begin by exploring the RStudio IDE and briefly introduce some of the basic building blocks of R code.

3 Navigating the RStudio Interface

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

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

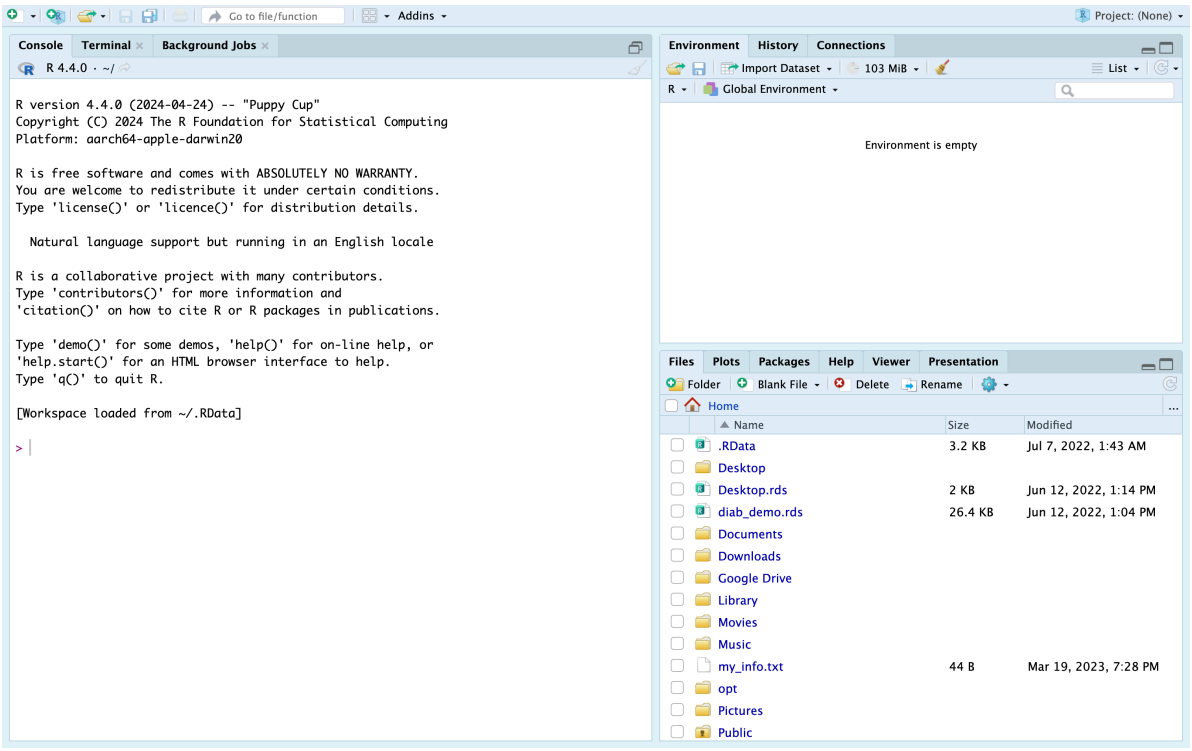


Figure 3.1: The default RStudio user interface.

3.1 The console pane

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

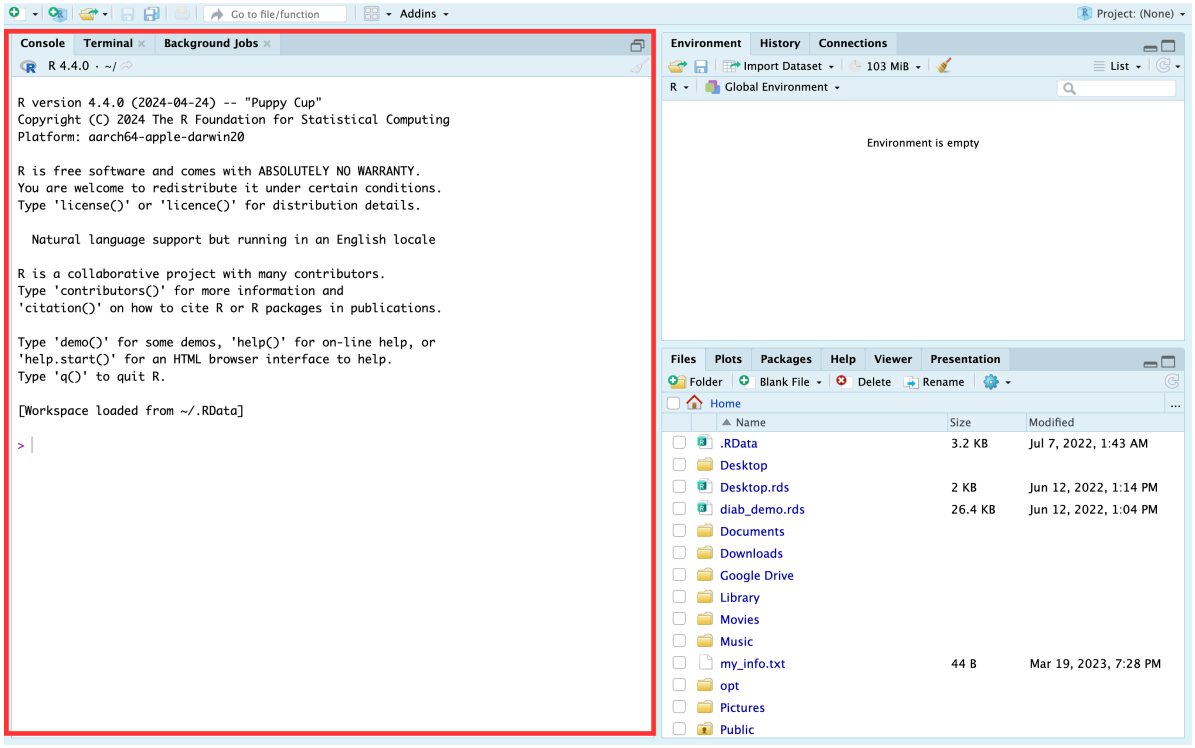


Figure 3.2: The R Console.

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

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

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

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

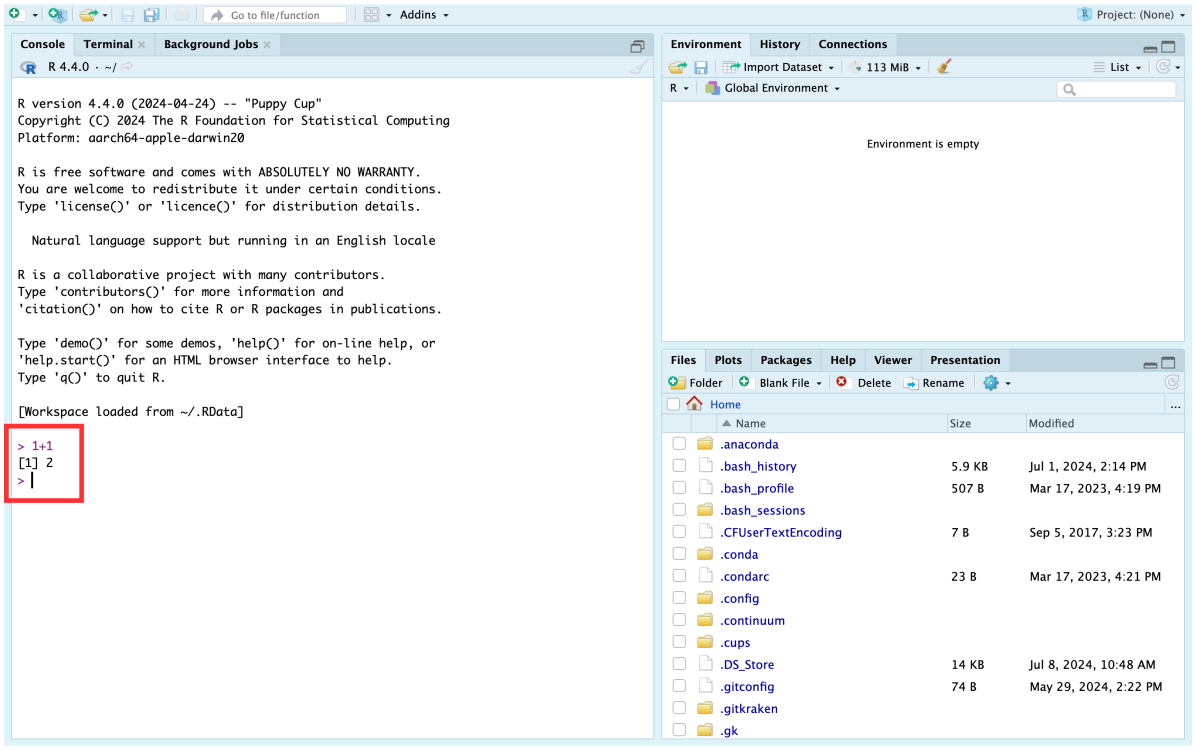


Figure 3.3: Doing some addition in the R console.

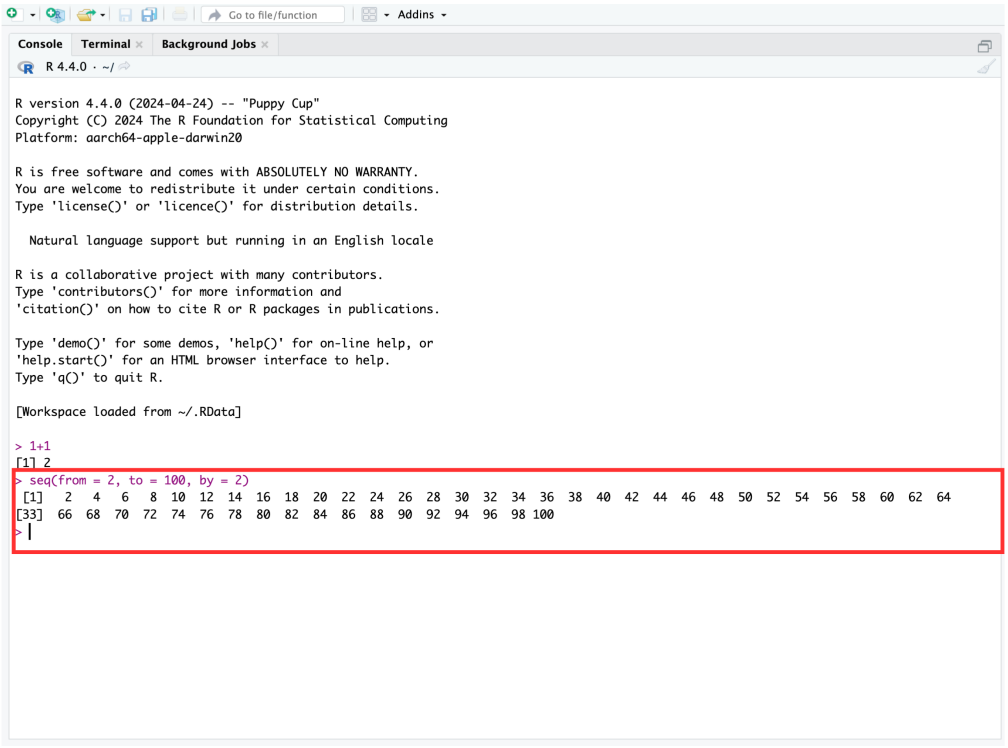


Figure 3.4: Demonstrating a function that returns multiple results.

First, as promised, we have more than one line of results (or output). The first line of results starts with a 1 in brackets (i.e., `[1]`), which indicates that this line of results starts with the first result. In this case, the first result is the number 2. The second line of results starts with a 33 in brackets (i.e., `[33]`), which indicates that this line of results starts with the thirty-third result. In this case, the thirty-third result is the number 66. If we count the numbers in the first line, there should be 32 – results 1 through 32. We also want to make it clear that `[1]` and `[33]` are *NOT* results themselves. They are just helping us count the number of results per line.

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

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

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

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

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

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

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

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

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

> **Note**
>
> 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.

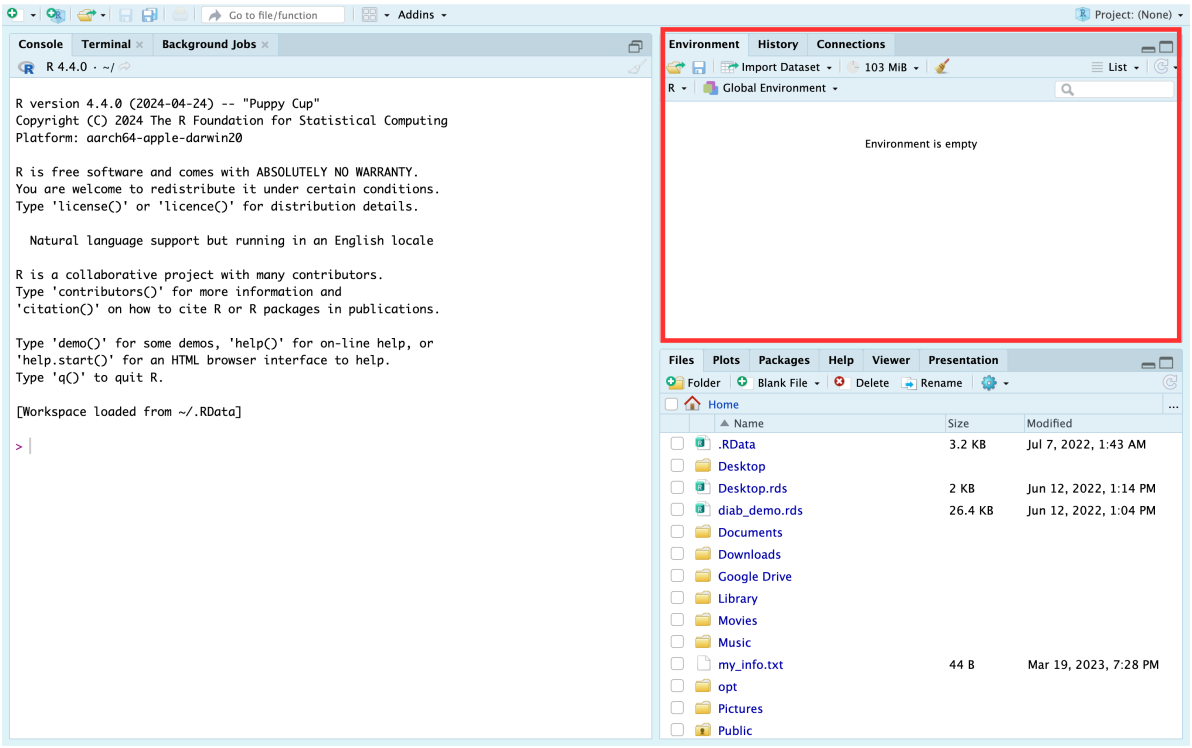


Figure 3.5: The environment pane

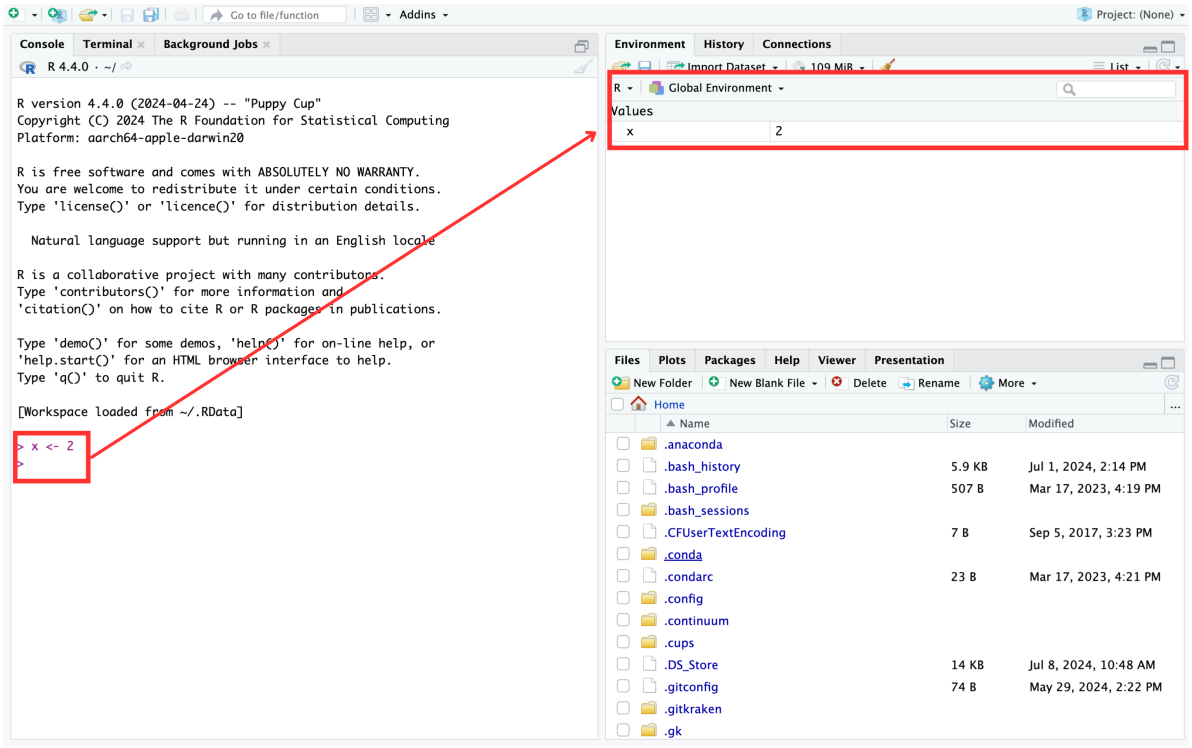


Figure 3.6: The vector x in the global environment.

> **Note**
>
> 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 we assign 2 to lower case x (`x <- 2`), and then later ask R to tell us what number we stored in uppercase X, we 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*.

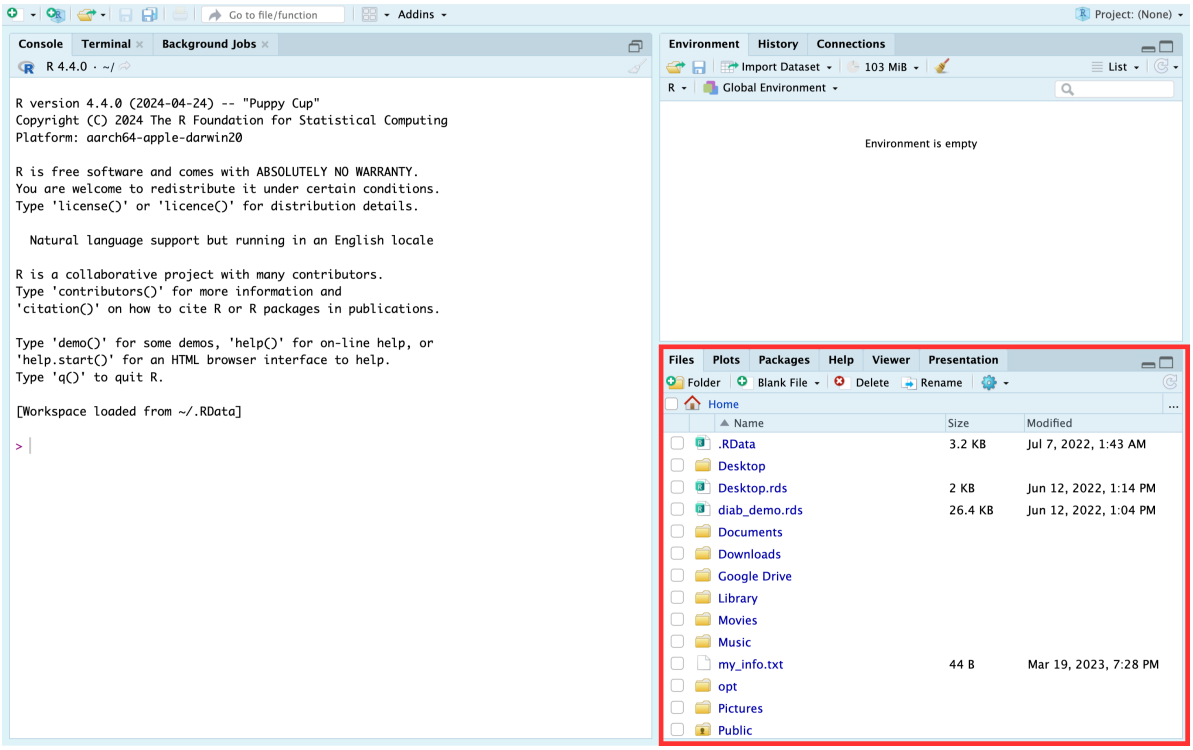


Figure 3.7: The Files/Plots/Packages/Help/Viewer pane.

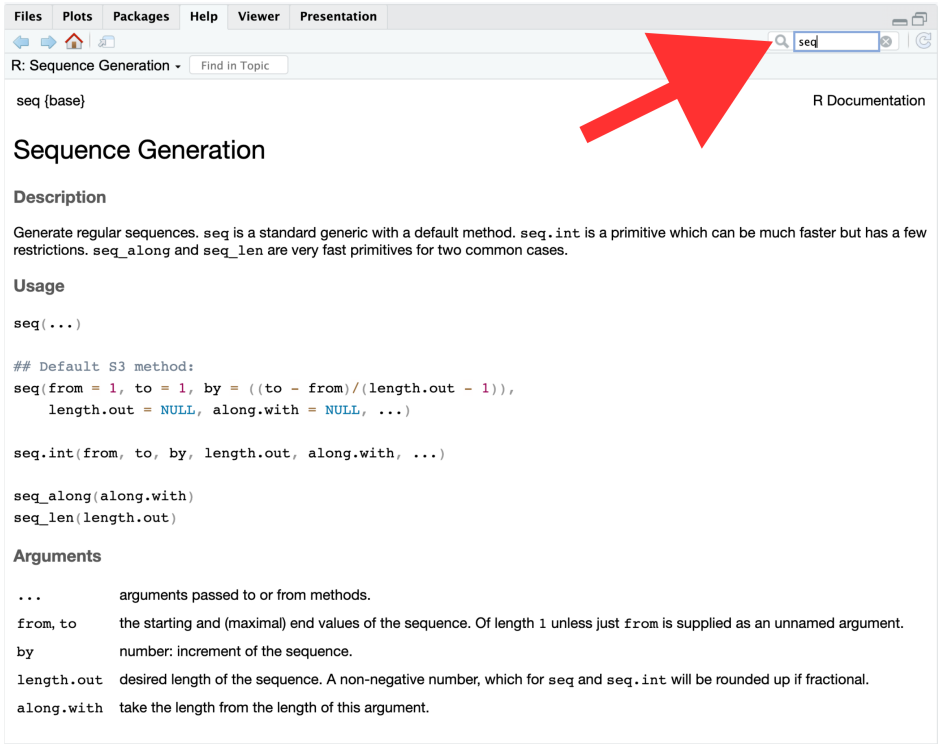


Figure 3.8: The help tab.

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

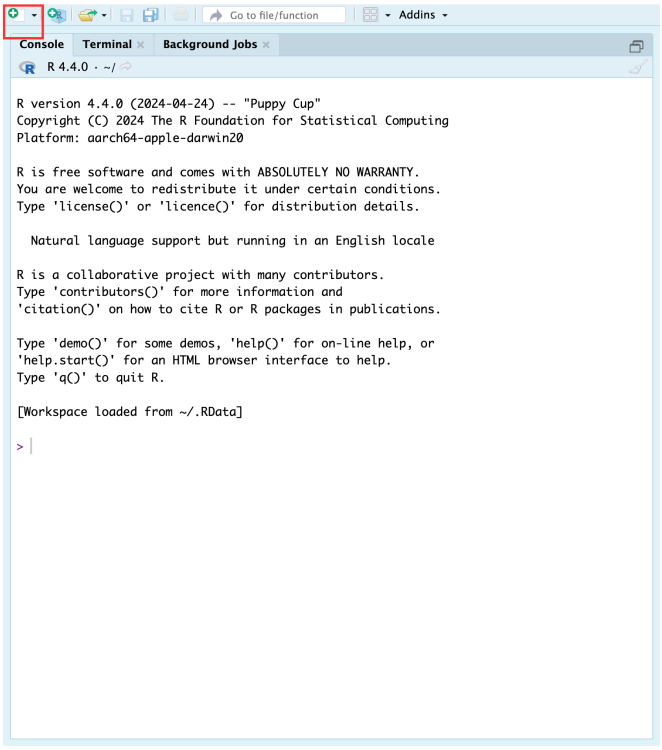


Figure 3.9: Click the new source file icon.

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.

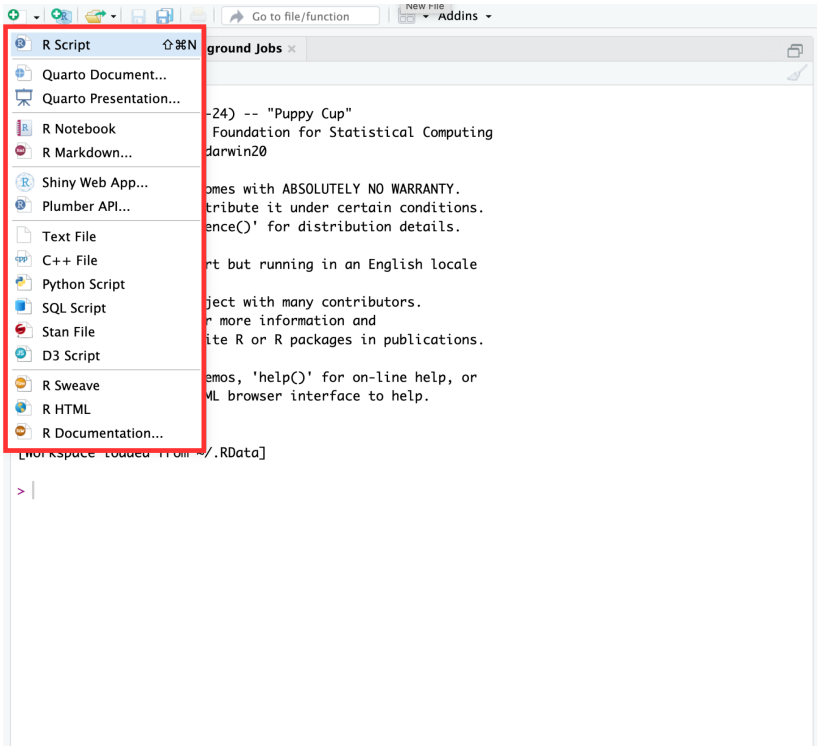


Figure 3.10: New source file options.

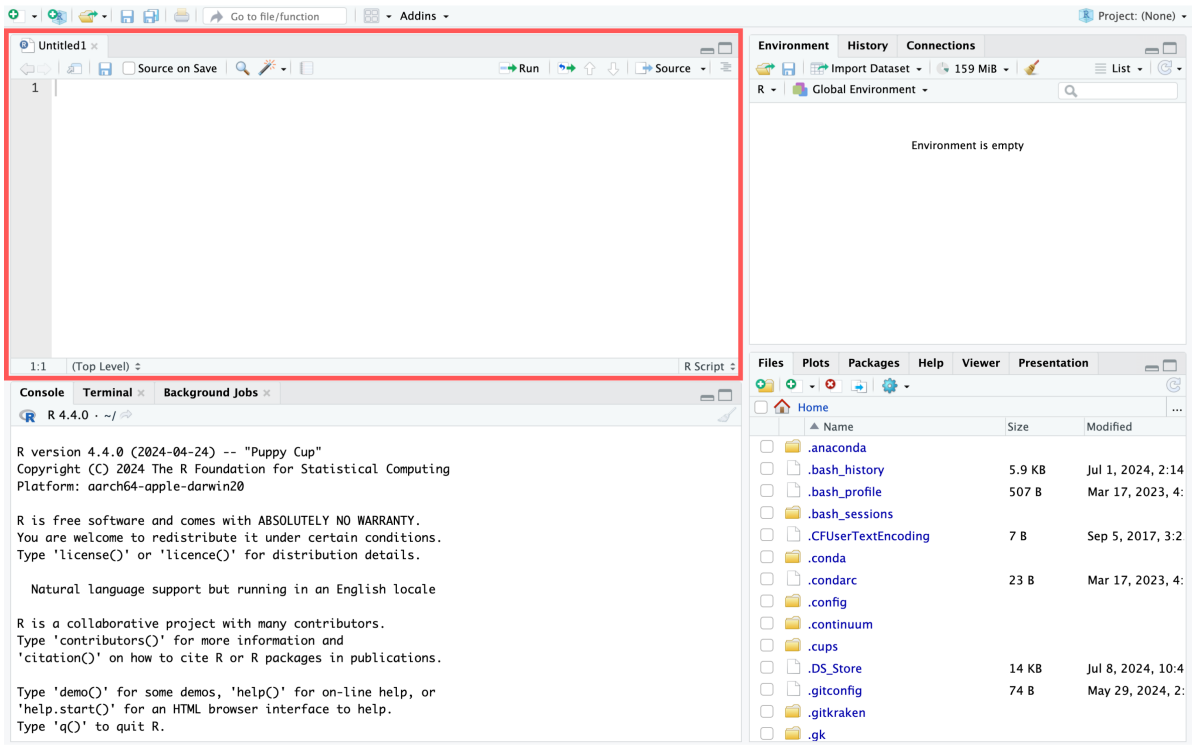


Figure 3.11: A blank R script in the source pane.

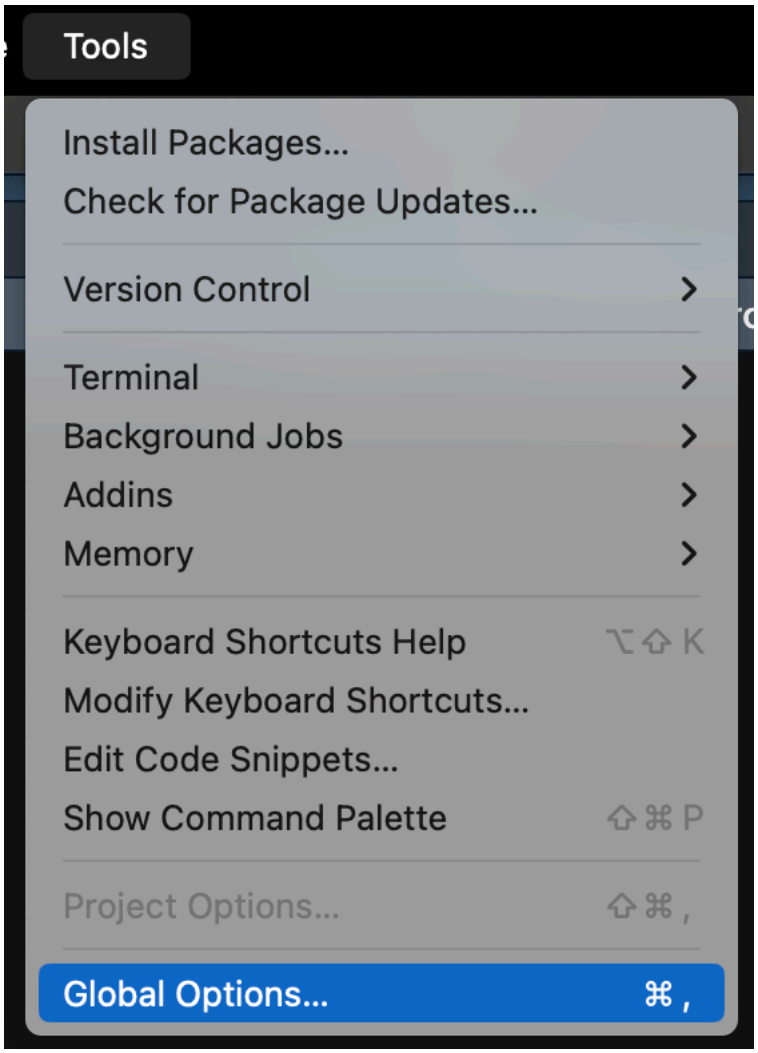


Figure 3.12: Select the preferences menu on Mac.

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.

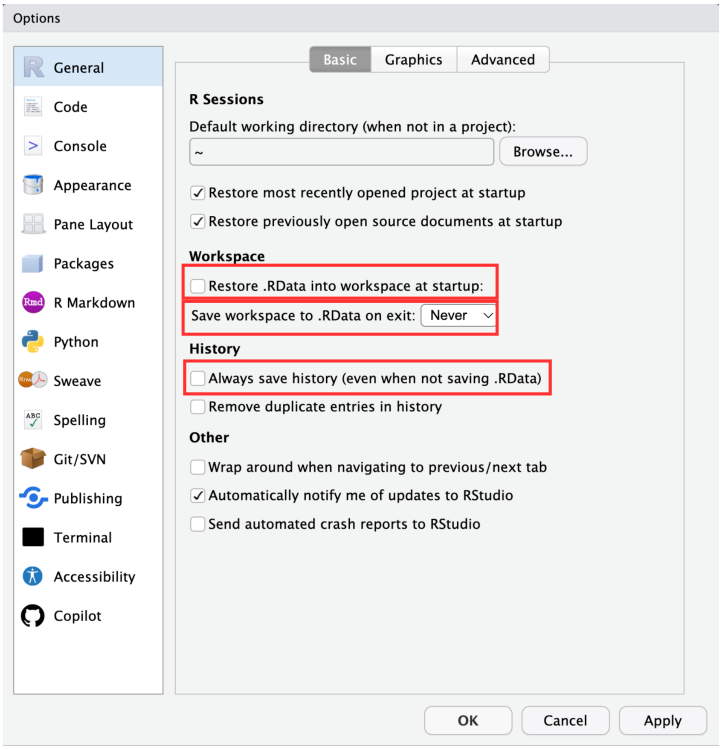


Figure 3.13: General options tab.

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?

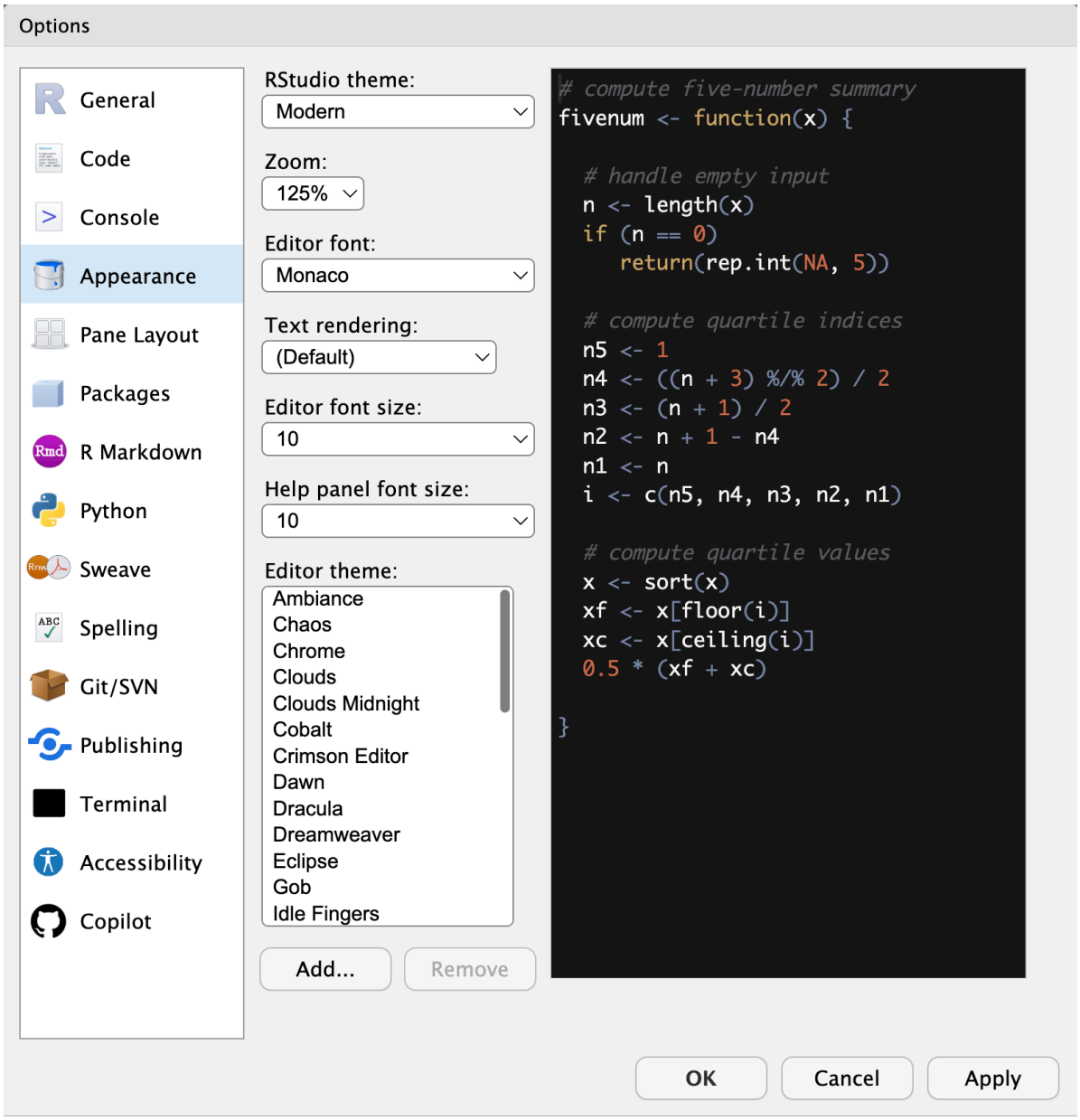


Figure 3.14: Appearance tab.

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**
>
> 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 we speak to R using the R language, who are we actually speaking to?

Well, we 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 us to understand about the R language is that it is **extremely literal!** Understanding the literal nature of R is important because it will be the underlying cause of a lot of the errors in our R code.

4.3 Errors

It's inevitable: errors will happen in your R code. Even experienced programmers who have been working with R for many years get errors when they write code. The goal of this section is to help us begin to understand *why* errors happen, and to give us a shared language for talking about them.

So, what exactly do we mean when we say that the R interpreter is extremely literal? In the previous lesson, we learned that R is a **case-sensitive** language. That means that uppercase `X` and lowercase `x` are treated as two completely different objects.

For example, if we assign the value `2` to lowercase `x` using `x <- 2`, and then later ask R to show us the contents of uppercase `X`, we'll get an error (`Error: object 'X' not found`):

```
x <- 2
X
```

```
Error: object 'X' not found
```

Specifically, this is an example of a logic error. Meaning, R understands what we are *asking* it to do – we want it to print the contents of the uppercase `X` object to the screen. However, it can't complete our request because we 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 we 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. We can sort of think of syntax as the grammar of the language. In English, we could say something like, "giving dog water drink." This sentence is grammatically incorrect; however, many people would roughly be able to figure out what's being asked based on their 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 we may be able to address them. The remainder of this chapter will discuss some key components of R's syntax and the data structures (i.e., ways of storing data) that the R syntax interacts with.

4.4 Functions

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

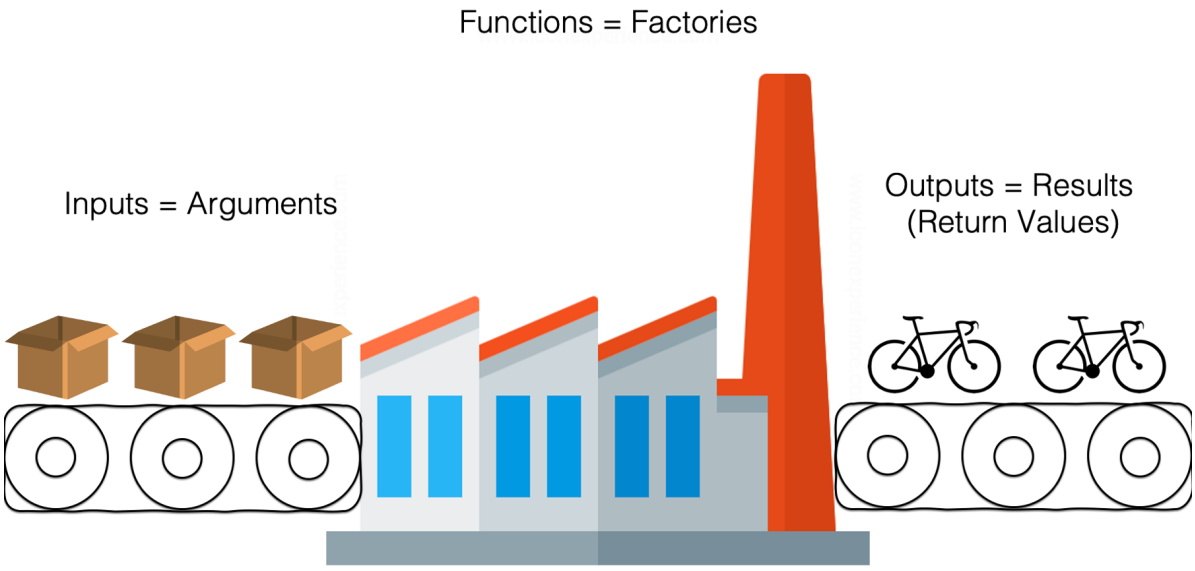


Figure 4.1: A factory making bicycles.

To make this concept more concrete, in Chapter 3 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.

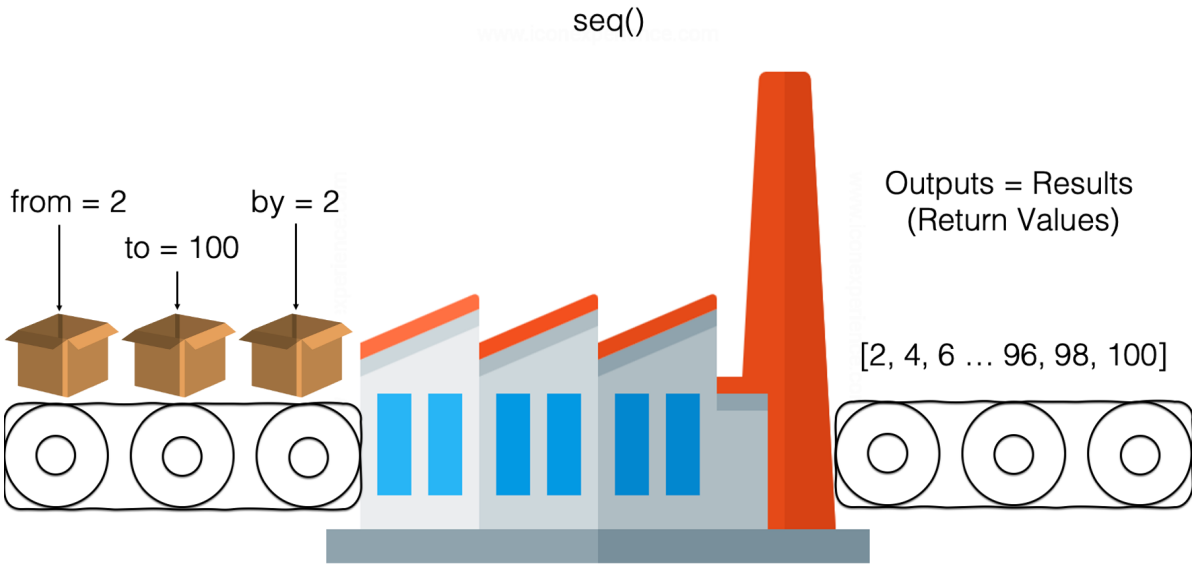


Figure 4.2: A function factory making numbers.

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 Section 3.3? There, we saw the documentation for the `seq()` function.

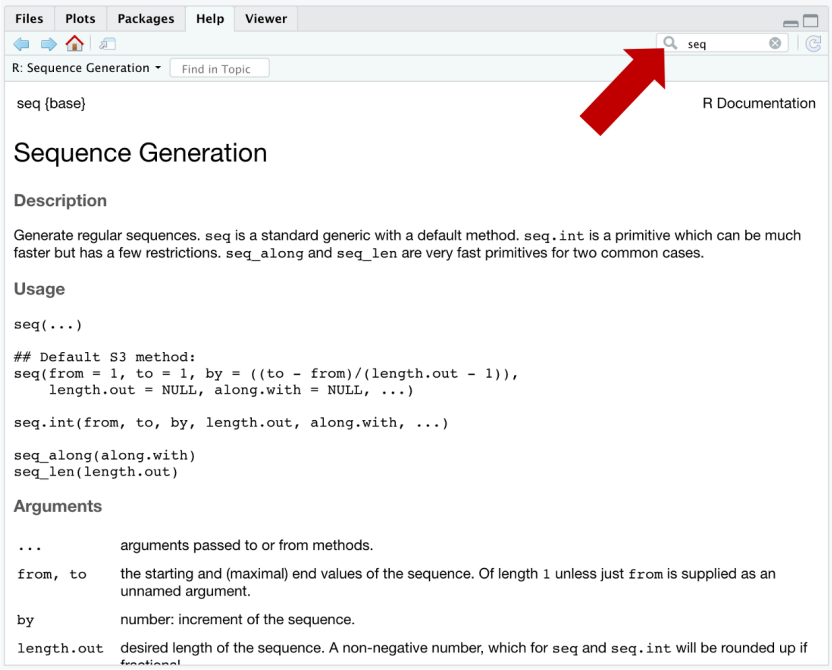


Figure 4.3: The help tab.

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

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

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

```
[1] 2
```

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

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

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

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

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

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

and

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

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

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

4.5 Objects

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

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

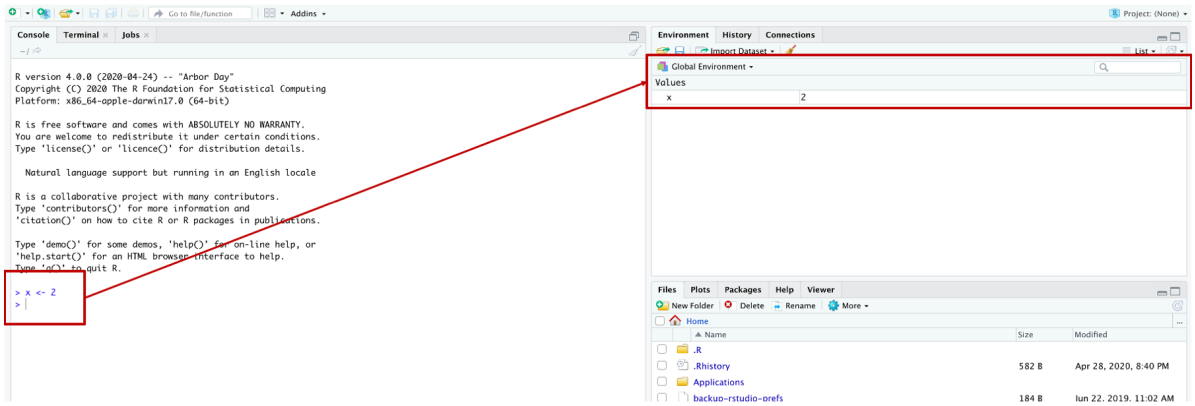
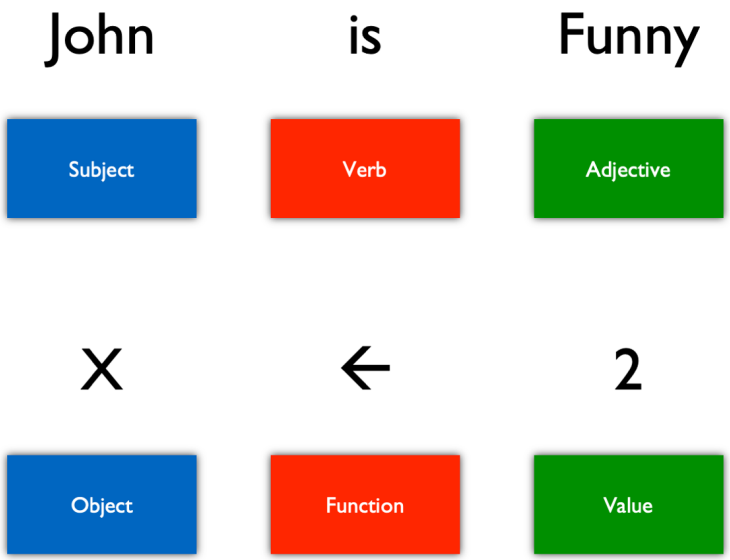


Figure 4.4: Creating the x object.

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

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



4.6 Comments

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

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

```
[1] 2
```

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

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

```
[1] 2
```

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

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

> **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. Chapter 10 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 primarily work with four vector types in this book: **character**, **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 Chapter 19. 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 Chapter 19. 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 Chapter 19.

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 Chapter 4. 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:

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

# Print my_df to the screen
my_df
```

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

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

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

Here's what we did above:

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

5.4.3 The tribble function

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

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

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

# Print my_df to the screen
my_df
```

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

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

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

Here's what we did above:

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

5.4.4 Why use tibbles

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

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

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

```
Error: object 'age_years' not found
```

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

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

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

# Print my_df to the screen
my_df
```

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

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

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

# Print my_df to the screen
my_df
```

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

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

5.5 Missing data

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

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

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

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

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

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

```r
typeof(NA)
```

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

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

```r
typeof(NA_character_)
```

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

```r
typeof(NA_integer_)
```

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

```r
typeof(NA_real_)
```

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

Most of the time, we won't have to worry about doing this because R will take care of converting `NA` for us What do we mean by that? Well, remember that every vector can have only one type. So, when we 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 us 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 us 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 we don't have to worry about type coercion – it will happen automatically. But, sometimes it doesn't and it will cause R to give us an error. we mostly encounter this when using the `if_else()` and `case_when()` functions, which we will discuss later.

5.6 Our first analysis

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

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

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

5.6.1 Manual calculation of the mean

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

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

```
[1] 68.5
```

Here's what we did above:

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

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

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

5.6.2 Dollar sign notation

```
class$heights
```

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

Here's what we did above:

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

5.6.3 Bracket notation

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

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

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

```
[1] 71
```

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

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

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

```
[1] 68.5
```

5.6.4 The sum function

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

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

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

```
[1] 274
```

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

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

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

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

```
[1] 68.5
```

Here's what we did above:

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

5.6.5 Nesting functions

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

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

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

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

```
[1] 2
```

Here's what we did above:

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

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

```
8 / 4
```

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

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

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

```
[1] 68.5
```

Here's what we did above:

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

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

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

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

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

5.6.6 The length function

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

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

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

```
[1] 4
```

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

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

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

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

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

```
[1] 68.5
```

Here's what we did above:

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

5.6.7 The mean function

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

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

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

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

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

```r
# 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.

```r
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.

```r
mean(weight)
```

```
Error: 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:

```r
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.

Before we dive into more advanced programming and data analysis, we'll take a moment to focus on an equally important skill: how to ask good questions and get help when we're stuck. That's the topic of the next chapter — and it's something that will serve us well throughout our entire journey as an R programmer.

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.

> **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.

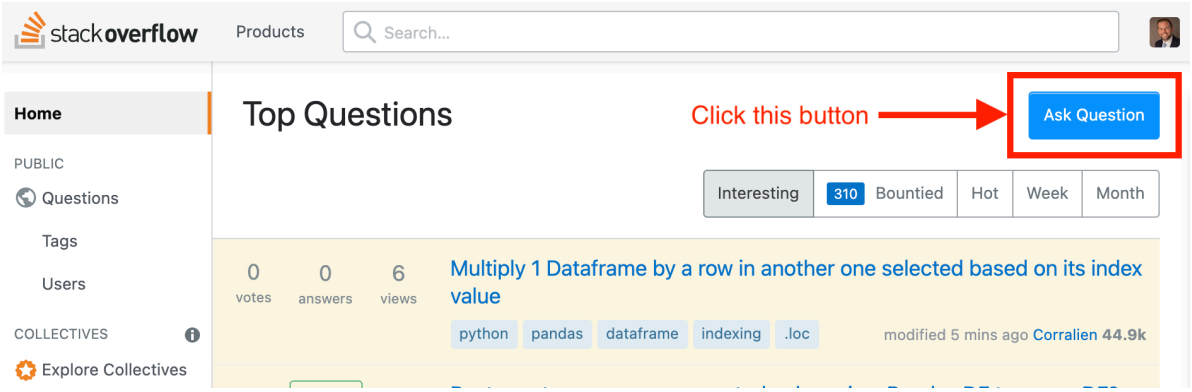
6.3 How should we seek help?

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?

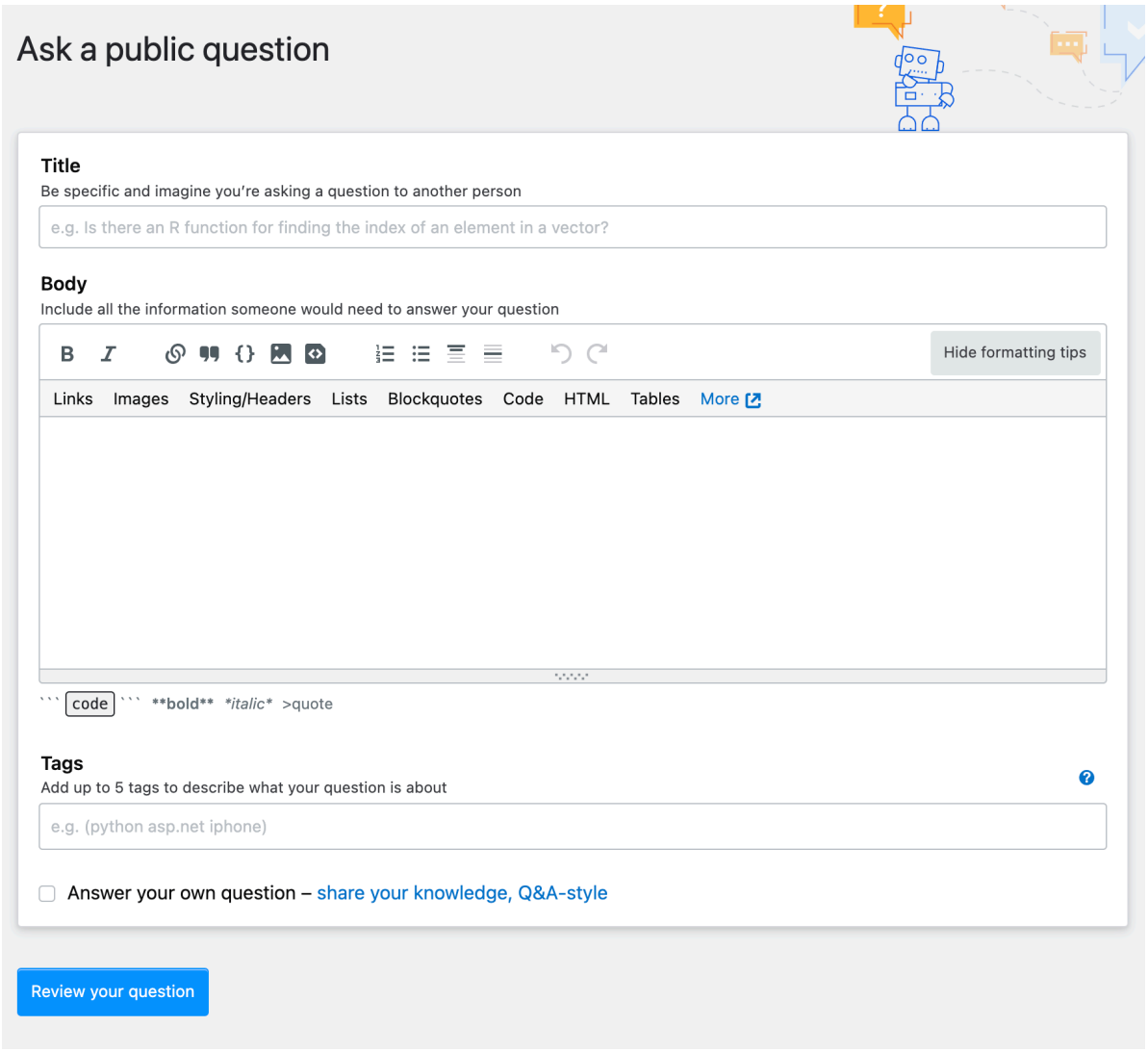
> **Note**
>
> We've decided to show you haw to create a post on Stack Overflow in this section, but the process for creating a post in the RStudio Community is very similar. Further, an RStudio Community tutorial is available here: https://community.rstudio.com/t/example-question-answer-topic-thread/70762.

6.3.1 Creating a post on Stack Overflow

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



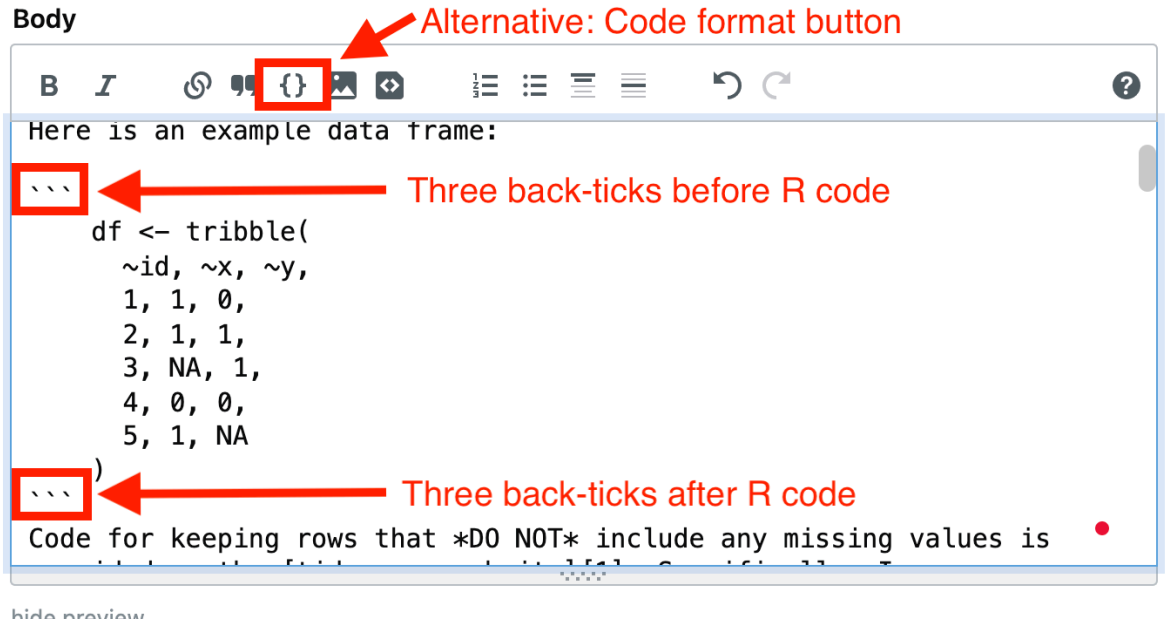
Next, you will click the blue “Ask Question” button. Doing so will take you to a screen like the following.



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

6.3.1.1 Inserting R code

To insert `R` code into your post (i.e., in the body), you will need to create **code blocks**. Then, you will type your `R` code inside of the code blocks. You can create code blocks using back-ticks (‘). The back-tick key is the upper-left key of most keyboards – right below the escape key. On our keyboard, the back-tick and the tilde (~) share the same key. We will learn more about code blocks in Chapter 8. For now, let's just take a look at an example of creating a code block in the screenshot below. This screenshot comes from the example Stack Overflow post introduced at the beginning of the chapter.



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

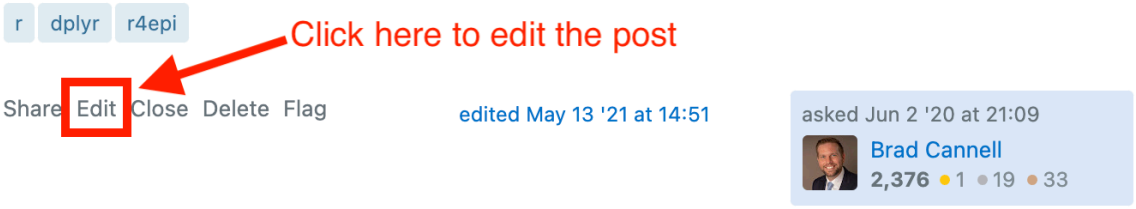
6.3.1.2 Reviewing the post

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

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

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

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



6.3.2 Creating better posts and asking better questions

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

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

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

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

6.3.2.1 Example posts

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

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

6.3.2.2 Question title

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

6.3.2.3 Explanation of the issue

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

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

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

6.3.2.4 Reproducible example

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

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

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

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

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

```
data()
```

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

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

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

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

6.4 Helping others

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

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

6.5 Summary

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

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

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

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

Part II

Coding Tools and Best Practices

7 R Scripts

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

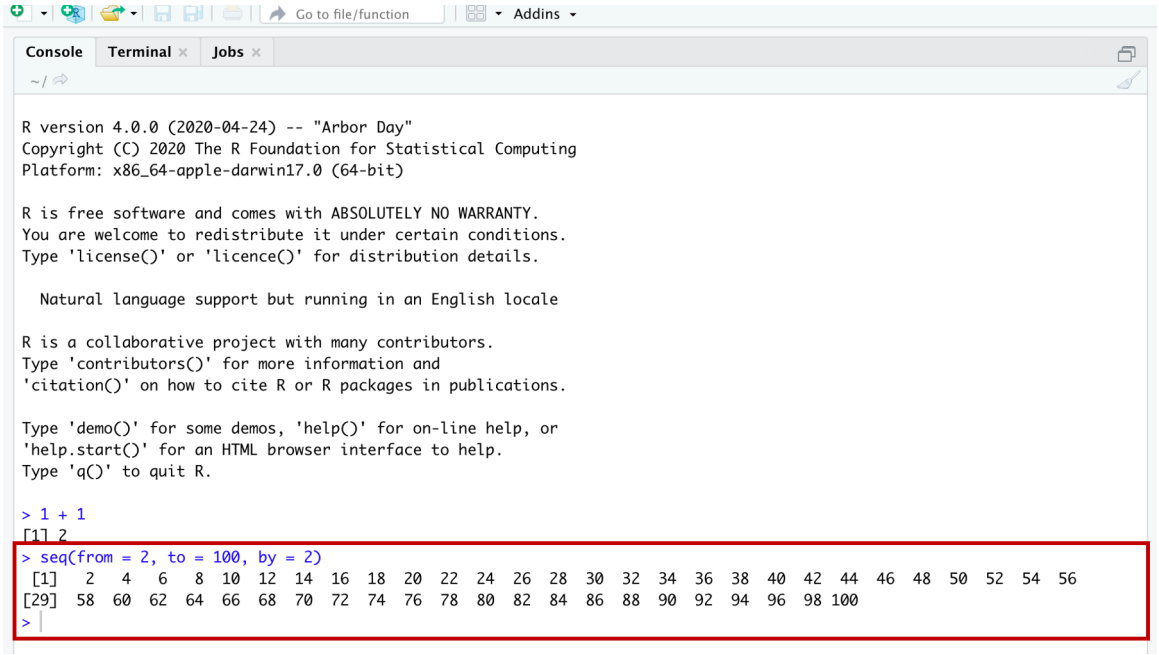


Figure 7.1: Submitting R code in the console.

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

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

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

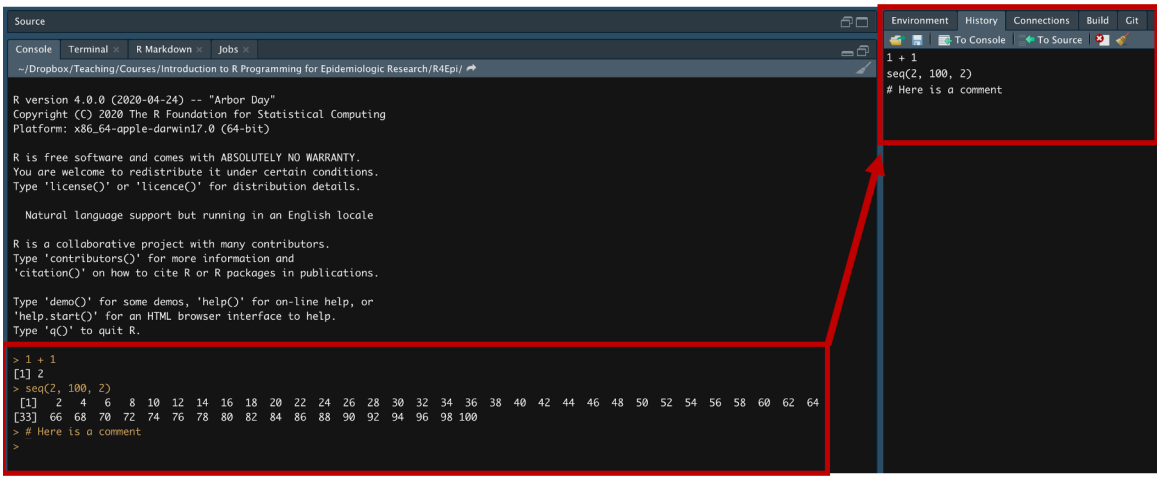


Figure 7.2: Console commands copied to the History pane.

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

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

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

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

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

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

4. Some people just prefer using R scripts.

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

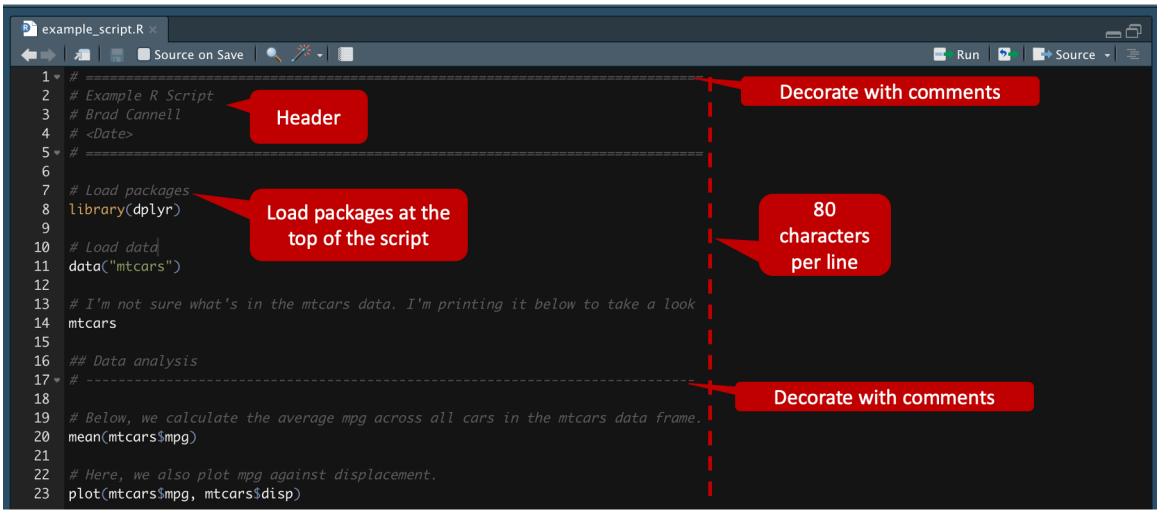


Figure 7.3: Example R script.

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).

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

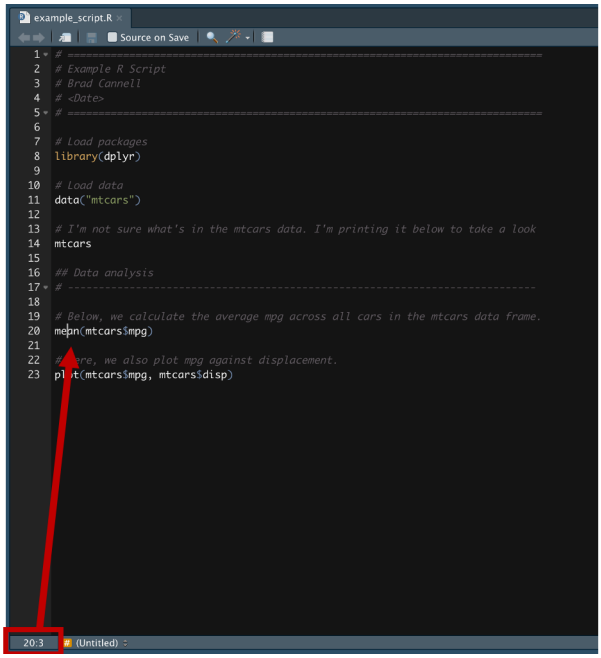


Figure 7.4: Cursor location.

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

7.1 Creating R scripts

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

Click the very first option – `R Script`.

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

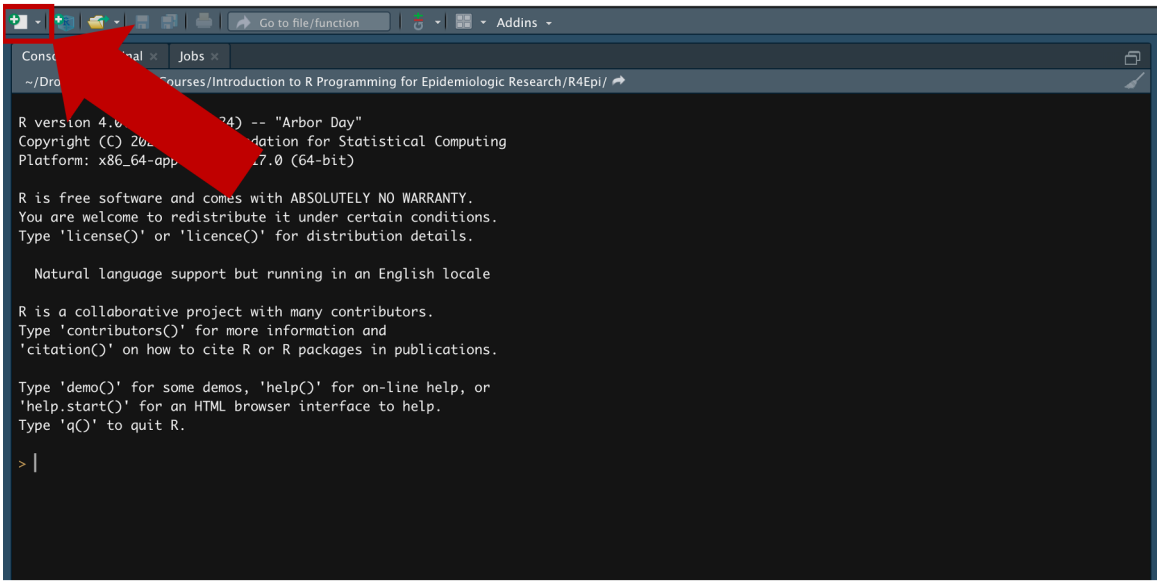


Figure 7.5: Click the new source file icon.

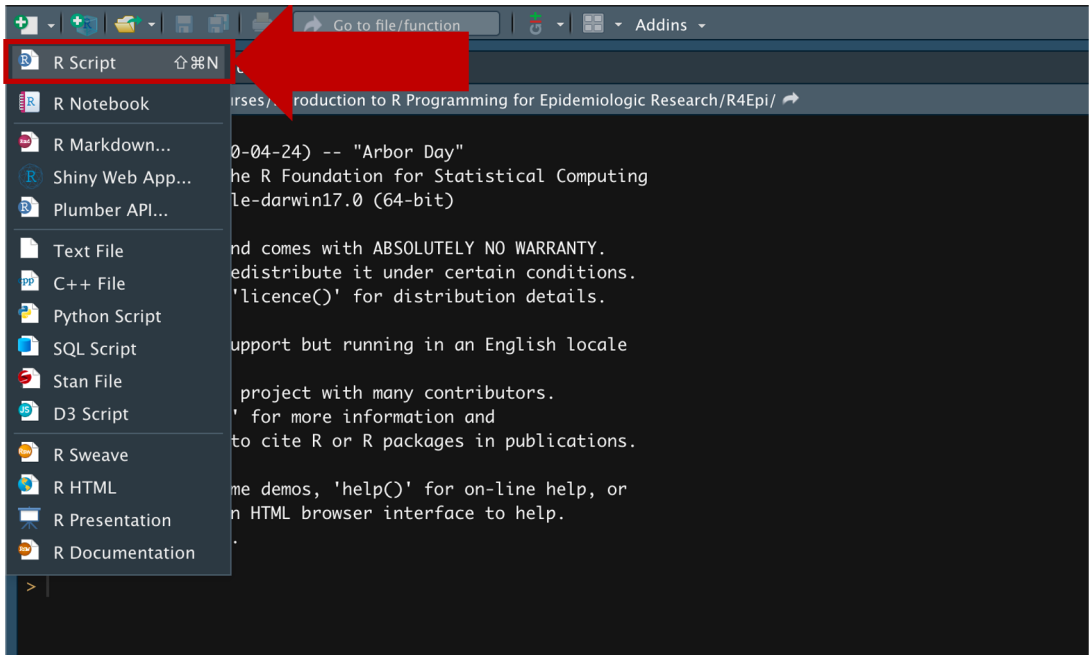


Figure 7.6: New source file options.

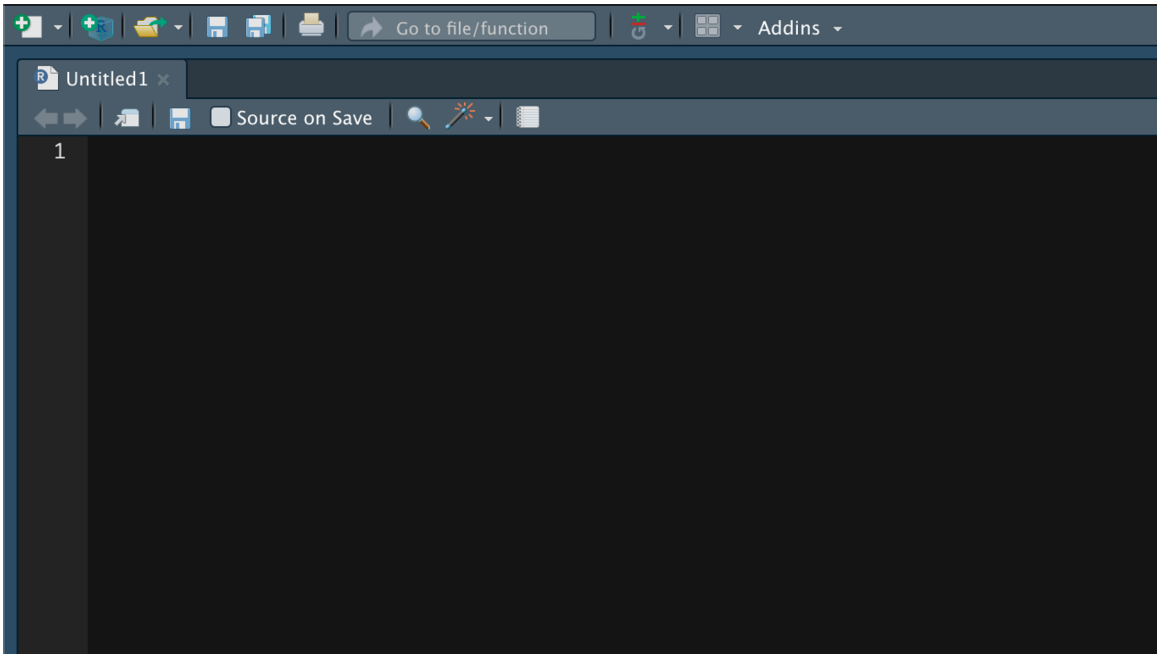


Figure 7.7: A blank R script in the source pane.

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

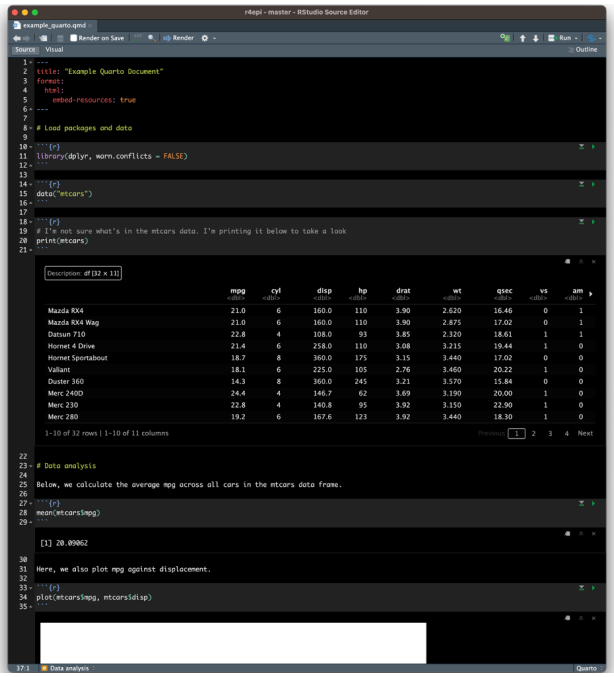
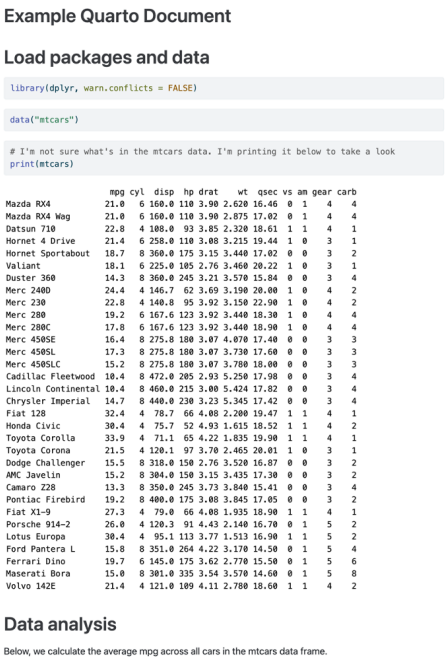


Figure 8.1: Example Quarto file.

Click here to download the Quarto file

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

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



Example Quarto Document

Load packages and data

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

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

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

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

Data analysis

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

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

Click here to download the rendered HTML file.

Notice how everything is nicely formatted and easy to read!

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

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

> **Warning**
>
> 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

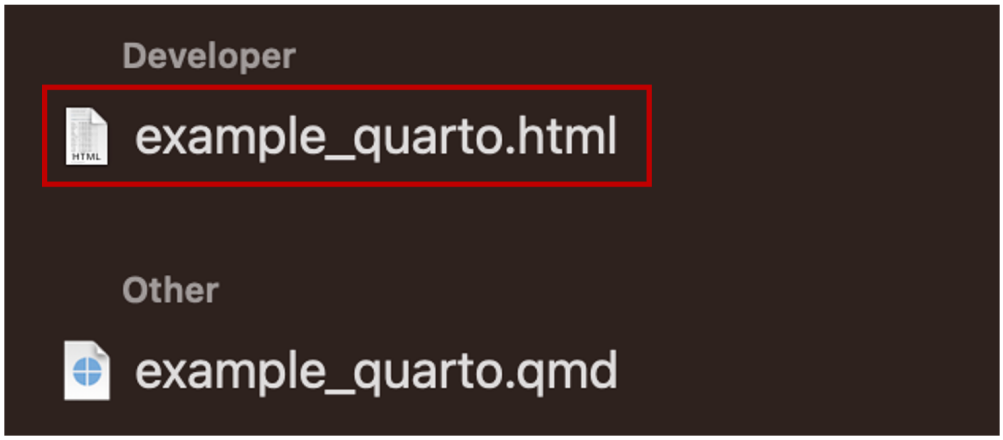


Figure 8.3: Quarto file and rendered HTML file and on MacOS.

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

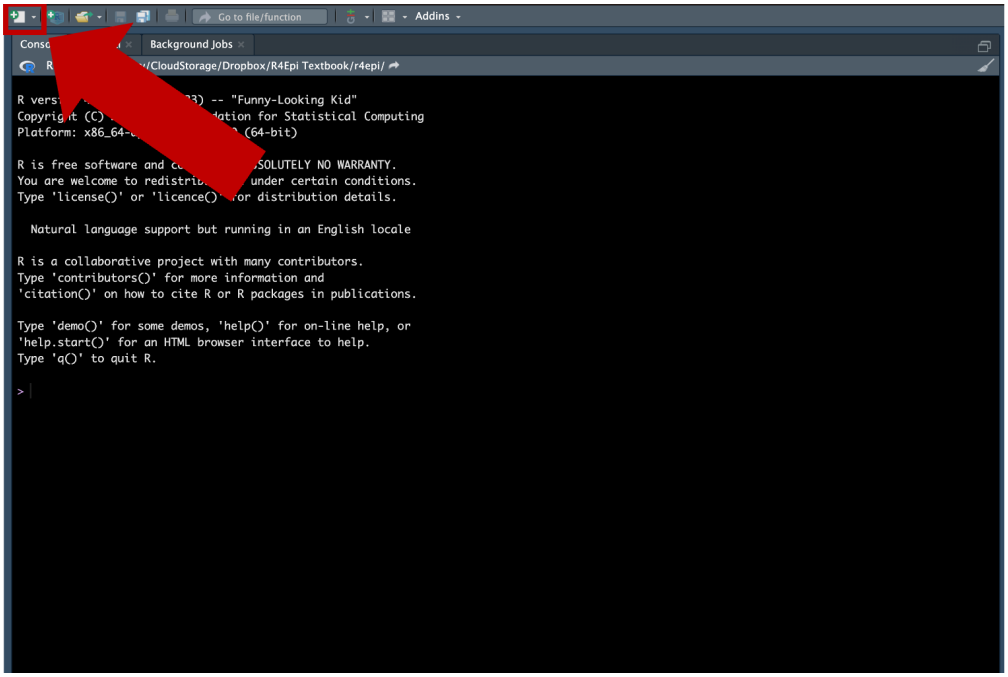


Figure 8.4: Click the new file icon.

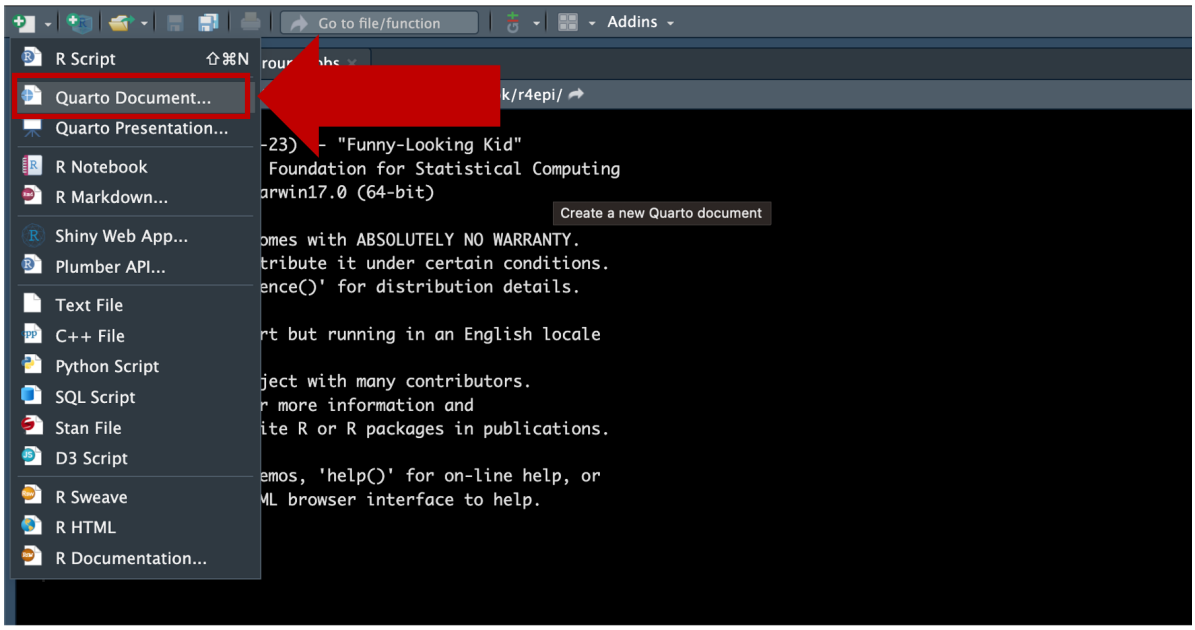


Figure 8.5: New source file options.

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

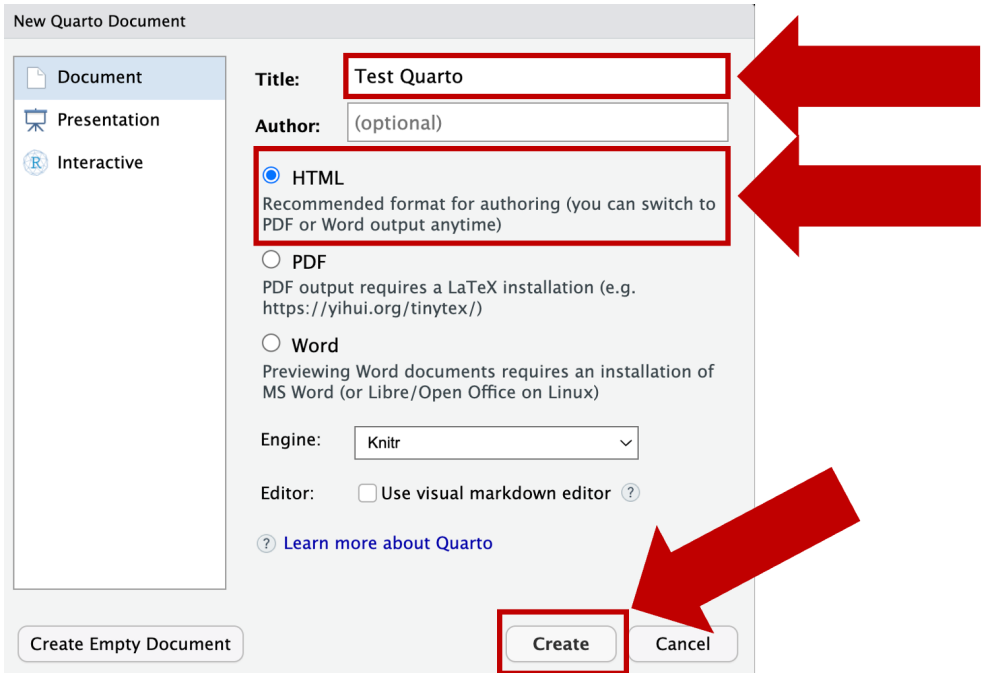


Figure 8.6: New Quarto document options.

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

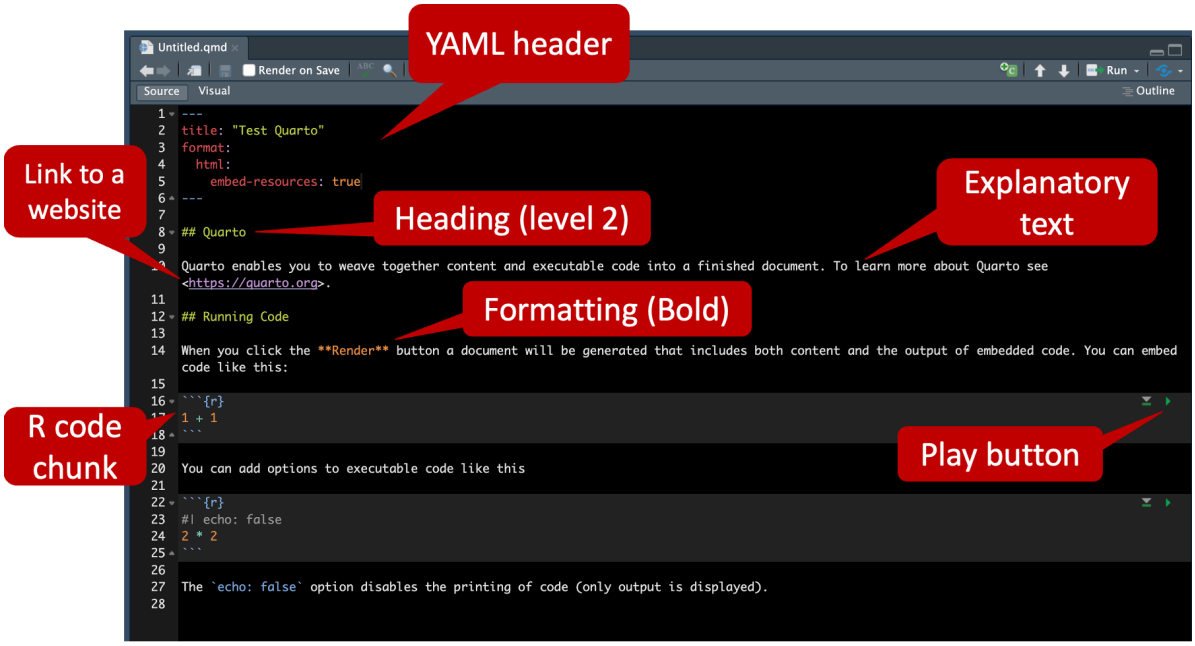


Figure 8.7: The 'Test Quarto' file in the RStudio source pane.

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

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

8.4 YAML headers

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

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

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

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

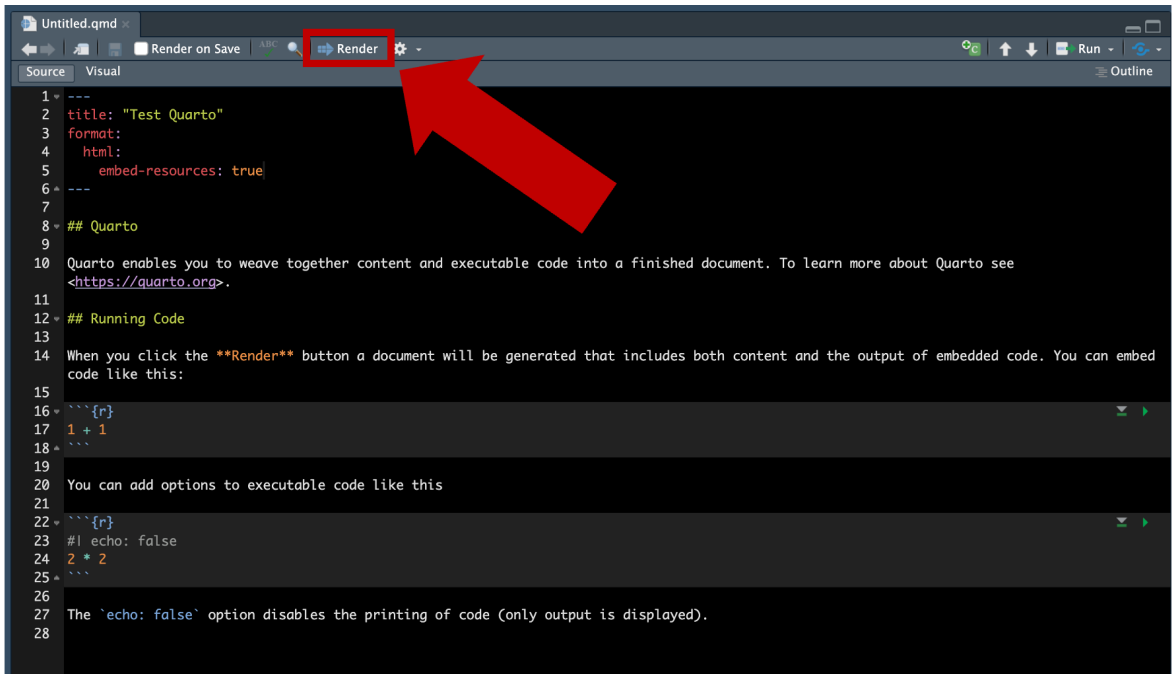


Figure 8.8: RStudio's render button. Only visible when a Quarto file is open.

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

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

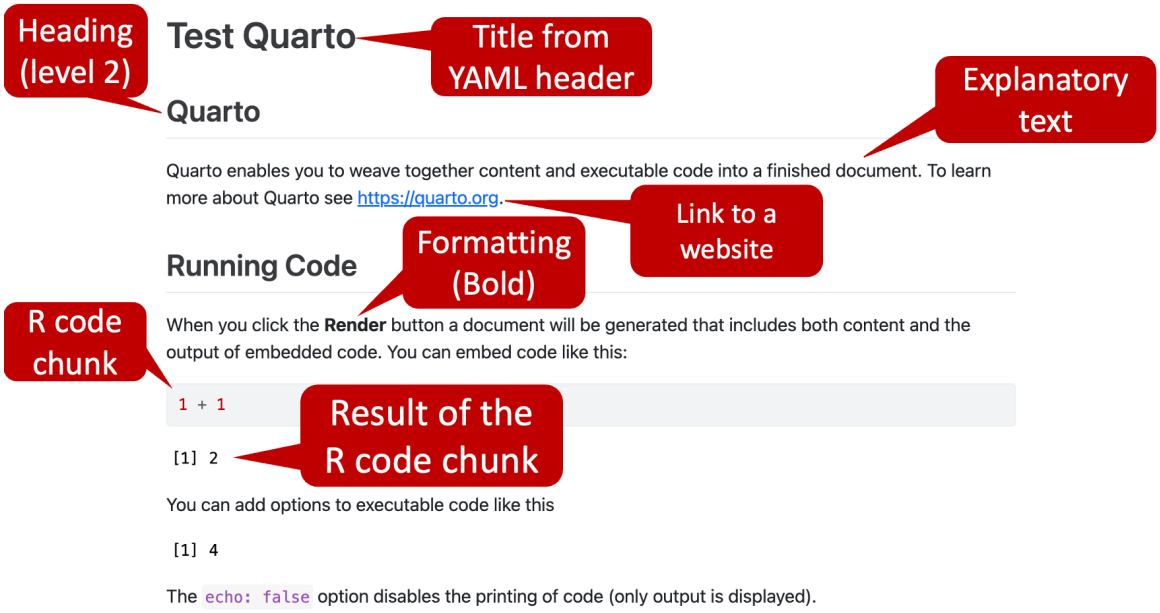


Figure 8.9: An HTML file created using a Quarto file.

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

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

8.5 R code chunks

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

On MacOS type `option + command + i`.

On Windows type `control + alt + i`

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

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

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

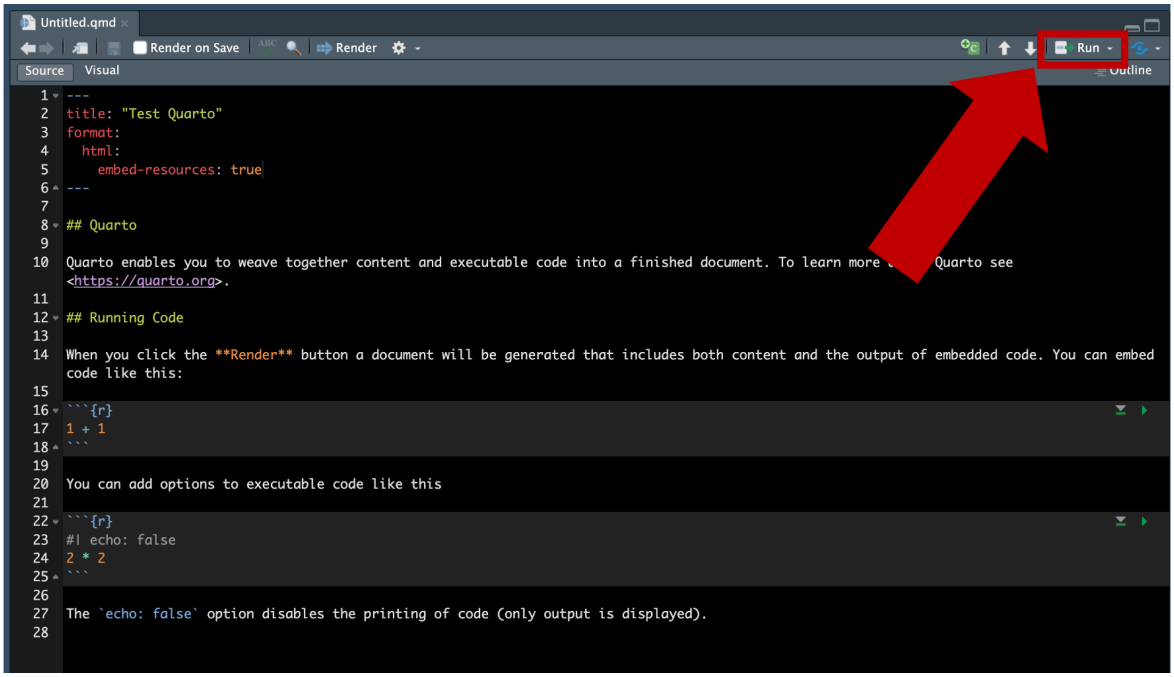


Figure 8.10: The run button in RStudio.

8.6 Markdown

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

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

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

8.6.1 Markdown headings

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

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

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

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

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

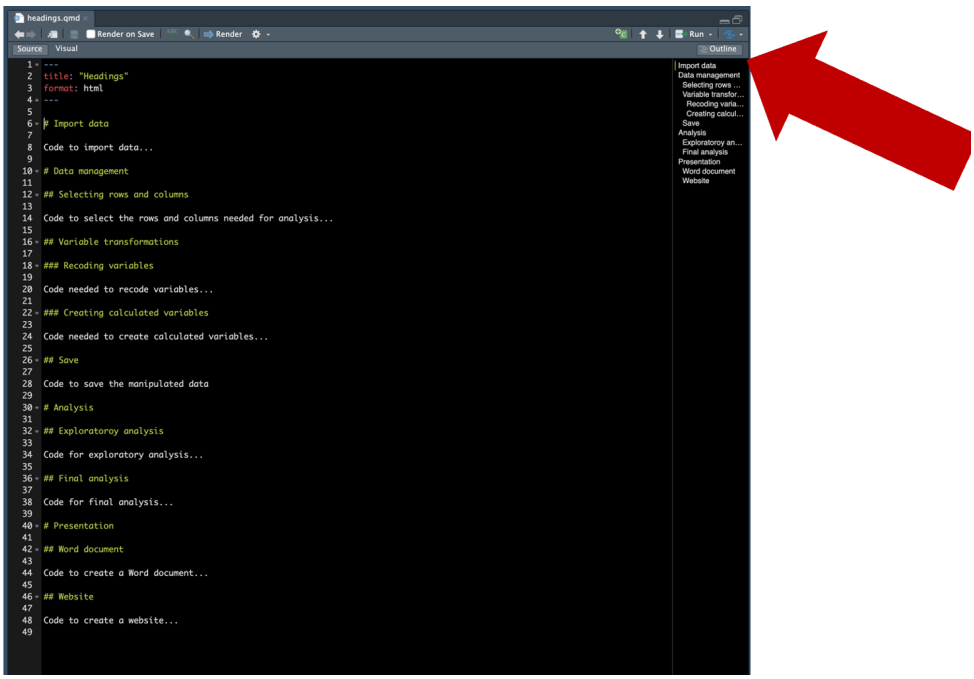


Figure 8.11: A Quarto file with nested headings.

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

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

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

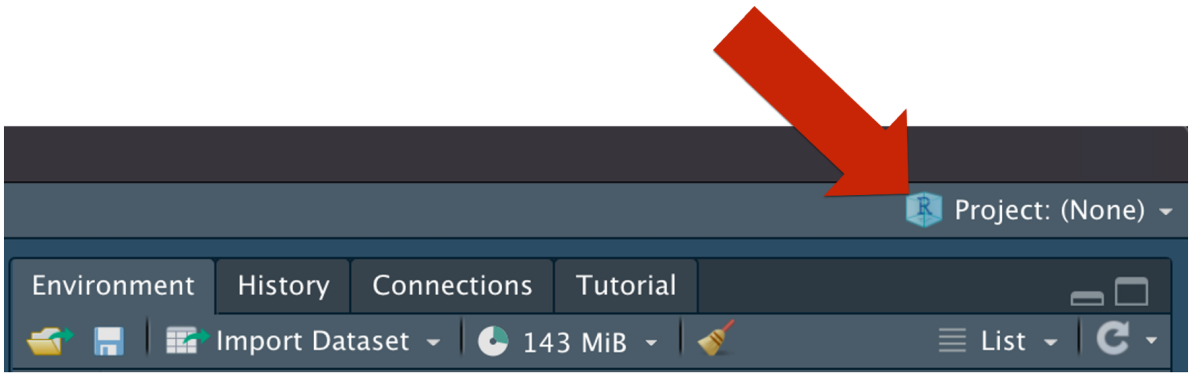
8.7 Summary

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

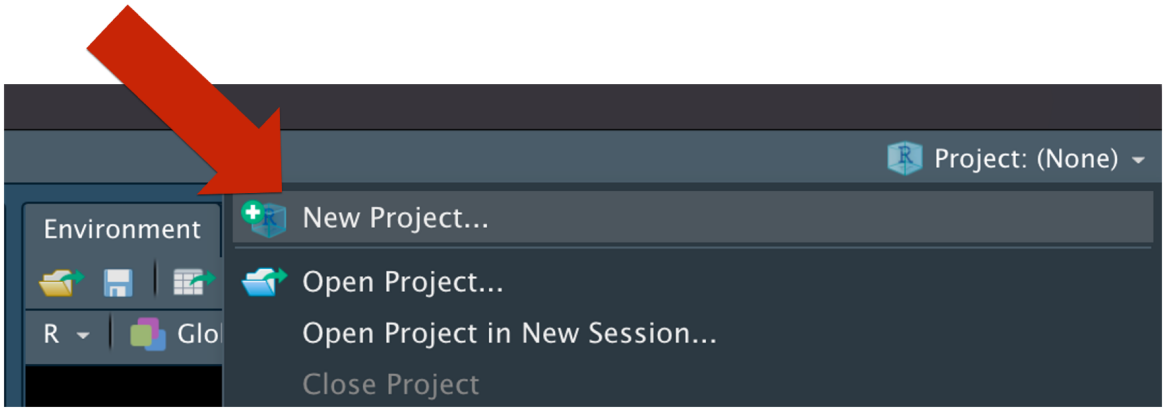
9 R Projects

In previous chapters of this book, we learned how to use R Scripts and 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.

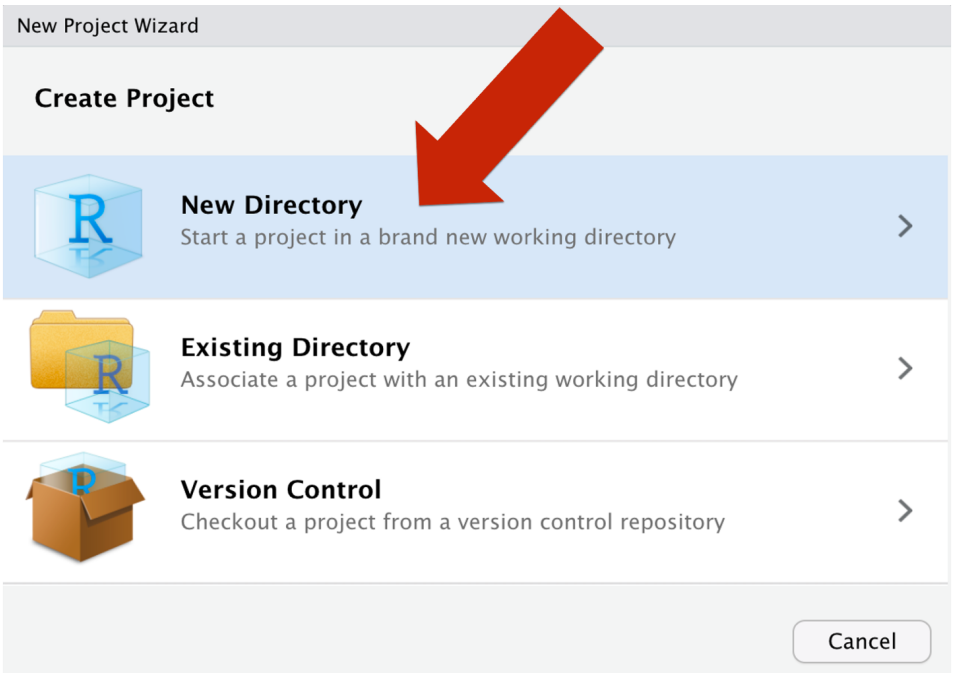
RStudio makes creating R projects really simple. For starters, let's take a look at the top right corner of our RStudio application window. Currently, we see an R project icon that looks like little blue 3-dimensional box with an "R" in the middle. To the right of the R project icon, we see words `Project: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



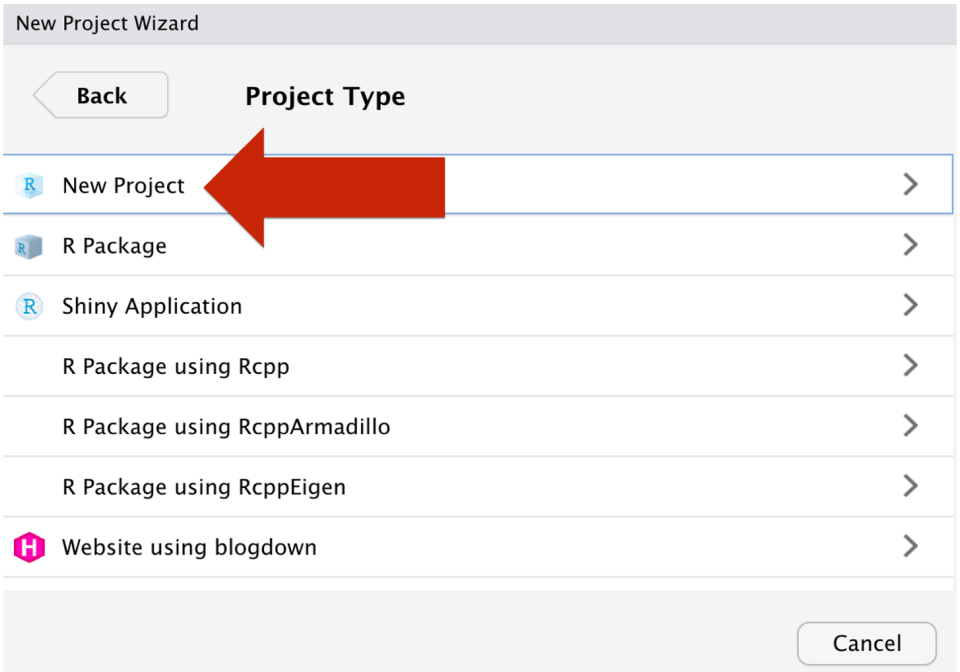
To create a new R project, we just need to click the drop-down arrow next to the words `Project: (None)` to open the projects menu. Then, we will click the `New Project...` option.



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.

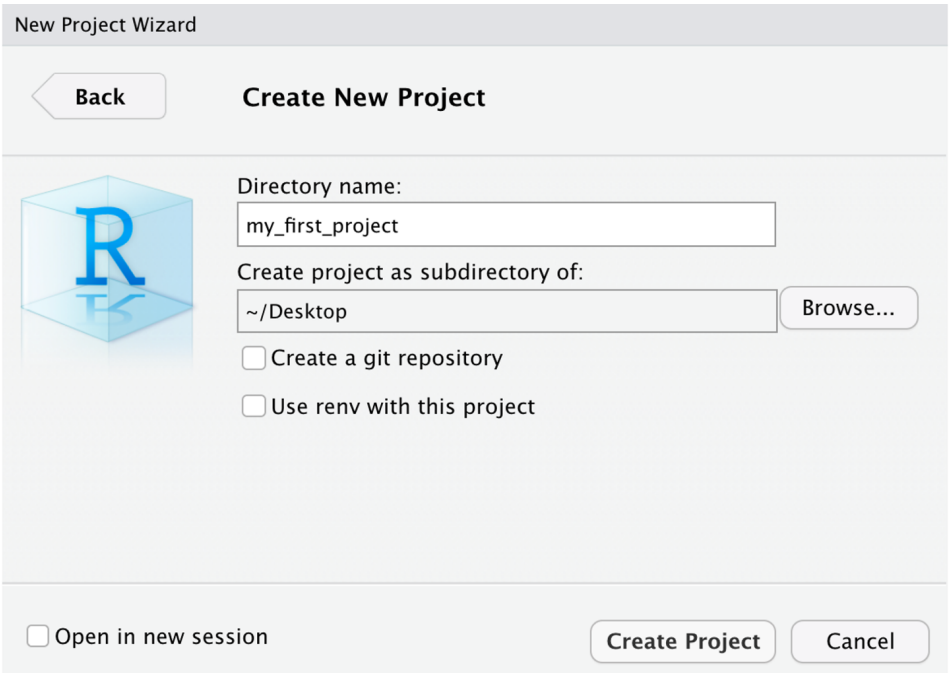


In the next window, we will have to make some choices and enter some information. The fist thing we will have to do is name our project. We do so by entering a value in the `Directory name:` box. Often, we can name our R project directory to match the name of the larger project

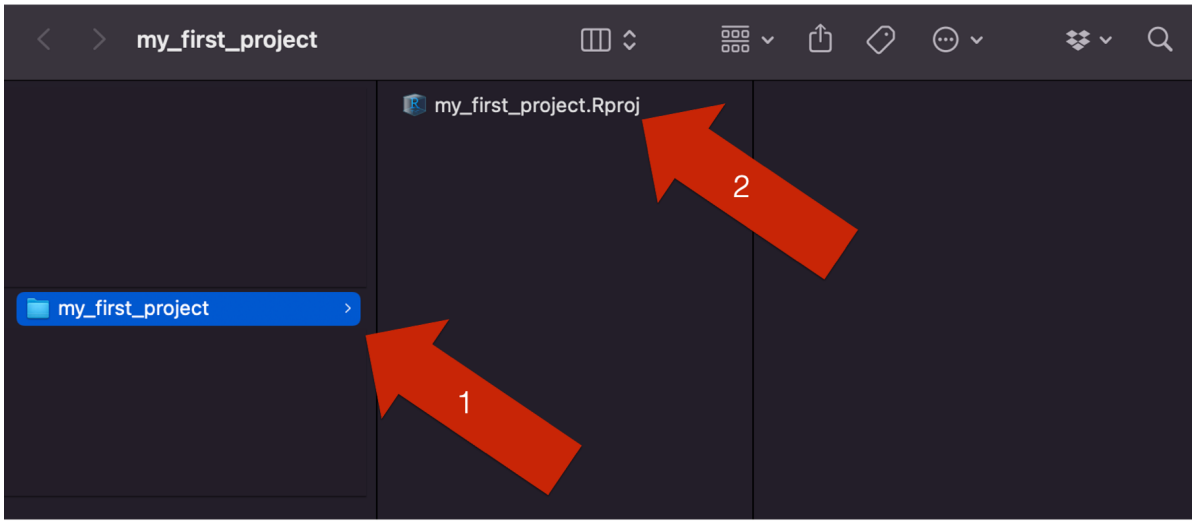
we are working on in a pretty natural way. If not, the name we choose for our project directory should essentially follow the same guidelines that we use for object (variable) names, which we will learn about soon. In this example, we went with the very creative `my_first_project` project name.

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

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



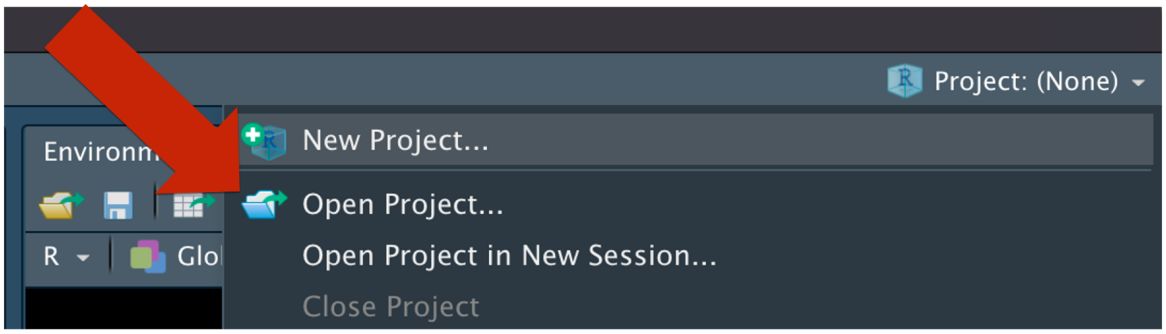
Doing so will create our new R project in the location we selected in the `Create project as subdirectory of:` text box in the new project wizard. In the screenshot below, we can see that a folder was created on our computer's desktop called `my_first_project`. Additionally, there is one file inside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



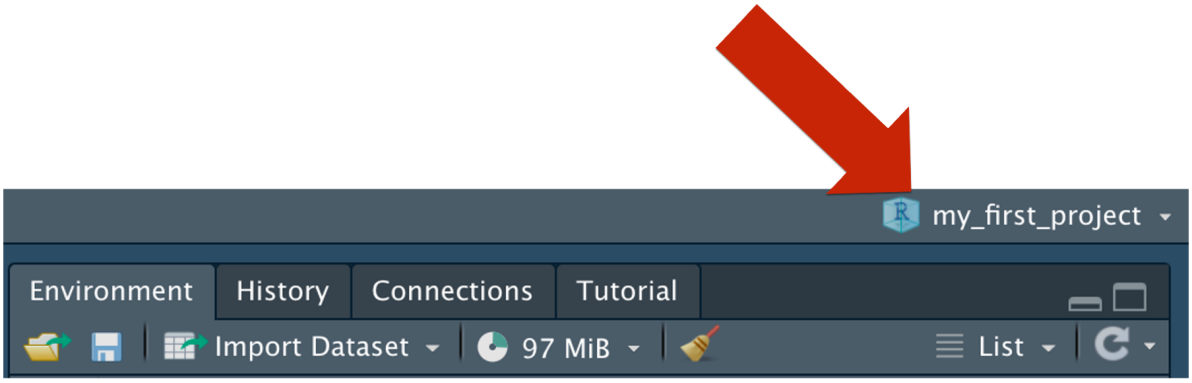
This file is called an R project file. Every time we create an R project, RStudio will create an R project file and add it to our project directory (i.e., the folder) for us. This file helps RStudio track and organize our R project.

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

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



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



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

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

10 Coding Best Practices

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

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

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

10.1 General principles

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

10.2 Code comments

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

10.2.1 Defining key variables

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

```
* **Case number definition**

    - Case / investigation number.

* **Intake stage definition**

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

* **Intake start definition**

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

10.2.2 What this code is trying to accomplish

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

```
## Standardize character strings

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

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

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

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

rm(vars)
```

10.2.3 Why we chose this particular strategy

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

```
### Create exact match dummy variables

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

10.3 Style guidelines

UsInG c_o_n_s_i_s_t_e_n_t STYLE i.s. import-ant!

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

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

10.3.1 Comments

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

10.3.2 Object (variable) names

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

10.3.3 Use names that are informative

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

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

# Informative names
employed
married
education
```

10.3.3.1 Use names that are concise

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

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

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

10.3.3.2 Use all lowercase letters

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

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

# Camel case - Don't use
myStudyData

# All lowercase - Use
my_study_data
```

10.3.3.3 Separate multiple words with underscores.

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

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

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

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

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

# Separate with underscores - Use
my_cancer_data
```

10.3.3.4 Prefix the names of similar variables

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

```
# Don't use
first_name
last_name
middle_name

# Use
name_first
name_last
name_middle

# Don't use
street
city
state

# Use
address_street
address_city
address_state
```

10.3.4 File Names

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

10.3.4.1 Managing multiple files in projects

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

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

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

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

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

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

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

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

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

11 Using Pipes

11.1 What are pipes?

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

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

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

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

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

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

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

```
[1] 3.662703
```

Here's what we did above:

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

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

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

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

Next, let's try nesting these functions instead:

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

```
[1] 3.662703
```

Here's what we did above:

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

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

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

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

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

```
[1] 3.662703
```

Here's what we did above:

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

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

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

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

# Nest functions with pipes
```

```r
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.

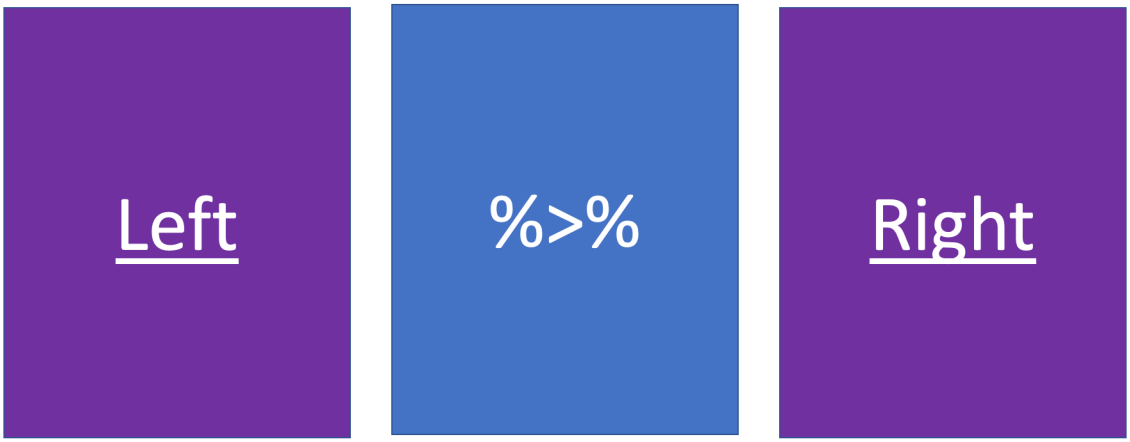


Figure 11.1: Pipes have a left side and a right side.

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.

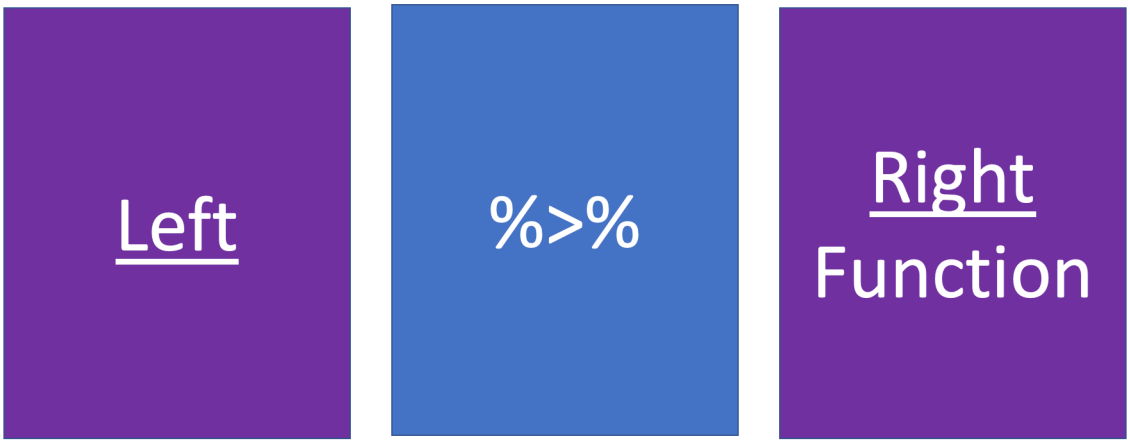


Figure 11.2: A function should always be to the right of the pipe operator.

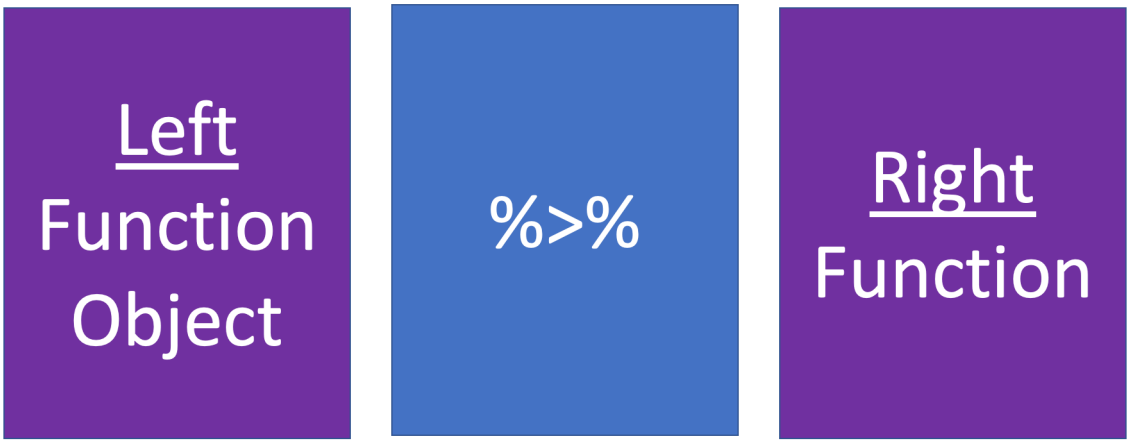


Figure 11.3: A function or an object can be to the left of the pipe operator.

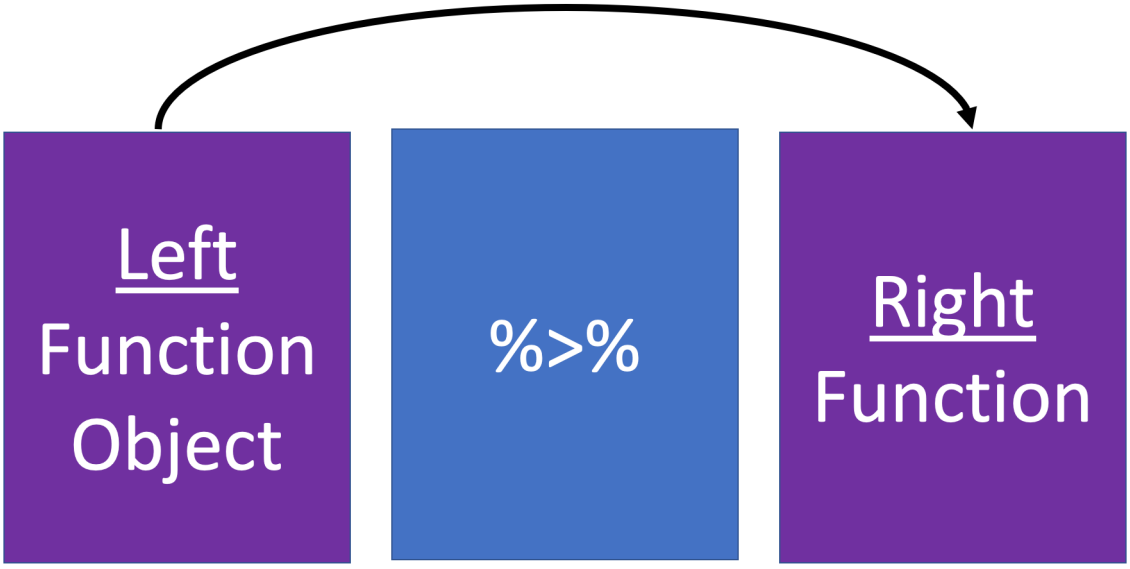


Figure 11.4: Pipe the left side to the first argument of the function on the right side.

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.

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**
>
> 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(                                     # Line 3
    mean_ht = mean(ht_in),                       # Line 4
    sd_ht   = sd(ht_in)                          # Line 5
  ) |>                                           # Line 6
  print()                                        # Line 7
```

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

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

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

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

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

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

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

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

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

11.3 Final thought on pipes

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

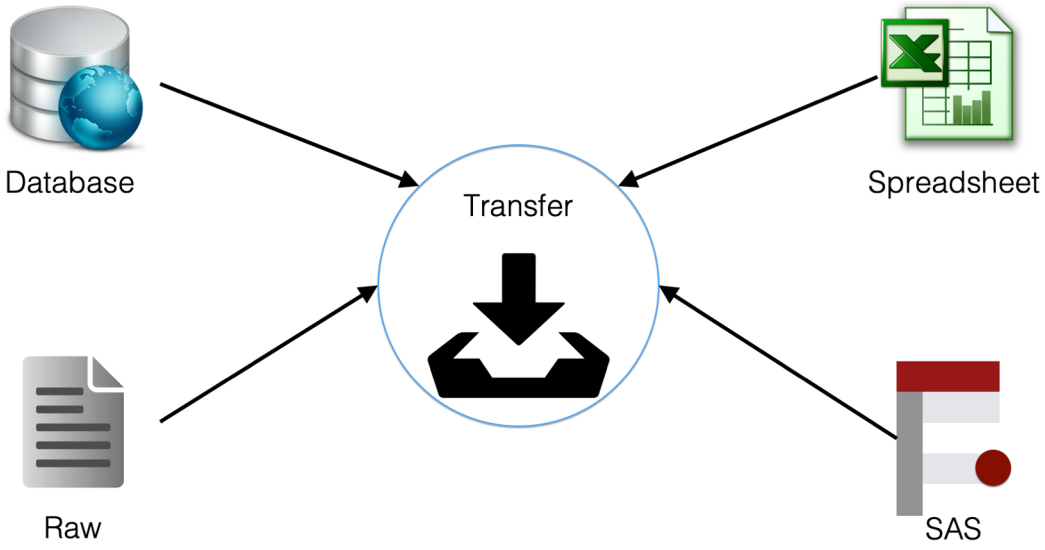
Part III

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.

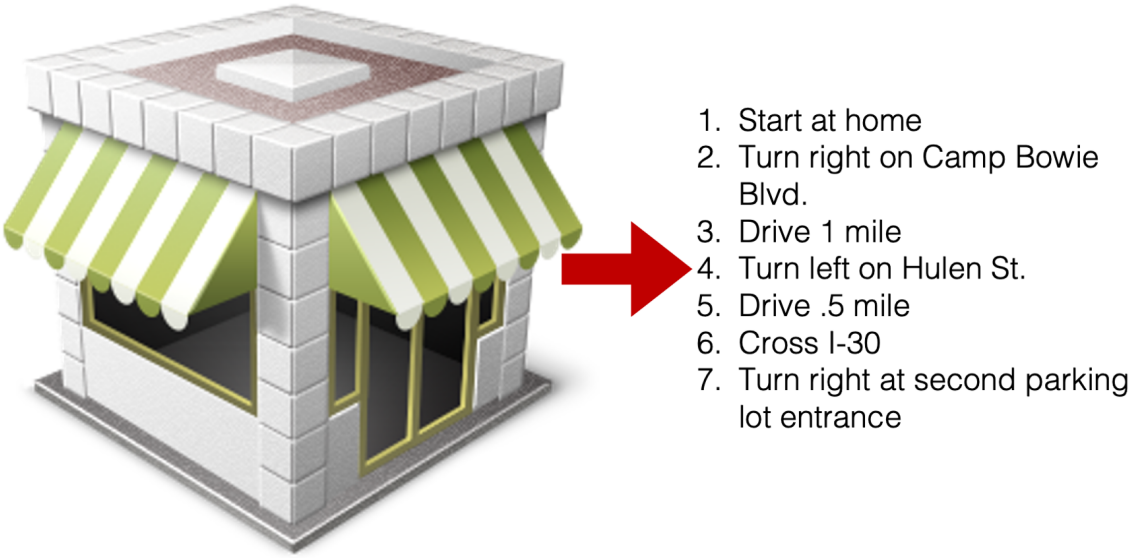
In this part of the book, we will learn to **import** data stored in various different file types into R for data management and analysis, we will learn to store R data frames in a more permanent way so that we can come back later to modify or analyze them, and we will learn to **export** data so that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

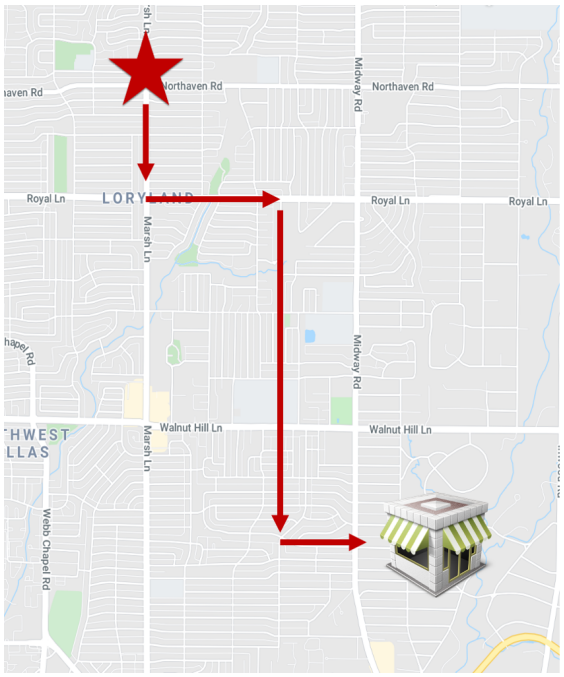
Let's say that we want you to go to the store and buy a loaf of bread.

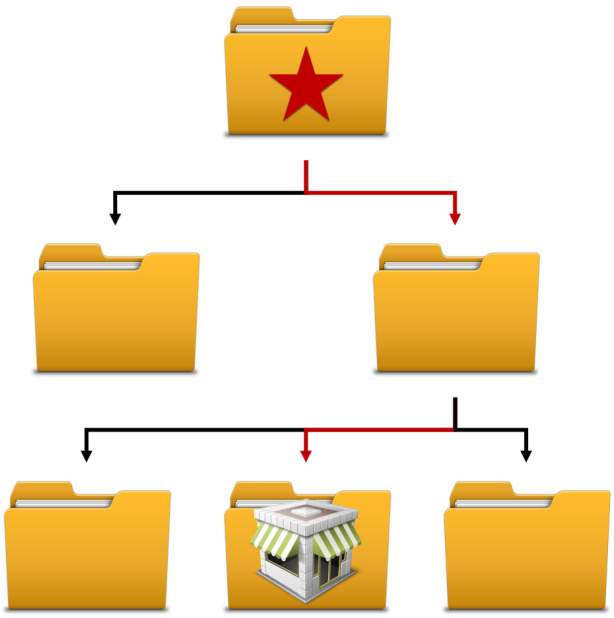
When we say, "go to the store", this is really a shorthand way of telling you a much more detailed set of directions.



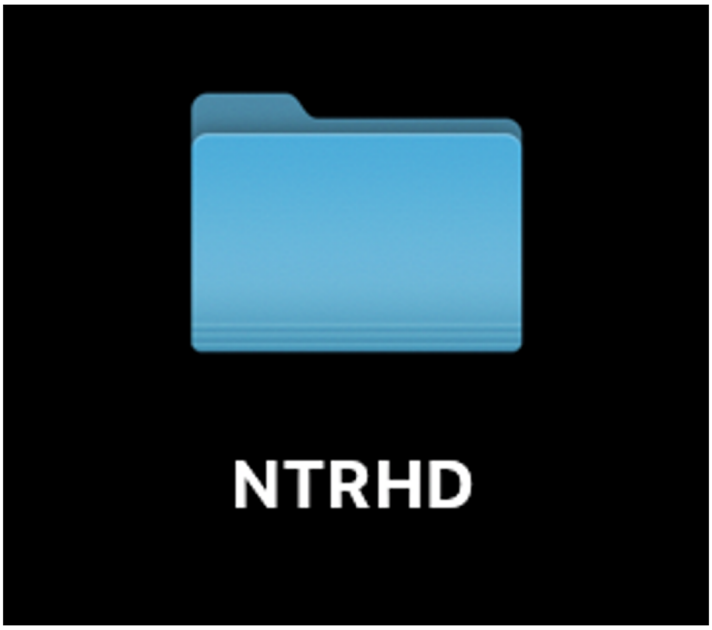
Not only do you need to do *all* of the steps in the directions above, but you also need to use the *exact sequence* above in order to arrive at the desired destination.

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.

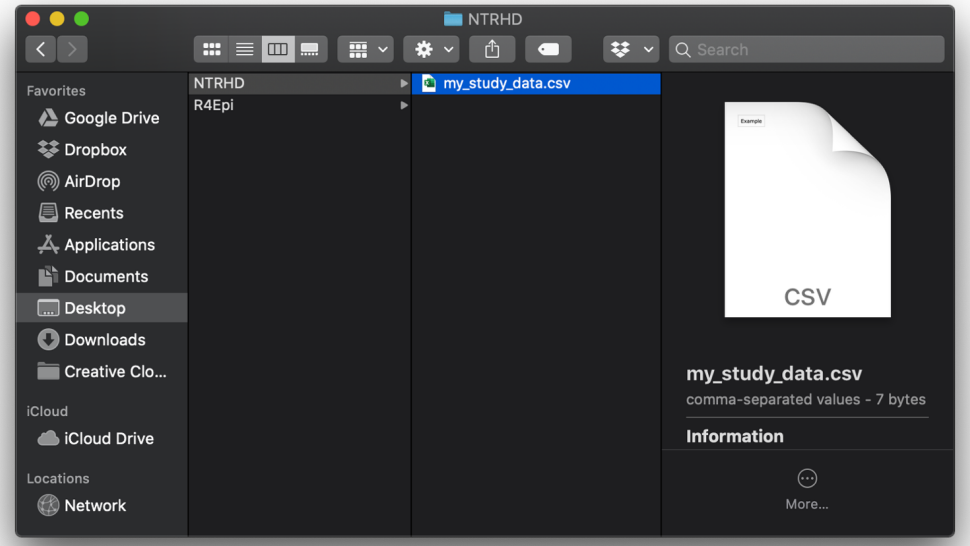




For example, let's say that we have a folder on our desktop called "NTRHD" for "North Texas Regional Health Department.



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 in file paths (`/`) 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)

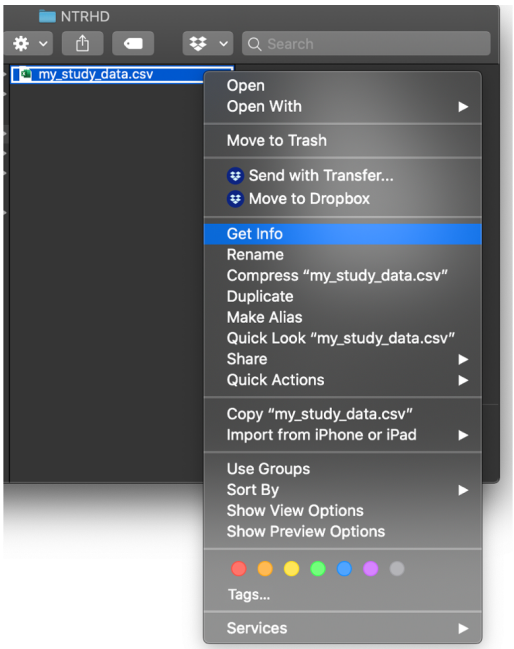
OR

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

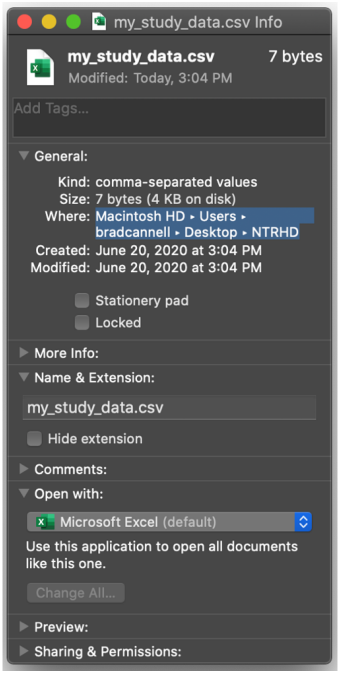
13.1 Finding file paths

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`

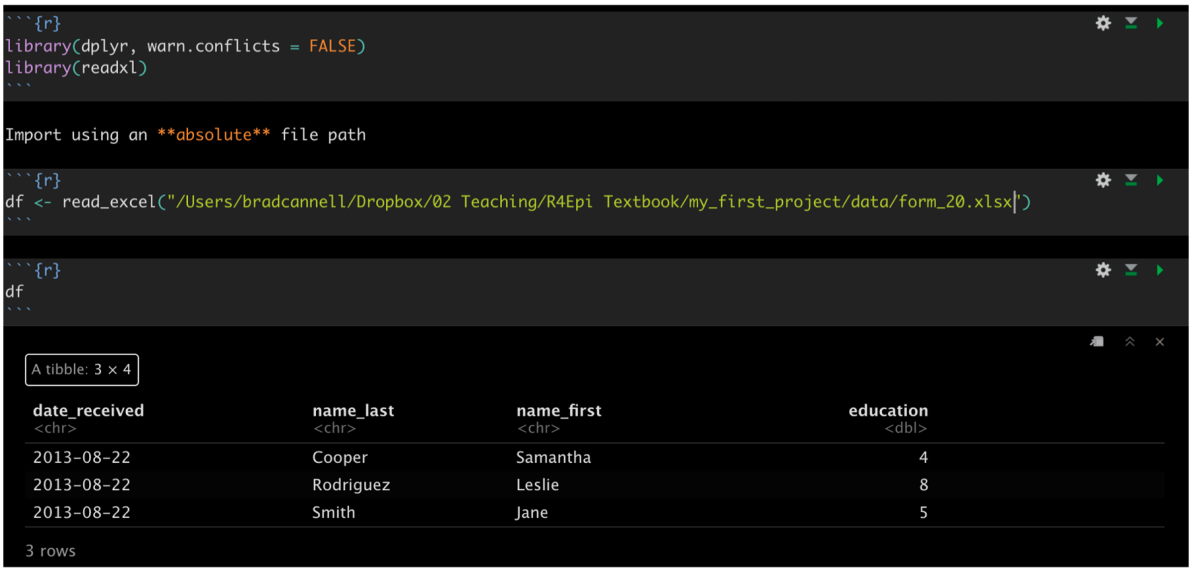
menu option changes to `Copy ... as Pathname`. We can then left-click that option to copy the path and paste it into our R code.

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

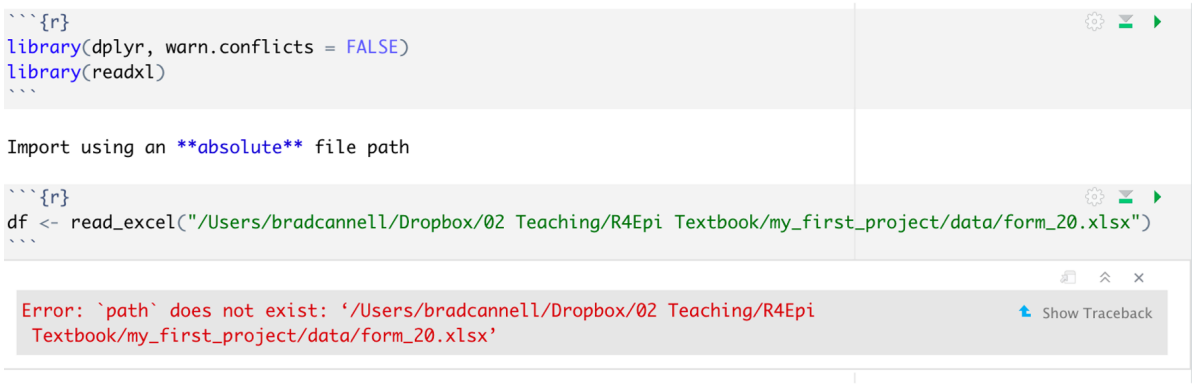
date_received <chr>	name_last <chr>	name_first <chr>	education <dbl>
2013-08-22	Cooper	Samantha	4
2013-08-22	Rodriguez	Leslie	8
2013-08-22	Smith	Jane	5

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?

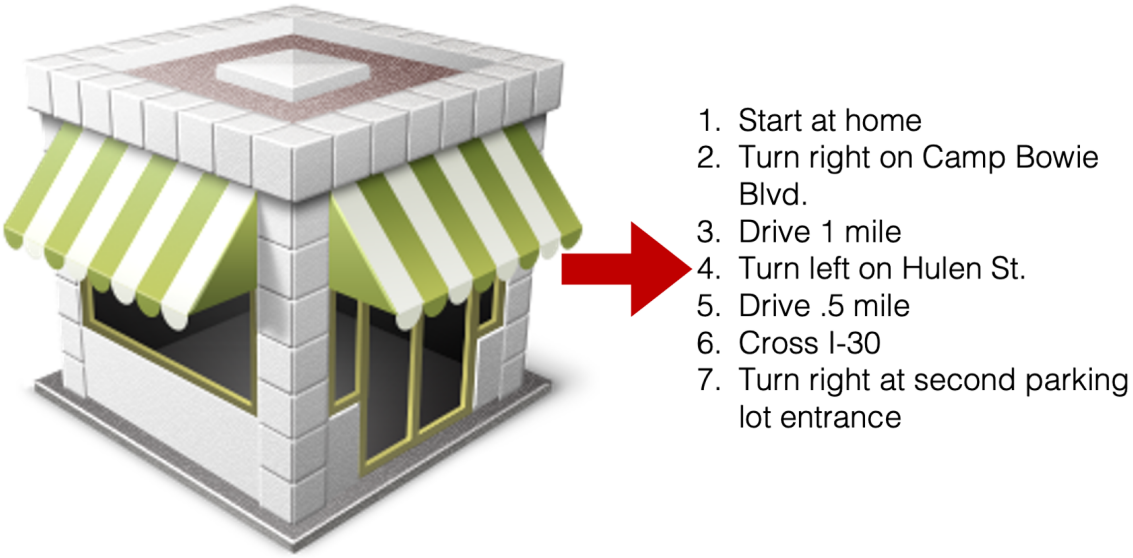
Yes! Our code does work. We can tell because there are no errors on the screen and the `df` object we created looks as we expect it to when we print it to the screen. Great!!

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?



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/arthurepi` 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.

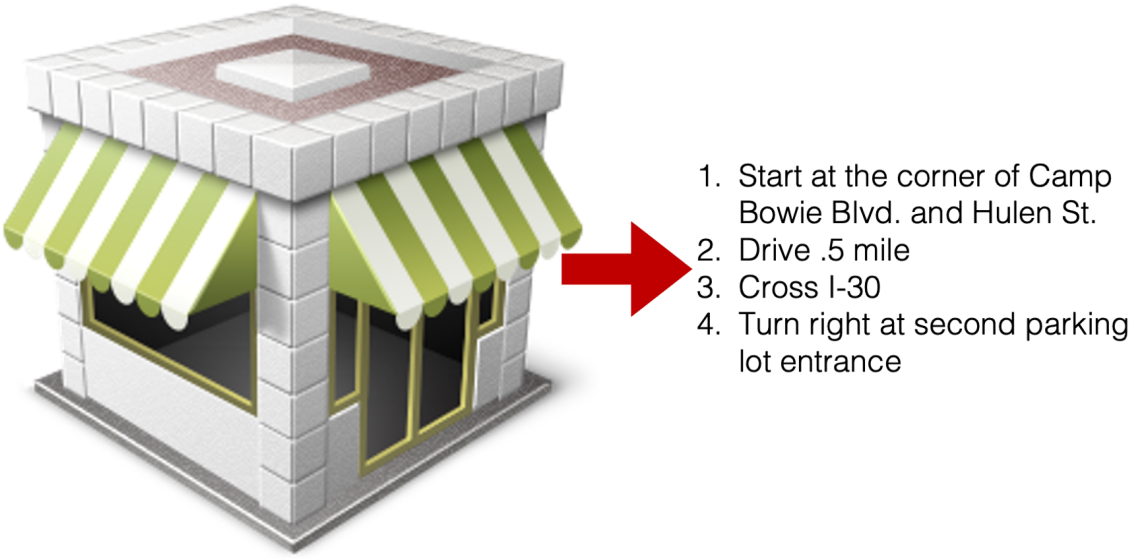


Notice that these directions assume that we are starting from our house. As long as we leave from our house, they work great! But what happens if we are at someone else's house and we ask you to go to the store and buy a loaf of bread? You'd walk out the front door and immediately discover that the directions 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?

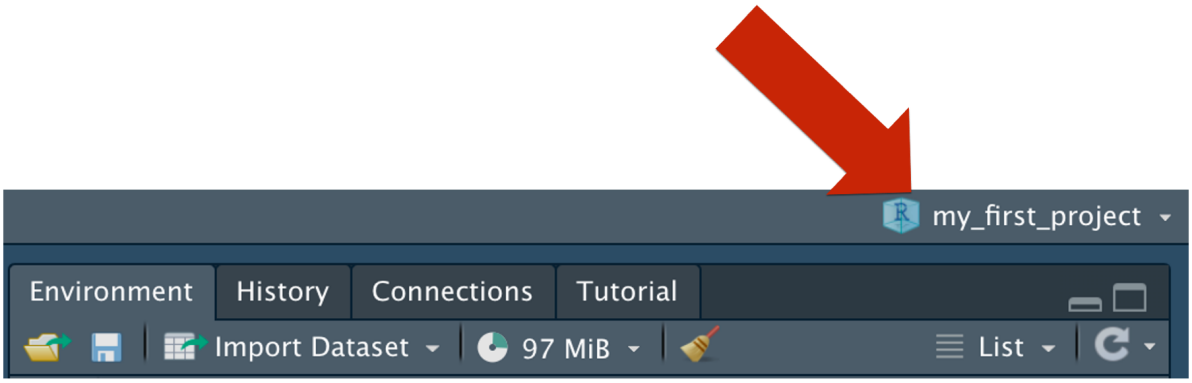
Yes! A better solution is to use a **relative file path**. Returning to our *directions to the store* example, it would be like giving directions to the store from a common starting point that everyone knows.



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?

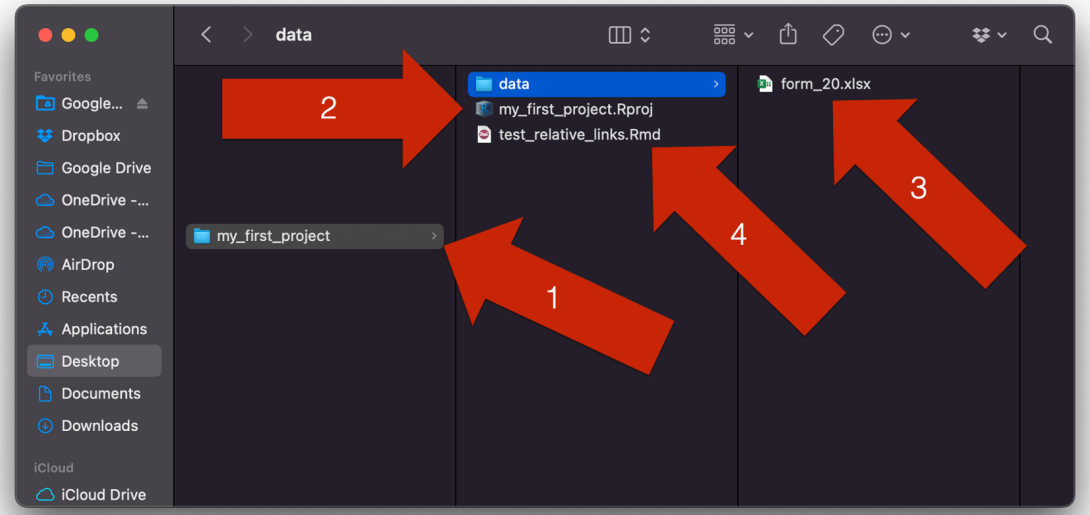
The answer is our R project's directory (i.e., folder)! In order to effectively use relative file paths in R, we start by creating an R project. If you don't remember how to create R projects, this would be a good time to go back and review Chapter 9.

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 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
```

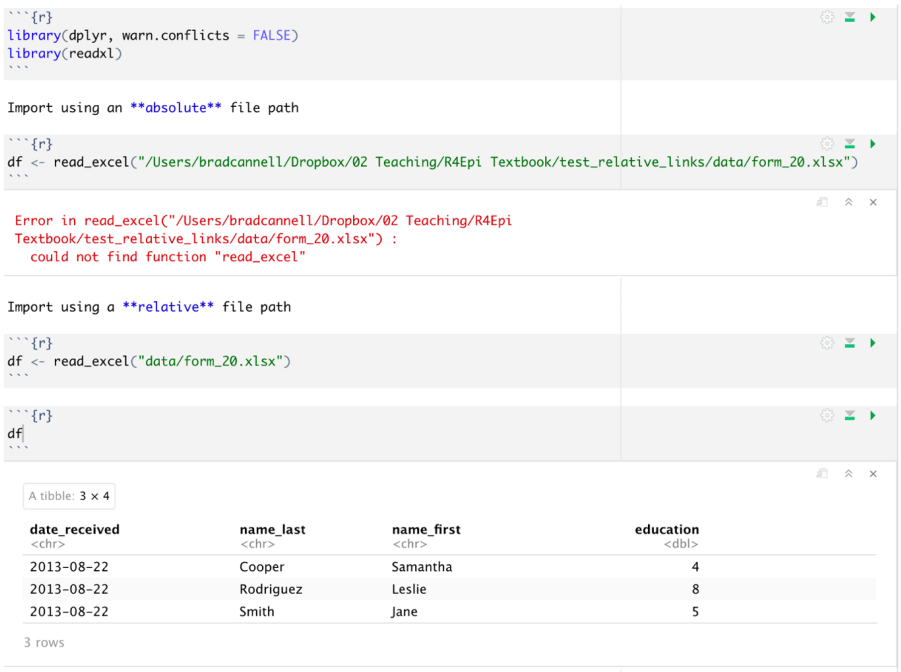
A tibble: 3 × 4

date_received <chr>	name_last <chr>	name_first <chr>	education <dbl>
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 <chr>	name_last <chr>	name_first <chr>	education <dbl>
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

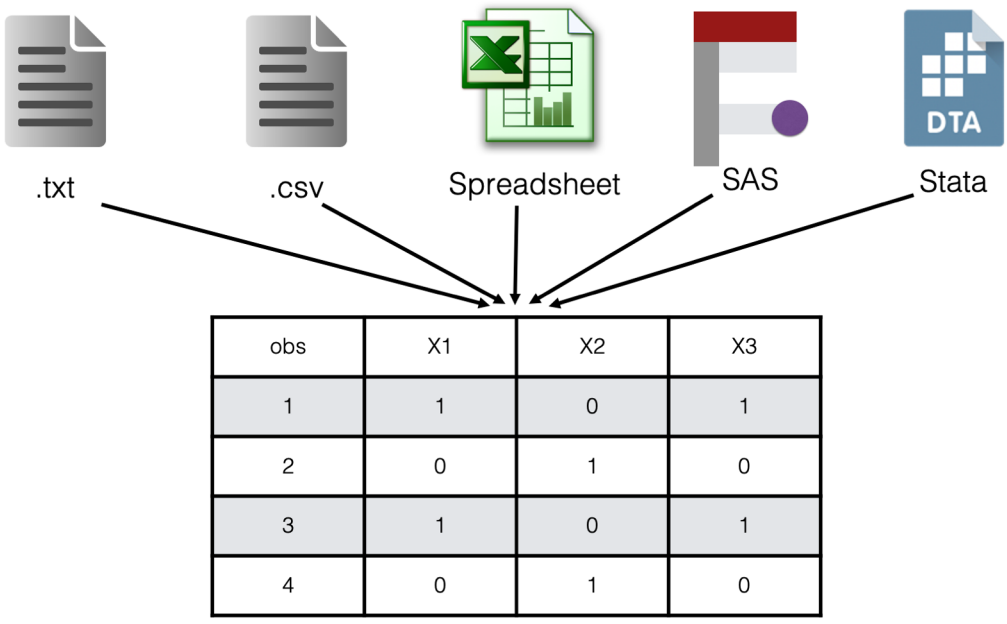
We previously learned how to manually create a data frame in RStudio with the `data.frame()` function, the `tibble()` function, or the `tribble()` function. This will get the job done, but it's not always very practical – particularly when you have larger data sets.

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 or TextEdit 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.



obs	X1	X2	X3
1	1	0	1
2	0	1	0
3	1	0	1
4	0	1	0

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.

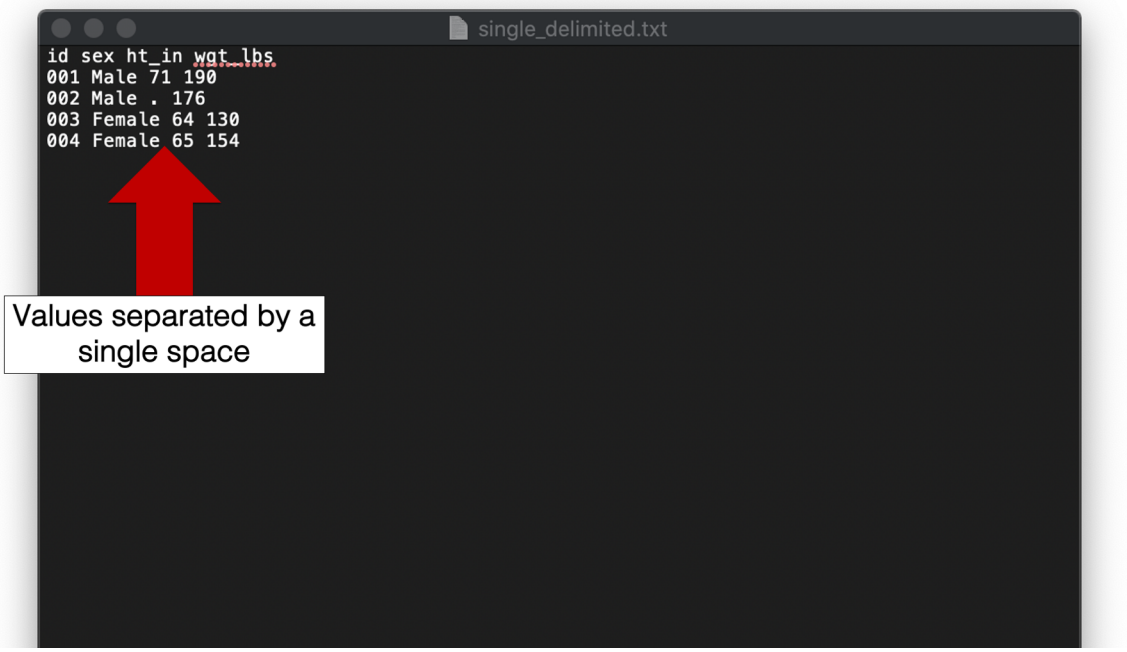
```
library(readr)
```

14.2 Importing space delimited files

We will start by importing data with values are 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.

> **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:

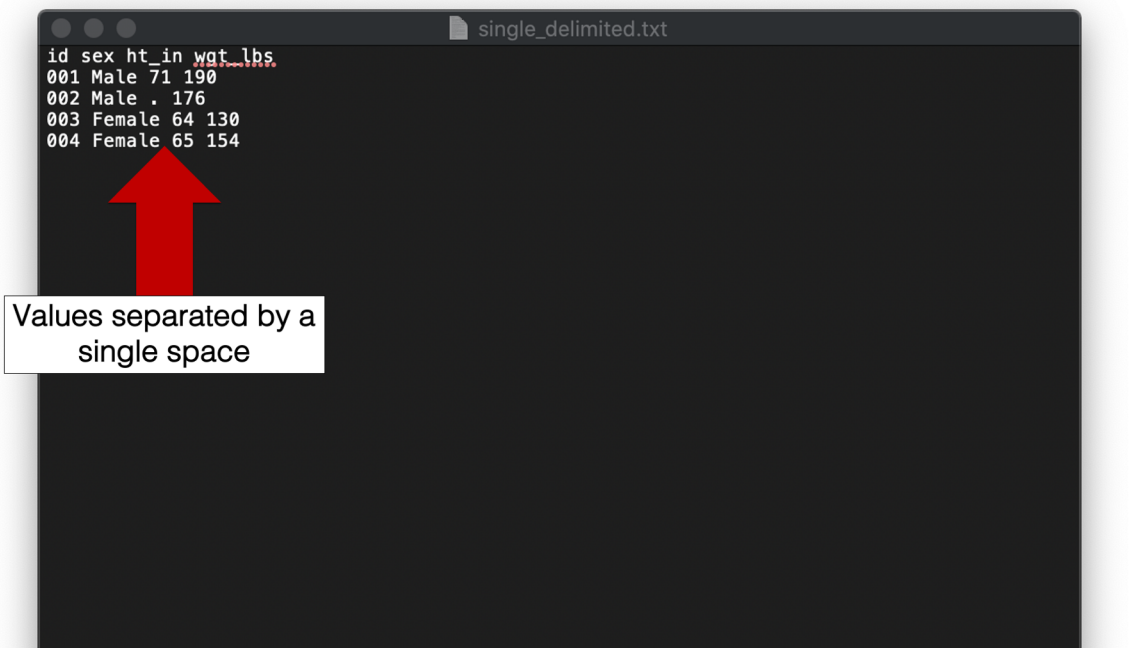
- We used `readr`'s `read_delim()` function to import a data set with values that are delimited by a single space. Those values were imported as a data frame, and we assigned that data frame to the R object called `single_space`.
- You can type `?read_delim` into your R console to view the help documentation 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 filed 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?



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
```

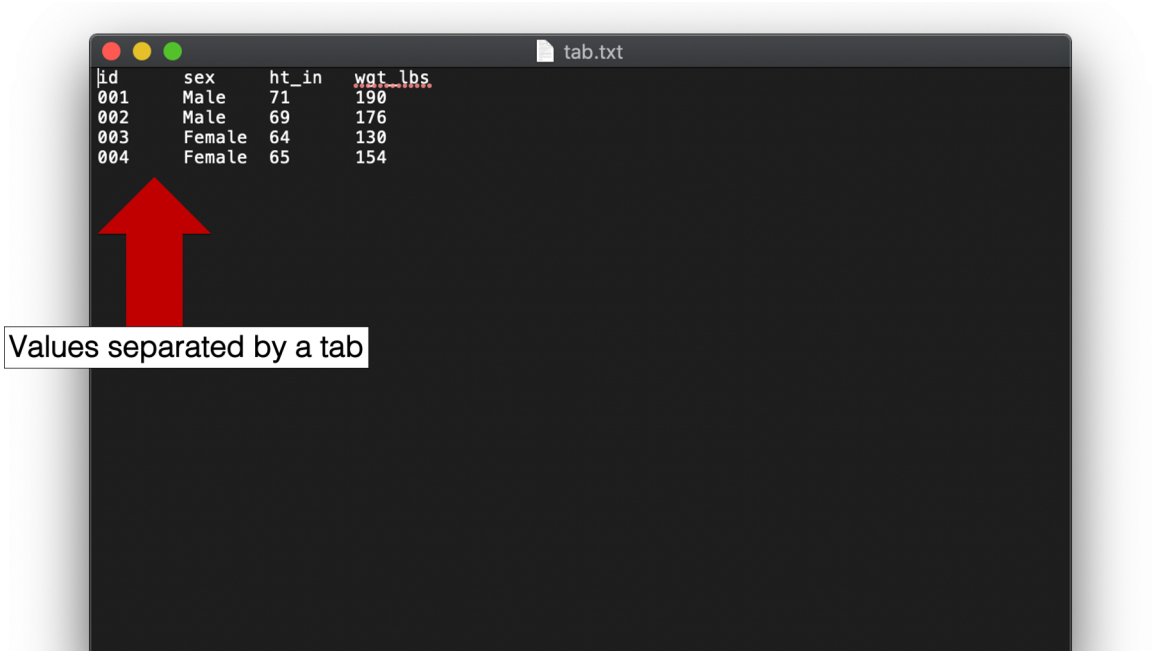
```
# A tibble: 4 x 4
  id    sex    ht_in wgt_lbs
  <chr> <chr>  <dbl>   <dbl>
1 001   Male      71     190
2 002   Male      NA     176
3 003   Female    64     130
4 004   Female    65     154
```

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.



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.
```

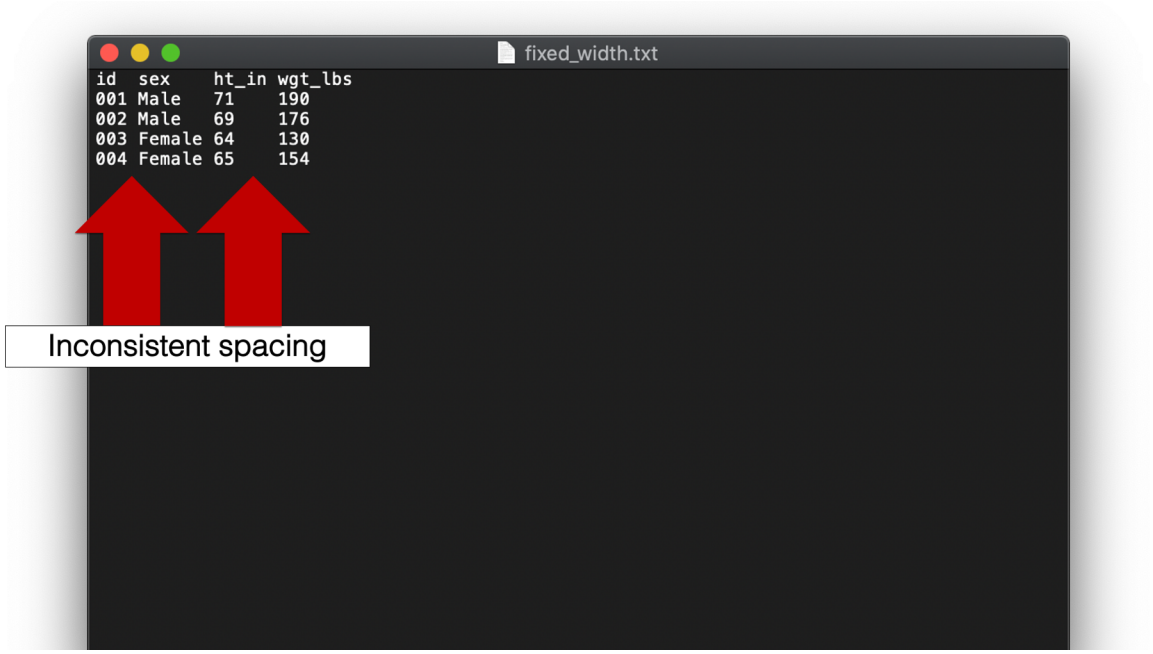
```
tab
```

```
# 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 003Female64   130           NA
4 004Female65   154           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 or TextEdit for Mac, many people view csv files in spreadsheet applications like Microsoft Excel, Numbers for Mac, or Google Sheets.

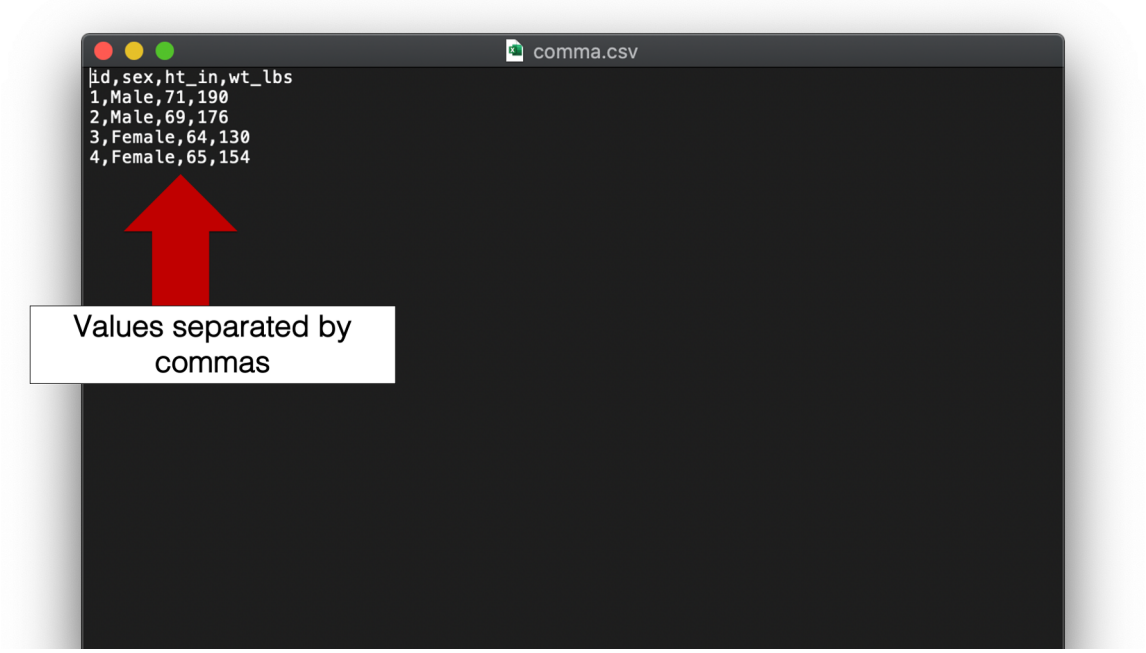


Figure 14.1: A csv file viewed in a plain text editor.

D10

	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: "," chr
(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 – staring 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
```

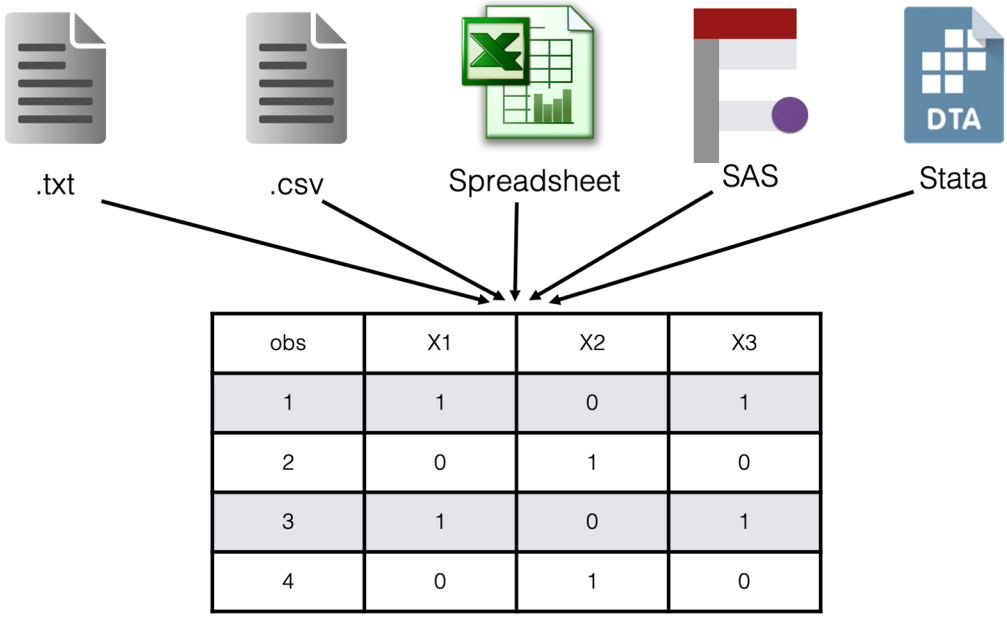
```
# A tibble: 4 x 4
  id    sex    ht_in wgt_lbs
  <chr> <chr>  <int>   <int>
1 1     Male      71     190
2 2     Male      NA     176
3 3     Female    64     130
4 4     Female    65      NA
```

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
```

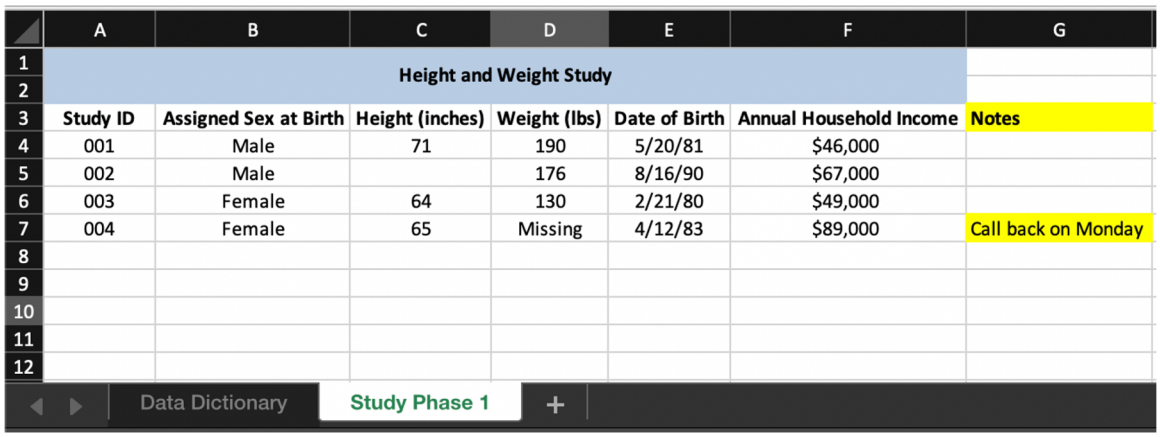
```
# A tibble: 4 x 4
  ID    sex    ht_in wgt_lbs
  <chr> <chr>  <dbl>   <dbl>
1 001   Male      71     190
2 002   Male      69     176
3 003   Female    64     130
4 004   Female    65     154
```

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 filed 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.

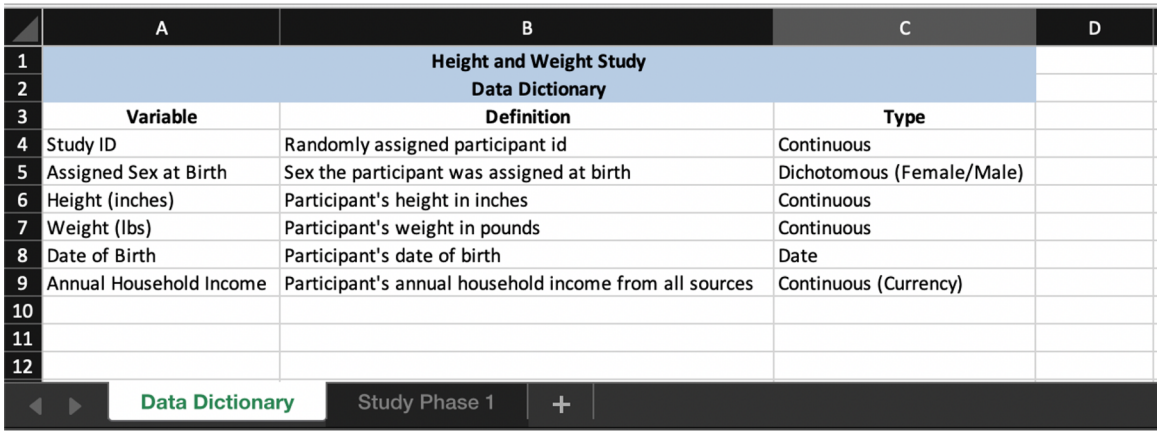


	A	B	C	D	E	F	G
1	Height and Weight Study						
2							
3	Study ID	Assigned Sex at Birth	Height (inches)	Weight (lbs)	Date of Birth	Annual Household Income	Notes
4	001	Male	71	190	5/20/81	$46,000	
5	002	Male		176	8/16/90	$67,000	
6	003	Female	64	130	2/21/80	$49,000	
7	004	Female	65	Missing	4/12/83	$89,000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

```
6 Weight (lbs)                                Participant's weight in po~ Cont~
7 Date of Birth                               Participant's date of birth Date
8 Annual Household Income                     Participant's annual house~ Cont~
```

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 were 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
```

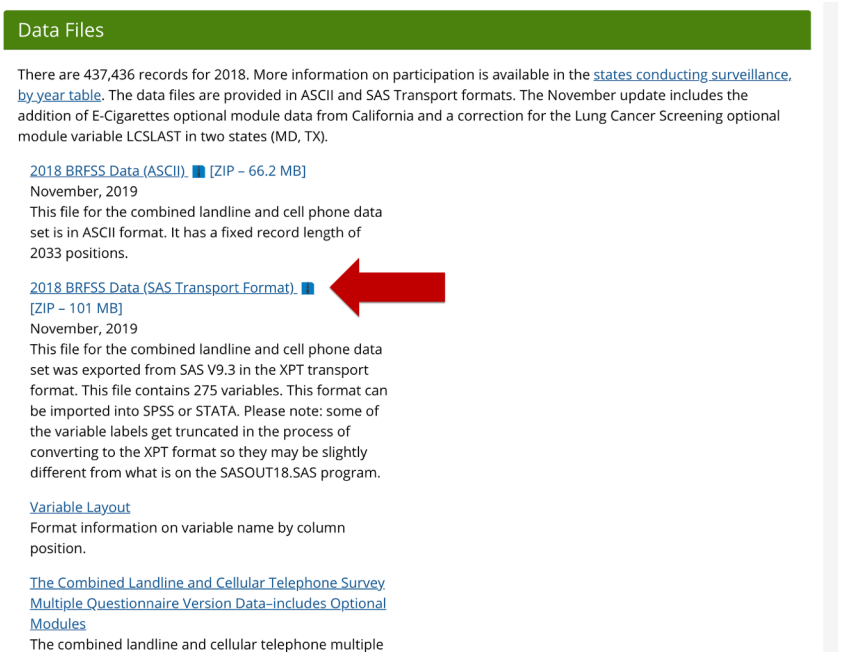
```
# A tibble: 4 x 4
  ID    sex    ht_in wgt_lbs
  <chr> <chr>  <dbl>   <dbl>
1 001   Male      71     190
2 002   Male      69     176
3 003   Female    64     130
4 004   Female    65     154
```

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)".



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
```

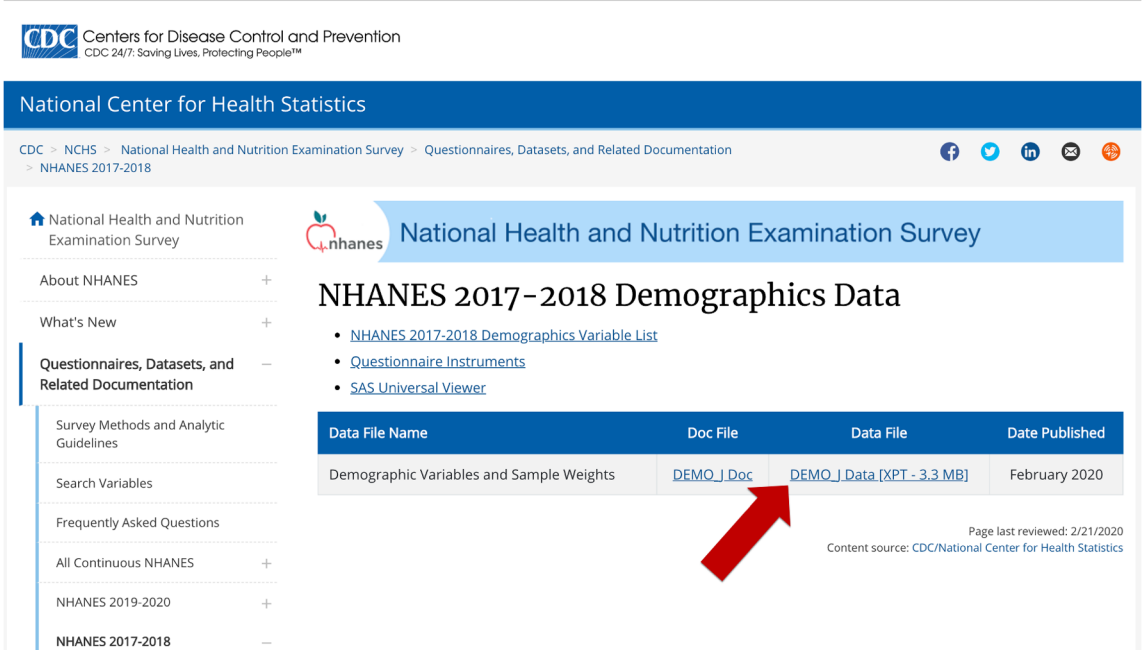
```
6        1      1 01112018 01     11    2018      1100 20180000~ 2.02e9        1
# i 265 more variables: PVTRESD1 <dbl>, COLGHOUS <dbl>, STATERE1 <dbl>,
#   CELLFON4 <dbl>, LADULT <dbl>, NUMADULT <dbl>, NUMMEN <dbl>, NUMWOMEN <dbl>,
#   SAFETIME <dbl>, CTELNUM1 <dbl>, CELLFON5 <dbl>, CADULT <dbl>,
#   PVTRESD3 <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:

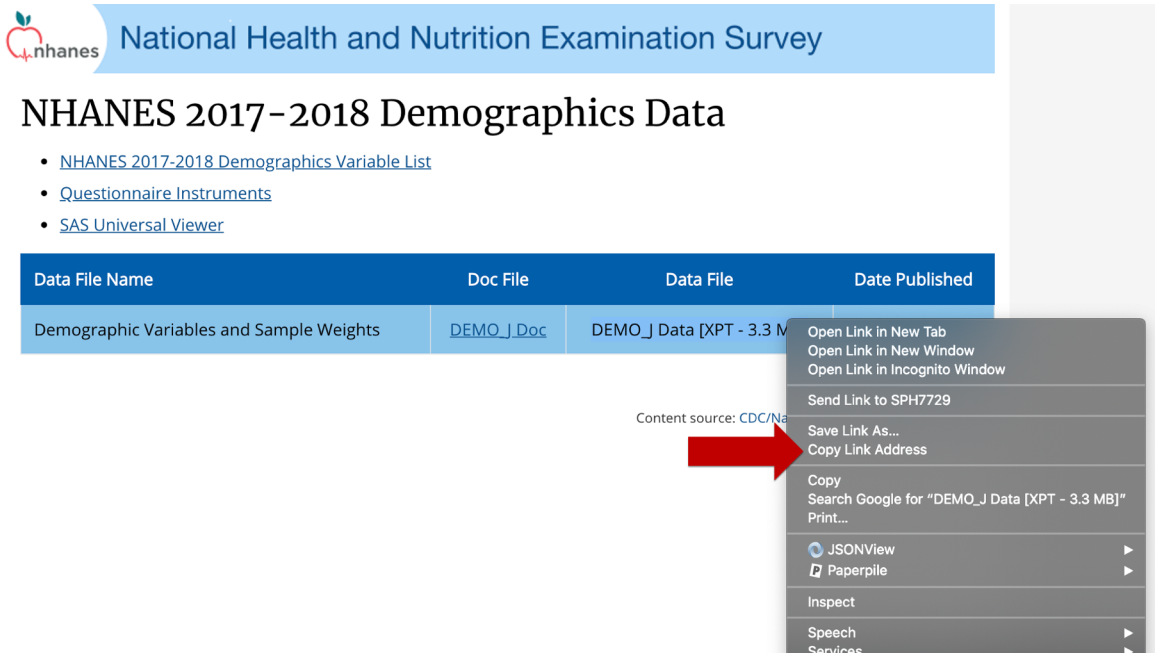
- We used `haven`'s `read_xpt()` function to import a zipped SAS Transport File. That data was imported as a data frame and we assigned that data frame to the R object called `brfss_2018`.
- Because this is a large data frame (437,436 observations and 275 variables), we used the `head()` function to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



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



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://wwwn.cdc.gov/Nchs/Data/Nhanes/Public/2017/DataFiles/DEMO_J.x
```

```
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.

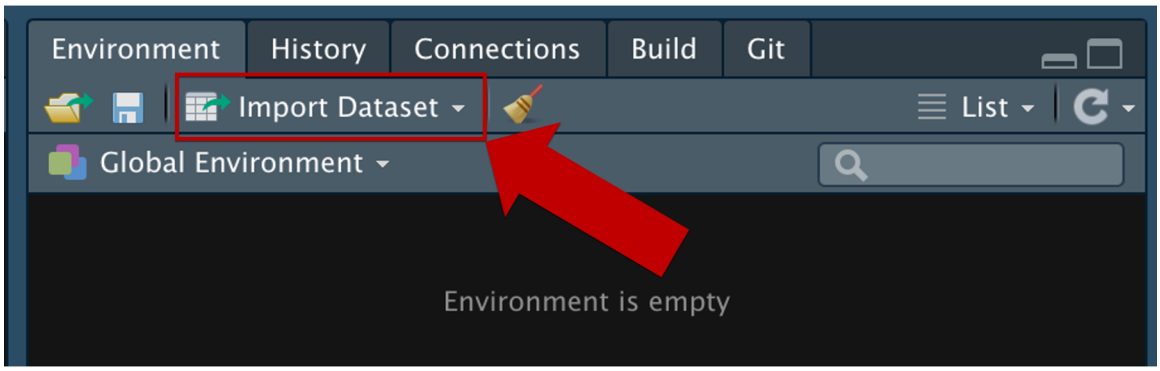
16 RStudio's Data Import Tool

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.

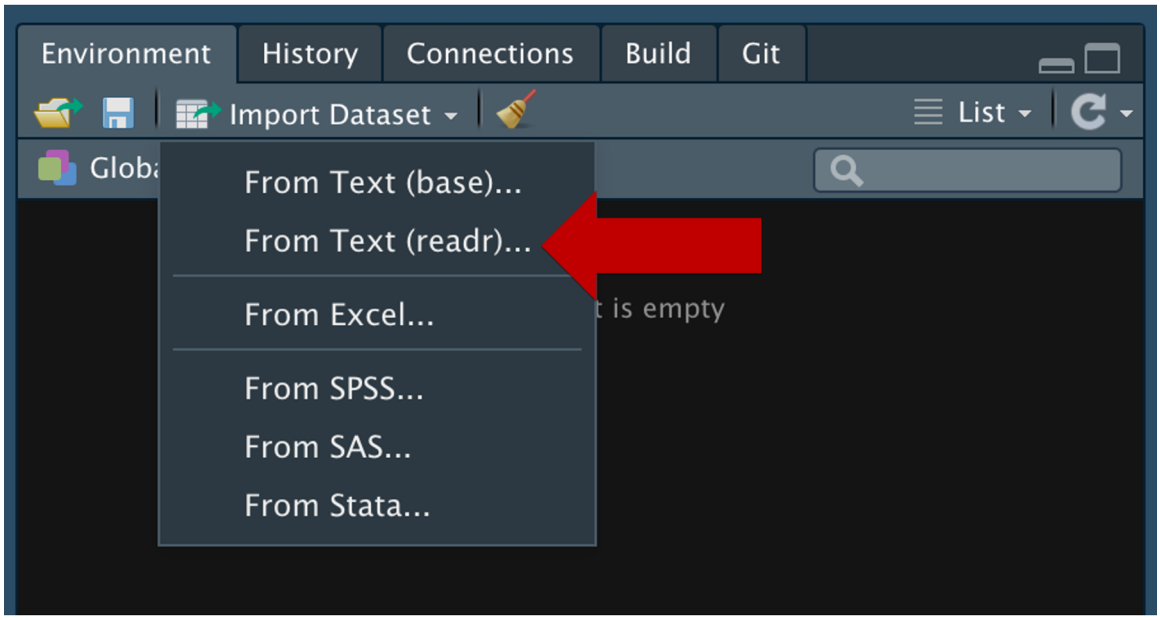
For this example, we will use the import tool to help us import the same height and weight csv file we imported in the chapter on importing plain text files.

You may click here to download this file to your compter.

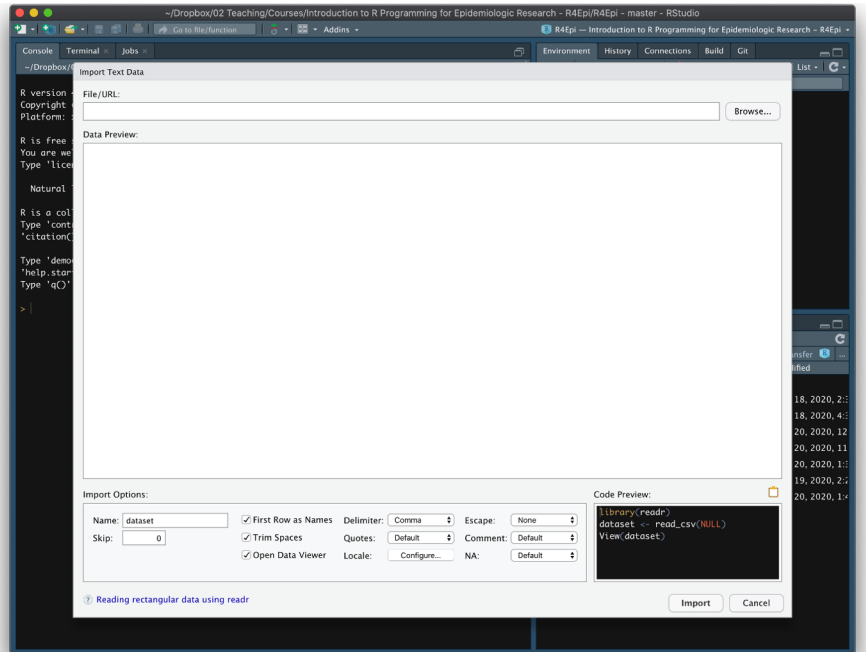
To open RStudio's data import tool, click the `Import Dataset` dropdown menu near the top of the environment pane.



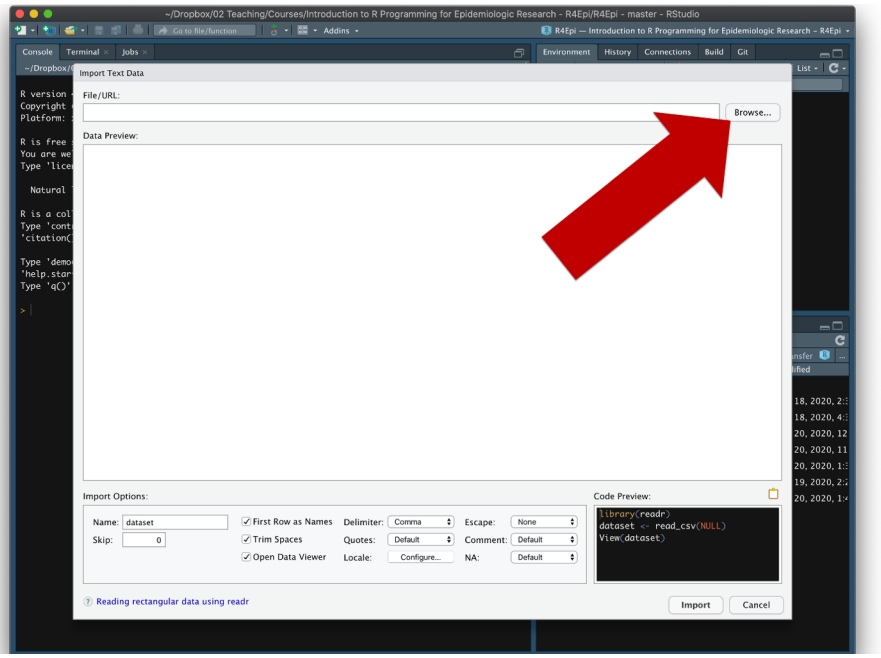
Next, because this is a csv file, we will choose the `From Text (readr)` option from the dropdown menu. The difference between `From Text (base)` and `From Text (readr)` is that `From Text (readr)` will use functions from the `readr` package to import the data and `From Text (base)` will use base R functions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



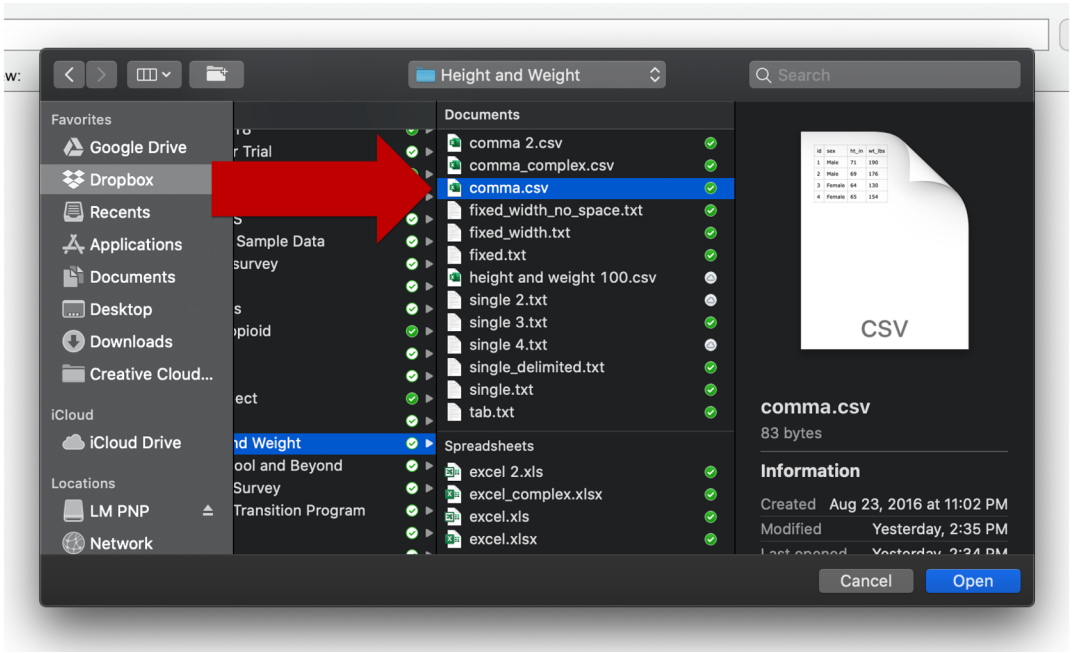
After you select a file type from the import tool dropdown menu, a separate data import window will open.



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



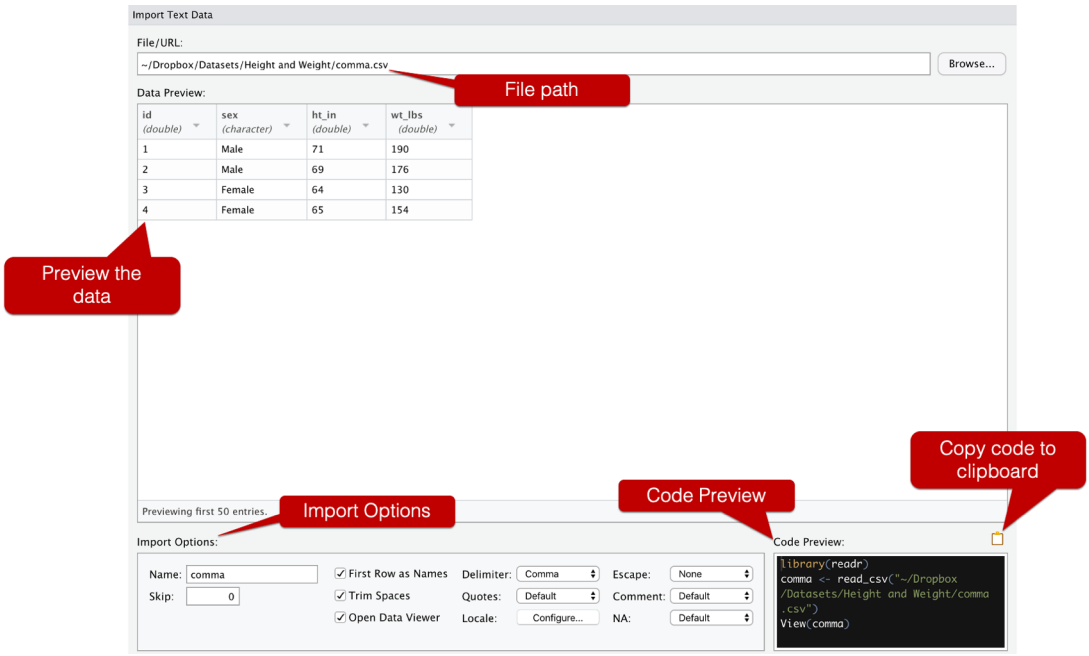
Doing so will open your operating system's file explorer window. Use that window to find and select the file you want to import. Again, we am using comma.csv for this demonstration.



After selecting you file, there will be some changes in the data import window. Specifically,

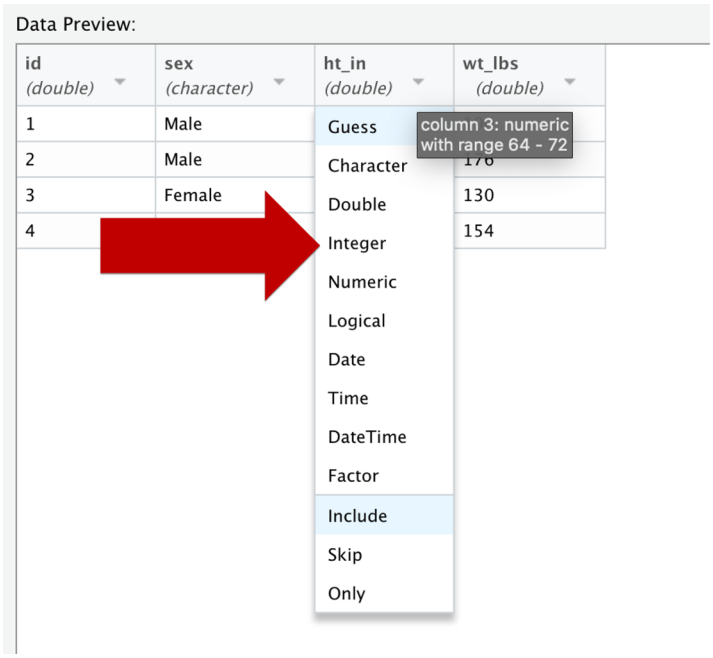
- The file path to the raw data you are importing will appear in the `File/URL` field.

- 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.



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.

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 a 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 you 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 Chapter 5. 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 globabl 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`s `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:

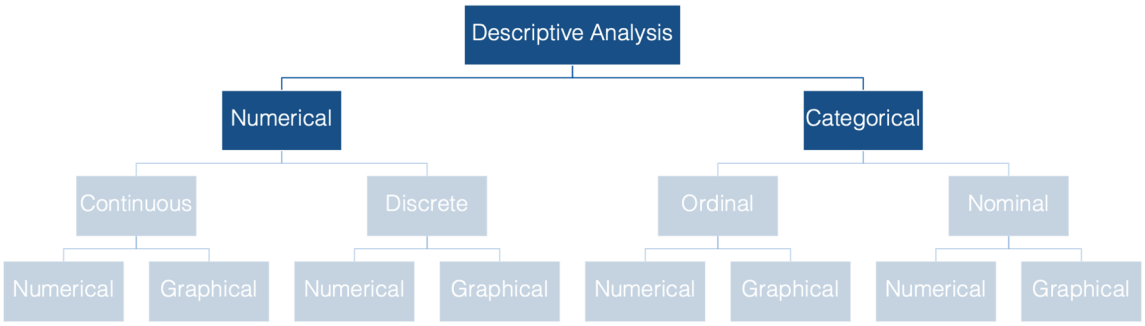
1. *We can use descriptive analysis to uncover errors in our data.*

2. *It helps us understand the distribution of values in our variables.*

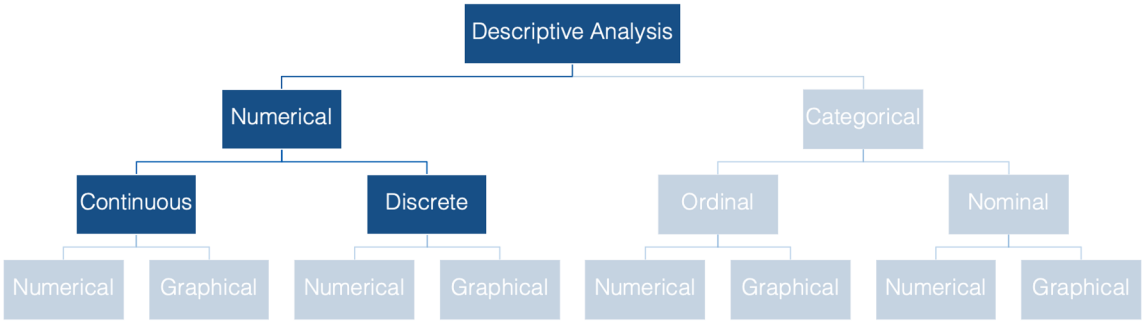
3. *Descriptive analysis serve as a starting point for understanding relationships between our variables.*

18.2 What kind of descriptive analysis should we perform?

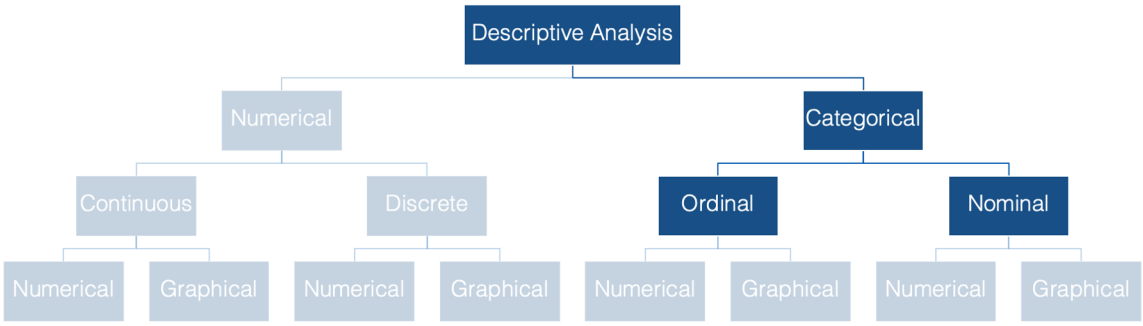
When conducting descriptive analysis, the method you choose will depend on the *type* of data you're analyzing. At the most basic level, variables can be described as numerical or categorical.



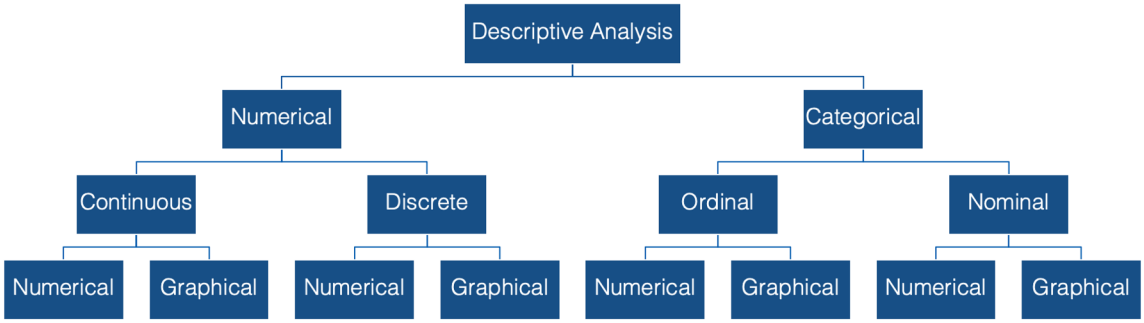
Numeric variables can then be further divided into continuous and discrete - the distinction being whether the variable can take on a continuum of values, or only set of certain values.



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.

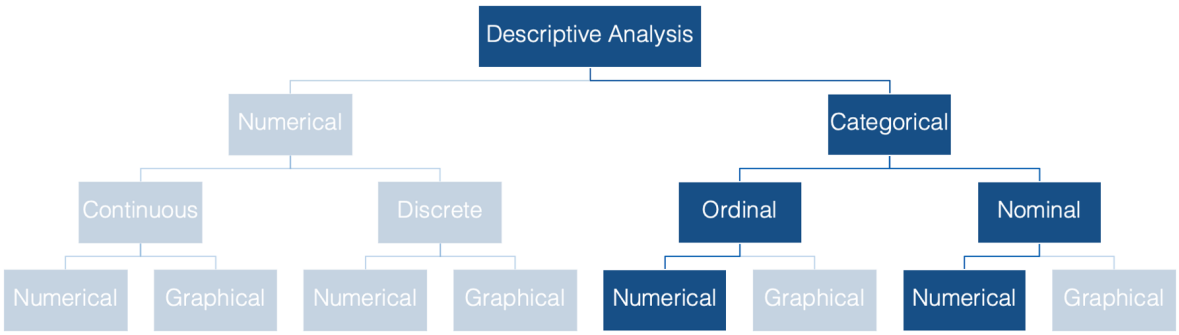


Figure 19.1: Numerical variable descriptive analysis flowchart.

Some examples of categorical variables commonly seen in public health data are: sex, race or ethnicity, and level of educational attainment.

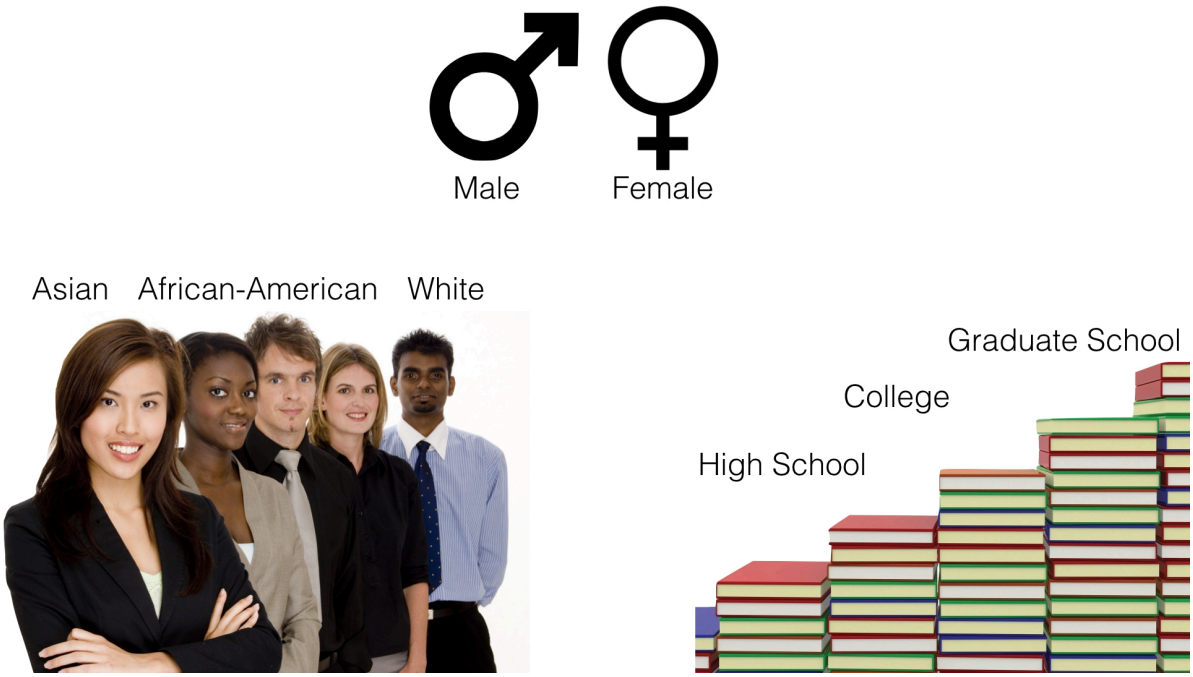
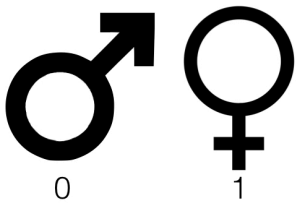


Figure 19.2: Examples of categorical variables.

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.



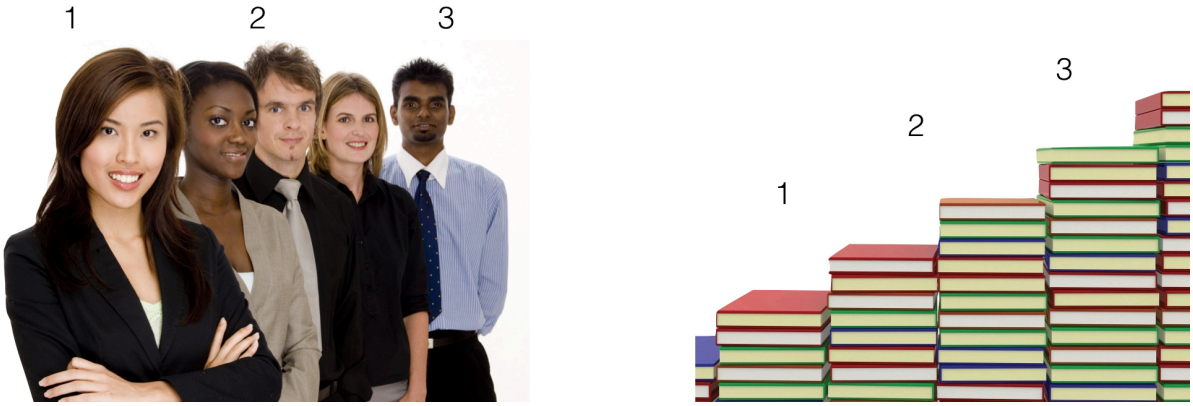


Figure 19.3: Examples of categorical variables with assigned numeric values.

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%

Figure 19.4: Frequency and count - common numeric descriptions of categorical variables.

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)
```

```r
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.

```r
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"
      )
    )
  )

demo
```

```
# 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 later in the book.
  - 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
  id      age   edu edu_char              edu_f       edu_5cat_f edu_f_from_char
  <chr> <dbl> <dbl> <chr>                 <fct>       <fct>      <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 gradua~
4 004      56     2 High school graduate  High schoo~ High scho~ High school gr~
```

```
table(demo$edu_f_from_char)
```

```
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.

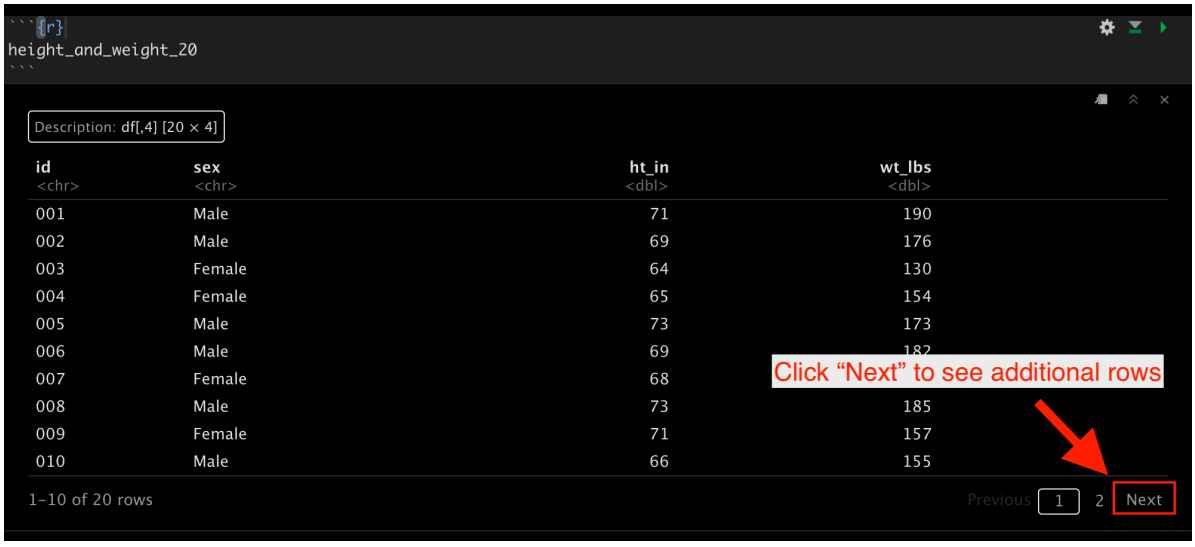


Figure 19.5: The "Next" button in RStudio.

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 saying 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 say `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 yo
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**
>
> 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**
>
> `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. 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.

20 Measures of Central Tendency

In previous sections you've seen methods for describing individual categorical variables. Now we'll switch over to numerically describing numerical variables.

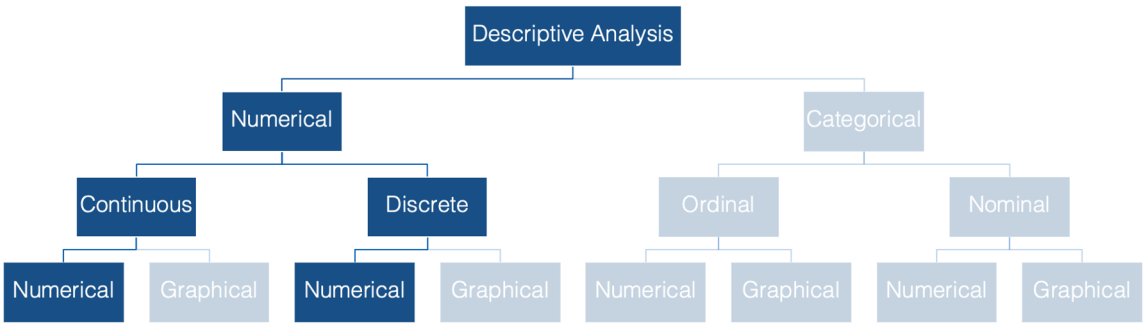


Figure 20.1: Numerical variable descriptive analysis flowchart.

In epidemiology, we often want to describe the "typical" person in a population with respect to some characteristic that is recorded as a numerical variable – like height or weight. The most basic, and probably most commonly used, way to do so is with a measure of central tendency.

In this chapter we'll discuss three measures of central tendency:

- **The mean**
- **The median**

- **The mode**

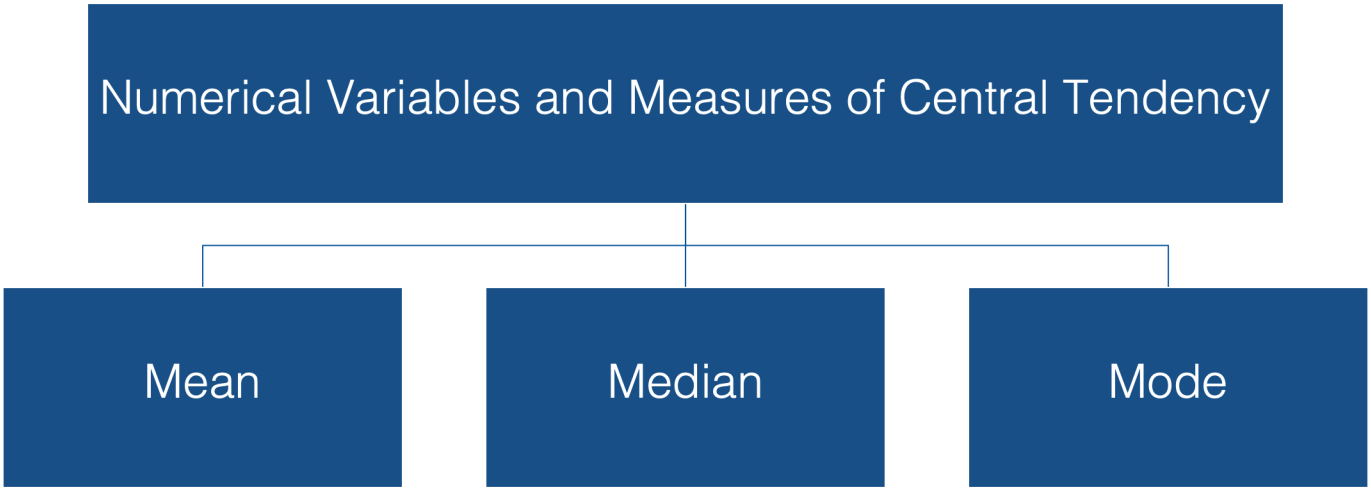


Figure 20.2: Measures of central tendency chart.

Now, this is not a statistics course. But we will briefly discuss these measures and some of their characteristics below to make sure that we're all on the same page when we discuss the interpretation of our results.

The mean

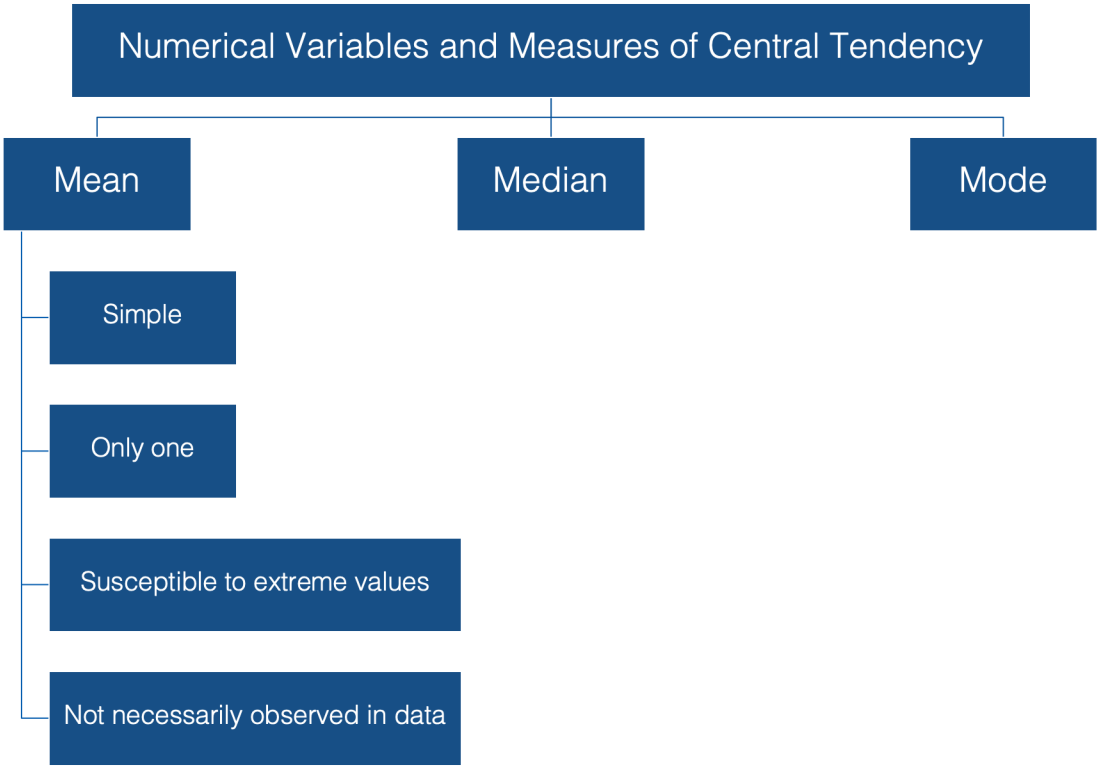


Figure 20.3: Mean chart.

When we talk about the typical, or "average", value of some variable measured on a continuous scale, we are usually talking about the mean value of that variable. To be even more specific, we are usually talking about the arithmetic mean value. This value has some favorable characteristics that make it a good description of central tendency.

For starters it's simple. Most people are familiar with the mean, and at the very least, have some intuitive sense of what it means (no pun intended).

In addition, there can be only one mean value for any set of values.

However, there are a couple of potentially problematic characteristics of the mean as well:

It's susceptible to extreme values in your data. In other words, a couple of people with very atypical values for the characteristic you are interested in can drastically alter the value of the mean, and your estimate for the typical person in your population of interest along with it.

Additionally, it's very possible to calculate a mean value that is not actually observed anywhere in your data.

> **Note**
>
> The sample mean is often referred to as $\bar{x}$, which pronounced "x bar."

The median

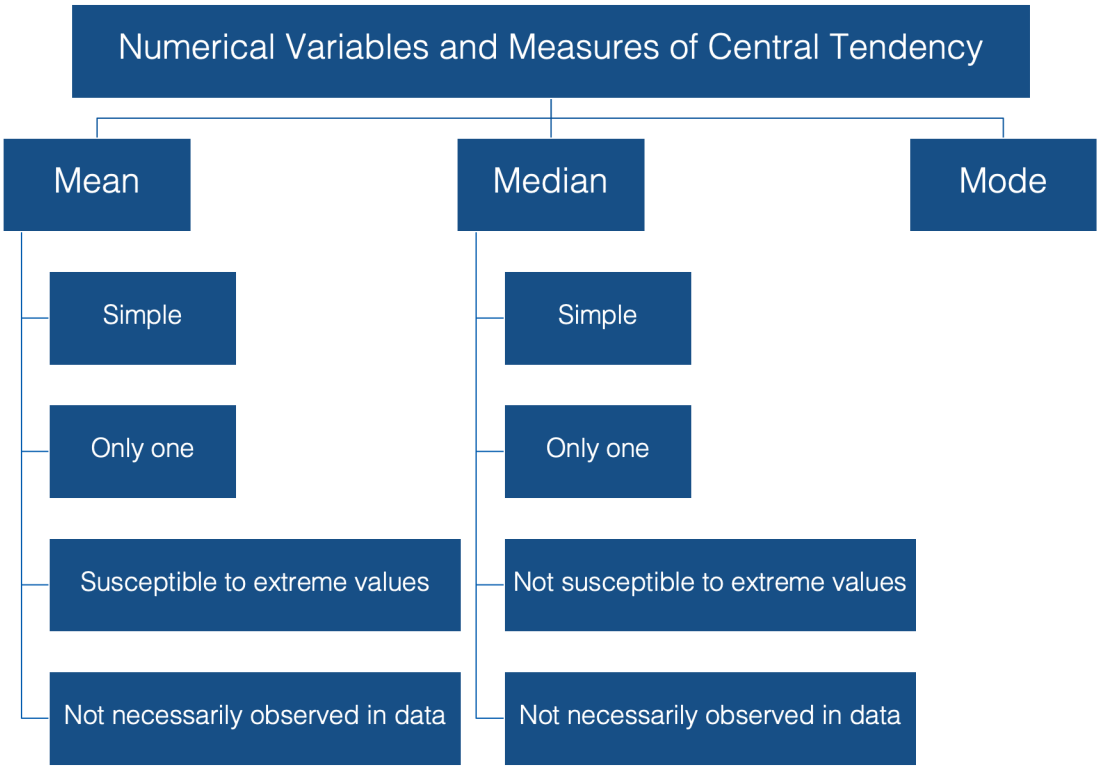


Figure 20.4: Mean and median chart

The median is probably the second most commonly used measure of central tendency. Like the mean, it's computationally simple and relatively straightforward to understand. There can be one, and only one, median. And, its value may also be unobserved in the data.

However, unlike the mean, it's relatively resistant to extreme values. In fact, when the median is used as the measure of central tendency, it's often because the person conducting the analysis suspects that extreme values in the data are likely to distort the mean.

The mode

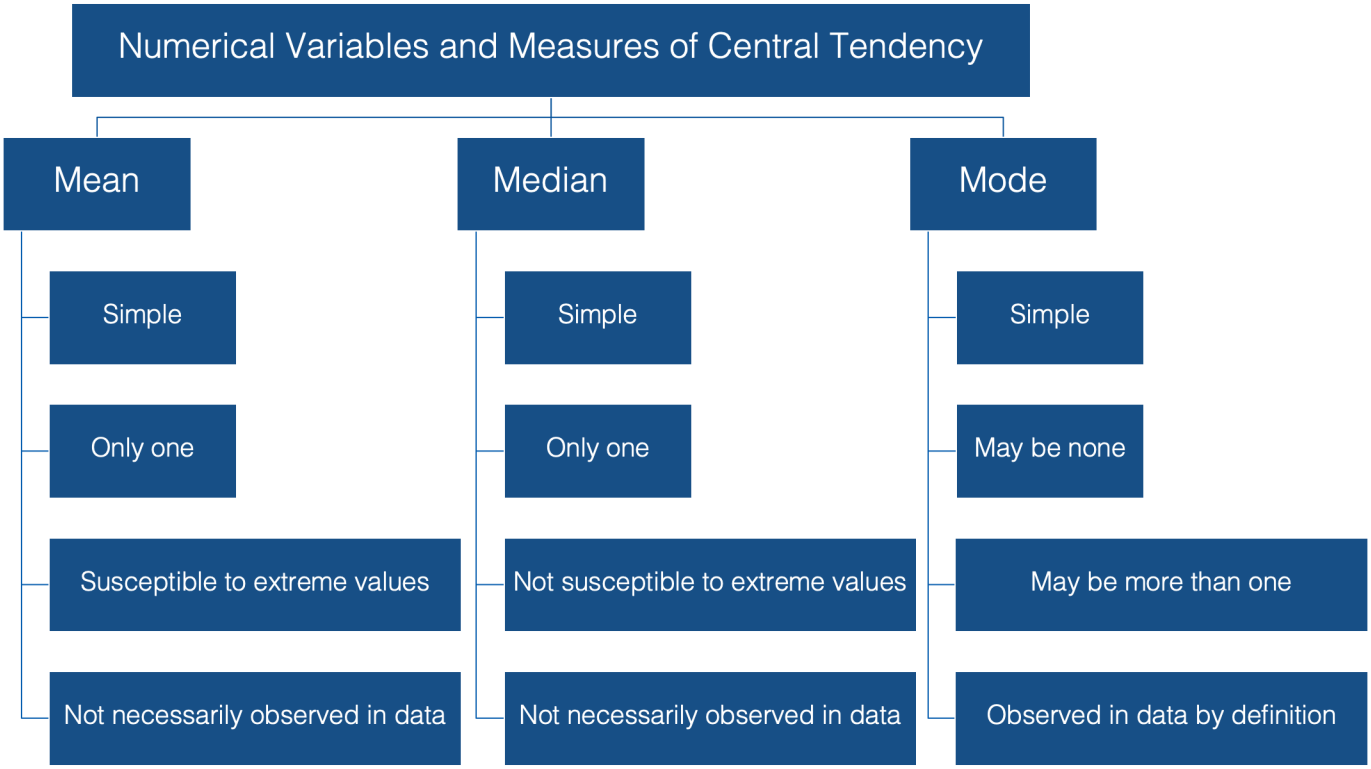


Figure 20.5: Mean, median and mode chart

And finally, we have the mode, or the value that is most often observed in the data. It doesn't get much simpler than that. But, unlike the mean and the median, there can be more than one mode for a given set of values. In fact, there can even be no mode if all the values are observed the exact same number of times.

However, if there is a mode, by definition it's observed in the data.

Now that we are all on the same page with respect to the fundamentals of central tendency, let's take a look at how to calculate these measures using R.

20.1 Calculate the mean

Calculating the mean is really straightforward. We can just use base R's built-in `mean()` function.

```
# 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 <- tribble(
  ~id,   ~sex,     ~ht_in, ~wt_lbs,
  "001", "Male",   71,     190,
  "002", "Male",   69,     177,
  "003", "Female", 64,     130,
  "004", "Female", 65,     153,
  "005", NA,       73,     173,
  "006", "Male",   69,     182,
  "007", "Female", 68,     186,
  "008", NA,       73,     185,
  "009", "Female", 71,     157,
  "010", "Male",   66,     155,
  "011", "Male",   71,     213,
  "012", "Female", 69,     151,
  "013", "Female", 66,     147,
  "014", "Female", 68,     196,
  "015", "Male",   75,     212,
  "016", "Female", 69,     19000,
  "017", "Female", 66,     194,
  "018", "Female", 65,     176,
  "019", "Female", 65,     176,
  "020", "Female", 65,     102
)
```

Here's what we did above:

- We loaded the `tibble` package so that we could use its `tribble()` function.
- We used the `tribble()` function to simulate some data – heights and weights for 20 hypothetical students.
  - The `tribble()` function creates something called a tibble. A tibble is the `tidyverse` version of a data frame. In fact, it *is* a data frame, but with some additional functionality. You can use the link to read more about it if you'd like.
  - We used the `tribble()` function instead of the `data.frame()` function to create our data frame above because we can use the `tribble()` function to create our data frames in rows (like you see above) instead of columns with the `c()` function.
  - Using the `tribble()` function to create a data frame isn't any better or worse than using the `data.frame()` function. You should just be aware that it exists and is sometimes useful.

```
mean(height_and_weight_20$ht_in)
```

```
[1] 68.4
```

Here's what we did above:

- We used base R's `mean()` function to calculate the mean of the column "ht_in" from the data frame "height_and_weight_20".
  - Note: if you just type `mean(ht_in)` you will get an error. That's because R will look for an object called "ht_in" in the global environment.
  - However, we didn't create an object called "ht_in". We created an object (in this case a data frame) called "height_and_weight_20". That object has a column in it called "ht_in".
  - So, we must specifically tell R to look for the "ht_in" column in the data frame "height_and_weight_20". Using base R, we can do that in one of two ways: `height_and_weight_20$ht_in` or `height_and_weight_20[["ht_in"]]`.

20.2 Calculate the median

Similar to above, we can use base R's `median()` function to calculate the median.

```
median(height_and_weight_20$ht_in)
```

```
[1] 68.5
```

Here's what we did above:

- We used base R's `median()` function to calculate the median of the column "ht_in" from the data frame "height_and_weight_20".

20.3 Calculate the mode

Base R does not have a built-in `mode()` function. Well, it actually does have a `mode()` function, but for some reason that function does not return the mode value(s) of a set of numbers. Instead, the `mode()` function gets or sets the type or storage mode of an object. For example:

```
mode(height_and_weight_20$ht_in)
```

```
[1] "numeric"
```

This is clearly not what we are looking for. So, how do we find the mode value(s)? Well, we are going to have to build our own mode function. Later in the book, we will return to this function and walk through how to build it one step at a time. For now, just copy and paste the code into R on your computer. Keep in mind, as is almost always the case with R, this way of writing this function is only one of multiple possible ways.

```
mode_val <- function(x) {

  # Count the number of occurrences for each value of x
  value_counts <- table(x)

  # Get the maximum number of times any value is observed
  max_count <- max(value_counts)

  # Create and index vector that identifies the positions that correspond to
  # count values that are the same as the maximum count value: TRUE if so
  # and false otherwise
  index <- value_counts == max_count

  # Use the index vector to get all values that are observed the same number
  # of times as the maximum number of times that any value is observed
  unique_values <- names(value_counts)
  result <- unique_values[index]

  # If result is the same length as value counts that means that every value
  # occured the same number of times. If every value occurred the same number
  # of times, then there is no mode
  no_mode <- length(value_counts) == length(result)

  # If there is no mode then change the value of result to NA
  if (no_mode) {
    result <- NA
  }

  # Return result
  result
}
```

```
mode_val(height_and_weight_20$ht_in)
```

```
[1] "65" "69"
```

Here's what we did above:

- We created our own function, `mode_val()`, that takes a vector (or data frame column) as a value to its "x" argument and returns the mode value(s) of that vector.
- We can also see that the function works as expected when there is more than one mode value. In this case, "65" and "69" each occur 4 times in the column "ht_in".

20.4 Compare mean, median, and mode

Now that you know how to calculate the mean, median, and mode, let's compare these three measures of central tendency. This is a good opportunity to demonstrate some of the different characteristics of each that we spoke about earlier.

```
height_and_weight_20 %>%
  summarise(
    min_weight    = min(wt_lbs),
    mean_weight   = mean(wt_lbs),
    median_weight = median(wt_lbs),
    mode_weight   = mode_val(wt_lbs) %>% as.double(),
    max_weight    = max(wt_lbs)
  )
```

```
# A tibble: 1 x 5
  min_weight mean_weight median_weight mode_weight max_weight
       <dbl>       <dbl>         <dbl>       <dbl>      <dbl>
1        102       1113.          176.         176      19000
```

Here's what we did above:

- We used the `mean()` function, `median()` function, and our `mode_val()` function inside of dplyr's `summarise()` function to find the mean, median, and mode values of the column "wt_lbs" in the "height_and_weight_20" data frame.
- We also used the `as.double()` function to convert the value returned by `mode_val()` – "176" – from a character string to a numeric double. This isn't strictly necessary, but does look better.

- Finally, we used base R's `min()` and `max()` functions to view the lowest and highest weights in our sample.

20.5 Data checking

Do you see any red flags as you scan the results? Do you really think a mean weight of 1,113 pounds sounds reasonable? This should definitely be a red flag for you. Now move your gaze three columns to the right and notice that the maximum value of weight is 19,000 lbs – an impossible value for a study in human populations. In this case the real weight was supposed to be 190 pounds, but the person entering the data accidentally got a little trigger-happy with the zero key.

This is an example of what was meant by "We can use descriptive analysis to uncover errors in our data" in the Introduction to descriptive analysis chapter. Often times, for various reasons, some observations for a given variable take on values that don't make sense. Starting by calculating some basic descriptive statistics for each variable is one approach you can use to try to figure out if you have values in your data that don't make sense.

In this case we can just go back and fix our data, but what if we didn't know this value was an error? What if it were a value that was technically possible, but very unlikely? Well, we can't just go changing values in our data. It's unethical, and in some cases illegal. Below, we discuss the how the properties of the median and mode can come in handy in situations such as this.

20.6 Properties of mean, median, and mode

Despite the fact that this impossibly extreme value is in our data, the median and mode estimates are reasonable estimates of the typical person's weight in this sample. This is what we mean when we say that the median and mode are more "resistant to extreme values" than the mean.

You may also notice that no person in our sample had an actual weight of 1,112.75 (the mean) or even 176.5 (the median). This is what we we mean when we say that the mean and median values are "not necessarily observed in the data."

In this case, the mode value (176) is also a more reasonable estimate of the average person's weight than the mean. And unlike the mean and the median, participants 18 and 19 actually weigh 176 pounds. This is **not** to say that the mode is always the best measure of central tendency to use. However, you can often learn useful information from your data by calculating and comparing these relatively simple descriptive statistics on each of your numeric variables.

20.7 Missing data

In numerical descriptions of categorical variables we saw that we could use the `dplyr::filter()` function to remove all the rows from our data frame that contained a missing value for any of our variables of interest. We learned that this is called a complete case analysis. This method should pretty much always work, but in this section, you will see an alternative method for dropping missing values from your analysis that you are likely to come across often when reading R documentation – the `na.rm` argument.

Many R functions that perform calculations on numerical variables include an `na.rm` – short for "Remove NA" – argument. By default, this argument is typically set to `FALSE`. By passing the value `TRUE` to this argument, we can perform a complete case analysis. Let's quickly take a look at how it works.

We already saw that we can calculate the mean value of a numeric vector using the `mean()` function:

```
mean(c(1, 2, 3))
```

```
[1] 2
```

But, what happens when our vector has a missing value?

```
mean(c(1, NA, 3))
```

```
[1] NA
```

As you can see, the `mean()` function returns `NA` by default when we pass it a numeric vector that contains a missing value. It can be confusing to understand why this is the case. The logic goes something like this. In R, an `NA` doesn't represent the *absence* of a value – a value that doesn't exist at all; rather, it represents a value that does exist, but is *unknown* to us. So, if you were asked to give the mean of a set of numbers that contains 1, some unknown number, and 3 what would your answer be? Well, you can't just give the mean of 1 and 2. That would imply that the unknown number doesn't exist. Further, you can't really give *any* numeric answer because that answer will depend on the value of the missing number. So, the only logical answer to give is something like "I don't know" or "it depends." That is essentially what R is telling us when it returns an `NA`.

While this answer is technically correct, it usually isn't very satisfying to us. Instead, we often want R to calculate the mean of the numbers that remain after all missing values are removed from the original set. The implicit assumption is that the mean of that reduced set of numbers will be "close enough" to the mean of the original set of numbers for our purposes. We can

ask R to do this by changing the value of the `na.rm` argument from `FALSE` – the default – to `TRUE`.

```
mean(c(1, NA, 3), na.rm = TRUE)
```

```
[1] 2
```

In this case, the mean of the original set of numbers (2) and the mean of our complete case analysis (2) are identical. That won't always be the case.

Finally, let's compare using `filter()` and `na.rm = TRUE` in a `dplyr` pipeline. We will first use the `replace()` function to add some missing values to our `height_and_weight_20` data.

```
height_and_weight_20 <- height_and_weight_20 %>%
  mutate(ht_in = replace(ht_in, c(1, 2), NA)) %>%
  print()
```

```
# A tibble: 20 x 4
   id    sex    ht_in wt_lbs
   <chr> <chr>  <dbl>  <dbl>
 1 001   Male      NA    190
 2 002   Male      NA    177
 3 003   Female    64    130
 4 004   Female    65    153
 5 005   <NA>      73    173
 6 006   Male      69    182
 7 007   Female    68    186
 8 008   <NA>      73    185
 9 009   Female    71    157
10 010   Male      66    155
11 011   Male      71    213
12 012   Female    69    151
13 013   Female    66    147
14 014   Female    68    196
15 015   Male      75    212
16 016   Female    69  19000
17 017   Female    66    194
18 018   Female    65    176
19 019   Female    65    176
20 020   Female    65    102
```

Here's what we did above:

- Replaced the 1st and 2nd value of `ht_in` with `NA` (missing) using the `replace()` function.

Here's what our results look like when we don't perform a complete case analysis.

```
height_and_weight_20 %>%
  summarise(
    min_height    = min(ht_in),
    mean_height   = mean(ht_in),
    median_height = median(ht_in),
    mode_height   = mode_val(ht_in),
    max_height    = max(ht_in)
  )
```

```
# A tibble: 1 x 5
  min_height mean_height median_height mode_height max_height
       <dbl>       <dbl>         <dbl> <chr>             <dbl>
1         NA          NA            NA 65                   NA
```

Here's what our results look like when we use the `filter()` function.

```
height_and_weight_20 %>%
  filter(!is.na(ht_in)) %>%
  summarise(
    min_height    = min(ht_in),
    mean_height   = mean(ht_in),
    median_height = median(ht_in),
    mode_height   = mode_val(ht_in),
    max_height    = max(ht_in)
  )
```

```
# A tibble: 1 x 5
  min_height mean_height median_height mode_height max_height
       <dbl>       <dbl>         <dbl> <chr>             <dbl>
1         64        68.2            68 65                   75
```

And, here's what our results look like when we change the `na.rm` argument to `TRUE`.

```
height_and_weight_20 %>%
  summarise(
    min_height    = min(ht_in, na.rm = TRUE),
    mean_height   = mean(ht_in, na.rm = TRUE),
```

```
    median_height = median(ht_in, na.rm = TRUE),
    mode_height   = mode_val(ht_in),
    max_height    = max(ht_in, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 5
  min_height mean_height median_height mode_height max_height
       <dbl>       <dbl>         <dbl> <chr>            <dbl>
1         64        68.2            68 65                  75
```

As you can see, both methods give us the same result. The method you choose to use will typically just come down to personal preference.

20.8 Using meantables

In the sections above, we learned how to use `dplyr` functions to calculate various measures of central tendency for continuous variables. 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, you will be shown how to write your own functions. For the time being, we suggest that you install and use the meantables package. The `meantables` 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 for continuous 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
# meantables uses. Go ahead and do so.
install.packages("meantables")
```

And then load it...

```
# After installing meantables on your computer, you can load it just like you
# would any other package.
library(meantables)
```

Now, let's use the `mean_table()` function from `meantables` package to rerun our analysis from above.

```
height_and_weight_20 %>%
  filter(!is.na(ht_in)) %>%
  mean_table(ht_in)
```

```
# A tibble: 1 x 9
  response_var     n  mean    sd   sem   lcl   ucl   min   max
  <chr>        <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 ht_in           18  68.2  3.28 0.774  66.6  69.8    64    75
```

Here's what we did above:

- We used `filter()` to keep only the rows that have a *non-missing* value for `ht_in` and passed the data frame on to the `mean_table()` function using a pipe.
- We told the `mean_table()` function to create a table of summary statistics for the variable `ht_in`. This is just an R data frame of useful statistics about a single continuous variable.
- The summary statistics in the table above include:
  - `response_var`: The name of the variable (column) we are analyzing.
  - `n`: The number of non-missing values of `response_var` being analyzed in the current analysis.
  - `mean`: The mean of all `n` values of `response_var`.
  - `sem`: The standard error of the mean of all `n` values of `response_var`.
  - `lcl`: The lower (95%, by default) confidence limit for the percentage `mean`.
  - `ucl`: The upper (95%, by default) confidence limit for the percentage `mean`.
  - `min`: The minimum value of `response_var`.
  - `max`: The maximum value of `response_var`.

We will continue using the `meantables` package at various points throughout the book. You will also be shown some other cool things we can do with `meantables`. For now, all you need to know how to do is use the `mean_table()` function to calculate basic descriptive statistics for single continuous variables.

21 Measures of Dispersion

In the chapter on measures of central tendency, we found the minimum value, mean value, median value, mode value, and maximum value of the weight variable in our hypothetical sample of students. We'll go ahead and start this lesson by rerunning that analysis below, but this time we will analyze heights instead of weights.

```
# 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 <- tribble(
  ~id,   ~sex,     ~ht_in, ~wt_lbs,
  "001", "Male",   71,     190,
  "002", "Male",   69,     177,
  "003", "Female", 64,     130,
  "004", "Female", 65,     153,
  "005", NA,       73,     173,
  "006", "Male",   69,     182,
  "007", "Female", 68,     186,
  "008", NA,       73,     185,
  "009", "Female", 71,     157,
  "010", "Male",   66,     155,
  "011", "Male",   71,     213,
  "012", "Female", 69,     151,
  "013", "Female", 66,     147,
  "014", "Female", 68,     196,
  "015", "Male",   75,     212,
  "016", "Female", 69,     19000,
  "017", "Female", 66,     194,
  "018", "Female", 65,     176,
  "019", "Female", 65,     176,
  "020", "Female", 65,     102
)
```

```r
# Recreate our mode function
mode_val <- function(x) {
  value_counts <- table(x)
  result <- names(value_counts)[value_counts == max(value_counts)]
  if (length(value_counts) == length(result)) {
    result <- NA
  }
  result
}
```

```r
height_and_weight_20 %>%
  summarise(
    min_height    = min(ht_in),
    mean_height   = mean(ht_in),
    median_height = median(ht_in),
    mode_height   = mode_val(ht_in) %>% paste(collapse = " , "),
    max_height    = max(ht_in)
  )
```

```
# A tibble: 1 x 5
  min_height mean_height median_height mode_height max_height
       <dbl>       <dbl>         <dbl> <chr>            <dbl>
1         64        68.4          68.5 65 , 69             75
```

> **Note**
>
> To get both mode height values to display in the output above we used the `paste()` function with the collapse argument set to " , " (notice the spaces). This forces R to display our mode values as a character string. The downside is that the "mode_height" variable no longer has any numeric value to R – it's simply a character string. However, this isn't a problem for us. We won't be using the mode in this lesson – and it is rarely used in practice.

Keep in mind that our interest is in describing the "typical" or "average" person in our sample. The result of our analysis above tells us that the average person who answered the height question in our hypothetical class was: 68.4 inches. This information gets us reasonably close to understanding the typical height of the students in our hypothetical class. But remember, our average person does not necessarily have the same height as any **actual person** in our class. So a natural extension of our original question is: "how much like the average person, are the other people in class."

For example, is everyone in class 68.4 inches?

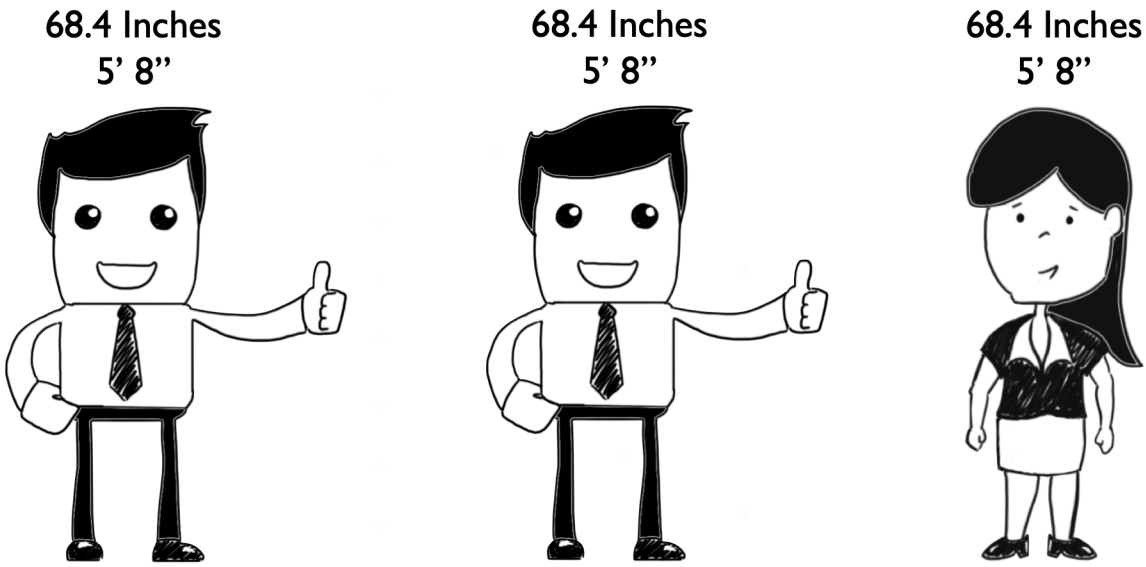


Figure 21.1: Example with people with the same height

Or are there differences in everyone’s height, with the average person’s height always having a value in the middle of everyone else’s?

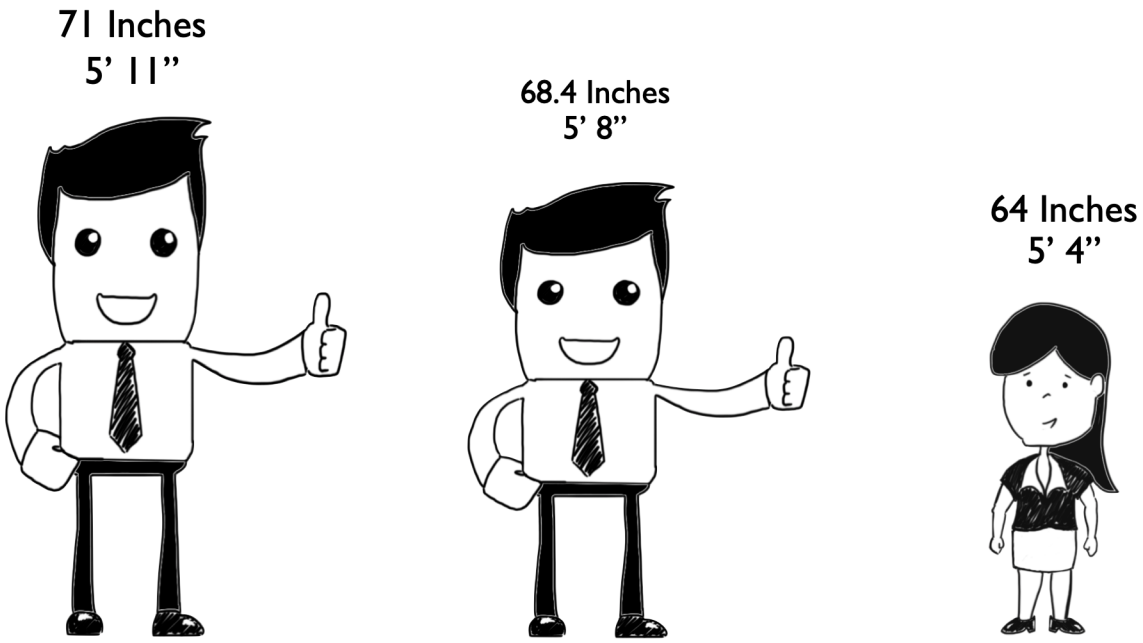


Figure 21.2: Example with people of different heights

The measures used to answer this question are called measures of dispersion, which we can say is the amount of difference between people in the class, or more generally, the amount of variability in the data.

Three common measures of dispersion used are the:

- **Range**
- **Variance**
- **Standard Deviation**

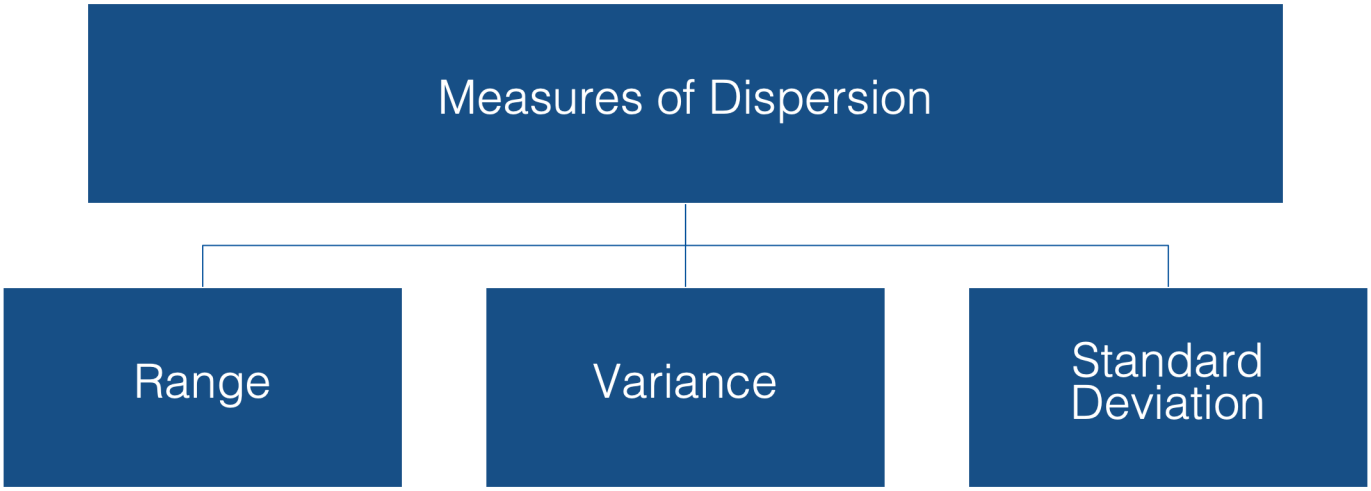


Figure 21.3: Measures of dispersion chart

Range

The range is simply the difference between the maximum and minimum value in the data.

```
height_and_weight_20 %>%
  summarise(
    min_height  = min(ht_in),
    mean_height = mean(ht_in),
    max_height  = max(ht_in),
    range       = max_height - min_height
  )
```

```
# A tibble: 1 x 4
  min_height mean_height max_height range
       <dbl>       <dbl>      <dbl> <dbl>
1         64        68.4         75    11
```

In this case, the range is 11. The range can be useful because it tells us how much difference there is between the tallest person in our class and the shortest person in our class – 11 inches. However, it doesn't tell us how close to 68.4 inches "most" people in the class are.

In other words, are most people in the class out at the edges of the range of values in the data?

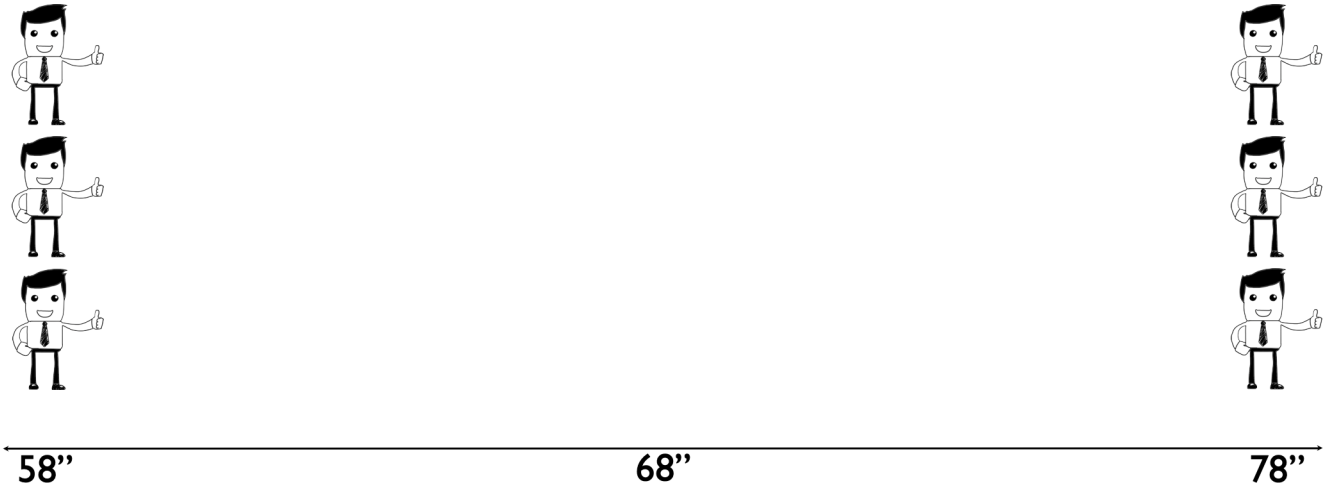


Figure 21.4: Example with people's heights on the edges of the range

Or are people "evenly distributed" across the range of heights for the class?

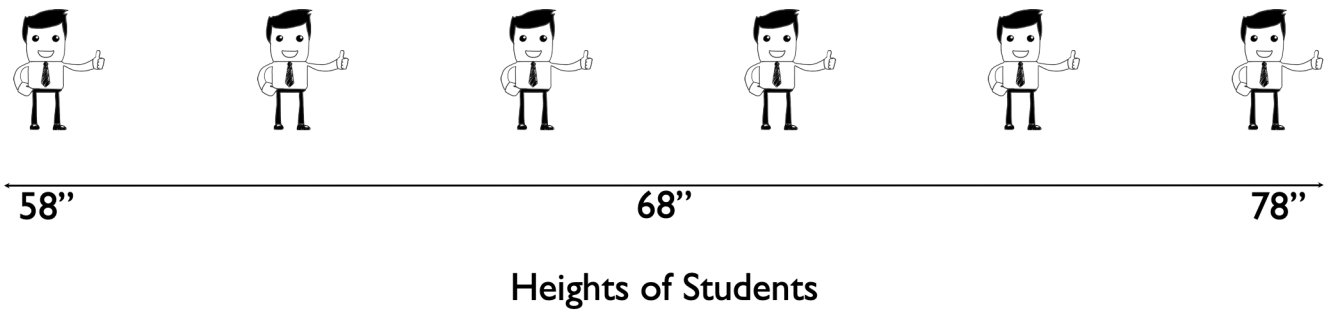


Figure 21.5: Example with people's heights evenly distributed across the range

Or something else entirely?

Variance

The variance is a measure of dispersion that is slightly more complicated to calculate, although not much, but gives us a number we can use to quantify the dispersion of heights around the mean. To do this, let's work through a simple example that only includes six observations: 3 people who are 58 inches tall and 3 people who are 78 inches tall. In this sample of six people from our population the average height is 68 inches.

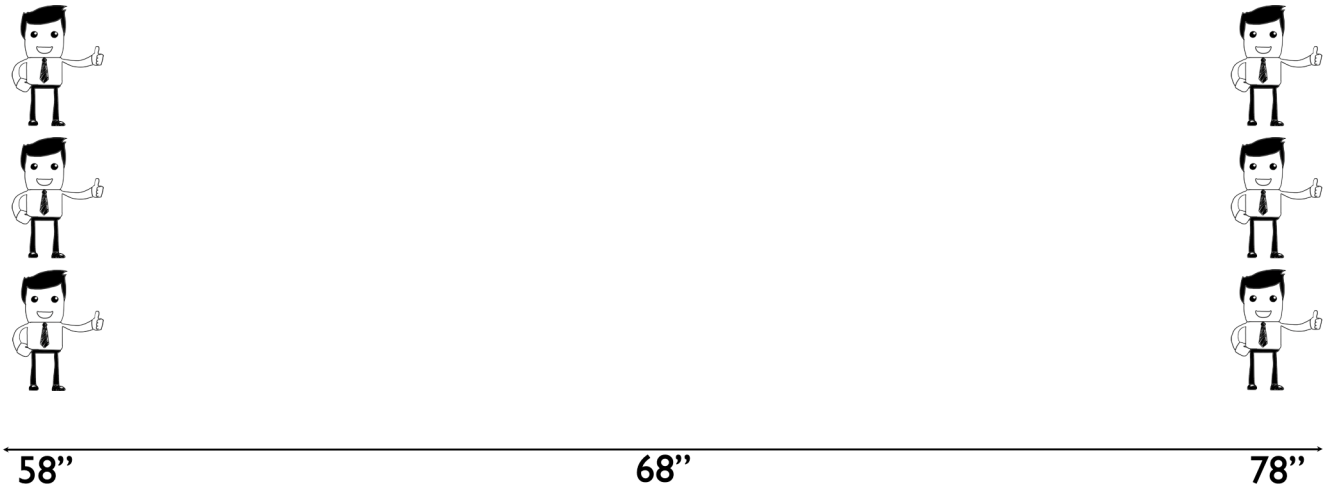


Figure 21.6: Example with people's heights on the edges of the range

Next, let's draw an imaginary line straight up from the mean.

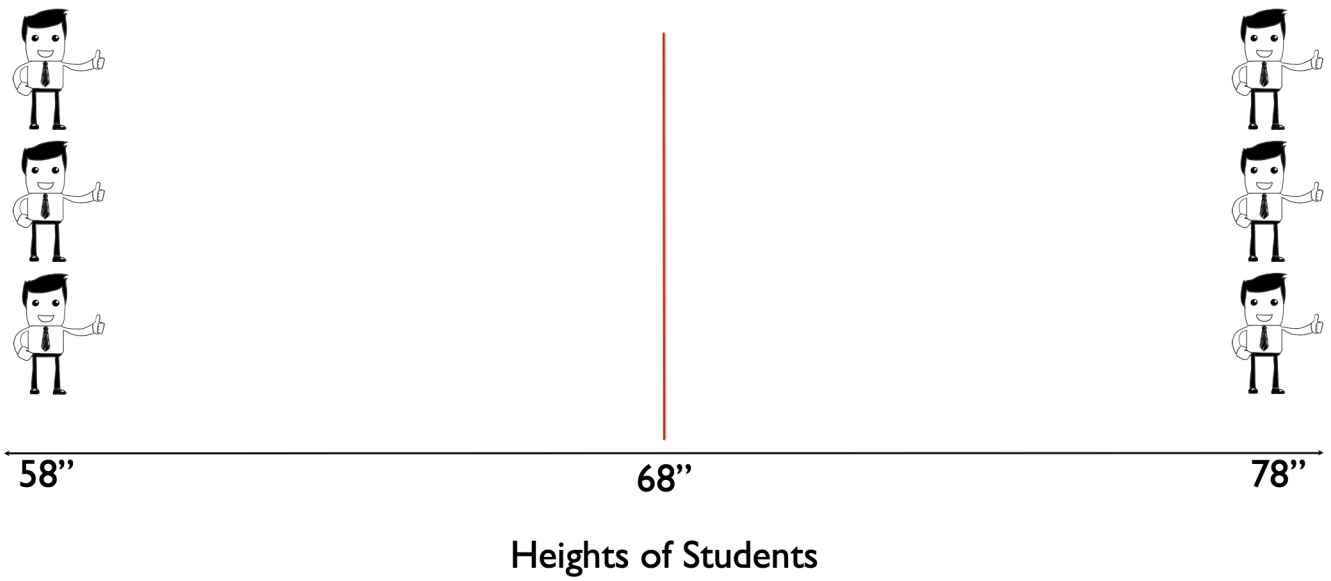


Figure 21.7: Drawing an maginary line at the mean height

Then, let’s measure the difference, or distance, between each person’s height and the mean height.

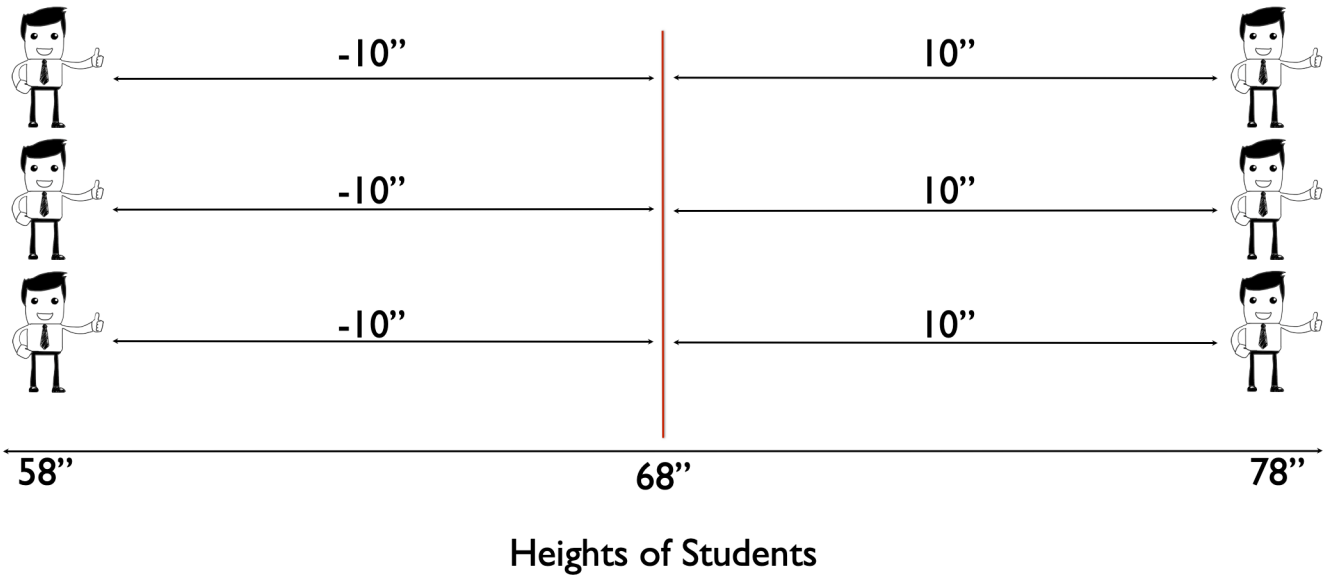


Figure 21.8: Calculating the differences between individual heights and the mean height

Then we square the differences.

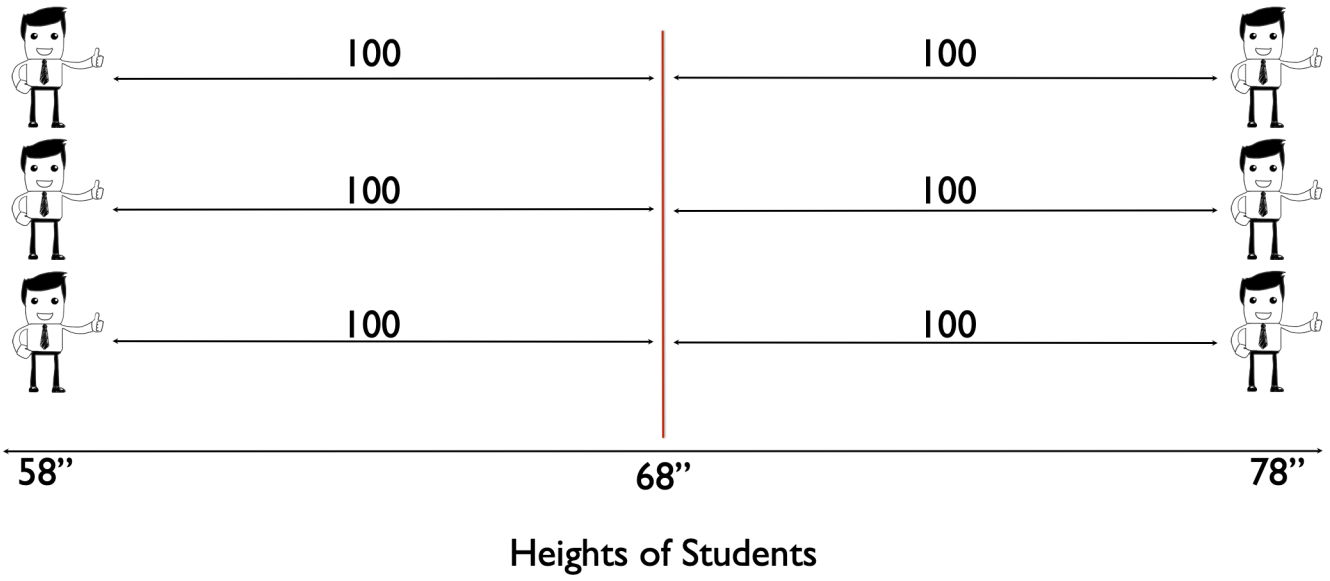


Figure 21.9: Squaring the differences between individual heights and the mean height

Then we add up all the squared differences.

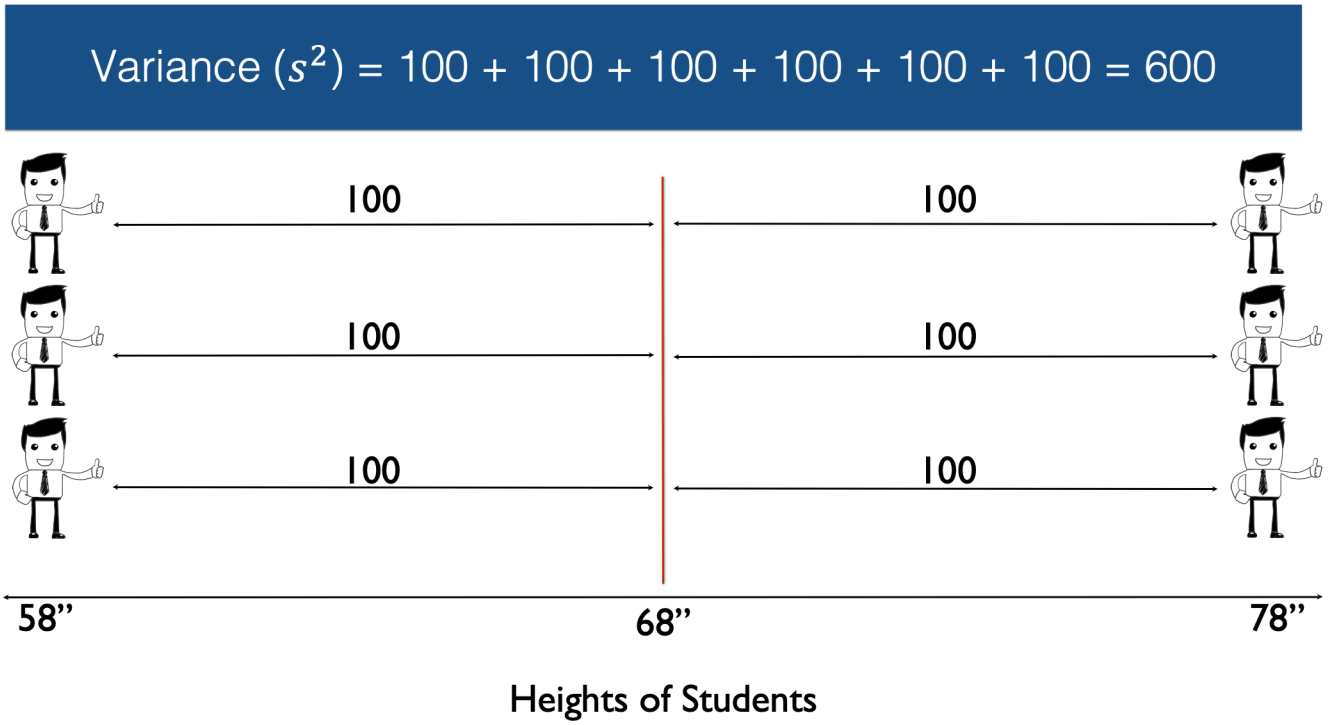


Figure 21.10: Adding the squared differences between individual heights and the mean height

And finally, we divide by n, the number of non-missing observations, minus 1. In this case n equals six, so n-1 equals five.

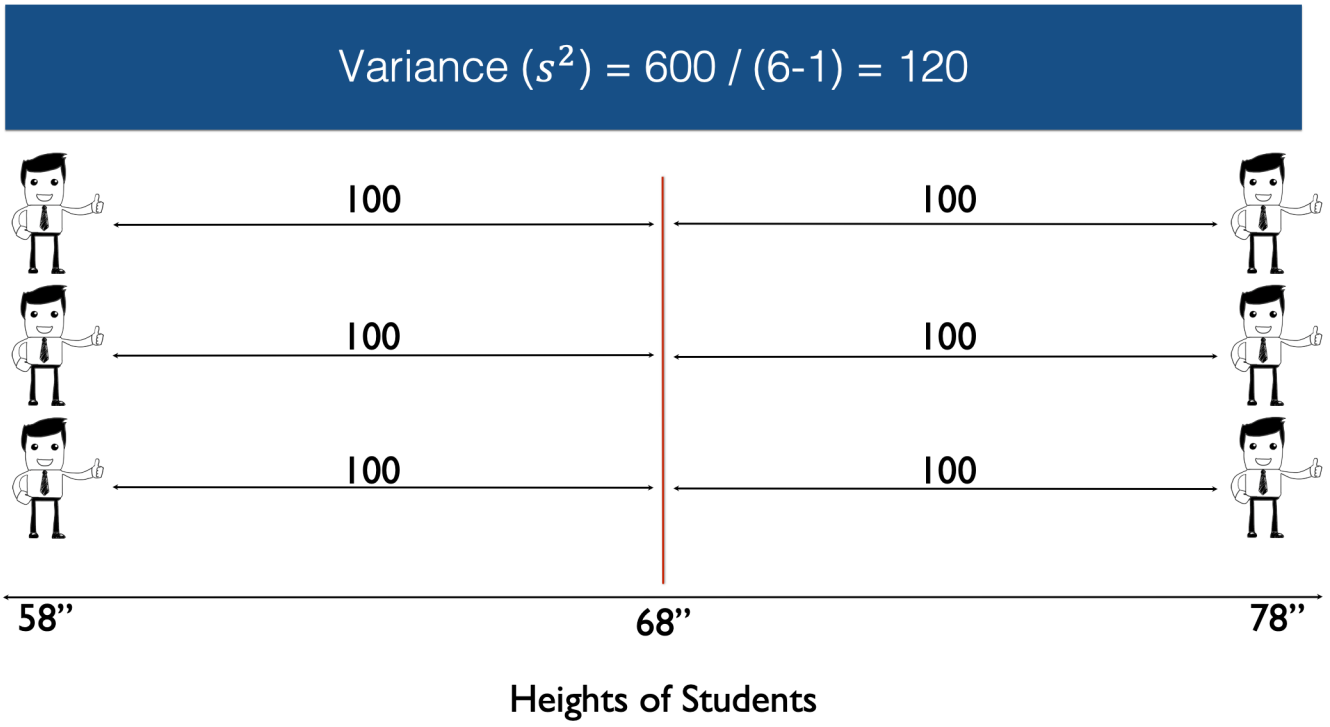


Figure 21.11: Dividing the sum of the squared differences between individual heights and the mean height by n

> **Note**
>
> The sample variance is often written as s^2.

> **Note**
>
> If the 6 observations here represented our entire population of interest, then we could simply divide by n instead of n-1.

Getting R to do this math for us is really straightforward. We simply use base R's `var()` function.

```
var(c(rep(58, 3), rep(78, 3)))
```

```
[1] 120
```

Here's what we did above:

- We created a numeric vector of heights using the `c()` function.
- Instead of typing `c(58, 58, 58, 78, 78, 78)` we used the `rep()` function. `rep(58, 3)` is equivalent to typing `c(58, 58, 58)` and `rep(78, 3)` is equivalent to typing `c(78, 78, 78)`.
- We passed this numeric vector to the `var()` function and R returned the variance – 120

So, 600 divided by 5 equals 120. Therefore, the sample variance in this case is 120. However, because the variance is expressed in squared units, instead of the original units, it isn't necessarily intuitive to interpret.

Standard deviation

If we take the square root of the variance, we get the standard deviation.

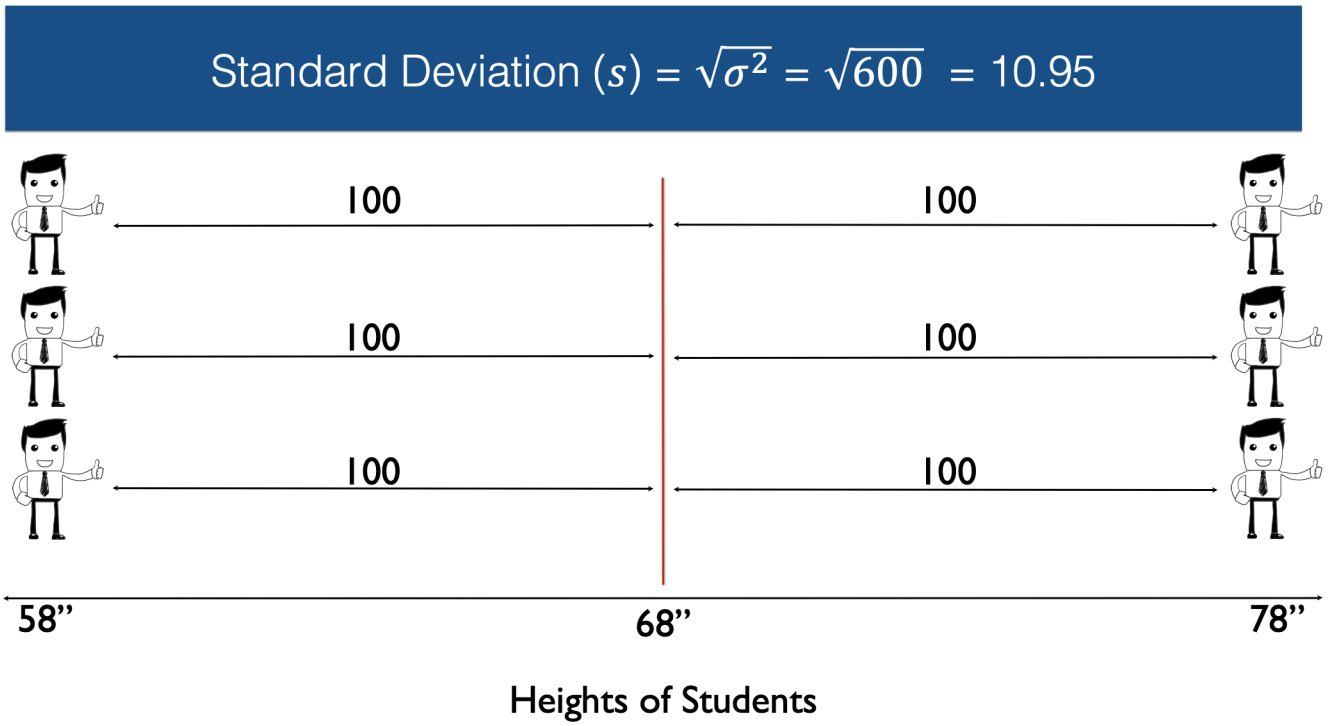


Figure 21.12: Obtaining the standard deviation by taking the square root of the variance

> **Note**
>
> The sample standard deviation is often written as s.

The standard deviation is 10.95 inches, which is much easier to interpret, and compare with other samples. Now that we know the sample standard deviation, we can use it to describe a value's distance from the mean. Additionally, when our data is approximately normally distributed, then the percentage of values within each standard deviation from the mean follow the rules displayed in this table:

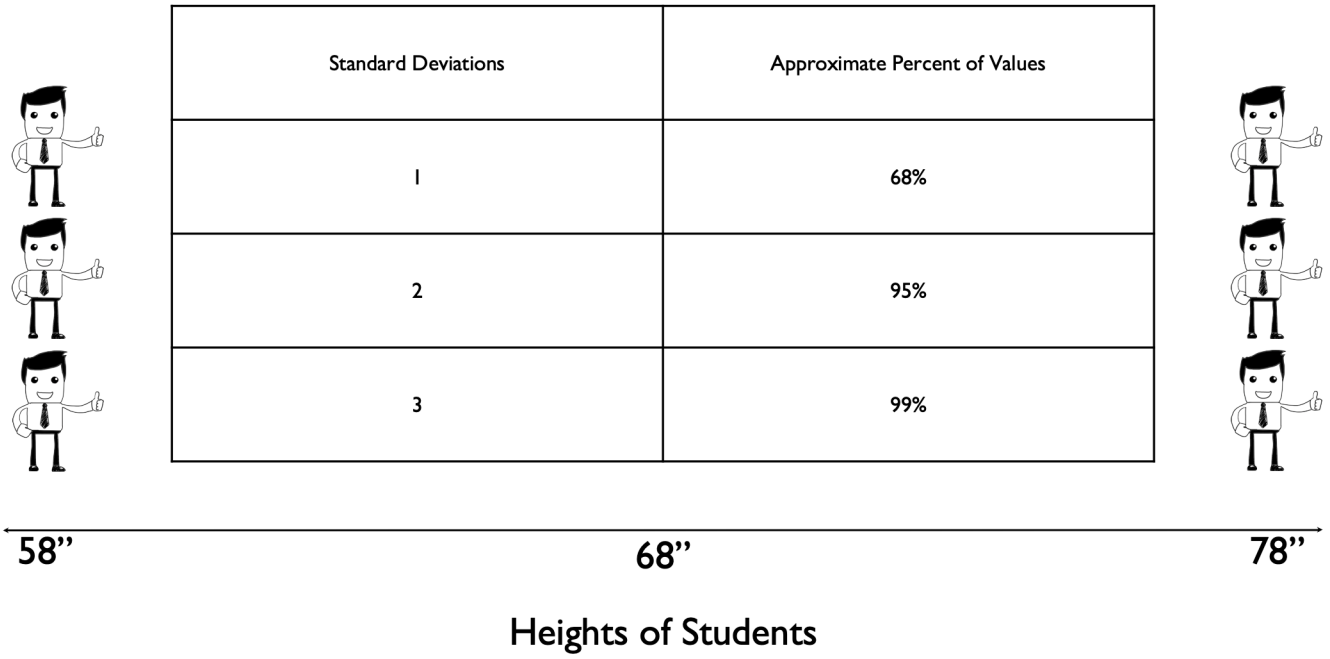
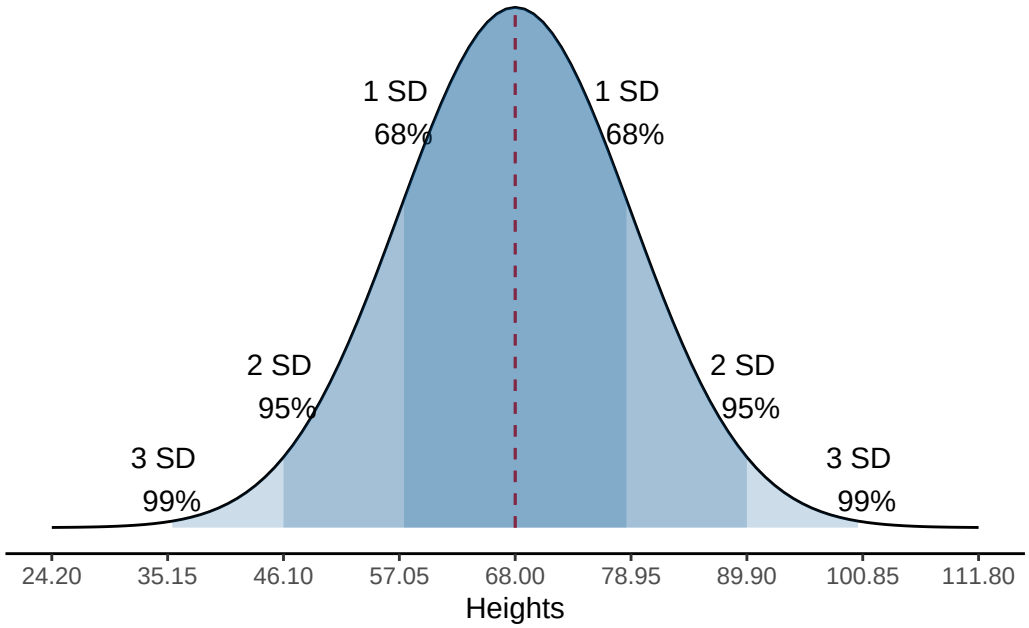


Figure 21.13: 68–95–99.7 rule for approximately normal data

That is, about 68% of all the observations fall within one standard deviation of the mean (that is, 10.95 inches). About 95% of all observations are within 2 standard deviations of the mean (that is, 10.95 * 2 = 21.9 inches), and about 99.9% of all observations are within 3 standard deviations of the mean (that is, 10.95 * 3 = 32.85 inches).

Don't forget that these percentage rules apply to values **around** the mean. In other words, half the values will be greater than the mean and half the values will be lower than the mean. You will often see this graphically illustrated with a "normal curve" or "bell curve."

```
Warning in geom_segment(aes(x = 68, y = 0, xend = 68, yend = peak), color = "red", : All aes
i Please consider using `annotate()` or provide this layer with data containing
  a single row.
```



Unfortunately, the current data is nowhere near normally distributed and does not make for a good example of this rule.

21.1 Comparing distributions

Now that you understand what the different measures of distribution are and how they are calculated, let's further develop your "feel" for interpreting them. We can do this by comparing different simulated distributions.

```
sim_data <- tibble(
  all_68     = rep(68, 20),
  half_58_78 = c(rep(58, 10), rep(78, 10)),
  even_58_78 = seq(from = 58, to = 78, length.out = 20),
  half_48_88 = c(rep(48, 10), rep(88, 10)),
  even_48_88 = seq(from = 48, to = 88, length.out = 20)
)
sim_data
```

```
# A tibble: 20 x 5
   all_68 half_58_78 even_58_78 half_48_88 even_48_88
    <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
 1     68         58       58           48       48
 2     68         58       59.1         48       50.1
 3     68         58       60.1         48       52.2
 4     68         58       61.2         48       54.3
 5     68         58       62.2         48       56.4
 6     68         58       63.3         48       58.5
 7     68         58       64.3         48       60.6
 8     68         58       65.4         48       62.7
 9     68         58       66.4         48       64.8
10     68         58       67.5         48       66.9
11     68         78       68.5         88       69.1
12     68         78       69.6         88       71.2
13     68         78       70.6         88       73.3
14     68         78       71.7         88       75.4
15     68         78       72.7         88       77.5
16     68         78       73.8         88       79.6
17     68         78       74.8         88       81.7
18     68         78       75.9         88       83.8
19     68         78       76.9         88       85.9
20     68         78       78           88       88
```

Here's what we did above:

- We created a data frame with 5 simulated distributions:
  - all_68 has a value of 68 repeated 20 times
  - half_58_78 is made up of the values 58 and 78, each repeated 10 times (similar to our example above)
  - even_58_78 is 20 evenly distributed numbers between 58 and 78
  - half_48_88 is made up of the values 48 and 88, each repeated 10 times
  - even_48_88 is 20 evenly distributed numbers between 48 and 88

We will use this simulated data to quickly demonstrate a couple of these concepts. Let's use R to calculate and compare the mean, variance, and standard deviation of each variable.

```r
tibble(
  Column   = names(sim_data),
  Mean     = purrr::map_dbl(sim_data, mean),
  Variance = purrr::map_dbl(sim_data, var),
  SD       = purrr::map_dbl(sim_data, sd)
)
```

```
# A tibble: 5 x 4
  Column      Mean Variance    SD
  <chr>      <dbl>    <dbl> <dbl>
1 all_68        68      0    0
2 half_58_78    68    105.  10.3
3 even_58_78    68     38.8  6.23
4 half_48_88    68    421.  20.5
5 even_48_88    68    155.  12.5
```

Here's what we did above:

- We created a data frame to hold some summary statistics about each column in the "sim_data" data frame.
- We used the `map_dbl()` function from the `purrr` package to iterate over each column in the data. Don't worry too much about this right now. We will talk more about iteration and the `purrr` package later in the book.

So, for all the columns the mean is 68 inches. And that makes sense, right? We set the middle value and/or most commonly occurring value to be 68 inches for each of these variables. However, the variance and standard deviation are quite different.

For the column "all_68" the variance and standard deviation are both zero. If you think about it, this should make perfect sense: all the values are 68 – they don't vary – and each observations distance from the mean (68) is zero.

When comparing the rest of the columns notice that all of them have a non-zero variance. This is because not all people have the same value in that column – they vary. Additionally, we can see very clearly that variance (and standard deviation) are affected by at least two things:

1. First is the distribution of values across the range of possible values. For example, half_58_78 and half_48_88 have a larger variance than even_58_78 and even_48_88 because all the values are clustered at the min and max - far away from the mean.
2. The second property of the data that is clearly influencing variance is the width of the range of values included in the distribution. For example, even_48_88 has a larger variance and standard deviation than even_58_78, even though both are evenly distributed

across the range of possible values. The reason is because the range of possible values is larger, and therefore the range of distances from the mean is larger too.

In summary, although the variance and standard deviation don't always have a really intuitive meaning all by themselves, we can get some useful information by **comparing** them. Generally speaking, the variance is larger when values are clustered at very low or very high values away from the mean, or when values are spread across a wider range.

22 Describing the Relationship Between a Continuous Outcome and a Continuous Predictor

Before covering anything new, let's quickly review the importance and utility of descriptive analysis.

1. We can use descriptive analysis to uncover errors in our data
2. Descriptive analysis helps us understand the distribution of values in our variables
3. Descriptive analysis serves as a starting point for understanding relationships between our variables

In the first few lessons on descriptive analysis we covered performing univariate analysis. That is, analyzing a single numerical or a single categorical variable. In this module, we'll learn methods for describing *relationships between* two variables. This is also called bivariate analysis.

For example, we may be interested in knowing if there is a relationship between heart rate and exercise. If so, we may ask ourselves if heart rate differs, on average, by daily minutes of exercise. And, we could answer that question with the using a bivariate descriptive analysis.

Before performing any such bivariate descriptive analysis, you should ask yourself what types of variables you will analyze. We've already discussed the difference between numerical variables and categorical variables, but we will also need to decide whether each variable is an outcome or a predictor.

1. **Outcome variable:** The variable whose value we are attempting to predict, estimate, or determine is the outcome variable. The outcome variable may also be referred to as the dependent variable or the response variable.
2. **Predictor variable:** The variable that we think will determine, or at least help us predict, the value of the outcome variable is called the predictor variable. The predictor variable may also be referred to as the independent variable or the explanatory variable.

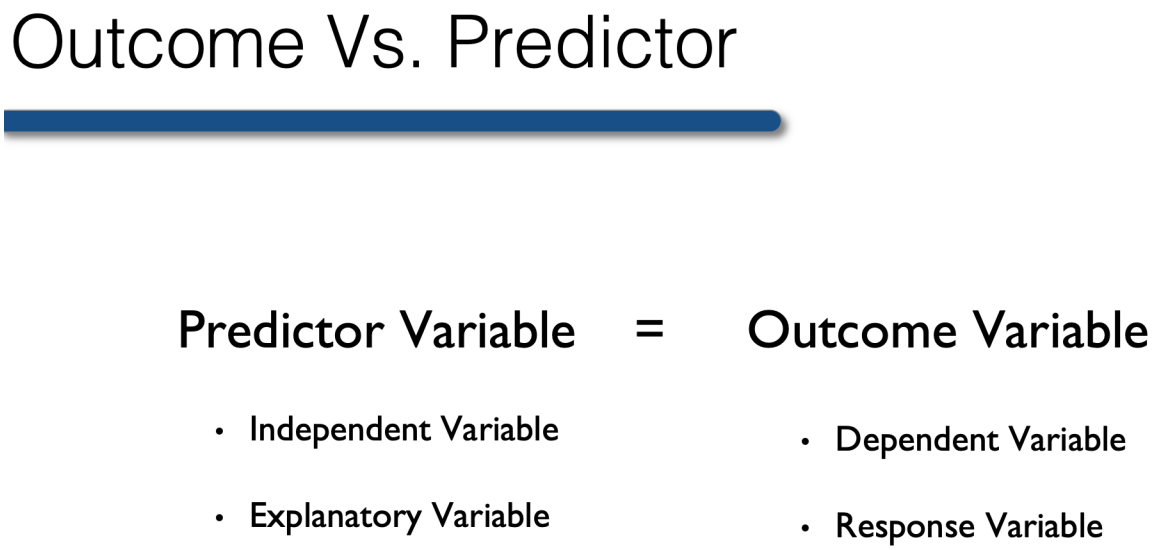


Figure 22.1: Comparing outcome and predictor variables

So, think back to our interest in whether or not heart rate differs by daily minutes of exercise. In this scenario, which variable is the predictor and which is the outcome?

In this scenario daily minutes of exercise is the predictor and heart rate is the outcome.

Heart rate is the variable we're interested in predicting or understanding, and exercise is a variable that we think helps to predict or explain heart rate.

In this first chapter on bivariate analysis, we will learn a simple method for describing the relationship between a continuous outcome variable and a continuous predictor variable – the Pearson Correlation Coefficient.

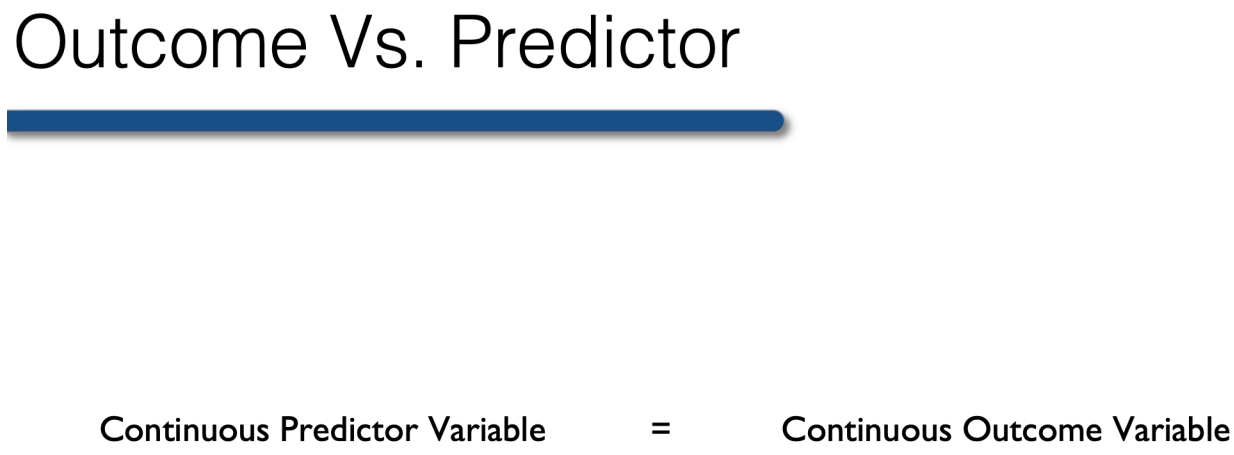


Figure 22.2: Describing the relationship between outcome and predictor variables

22.1 Pearson Correlation Coefficient

Pearson's Correlation Coefficient is a parametric measure of the *linear* relationship between two numerical variables. It's also referred to as rho (pronounced like "row") and can be written shorthand as a lowercase r. The Pearson Correlation Coefficient can take on values between -1 and 1, including zero.

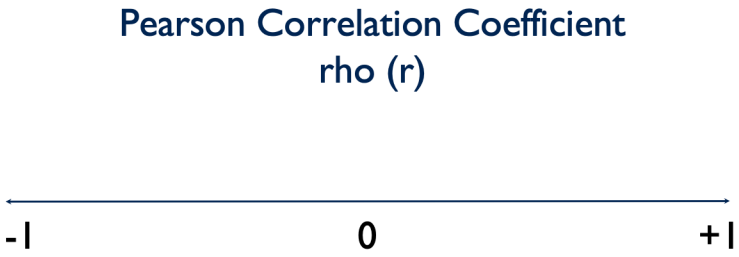


Figure 22.3: Pearson's correlation coefficient range of values

A value of 0 indicates that there is no linear correlation between the two variables.

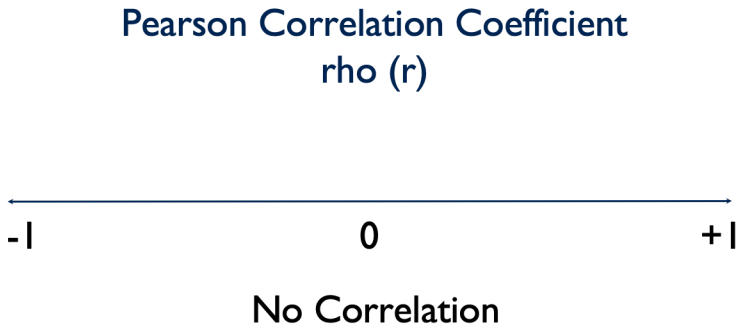


Figure 22.4: Pearson's correlation coefficient value of 0

A negative value indicates that there is a negative linear correlation between the two variables. In other words, as the value of x increases, the value of y decreases. Or, as the value of x decreases, the value of y increases.

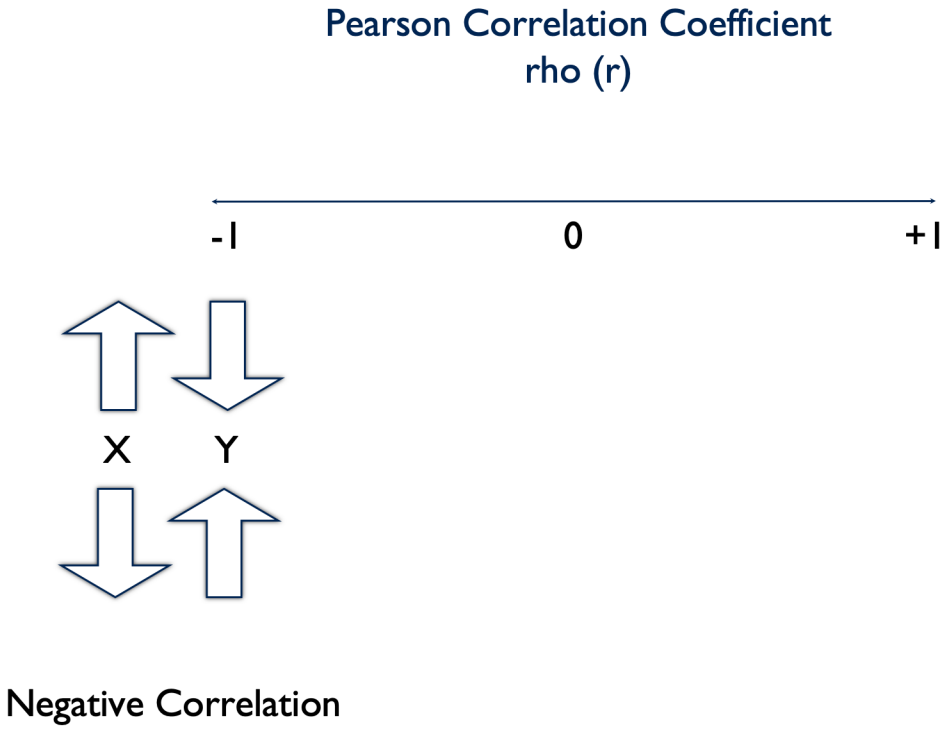


Figure 22.5: Negative Pearson's correlation coefficient values

A positive value indicates that there is a positive linear correlation between the two variables. As the value of x increases, the value of y increases. Or as the value of x decreases, the value of y decreases.

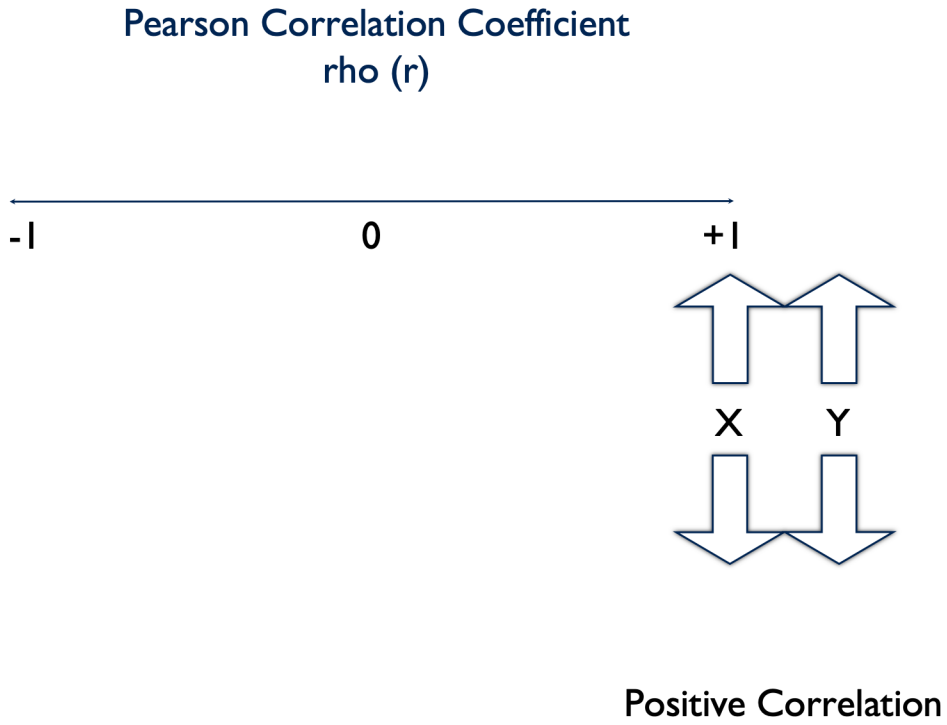


Figure 22.6: Positive Pearson's correlation coefficient values

> **Warning**
>
> When the relationship between two variables is nonlinear, or when outliers are present, the correlation coefficient might incorrectly estimate the strength of the relationship. Plotting the data enables you to verify the linear relationship and to identify the potential outliers.

22.1.1 Calculating r

In this first code chunk, we're going to use some simple simulated data to develop an intuition about describing the relationship between two continuous variables.

```
# Load the dplyr package
library(dplyr)
# Load the ggplot2 package
library(ggplot2)
```

```r
set.seed(123)
df <- tibble(
  id = 1:20,
  x  = sample(x = 0:100, size = 20, replace = TRUE),
  y  = sample(x = 0:100, size = 20, replace = TRUE)
)
df
```

```
# A tibble: 20 x 3
      id     x     y
   <int> <int> <int>
 1     1    30    71
 2     2    78    25
 3     3    50     6
 4     4    13    41
 5     5    66     8
 6     6    41    82
 7     7    49    35
 8     8    42    77
 9     9   100    80
10    10    13    42
11    11    24    75
12    12    89    14
13    13    90    31
14    14    68     6
15    15    90     8
16    16    56    40
17    17    91    73
18    18     8    22
19    19    92    26
20    20    98    59
```

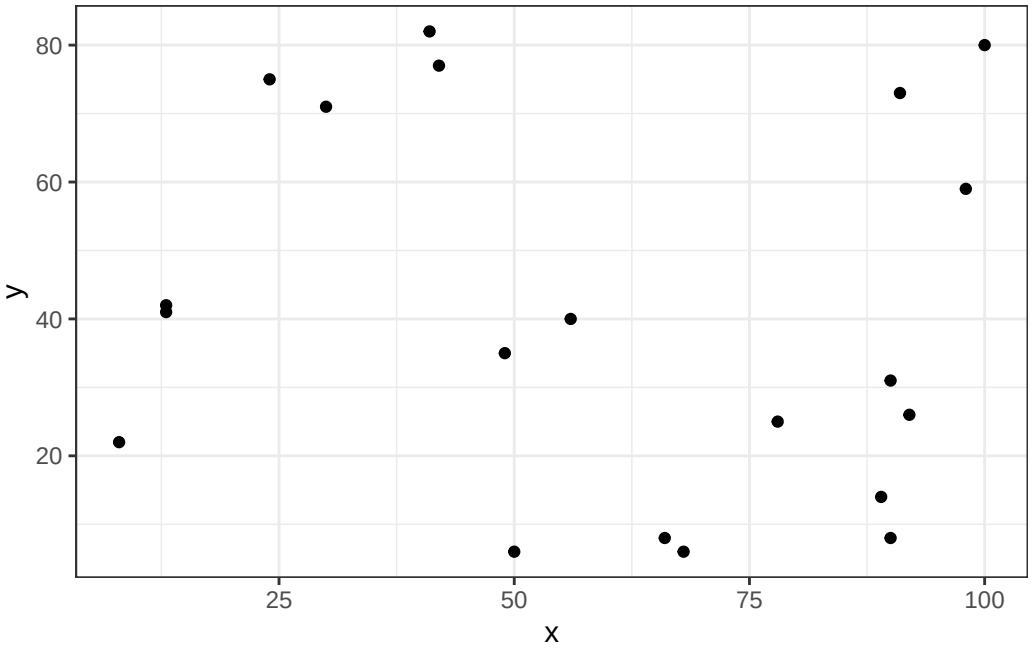
Here's what we did above:

- We created a data frame with 3 simulated variables – id, x, and y.
- We used the `sample()` function to create x and y by sampling a number between 0 and 100 at random, 20 times.
- The `replace = TRUE` option tells R that the same number can be selected more than once.
- The `set.seed()` function is to ensure that we get the same random numbers every time we run the code chunk.

There is nothing special about 0 and 100; they are totally arbitrary. But, because all of these values are chosen at random, we have no reason to believe that there should be any relationship between them. Accordingly, we should also expect the Pearson Correlation Coefficient to be 0 (or very close to it).

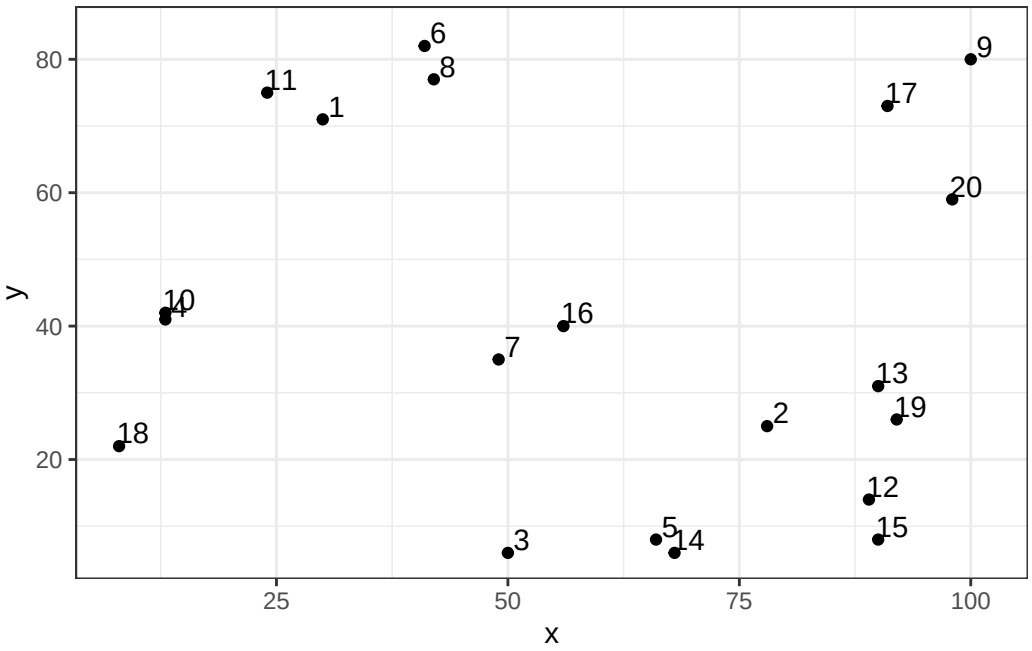
In order to develop an intuition, let's first plot this data, and get a feel for what it looks like.

```
ggplot(df, aes(x, y)) +
  geom_point() +
  theme_bw()
```



Above, we've created a nice scatter plot using `ggplot2()`. But, how do we interpret it? Well, each dot corresponds to a person in our data at the point where their x value intersects with their y value. This is made clearer by adding a `geom_text()` layer to our plot.

```
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_text(aes(label = id), nudge_x = 1.5, nudge_y = 2) +
  theme_bw()
```



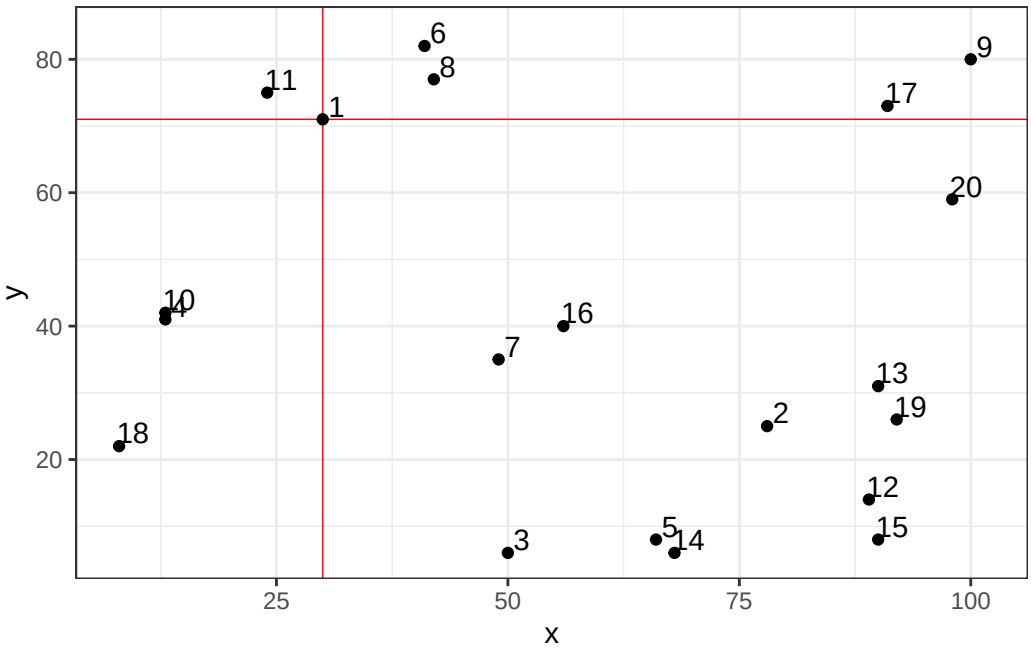
Here's what we did above:

- We added a `geom_text()` layer to our plot in order to make it clear which person each dot represents.
- The `nudge_x = 1.5` option moves our text (the id number) to the right 1.5 units. The `nudge_y = 2` option moves our text 2 units up. We did this to make the id number easier to read. If we had not nudged them, they would have been placed directly on top of the points.

For example, person 1 in our simulated data had an x value of 30 and a y value of 71. When you look at the plot above, does it look like person 1's point is approximately at (x = 30, y = 71)? If we want to emphasize the point even further, we can plot a vertical line at x = 30 and a horizontal line at y = 71. Let's do that below.

```
ggplot(df, aes(x, y)) +
  geom_text(aes(label = id), nudge_x = 1.5, nudge_y = 2) +
  geom_vline(xintercept = 30, col = "red", size = 0.25) +
  geom_hline(yintercept = 71, col = "red", size = 0.25) +
  geom_point() +
  theme_bw()
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```



As you can see, the dot representing id 1 is at the intersection of these two lines.

So, we know how to read the plot now, but we still don't really know anything about the *relationship* between x and y. Remember, we want to be able to characterize x and y as having one of these 5 relationships:

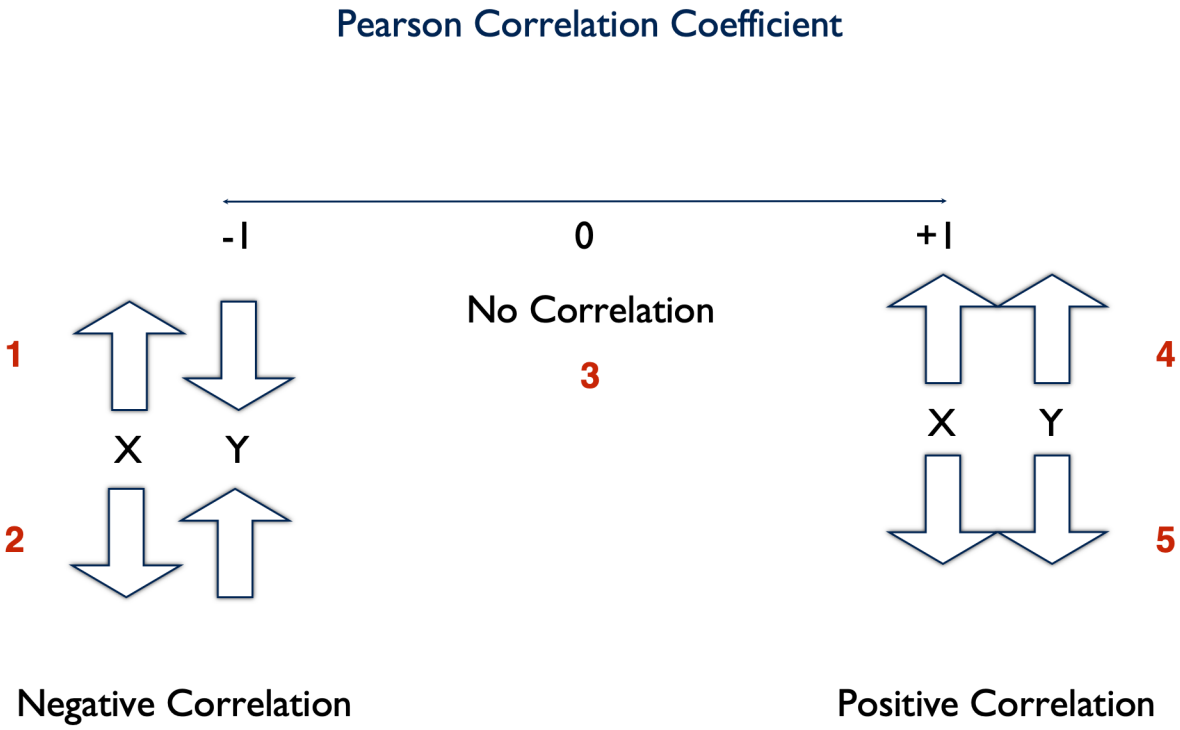
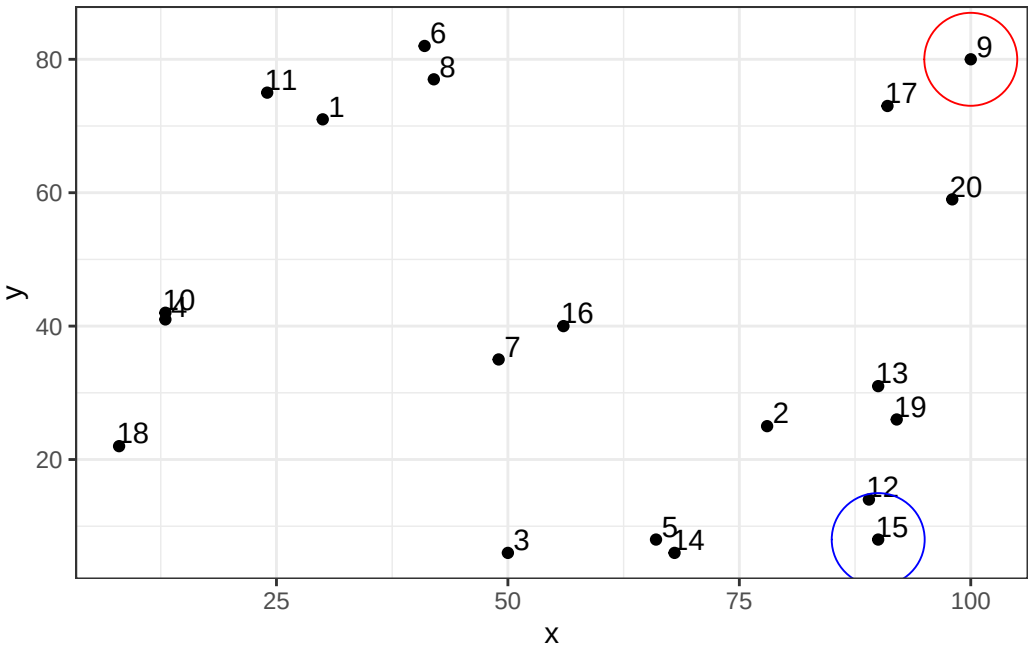


Figure 22.7: Relationship between the outcome and the predictor

Looking again at our scatter plot, which relationship do you think x and y have?

```
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_text(aes(label = id), nudge_x = 1.5, nudge_y = 2) +
  geom_point(aes(x, y), tibble(x = 100, y = 80), shape = 1, size = 16, col = "red") +
  geom_point(aes(x, y), tibble(x = 90, y = 8), shape = 1, size = 16, col = "blue") +
  theme_bw()
```



Well, if you look at id 9 above, x is a high number (100) and y is a high number (80). But if you look at id 15, x is a high number (90) and y is a low number (8). In other words, these dots are scattered all over the chart area. There doesn't appear to be much of a pattern, trend, or relationship. And that's exactly what we would expect from randomly generated data.

Now that we know what this data looks like, and we intuitively feel as though x and y are unrelated, it would be nice to quantify our results in some way. And, that is precisely what the Pearson Correlation Coefficient does.

```
cor.test(x = df$x, y = df$y)
```

```
    Pearson's product-moment correlation

data:  df$x and df$y
t = -0.60281, df = 18, p-value = 0.5542
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.5490152  0.3218878
sample estimates:
       cor
-0.1406703
```

Here's what we did above:

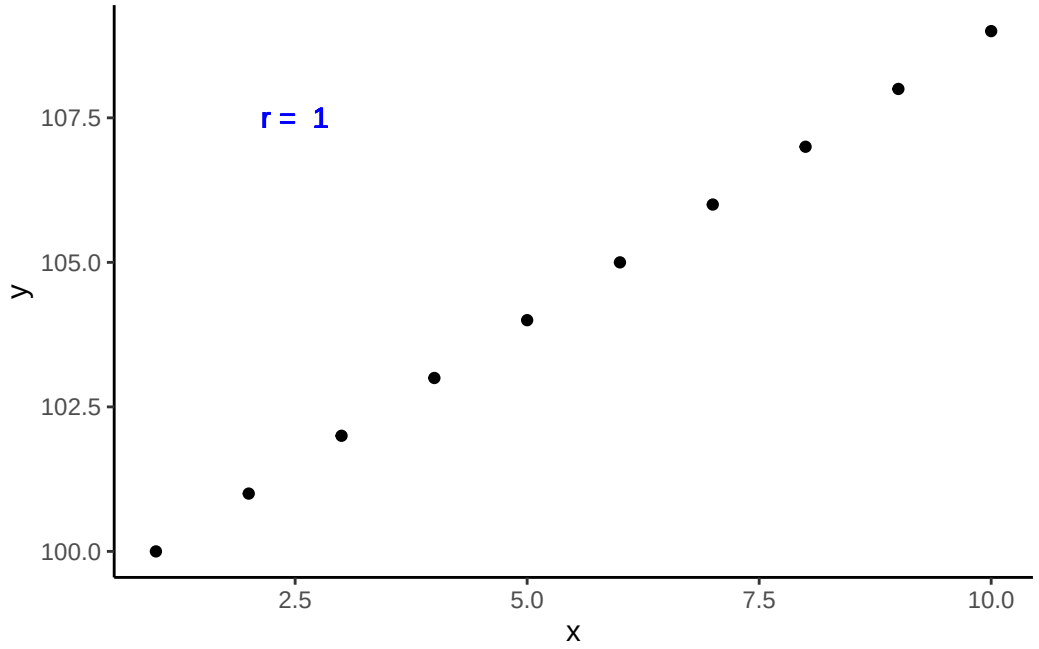
- By default, R's `cor.test()` function gives us a list of information about the relationship between x and y. The very last number in the output (-0.1406703) is the Pearson Correlation Coefficient.
- The fact that this value is negative (between -1 and 0) tells us that x and y tend to vary in opposite directions.
- The numeric value (0.1406703) tells us something about the strength of the relationship between x and y. In this case, the relationship is not strong – exactly what we expected.
  - You will sometimes hear rules of thumb for interpreting the strength of r such as[7]:
    - ± 0.1 = Weak correlation
    - ± 0.3 = Medium correlation
    - ± 0.5 = Strong correlation
  - Rules of thumb like this are useful as you are learning; however, you want to make sure you don't become overly reliant on them. As you get more experience, you will want to start interpreting effect sizes in the context of your data and the specific research question at hand.
- The p-value (0.5542) tells us that we'd be pretty likely to get the result we got even if there really were no relationship between x and y – **assuming all other assumptions are satisfied and the sample was collected without bias.**
- Taken together, the weak negative correlation and p-value tell us that there is not much – if any – relationship between x and y. Another way to say the same thing is, "x and y are statistically independent."

22.1.2 Correlation intuition

To further bolster our intuition about these relationships, let's look at a few positively and negatively correlated variables.

```
# Positively correlated data
tibble(
  x = 1:10,
  y = 100:109,
  r = cor(x, y)
) %>%
  ggplot() +
    geom_point(aes(x, y)) +
```

```
    geom_text(aes(x = 2.5, y = 107.5, label = paste("r = ", r)), col = "blue") +
    theme_classic()
```



Above, we created positively correlated data. In fact, this data is perfectly positively correlated. That is, every time the value of x increases, the value of y increases by a proportional amount. Now, instead of being randomly scattered around the plot area, the dots line up in a perfect, upward-sloping, diagonal line. We also added the correlation coefficient directly to the plot. As you can see, it is exactly 1. This is what you should expect from perfectly positively correlated data.

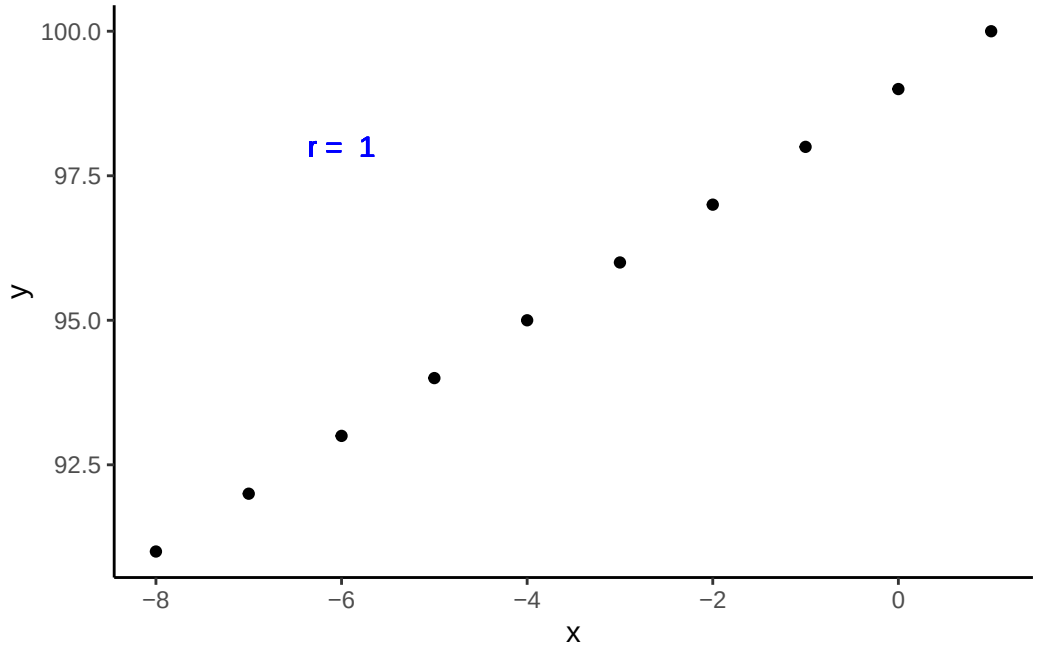
How about this next data set? Now, every time x decreases by one, y decreases by one. Is this positively or negatively correlated data?

```
df <- tibble(
  x = 1:-8,
  y = 100:91
)
df
```

```
# A tibble: 10 x 2
       x     y
   <int> <int>
 1     1   100
```

```
 2     0    99
 3    -1    98
 4    -2    97
 5    -3    96
 6    -4    95
 7    -5    94
 8    -6    93
 9    -7    92
10    -8    91
```

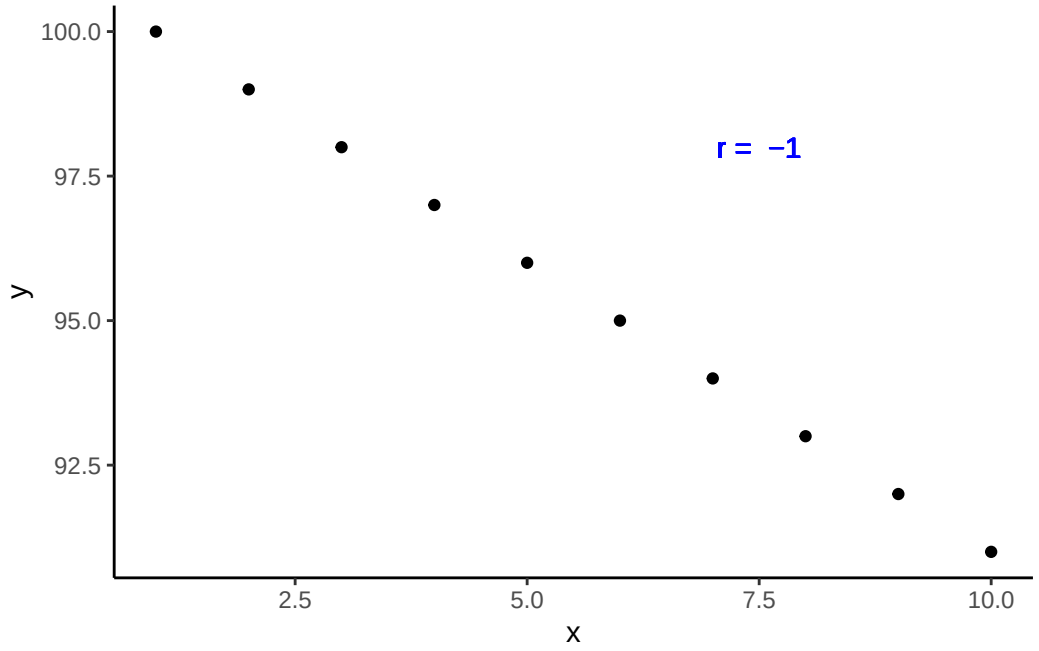
```
df %>%
  mutate(r = cor(x, y)) %>%
  ggplot() +
      geom_point(aes(x, y)) +
      geom_text(aes(x = -6, y = 98, label = paste("r = ", r)), col = "blue") +
      theme_classic()
```



This is still perfectly positively correlated data. The values for x and y are still changing in the same direction proportionately. The fact that the direction is one of decreasing value makes no difference.

One last simulated example here. This time, as x increases by one, y decreases by one. Let's plot this data and calculate the Pearson Correlation Coefficient.

```
tibble(
  x = 1:10,
  y = 100:91,
  r = cor(x, y)
) %>%
  ggplot() +
    geom_point(aes(x, y)) +
    geom_text(aes(x = 7.5, y = 98, label = paste("r = ", r)), col = "blue") +
    theme_classic()
```



This is what perfectly negatively correlated data looks like. The dots line up in a perfect, downward-sloping diagonal line, and when we check the value of rho, we see that it is exactly -1.

Of course, as you may have suspected, *in real life things are almost never this cut and dry.* So, let's investigate the relationship between continuous variables using more realistic data.

In this example, we will use data from an actual class survey conducted in the past:

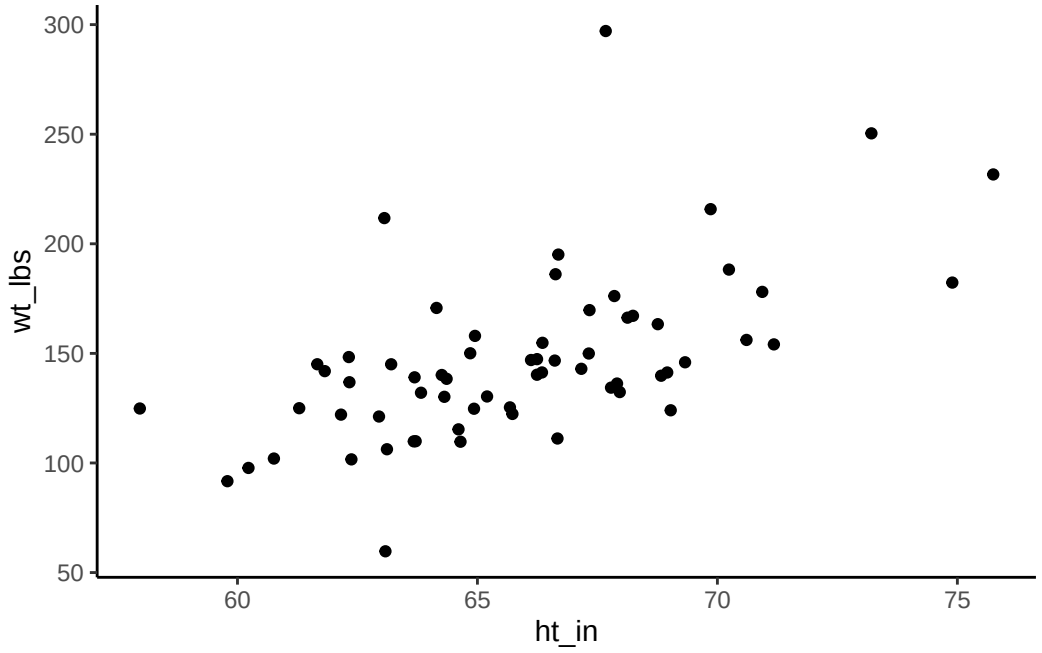
```
class <- tibble(
  ht_in = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
            64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
            64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
```

```
            69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
            61, 69, 66, NA),
  wt_lbs = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
             125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
             186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
             147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
             110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
             163, 141, NA)
)
```

Next, we're going to use a scatter plot to explore the relationship between height and weight in this data.

```
ggplot(class, aes(ht_in, wt_lbs)) +
  geom_jitter() +
  theme_classic()
```

```
Warning: Removed 4 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Quickly, what do you think? Will height and weight be positively correlated, negatively correlated, or not correlated?

```
cor.test(class$ht_in, class$wt_lbs)
```

```
    Pearson's product-moment correlation

data:  class$ht_in and class$wt_lbs
t = 5.7398, df = 62, p-value = 3.051e-07
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.4013642 0.7292714
sample estimates:
      cor
0.5890576
```

The dots don't line up in a perfectly upward – or downward – slope. But the general trend is still an upward slope. Additionally, we can see that height and weight are positively correlated because the value of the correlation coefficient is between 0 and positive 1 (0.5890576). By looking at the p-value (3.051e-07), we can also see that the probability of finding a correlation value this large or larger in our sample if the true value of the correlation coefficient in the population from which our sample was drawn is zero, is very small.

That's quite a mouthful, right? In more relatable terms, you can just think of it this way. In our data, as height increases weight tends to increase as well. Our p-value indicates that it's pretty unlikely that we would get this result if there were truly no relationship in the population this sample was drawn from – assuming it's an unbiased sample.

Quick detour: The p-value above is written in scientific notation, which you may not have seen before. We'll quickly show you how to basically disable scientific notation in R.

```
options(scipen = 999)
cor.test(class$ht_in, class$wt_lbs)
```

```
    Pearson's product-moment correlation

data:  class$ht_in and class$wt_lbs
t = 5.7398, df = 62, p-value = 0.0000003051
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.4013642 0.7292714
sample estimates:
      cor
0.5890576
```

Here's what we did above:

- We used the R global option `options(scipen = 999)` to display decimal numbers instead of scientific notation. Because this is a global option, it will remain in effect until you restart your R session. If you do restart your R session, you will have to run `options(scipen = 999)` again to disable scientific notation.

Finally, wouldn't it be nice if we could draw a line through this graph that sort of quickly summarizes this relationship (or lack thereof). Well, that is exactly what an Ordinary Least Squares (OLS) regression line does.

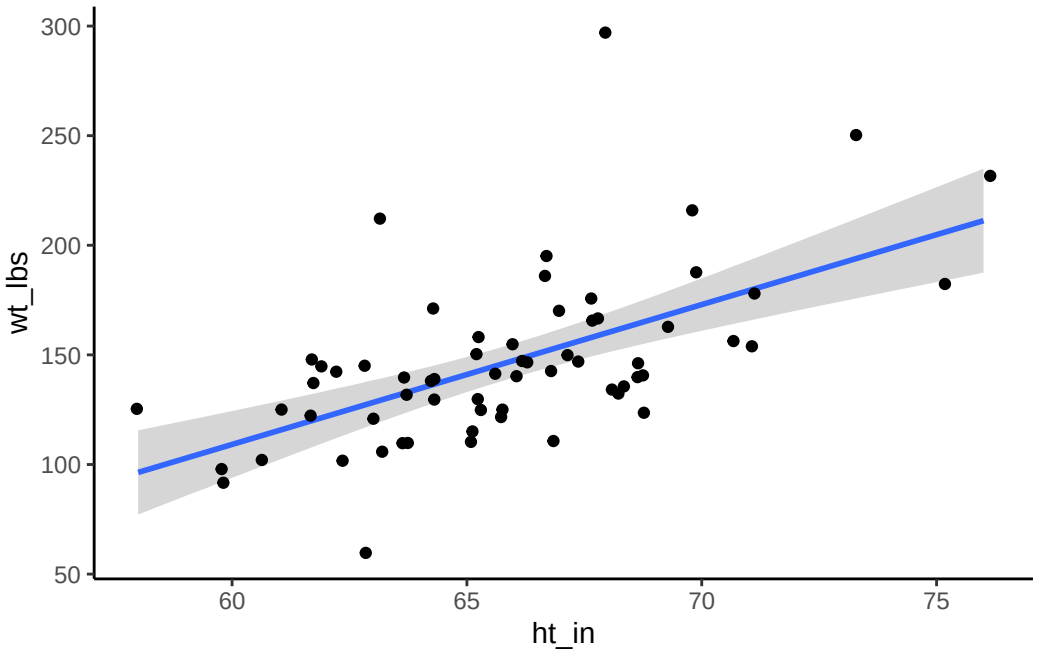
To add a regression line to our plot, all we need to do is add a `geom_smooth()` layer to our scatterplot with the `method` argument set to `lm`. Let's do that below and take a look.

```
ggplot(class, aes(ht_in, wt_lbs)) +
  geom_smooth(method = "lm") +
  geom_jitter() +
  theme_classic()
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 4 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 4 rows containing missing values or values outside the scale range
(`geom_point()`).
```



The exact calculation for deriving this line is beyond the scope of this chapter. In general, though, you can think of the line as cutting through the middle of all of your points and representing the average change in the y value given a one-unit change in the x value. So here, the upward slope indicates that, on average, as height (the x value) increases, so does weight (the y value). And that is completely consistent with our previous conclusions about the relationship between height and weight.

23 Describing the Relationship Between a Continuous Outcome and a Categorical Predictor

Up until now, we have only ever looked at the overall mean of a continuous variable. For example, the mean height for the entire class. However, we often want to estimate the means within levels, or categories, of another variable. For example, we may want to look at the mean height within gender. Said another way, we want to know the mean height for men and separately the mean height for women.

More generally, in this lesson you will learn to perform bivariate analysis when the outcome is continuous and the predictor is categorical.

Outcome Vs. Predictor

Figure 23.1: Continuous outcome and categorical predictor

Typically in a situation such as this, all we need to do is apply the analytic methods we've already learned for a single continuous outcome, but apply them separately within levels of our categorical predictor variable. Below, we'll walk through doing so with R. To start with, we will again use our previously collected class survey data.

```
library(dplyr)
library(ggplot2)
```

```
class <- tibble(
  age       = c(32, 30, 32, 29, 24, 38, 25, 24, 48, 29, 22, 29, 24, 28, 24, 25,
                25, 22, 25, 24, 25, 24, 23, 24, 31, 24, 29, 24, 22, 23, 26, 23,
                24, 25, 24, 33, 27, 25, 26, 26, 26, 26, 26, 27, 24, 43, 25, 24,
                27, 28, 29, 24, 26, 28, 25, 24, 26, 24, 26, 31, 24, 26, 31, 34,
                26, 25, 27, NA),
  age_group = c(2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2,
```

```
                2, 1, 1, 1, NA),
  gender    = c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1,
                1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1,
                1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 1,
                1, 1, 2, 1, NA),
  ht_in     = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
                64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
                64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
                69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
                61, 69, 66, NA),
  wt_lbs    = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
                125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
                186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
                147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
                110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
                163, 141, NA),
  bmi       = c(30.99, 18.78, 26.52, 30.54, 22.39, 26.12, 23.69, 20.67, 26.29,
                25.39, 25.68, 45.15, 21.56, 20.17, 17.38, 20.8, 22.31, 22.75,
                26.62, 21.43, 19.14, 23.49, 22.66, 32.98, 25.05, 18.31, 29.13,
                27.07, 20.37, 25.01, 19.69, 25.97, 18.88, 20.07, NA, 26.76,
                26.97, 25.24, 20.68, 23.72, 24.82, 23.62, 18.65, 24.03, 23.86,
                10.63, 23.02, 23.72, 20.82, 28.24, NA, 37.55, 18.88, 18.3,
                19.13, 21.48, 22.59, 24.96, 21.63, NA, 29.35, 21.76, 17.97,
                22.31, 19.27, 24.07, 22.76, NA),
  bmi_3cat  = c(3, 1, 2, 3, 1, 2, 1, 1, 2, 2, 2, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1,
                1, 1, 3, 2, 1, 2, 2, 1, 2, 1, 2, 1, 1, NA, 2, 2, 2, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 2, NA, 3, 1, 1, 1, 1, 1, 1, 1, NA, 2, 1,
                1, 1, 1, 1, 1, NA)
) %>%
  mutate(
    age_group = factor(age_group, labels = c("Younger than 30", "30 and Older")),
    gender = factor(gender, labels = c("Female", "Male")),
    bmi_3cat = factor(bmi_3cat, labels = c("Normal", "Overweight", "Obese"))
  ) %>%
  print()
```

```
# A tibble: 68 x 7
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese
 2    30 30 and Older    Female    63    106  18.8 Normal
 3    32 30 and Older    Female    62    145  26.5 Overweight
```

```
 4    29 Younger than 30 Male      67    195  30.5 Obese
 5    24 Younger than 30 Female    67    143  22.4 Normal
 6    38 30 and Older    Female    58    125  26.1 Overweight
 7    25 Younger than 30 Female    64    138  23.7 Normal
 8    24 Younger than 30 Male      69    140  20.7 Normal
 9    48 30 and Older    Male      65    158  26.3 Overweight
10    29 Younger than 30 Male      68    167  25.4 Overweight
# i 58 more rows
```

23.1 Single predictor and single outcome

We can describe our continuous outcome variables using the same methods we learned in previous lessons. However, this time we will use `dplyr's group_by()` function to calculate these statistics within subgroups of interests. For example:

```
class_summary <- class %>%
  filter(!is.na(ht_in)) %>%
  group_by(gender) %>%
  summarise(
    n                    = n(),
    mean                 = mean(ht_in),
    `standard deviation` = sd(ht_in),
    min                  = min(ht_in),
    max                  = max(ht_in)
  ) %>%
  print()
```

```
# A tibble: 2 x 6
  gender     n  mean `standard deviation`   min   max
  <fct>  <int> <dbl>                <dbl> <dbl> <dbl>
1 Female    43  64.3                 2.59    58    69
2 Male      22  69.2                 2.89    65    76
```

Here's what we did above:

- We used base R's statistical functions inside `dplyr's summarise()` function to calculate the number of observations, mean, standard deviation, minimum value and maximum value of height within levels of gender.

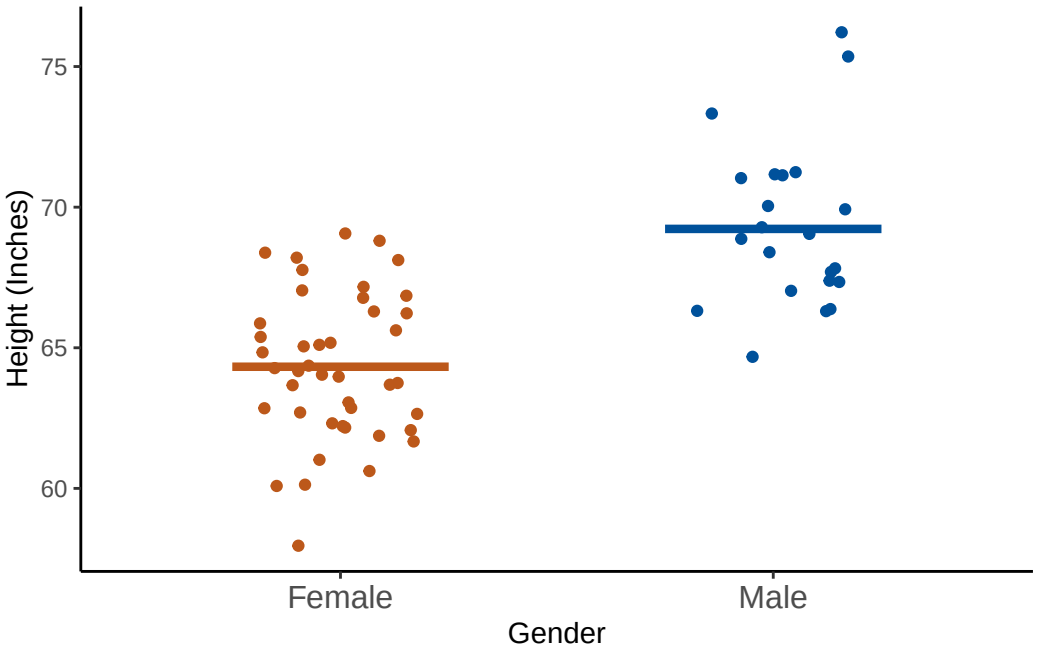
- We used `filter(!is.na(ht_in))` to remove all rows from the data that have a missing value for "ht_in". If we had not done so, R would have returned a value of "NA" for mean, standard deviation, min, and max. Alternatively, we could have added the `na.rm = TRUE` option to each of the `mean()`, `sd()`, `min()`, and `max()` functions.
- We used `group_by(gender)` to calculate our statistics of interest separately within each category of the variable "gender." In this case, "Female" and "Male."
- You may notice that we used back ticks around the variable name "standard deviation" – NOT single quotes. If you want to include a space in a variable name in R, you must surround it with back ticks. In general, it's a **really** bad idea to create variable names with spaces in them. It is recommend that you only do so in situations where you are using a data frame to display summary information, as we did above.
- Notice too that we saved our summary statistics table as data frame named "class_summary." Doing so is sometimes useful, especially for plotting as we will see below.

As you look over this table, you should have an idea of whether male or female students in the class appear to be taller on average, and whether male or female students in the class appear to have more dispersion around the mean value.

Finally, let's plot this data to get a feel for the relationship between gender and height graphically.

```
class %>%
  filter(!is.na(ht_in)) %>%
  ggplot(aes(x = gender, y = ht_in)) +
    geom_jitter(aes(col = gender), width = 0.20) +
    geom_segment(
      aes(x = c(0.75, 1.75), y = mean, xend = c(1.25, 2.25), yend = mean, col = gender),
      size = 1.5, data = class_summary
    ) +
    scale_x_discrete("Gender") +
    scale_y_continuous("Height (Inches)") +
    scale_color_manual(values = c("#BC581A", "#00519B")) +
    theme_classic() +
    theme(legend.position = "none", axis.text.x = element_text(size = 12))
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```



Here's what we did above:

- We used `ggplot2` to plot each student's height as well as the mean heights of female and male students respectively.
- The `geom_jitter()` function plots a point for each student's height, and then makes slight random adjustments to the location of the points so that they are less likely to overlap. One of the great things about plotting our data like this is that we can quickly see if there are many more observations in one category than another. That information would be obscured if we were to use a box plot.
- The `geom_segment()` function creates the two horizontal lines at the mean values of height. Notice we used a different data frame – class_summary – using the `data = class_summary` argument to plot the mean values.
- We changed the x and y axis titles using the `scale_x_discrete()` and `scale_y_continuous()` functions.
- We changed the default ggplot colors to orange and blue (Go Gators!) using the `scale_color_manual()` function.
- We simplified the plot using the `theme_classic()` function.
- `theme(legend.position = "none", axis.text.x = element_text(size = 12))` removed the legend and increased the size of the x-axis labels a little bit.

After checking both numerical and graphical descriptions of the relationship between gender and height we may conclude that male students were taller, on average, than female students.

23.2 Multiple predictors

At times we may be interested in comparing continuous outcomes across levels of two or more categorical variables. As an example, perhaps we want to describe BMI by gender *and* age group. All we have to do is add age group to the `group_by()` function.

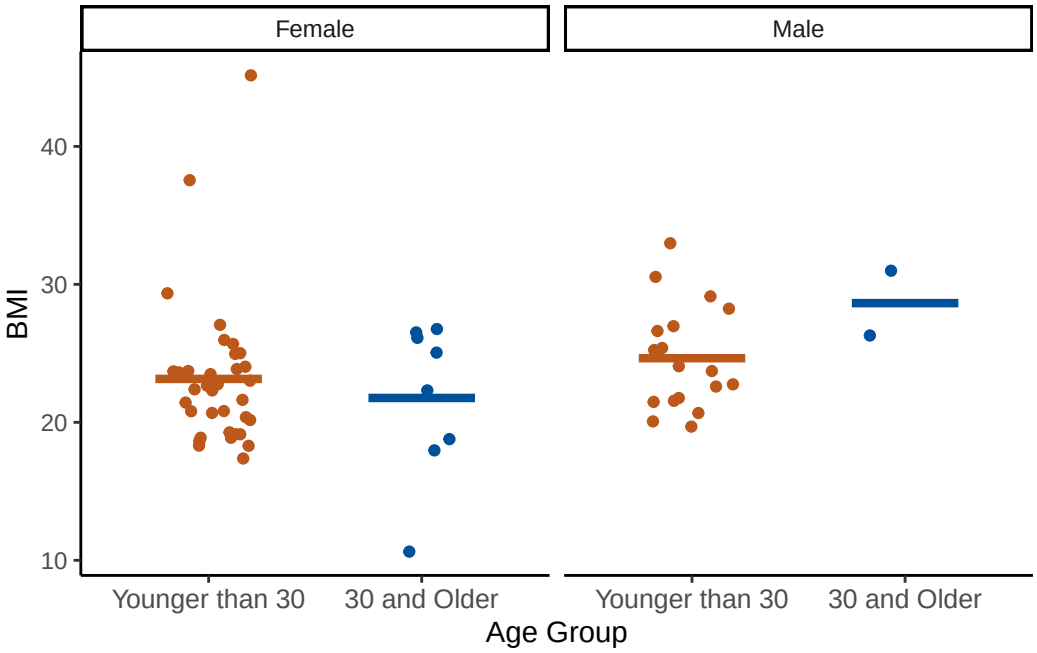
```
class_summary <- class %>%
  filter(!is.na(bmi)) %>%
  group_by(gender, age_group) %>%
  summarise(
    n                    = n(),
    mean                 = mean(bmi),
    `standard deviation` = sd(bmi),
    min                  = min(bmi),
    max                  = max(bmi)
  ) %>%
  print()
```

```
# A tibble: 4 x 7
# Groups:   gender [2]
  gender age_group           n  mean `standard deviation`   min   max
  <fct>  <fct>           <int> <dbl>                <dbl> <dbl> <dbl>
1 Female Younger than 30    35  23.1                 5.41  17.4  45.2
2 Female 30 and Older        8  21.8                 5.67  10.6  26.8
3 Male   Younger than 30    19  24.6                 3.69  19.7  33.0
4 Male   30 and Older        2  28.6                 3.32  26.3  31.0
```

And we can see these statistics for BMI within levels of gender separately for younger and older students. Males that are 30 and older report, on average, the highest BMI (28.6). Females age 30 and older report, on average, the lowest BMI (21.8). This is good information, but often when comparing groups a picture really is worth a thousand words. Let's wrap up this chapter with one final plot.

```
class %>%
  filter(!is.na(bmi)) %>%
  ggplot(aes(x = age_group, y = bmi)) +
    facet_wrap(vars(gender)) +
```

```
  geom_jitter(aes(col = age_group), width = 0.20) +
  geom_segment(
    aes(x = rep(c(0.75, 1.75), 2), y = mean, xend = rep(c(1.25, 2.25), 2), yend = mean,
        col = age_group),
    size = 1.5, data = class_summary
  ) +
  scale_x_discrete("Age Group") +
  scale_y_continuous("BMI") +
  scale_color_manual(values = c("#BC581A", "#00519B")) +
  theme_classic() +
  theme(legend.position = "none", axis.text.x = element_text(size = 10))
```



Here's what we did above:

- We used the same code for this plot that we used for the first height by gender plot. The only difference is that we added `facet_wrap(vars(gender))` to plot males and females on separate plot panels.

24 Describing the Relationship Between a Categorical Outcome and a Categorical Predictor

Generally speaking, there is no good way to describe the relationship between a continuous predictor and a categorical outcome.

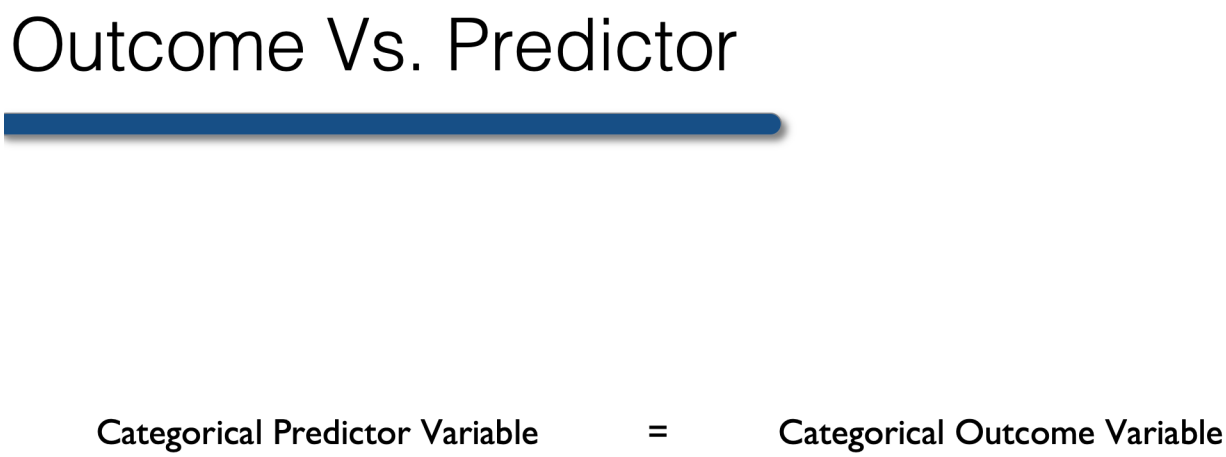


Figure 24.1: Categorical outcome and categorical predictor

So, when your outcome is categorical, the predictor must also be categorical. Therefore, any continuous predictor variables must be collapsed into categories before conducting bivariate analysis when your outcome is categorical. The best categories are those that have scientific

or clinical meaning. For example, collapsing raw scores on a test of cognitive function into a categorical variable for cognitive impairment. The variable could be dichotomous (yes, no) or it could have multiple levels (no, mild cognitive impairment, dementia).

Categorizing Outcomes

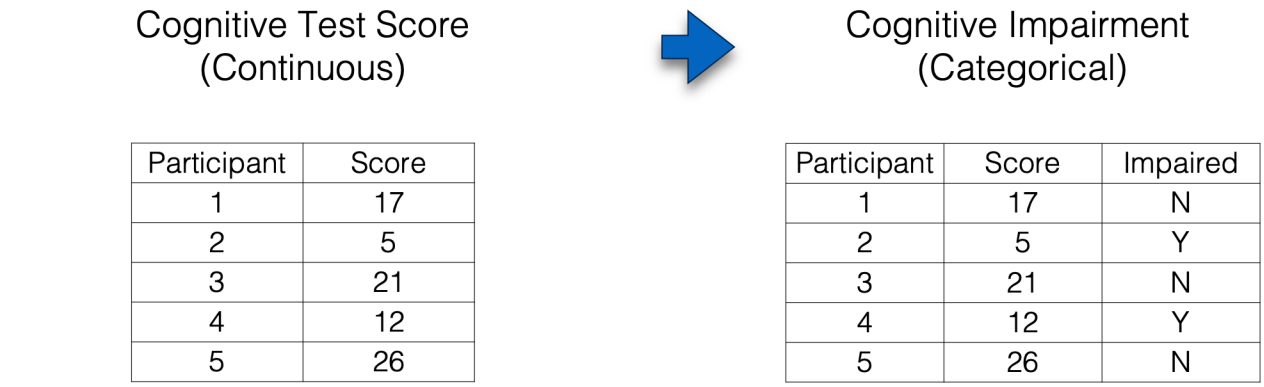


Figure 24.2: Categorizing outcomes

Once your continuous variables are collapsed you're ready to create **n-way frequency tables** that will allow you to describe the relationship between two or more categorical variables. To start with, we will once again use our previously collected class survey data.

```
library(dplyr)
library(ggplot2)
```

```
class <- tibble(
  age       = c(32, 30, 32, 29, 24, 38, 25, 24, 48, 29, 22, 29, 24, 28, 24, 25,
                25, 22, 25, 24, 25, 24, 23, 24, 31, 24, 29, 24, 22, 23, 26, 23,
                24, 25, 24, 33, 27, 25, 26, 26, 26, 26, 26, 27, 24, 43, 25, 24,
                27, 28, 29, 24, 26, 28, 25, 24, 26, 24, 26, 31, 24, 26, 31, 34,
                26, 25, 27, NA),
```

```
  age_group = c(2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2,
                2, 1, 1, 1, NA),
  gender    = c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1,
                1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1,
                1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 1,
                1, 1, 2, 1, NA),
  ht_in     = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
                64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
                64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
                69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
                61, 69, 66, NA),
  wt_lbs    = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
                125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
                186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
                147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
                110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
                163, 141, NA),
  bmi       = c(30.99, 18.78, 26.52, 30.54, 22.39, 26.12, 23.69, 20.67, 26.29,
                25.39, 25.68, 45.15, 21.56, 20.17, 17.38, 20.8, 22.31, 22.75,
                26.62, 21.43, 19.14, 23.49, 22.66, 32.98, 25.05, 18.31, 29.13,
                27.07, 20.37, 25.01, 19.69, 25.97, 18.88, 20.07, NA, 26.76,
                26.97, 25.24, 20.68, 23.72, 24.82, 23.62, 18.65, 24.03, 23.86,
                10.63, 23.02, 23.72, 20.82, 28.24, NA, 37.55, 18.88, 18.3,
                19.13, 21.48, 22.59, 24.96, 21.63, NA, 29.35, 21.76, 17.97,
                22.31, 19.27, 24.07, 22.76, NA),
  bmi_3cat  = c(3, 1, 2, 3, 1, 2, 1, 1, 2, 2, 2, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1,
                1, 1, 3, 2, 1, 2, 2, 1, 2, 1, 2, 1, 1, NA, 2, 2, 2, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 2, NA, 3, 1, 1, 1, 1, 1, 1, 1, NA, 2, 1,
                1, 1, 1, 1, 1, NA),
  genhlth   = c(2, 2, 3, 3, 2, 1, 2, 2, 2, 1, 3, 3, 1, 2, 2, 1, 2, NA, 3, 2, 3,
                1, 2, 2, 2, 4, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 3, 3, 2, 1, 3, 3,
                2, 2, 3, 3, 2, 3, 2, 2, 3, 5, 3, 2, 3, 2, 3, 3, 2, 2, 3, 3, 3,
                1, 2, 2, 1, 3),
  persdoc   = c(1, 2, 2, 1, 2, 0, 0, 1, 2, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0,
                0, 1, 1, 1, 1, 2, 0, 0, 1, 1, 2, 1, 2, 0, 0, 2, 0, 0, 2, 2, 0,
                NA, 0, 0, 0, 2, 0, 2, NA, 0, 2, 1, 1, 1, 2, 2, 0, 0, 0, 1, 2,
                1, 1, 0, 0, 0, NA)
) %>%
  mutate(
    age_group = factor(age_group, labels = c("Younger than 30", "30 and Older")),
```

```
    gender    = factor(gender, labels = c("Female", "Male")),
    bmi_3cat  = factor(bmi_3cat, labels = c("Normal", "Overweight", "Obese")),
    genhlth   = factor(genhlth, labels = c("Excellent", "Very Good", "Good", "Fair", "Poor")),
    persdoc    = factor(persdoc, labels = c("No", "Yes, only one", "Yes, more than one"))
  ) %>%
  print()
```

```
# A tibble: 68 x 9
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat   genhlth   persdoc
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>      <fct>     <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese      Very Good Yes, on~
 2    30 30 and Older    Female    63    106  18.8 Normal     Very Good Yes, mo~
 3    32 30 and Older    Female    62    145  26.5 Overweight Good      Yes, mo~
 4    29 Younger than 30 Male      67    195  30.5 Obese      Good      Yes, on~
 5    24 Younger than 30 Female    67    143  22.4 Normal     Very Good Yes, mo~
 6    38 30 and Older    Female    58    125  26.1 Overweight Excellent No
 7    25 Younger than 30 Female    64    138  23.7 Normal     Very Good No
 8    24 Younger than 30 Male      69    140  20.7 Normal     Very Good Yes, on~
 9    48 30 and Older    Male      65    158  26.3 Overweight Very Good Yes, mo~
10    29 Younger than 30 Male      68    167  25.4 Overweight Excellent No
# i 58 more rows
```

24.1 Comparing two variables

We've already used R to create one-way descriptive tables for categorical variables. One-way frequency tables can be interesting in their own right; however, most of the time we are interested in the relationships between two variables. For example, think about when we looked at mean height within levels of gender. This told us something about the relationship between height and gender. While far from definite, our little survey provides some evidence that women, on average, are shorter than men.

Well, we can describe the relationship between two categorical variables as well. One way of doing so is with two-way frequency tables, which are also sometimes referred to as **crosstabs** or **contingency tables**. Let's start by simply looking at an example.

Below we use the same `CrossTable()` function that we used in the lesson on univariate analysis of categorical data. The only difference is that we pass two vectors to the function instead of one. The first variable will always form the rows, and the second variable will always form the columns. In other words, we can say that we are creating a two-way table of persdoc by genhealth.

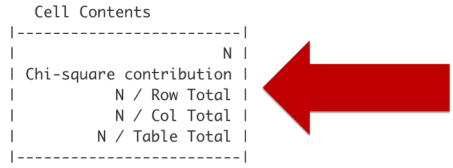
```r
df <- filter(class, !is.na(bmi_3cat)) # Drop rows with missing bmi
gmodels::CrossTable(df$persdoc, df$genhlth)
```

```
   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  61


                   | df$genhlth
         df$persdoc | Excellent | Very Good |      Good |      Fair |      Poor | Row Total |
--------------------|-----------|-----------|-----------|-----------|-----------|-----------|
                 No |         4 |         9 |         8 |         0 |         0 |        21 |
                    |     0.090 |     0.097 |     0.180 |     0.344 |     0.344 |           |
                    |     0.190 |     0.429 |     0.381 |     0.000 |     0.000 |     0.344 |
                    |     0.400 |     0.310 |     0.400 |     0.000 |     0.000 |           |
                    |     0.066 |     0.148 |     0.131 |     0.000 |     0.000 |           |
--------------------|-----------|-----------|-----------|-----------|-----------|-----------|
      Yes, only one |         4 |        12 |         6 |         1 |         0 |        23 |
                    |     0.014 |     0.104 |     0.315 |     1.029 |     0.377 |           |
                    |     0.174 |     0.522 |     0.261 |     0.043 |     0.000 |     0.377 |
                    |     0.400 |     0.414 |     0.300 |     1.000 |     0.000 |           |
                    |     0.066 |     0.197 |     0.098 |     0.016 |     0.000 |           |
--------------------|-----------|-----------|-----------|-----------|-----------|-----------|
Yes, more than one |         2 |         8 |         6 |         0 |         1 |        17 |
                    |     0.222 |     0.001 |     0.033 |     0.279 |     1.867 |           |
                    |     0.118 |     0.471 |     0.353 |     0.000 |     0.059 |     0.279 |
                    |     0.200 |     0.276 |     0.300 |     0.000 |     1.000 |           |
                    |     0.033 |     0.131 |     0.098 |     0.000 |     0.016 |           |
--------------------|-----------|-----------|-----------|-----------|-----------|-----------|
       Column Total |        10 |        29 |        20 |         1 |         1 |        61 |
                    |     0.164 |     0.475 |     0.328 |     0.016 |     0.016 |           |
--------------------|-----------|-----------|-----------|-----------|-----------|-----------|
```

Okay, let's walk through this output together...



```
   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|
```

```
Total Observations in Table:  61
```

	df$genhlth				
df$persdoc	Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.3: Cell contents

Think of little box labeled "Cell Contents" as a legend that tells you how to interpret the rest of the boxes. Reading from top to bottom, the first number you encounter in a box will be the frequency or count of observations (labeled `N`). The second number you encounter will be the chi-square contribution. Please ignore that number for now. The third number will be the row proportion. The fourth number will be the column proportion. And the fifth number will be the overall proportion.

df$persdoc	df$genhlth Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.4: Table of summary statistics row headers

Reading the table of summary statistics from top to bottom, the row headers describe categories of persdoc, which are `one`, `only one`, and `more than one`.

df$persdoc	df$genhlth Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.5: Table of summary statistics column headers

Reading from left to right, the column headers describe categories of genhealth, which are `excellent`, `very good`, `good`, `fair`, and `poor`.

df$persdoc	df$genhlth Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.6: Total frequency and proportion of observations in each category defined by columns

The bottom row gives the total frequency and proportion of observations that fall in each of the categories defined by the columns. For example, 10 students – about 0.164 of the entire class – reported being in excellent general health.

df$persdoc	df$genhlth Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.7: Total frequency and proportion of observations in each category defined by rows

The far-right column gives the total frequency and proportion of observations that fall in each of the categories defined by the rows. For example, 23 students – about 0.377 of the entire class – reported that they have exactly one person that they think of as their personal doctor or healthcare provider.

	df$genhlth					
df$persdoc	Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.8: Total frequency and proportion of observations in each category defined by rows

And the bottom right corner gives the overall total frequency of observations in the table. Together, the last row, the far-right column, and the bottom right cell make up what are called the marginal totals because they are on the outer margin of the table.

Next, let's interpret the data contained in the first cell with data.

	df$genhlth					
df$persdoc	Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.9: First cell

The first number is the frequency. There are 4 students that do not have a personal doctor *and* report being in excellent health.

	df$genhlth					
df$persdoc	Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.10: First number - cell frequency

The third number is the row proportion. The row this cell is in is the `No` row, which includes 21 students. Out of the 21 total students in the `No` row, 4 reported being in excellent health. 4 divided by 21 is 0.190. Said another way, 19% of students with no personal doctor reported being in excellent health.

df$persdoc	df$genhlth Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.11: Third number - row proportion

The fourth number is the column proportion. This cell is in the `Excellent` column. Of the 10 students in the `Excellent` column, 4 reported that they do not have a personal doctor. 4 out of 10 is 0.4. Said another way, 40% of students who report being in excellent health have no personal doctor.

	df$genhlth					
df$persdoc	Excellent	Very Good	Good	Fair	Poor	Row Total
No	4	9	8	0	0	21
	0.090	0.097	0.180	0.344	0.344	
	0.190	0.429	0.381	0.000	0.000	0.344
	0.400	0.310	0.400	0.000	0.000	
	0.066	0.148	0.131	0.000	0.000	
Yes, only one	4	12	6	1	0	23
	0.014	0.104	0.315	1.029	0.377	
	0.174	0.522	0.261	0.043	0.000	0.377
	0.400	0.414	0.300	1.000	0.000	
	0.066	0.197	0.098	0.016	0.000	
Yes, more than one	2	8	6	0	1	17
	0.222	0.001	0.033	0.279	1.867	
	0.118	0.471	0.353	0.000	0.059	0.279
	0.200	0.276	0.300	0.000	1.000	
	0.033	0.131	0.098	0.000	0.016	
Column Total	10	29	20	1	1	61
	0.164	0.475	0.328	0.016	0.016	

Figure 24.12: Fourth number - column proportion

The last number is the overall proportion. So, 4 out of the 61 total students in this analysis have no personal doctor *and* report being in excellent health. Four out of 61 is 0.066. So, about 7% of **all** the students in the class have no personal doctor *and* are in excellent health.

Now that you know how to read the table, let's point out a couple subtleties that may not have jumped out at you above.

1. **The changing denominator.** As we moved from the row proportion to the column proportion and then the overall proportion, all that changed was the denominator (the blue circle). And each time we did so we were describing the characteristics of a different group of people: (1) students without a personal doctor, (2) students in excellent general health, (3) all students – regardless of personal doctor or general health.

2. **Language matters.** Because we are actually describing the characteristics of different subgroups, the language we use to interpret our results is important. For example, when interpreting the row proportion above, we wrote, "19% of students with no personal doctor reported being in excellent health." This language implies that we're describing the health (characteristic) of students with no personal doctor (subgroup). It would be completely incorrect to instead say, "19% of students in excellent health have no personal

doctor" or "19% of students have no personal doctor." Those are interpretations of the column percent and overall percent respectively. They are not interchangeable.

Part V

Data Management

25 Introduction to Data Management

Way back in Section 2.2.2, we told you that managing data includes all the things you may have to do to your data to get it ready for analysis. We also talked about the 80/20 "rule." The basic idea of the 80/20 rule is that data management is where we will spend the majority of our time and effort when we are involved in just about any project that makes use of data. Unfortunately, we can't cover strategies for overcoming every single data management challenge that you will encounter in epidemiology. However, in this part of the book, we will try to give you a foundation in some of the most common data management tasks that you will encounter. We will also try to point you towards some of the best tools and resources for data management that the R community has to offer.

25.1 Multiple paradigms for data management in R

Before moving on to providing you with examples of how to accomplish specific data management tasks, we think this is the right point in the book to touch on a couple of high-level concepts that we have more or less ignored thus far.

R is pretty unique among the major statistical programming applications used in epidemiology in many ways. Among them is that R has multiple paradigms for data management. That's what we're calling them anyway. What we mean by that is that there are 3 primary packages that the vast majority of R users use for data management. They are base R, `data.table`, and `dplyr`. There is a tremendous amount of overlap in the data management tasks you can perform with base R, `data.table`, and `dplyr`, but the syntax for each is very different. As are the relative strengths and weaknesses.

In this book, we will primarily use the `dplyr` paradigm for data management. We will do so because we believe in using the best tool to get the job done. Currently, we believe that the best tool for managing data in R is usually `dplyr`, and especially when you are new to R. However, there will be cases where we will show you how to use base R to accomplish a task. Where we do this, it's because we think that base R is the best tool for the job or because we think you are very likely to see base R way used when you go looking for help with a related data management challenge and we don't want you to be totally clueless about what you're looking at.

As of this writing, we've decided not to specifically discuss using the `data.table` package for data management. We think the `data.table` package is a great package, and we use it when

we think it's the best tool for the job. However, we think the confusion caused by introducing `data.table` in this text aimed primarily at inexperienced R users would cause more problems than it would solve. The last thing we'll say about `data.table` for now is that you may want to consider learning more about `data.table` if you routinely work with very large data sets (e.g., millions of rows). For reasons that are beyond the scope of this book, `data.table` is currently much faster than `dplyr`. However, for most of the work we do, and all of what we will do in this book, the time difference will be imperceptible to you. Literally milliseconds.

25.2 The dplyr package

At this point in the book, you've already been exposed to several of the most important functions in the `dplyr` package. You saw the `filter()` function in the Speaking R's language chapter, the `mutate()` function in the chapter on exporting data, and the `summarise()` function all over the descriptive analysis part of the book. However, we mostly glossed over the details at those points. In this section, we want to dive just a tiny bit deeper into how the `dplyr` functions work – but not too deep.

25.2.1 The dplyr verbs

The `dplyr` package includes five main functions for managing data: `mutate()`, `select()`, `filter()`, `arrange()`, and `summarise()`. These five functions are often referred to as the dplyr verbs. And, the first two arguments to all five of these functions are `.data` and `....` Let's go ahead and discuss those two arguments a little bit more.

> **Note**
>
> we don't want to give you the impression that `dplyr` only contains 5 functions. In fact, `dplyr` contains many functions, and they are all designed to work together in a very intentional way.

25.2.2 The .data argument

I first introduced you to data frames in the Let's get programming chapter and we've been using them as our primary structure for storing and analyzing data in ever since. The R language allows for other data structures (e.g., vectors, lists, and matrices), but data frames are the most commonly used data structure for most of the kinds of things we do in epidemiology. Thankfully, the `dplyr` package is designed specifically to help people like you and us manage data stored in *data frames*. Therefore, `dplyr` verbs always receive a data frame as an input and return a data frame as an output. Specifically, the value passed to the `.data` argument

must always be a data frame, and you will get an error if you attempt to pass any other data structure to the `.data` argument. For example:

```
# No problem
df <- tibble(
  id = c(1, 2, 3),
  x  = c(0, 1, 0)
)

df %>%
  filter(x == 0)
```

```
# A tibble: 2 x 2
     id     x
  <dbl> <dbl>
1     1     0
2     3     0
```

```
# Problem
l <- list(
  id = c(1, 2, 3),
  x  = c(0, 1, 0)
)

l %>%
  filter(x == 0)
```

```
Error in UseMethod("filter"): no applicable method for 'filter' applied to an object of class
```

25.2.3 The ... argument

The second value passed to all of the `dplyr` verbs is the `...` argument. If you are new to R, this probably seems like a really weird argument. And, it kind of is! But, it's also really useful. The `...` argument (pronounced "dot dot dot") has special meaning in the R language. This is true for all functions that use the `...` argument – not just `dplyr` verbs. The `...` argument can be used to pass multiple arguments to a function without knowing exactly what those arguments will look like ahead of time – including entire expressions. For example:

```
df %>%
  filter(x == 0)
```

```
# A tibble: 2 x 2
     id     x
  <dbl> <dbl>
1     1     0
2     3     0
```

Above we passed a data frame to the `.data` argument of the `filter()` function. The second value we passed to the `filter()` function was `x == 1`. Think about it, `x` is an object (i.e. a column in the data frame), `==` is a function (remember that operators are functions in R), and `0` is a value. Together, they form an expression (`x == 0`) that tells R to perform a relatively complex operation – compare every value of `x` to the value `0` and tell me if they are the same. If you are new to programming, this may not seem like any big deal, but it's really handy to be able to pass that much information to a single argument of a single function.

If this is all really confusing to you, don't get too hung up on it right now. The `...` argument is an important component of the R language, but it isn't important that you fully understand it in order to use R. If nothing else, just know that that the `...` is the second argument to all the `dplyr` verbs, and it is generally where you will tell R what you want to *do* to the columns of our data frame (i.e., keep them, drop them, create them, sort them, etc.).

25.2.4 Non-standard evaluation

A final little peculiarity about the `tidyverse` packages – `dplyr` being one of them – that we want to discuss in this chapter is something called **non-standard evaluation**. *How* non-standard evaluation works really isn't that important for us. If we're being honest, we don't even fully understand how it works "under the hood." But, it is one of the big advantages of using `dplyr`, and therefore worth mentioning. Do you remember the section in the Let's get programming chapter on common errors? In that section I wrote about how a vector that lives in the global environment is a different thing to R than a vector that lives as a column in a data frame in the global environment. So, `weight` and `class$weight` are different things, and if you want to access the weight values in `class$weight` then you have to make sure and write the whole thing out. But, have you noticed that we don't have to do that in `dplyr` verbs? For example:

```
df %>%
  filter(df$x == 0)
```

```
# A tibble: 2 x 2
     id     x
  <dbl> <dbl>
1     1     0
2     3     0
```

In the example above we wrote out the column name using dollar sign notation. But, we don't *have* to:

```
df %>%
  filter(x == 0)
```

```
# A tibble: 2 x 2
     id     x
  <dbl> <dbl>
1     1     0
2     3     0
```

When we don't tell a `dplyr` verb exactly which data frame a column lives in, then the `dplyr` verb will assume it lives in the data frame that is passed to the `.data` argument. This is really handy for at least two reasons:

- It reduces the amount of typing we have to do when we write our code.
- It makes it easier to glance at our code and see what it's doing. Without all the data frame names and dollar signs strewn about our code, it's much easier to see what the code is actually doing.

Overall, non-standard evaluation is a great thing – at least in our opinion. However, it will present some challenges that we will have to overcome if we plan to use `dplyr` verbs inside of functions and loops. Don't worry, we'll come back to this topic later in the book.

Now that you (hopefully) have a better general understanding of the `dplyr` verbs, let's go take a look at how to use them for data management.

26 Creating and Modifying Columns

Two of the most fundamental data management tasks are to create new columns in your data frame and to modify existing columns in your data frame. In fact, we've already talked about creating and modifying columns at a few different places in the book.

In this book, we are actually going to learn 4 different methods for creating and modifying columns of a data frame. They are:

1. Using name-value pairs to add columns to a data frame during its initial creation. This was one of the first methods we used in this book for creating columns in a data frame. However, this method does not apply to creating or modifying columns in *a data frame that already exists*. Therefore, we won't discuss it much in this chapter.
2. Dollar sign notation. This is probably the most commonly used base R way of creating and modifying columns in a data frame. In this book, we won't use it as much as we use `dplyr::mutate()`, but you will see it all over the place in the R community.
3. Bracket notation. Again, we won't use bracket notation very often in this book. However, we will use it later on when we learn about for loops. Therefore, we're going to introduce you to using bracket notation to create and modify data frame columns now.
4. The `mutate()` function from the `dplyr` package. This is the method that we will use the vast majority of the time in this book (and in our real-life projects). We're going to recommend that you do the same.

26.1 Creating data frames

Very early on, in the Let's get programming chapter, we learned how to create data frame columns using name-value pairs passed directly into the `tibble()` function.

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

```
# A tibble: 4 x 2
  names heights
  <chr>   <dbl>
1 John       68
2 Sally      63
3 Brad       71
4 Anne       72
```

This is an absolutely fundamental R programming skill, and one that you will likely use often. However, most people would not consider this to be a "data management" task, which is the focus of this part of the book. Further, we've really already covered all we need to cover about creating columns this way. So, we're not going to write anything further about this method.

26.2 Dollar sign notation

Later in the Let's get programming chapter, we learned about **dollar sign notation**. At that time, we used dollar sign notation to access or "get" values from a column.

```
class$heights
```

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

However, we can also use dollar sign notation to create and/or modify columns in our data frame. For example:

```
class$heights <- class$heights / 12
class
```

```
# A tibble: 4 x 2
  names heights
  <chr>   <dbl>
1 John     5.67
2 Sally    5.25
3 Brad     5.92
4 Anne     6
```

Here's what we did above:

- we modified the values in the `heights` column of our `class` data frame using dollar sign notation. More specifically, we converted the values in the `heights` column from inches to feet. We did this by telling R to "get" the values for the `heights` column and divide them by 12 (`class$heights / 12`) and then assign those new values back to the `heights` column (`class$heights <-`). In this case, that has the effect of modifying the values of a column that already exists.

> **Note**
>
> we would actually suggest that you don't typically do what we just did above in a real-world analysis. It's typically safer to create a new variable with the modified values (e.g. `height_feet`) and leave the original values in the original variable as-is.

we can also create a *new* variable in our data frame in a similar way. All we have to do is use a valid column name (that doesn't already exist in the data frame) on the left side of our assignment arrow. For example:

```
class$grades <- c(89, 92, 86, 98)
class
```

```
# A tibble: 4 x 3
  names heights grades
  <chr>   <dbl>  <dbl>
1 John     5.67     89
2 Sally    5.25     92
3 Brad     5.92     86
4 Anne     6        98
```

Here's what we did above:

- we created a new column in our `class` data frame using dollar sign notation. We assigned the values 89, 92, 86, and 98 to that column with the assignment arrow.

26.3 Bracket notation

we also learned how to access or "get" values from a column using bracket notation in the Let's get programming chapter. There, we actually used a combination of dollar sign and bracket notation to access single individual values from a data frame column. For example:

```
class$heights[3]
```

```
[1] 5.916667
```

But, we can also use bracket notation to access or "get" the entire column. For example:

```
class[["heights"]]
```

```
[1] 5.666667 5.250000 5.916667 6.000000
```

Here's what we did above:

- we used bracket notation to get all of the values from the `heights` column of the `class` data frame.

we'd like you to notice a couple of things about the example above. First, notice that this is the exact same result we got from (`class$heights`). Well, technically, the heights are now in feet instead of inches, but you know what we mean. R returned a numeric vector containing the values from the `heights` column to us. Second, notice that we used double brackets (i.e., two brackets on each side of the column name), and that the column name is wrapped in quotation marks. Both are required to get this result.

Similar to dollar sign notation, we can also create and/or modify columns in our data frame using bracket notation. For example, let's convert those heights back to inches using bracket notation:

```
class[["heights"]] <- class[["heights"]] * 12
class
```

```
# A tibble: 4 x 3
  names heights grades
  <chr>   <dbl>  <dbl>
1 John       68     89
2 Sally      63     92
3 Brad       71     86
4 Anne       72     98
```

And, let's go ahead and add one more variable to our data frame using bracket notation.

```
class[["rank"]] <- c(3, 2, 4, 1)
class
```

```
# A tibble: 4 x 4
  names heights grades  rank
  <chr>   <dbl>  <dbl> <dbl>
1 John       68     89     3
2 Sally      63     92     2
3 Brad       71     86     4
4 Anne       72     98     1
```

Somewhat confusingly, we can also access, create, and modify data frame columns using single brackets. For example:

```
class["heights"]
```

```
# A tibble: 4 x 1
  heights
    <dbl>
1      68
2      63
3      71
4      72
```

Notice, however, that this returns a different result than `class$heights` and `class[["heights]]`. The results returned from `class$heights` and `class[["heights]]` were numeric vectors with 4 elements. The result returned from `class["heights"]` was a data frame with 1 column and 4 rows.

we don't want you to get too hung up on the difference between single and double brackets right now. As we said, we are primarily going to use `mutate()` to create and modify data frame columns in this book. For now, it's enough for you to simply be aware that single brackets and double brackets are a thing, and they can sometimes return different results. We will make sure to point out whether or not that matters when we use bracket notation later in the book.

26.4 Modify individual values

Before moving on to the `mutate()` function, we wanted to quickly discuss using dollar sign and bracket notation for modifying individual values in a column. Recall that we already learned how to access individual column values in the Let's get programming chapter.

```
class$heights[3]
```

```
[1] 71
```

As you may have guessed, we can also get the result above using only bracket notation.

```
class[["heights"]][3]
```

```
[1] 71
```

Not only can we use these methods to get individual values from a column in a data frame, but we can also use these methods to *modify* an individual value in a column of a data frame. When might we want to do this? Well, we generally do this in one of two different circumstances.

- First, we may do this when we're writing our own R functions (you'll learn how to do this later) and we want to make sure the function still behaves in the way we intended when there are small changes to the data. So, we may add a missing value to a column or something like that.
- The second circumstance is when there are little one-off typos in the data. For example, let's say we imported a data frame that looked like this:

```
study_data <- tibble(
  id = c(1, 2, 3, 4),
  site = c("TX", "CA", "tx", "CA")
)

study_data
```

```
# A tibble: 4 x 2
     id site
  <dbl> <chr>
1     1 TX
2     2 CA
3     3 tx
4     4 CA
```

Notice that `tx` in the third row of data isn't capitalized. Remember, R is a case-sensitive language, so this will likely cause us problems down the road if we don't fix it. The easiest way to do so is probably:

```
study_data$site[3] <- "TX"
study_data
```

```
# A tibble: 4 x 2
     id site
  <dbl> <chr>
1     1 TX
2     2 CA
3     3 TX
4     4 CA
```

Keep in mind that we said that we fix *little one-off typos*. If we needed to change `tx` to `TX` in multiple different places in the data, we wouldn't use this method. Instead, we would use a conditional operation, which we will discuss later in the book.

26.5 The mutate() function

```
# Load dplyr for the mutate function
library(dplyr)
```

we first discussed `mutate()` in Chapter 17, and again in Chapter 25. As we said there, the first two arguments to `mutate()` are `.data` and `....`

The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% mutate()`).

The value passed to the `...` argument should be a name-value pair or multiple name value pairs separated by commas. The `...` argument is where you will tell `mutate()` to create or modify columns in your data frame and how.

- Name-value pairs look like this: `column name = value`. The only thing that distinguishes whether you are creating or modifying a column is the column name in the name-value pair. If the column name in the name-value pair matches the name of an existing column in the data frame, then `mutate()` will modify that existing column. If the column name in the name-value pair does NOT match the name of an existing column in the data frame, then `mutate()` will create a *new* column in the data frame with a matching column name.

Let's take a look at a couple of examples. To get us started, let's simulate some data that is a little more interesting than the class data we used above.

```
set.seed(123)

drug_trial <- tibble(
  # Study id, there are 20 people enrolled in the trial.
  id = rep(1:20, each = 3),
  # Follow-up year, 0 = baseline, 1 = year one, 2 = year two.
  year = rep(0:2, times = 20),
  # Participant age a baseline. Must be between the ages of 35 and 75 at
  # baseline to be eligible for the study
  age = sample(35:75, 20, TRUE) %>% rep(each = 3),
  # Drug the participant received, Placebo or active
  drug = sample(c("Placebo", "Active"), 20, TRUE) %>%
    rep(each = 3),
  # Reported headaches side effect, Y/N
  se_headache = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.95,.05)),
    sample(0:1, 60, TRUE, c(.10, .90))
  ),
  # Report diarrhea side effect, Y/N
  se_diarrhea = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.98,.02)),
    sample(0:1, 60, TRUE, c(.20, .80))
  ),
  # Report dry mouth side effect, Y/N
  se_dry_mouth = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.97,.03)),
    sample(0:1, 60, TRUE, c(.30, .70))
  ),
  # Participant had myocardial infarction in study year, Y/N
  mi = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.85, .15)),
    sample(0:1, 60, TRUE, c(.80, .20))
  )
)
```

Here's what we did above:

- we are simulating some drug trial data that includes the following variables:
  - id: Study id, there are 20 people enrolled in the trial.
  - year: Follow-up year, 0 = baseline, 1 = year one, 2 = year two.
  - age: Participant age a baseline. Must be between the ages of 35 and 75 at baseline to be eligible for the study.
  - drug: Drug the participant received, Placebo or active.
  - se_headache: Reported headaches side effect, Y/N.
  - se_diarrhea: Report diarrhea side effect, Y/N.
  - se_dry_mouth: Report dry mouth side effect, Y/N.
  - mi: Participant had myocardial infarction in study year, Y/N.
- we used the `tibble()` function above to create our data frame instead of the `data.frame()` function. This allows us to pass the `drug` column as a value to the `if_else()` function when we create `se_headache`, `se_diarrhea`, `se_dry_mouth`, and `mi`. If we had used `data.frame()` instead, we would have had to create `se_headache`, `se_diarrhea`, `se_dry_mouth`, and `mi` in a separate step.
- we used a new function, `if_else()`, above to help us simulate this data. This function allows us to do something called **conditional operations**. There will be an entire chapter on conditional operations later in the book.
- we used a new function, `sample()`, above to help us simulate this data. We used this function to randomly assign values to `age`, `drug`, `se_headache`, `se_diarrhea`, `se_dry_mouth`, and `mi` instead of manually assigning each value ourselves.
  - You can type `?sample` into your R console to view the help documentation for this function and follow along with the explanation below.
  - The first argument to the `sample()` function is the `x` argument. You should pass a vector of values you want R to randomly choose from. For example, we told R to select values from a vector of numbers that spanned between 35 and 75 to fill-in the `age` column. Alternatively, we told R to select values from a character vector that included the values "Placebo" and "Active" to fill-in the `drug` column.
  - The second argument to the `sample()` function is the `size` argument. You should pass a number to the size argument. That number tells R how many times to choose a value from the vector of possible values passed to the `x` argument.

- The third argument to the `sample()` function is the `replace` argument. The default value passed to the `replace` argument is `FALSE`. This tells R that once it has chosen a value from the vector of possible values passed to the `x` argument, it can't choose that value again. If you want R to be able to choose the same value more than once, then you have to pass the value `TRUE` to the `replace` argument.
- The fourth argument to the `sample()` function is the `prob` argument. The default value passed to the `prob` argument is `NULL`. This just means that this argument is *optional*. Passing a vector of probabilities to this argument allows you to adjust how likely it is that R will choose certain values from the vector of possible values passed to the `x` argument.
- Finally, notice that we also used the `set.seed()` function at the very top of the code chunk. We did this because, the `sample()` function chooses values at random. That means, every time we run the code above, we get different values. That makes it difficult for me to write about the data because it's constantly changing. When we use the `set.seed()` function, the values will still be randomly selected, but they will be the *same* randomly selected values every time. It doesn't matter what numbers you pass to the `set.seed()` function as long as you pass the same numbers every time you want to get the same random values. For example:

```
# No set.seed - Random values
sample(1:100, 10, TRUE)
```

```
 [1]  5 29 50 70 74 26 73 11  6 96
```

```
# No set.seed - Different random values
sample(1:100, 10, TRUE)
```

```
 [1] 76 83 91 56 96 27 94 68 88 28
```

```
# Use set.seed - Random values
set.seed(456)
sample(1:100, 10, TRUE)
```

```
 [1] 35 38 85 27 25 78 31 73 79 90
```

```
# Use set.seed again - Same random values
set.seed(456)
sample(1:100, 10, TRUE)
```

```
 [1] 35 38 85 27 25 78 31 73 79 90
```

```
# Use set.seed with different value - Different random values
set.seed(789)
sample(1:100, 10, TRUE)
```

```
 [1]  45  12  42  26  99  37 100  43  67  70
```

- It's not important that you fully understand the `sample()` function at this point. We're just including it for those of you who are interested in simulating some slightly more complex data than we have simulated so far. The rest of you can just copy and paste the code if you want to follow along.

26.5.1 Adding or modifying a single column

This is probably the simplest case of adding a new column. We are going to use `mutate()` to add a single new column to the `drug_trial` data frame. Let's say we want to add a column called `complete` that is equal to `1` if the participant showed up for all follow-up visits and equal to `0` if they didn't. In this case, we simulated our data in such a way that we have complete follow-up for every participant. So, the value for complete should be `0` in all 60 rows of the data frame. We can do this in a few different ways.

```
drug_trial %>%
  mutate(complete = c(
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
  )
```

```
# A tibble: 60 x 9
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi complete
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>    <dbl>
 1     1     0    65 Active            0           1            1     0        0
 2     1     1    65 Active            1           1            1     0        0
 3     1     2    65 Active            1           1            0     0        0
 4     2     0    49 Active            1           1            1     0        0
 5     2     1    49 Active            0           0            1     0        0
 6     2     2    49 Active            1           1            1     0        0
 7     3     0    48 Placebo           0           0            0     0        0
 8     3     1    48 Placebo           0           0            0     0        0
 9     3     2    48 Placebo           0           0            0     0        0
```

```
10     4     0    37 Placebo           0           0             0     0        0
# i 50 more rows
```

So, that works, but typing that out is no fun. Not to mention, this isn't scalable at all. What if we needed 1,000 zeros? There's actually a much easier way to get the result above, which may surprise you. Take a look :

```
drug_trial %>%
  mutate(complete = 0)
```

```
# A tibble: 60 x 9
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi complete
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>    <dbl>
 1     1     0    65 Active            0           1            1     0        0
 2     1     1    65 Active            1           1            1     0        0
 3     1     2    65 Active            1           1            0     0        0
 4     2     0    49 Active            1           1            1     0        0
 5     2     1    49 Active            0           0            1     0        0
 6     2     2    49 Active            1           1            1     0        0
 7     3     0    48 Placebo           0           0            0     0        0
 8     3     1    48 Placebo           0           0            0     0        0
 9     3     2    48 Placebo           0           0            0     0        0
10     4     0    37 Placebo           0           0            0     0        0
# i 50 more rows
```

How easy is that? Just pass the value to the name-value pair once and R will use it in every row. This works because of something called the recycling rules . In a nutshell, this means that R will change the length of vectors in certain situations all by itself when it thinks it knows what you "meant." So, above we passed gave R a length 1 vector `0` (i.e. a numeric vector with one value in it), and R changed it to a length 60 vector behind the scenes so that it could complete the operation it thought you were trying to complete.

26.5.2 Recycling rules

The recycling rules work as long as the length of the longer vector is an integer multiple of the length of the shorter vector. For example, every vector (column) in R data frames must have the same length. In this case, 60. The length of the value we used in the name-value pair above was 1 (i.e., a single `0`). Therefore, the longer vector had a length of 60 and the shorter vector had a length of 1. Because 60 * 1 = But, what if we had tried to pass the values 0 and 1 to the column instead of just zero?

```
drug_trial %>%
  mutate(complete = c(0, 1))
```

```
Error in `mutate()`:
i In argument: `complete = c(0, 1)`.
Caused by error:
! `complete` must be size 60 or 1, not 2.
```

This doesn't work, but it actually isn't for the reason you may be thinking. Because 30 * 2 = 60, the length of the longer vector (60) is an integer multiple (30) of the length of the shorter vector (2). However, `tidyverse` functions throw errors when you try to recycle anything other than a single number. They are designed this way to protect you from accidentally getting unexpected results. So, we're going to switch back over to using base R to round out our discussion of the recycling rules. Let's try our example above again using base R:

```
drug_trial$complete <- c(0,1)
```

```
Error in `$<-`:
! Assigned data `c(0, 1)` must be compatible with existing data.
x Existing data has 60 rows.
x Assigned data has 2 rows.
i Only vectors of size 1 are recycled.
Caused by error in `vectbl_recycle_rhs_rows()`:
! Can't recycle input of size 2 to size 60.
```

```
drug_trial
```

```
# A tibble: 60 x 8
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>
 1     1     0    65 Active            0           1            1     0
 2     1     1    65 Active            1           1            1     0
 3     1     2    65 Active            1           1            0     0
 4     2     0    49 Active            1           1            1     0
 5     2     1    49 Active            0           0            1     0
 6     2     2    49 Active            1           1            1     0
 7     3     0    48 Placebo           0           0            0     0
 8     3     1    48 Placebo           0           0            0     0
 9     3     2    48 Placebo           0           0            0     0
10     4     0    37 Placebo           0           0            0     0
# i 50 more rows
```

Wait, why are we still getting an error? Well, take a look at the output below and see if you can figure it out.

```
class(drug_trial)
```

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

It may not be totally obvious, but this is telling us that `drug_trial` is a tibble – an enhanced data frame. Remember, we created `drug_trial` using the `tibble()` function instead of the `tibble()` function. Because tibbles are part of the `tidyverse` they throw the same recycling errors that the `mutate()` function did above. So, we'll need to create a non-tibble version of `drug_trial` to finish our discussion of recycling rules.

```
drug_trial_df <- as.data.frame(drug_trial)
class(drug_trial_df)
```

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

There we go! A regular old data frame.

```
drug_trial_df$complete <- c(0,1)
drug_trial_df
```

```
   id year age    drug se_headache se_diarrhea se_dry_mouth mi complete
1   1    0  65  Active           0           1            1  0        0
2   1    1  65  Active           1           1            1  0        1
3   1    2  65  Active           1           1            0  0        0
4   2    0  49  Active           1           1            1  0        1
5   2    1  49  Active           0           0            1  0        0
6   2    2  49  Active           1           1            1  0        1
7   3    0  48 Placebo           0           0            0  0        0
8   3    1  48 Placebo           0           0            0  0        1
9   3    2  48 Placebo           0           0            0  0        0
10  4    0  37 Placebo           0           0            0  0        1
11  4    1  37 Placebo           0           0            0  0        0
12  4    2  37 Placebo           0           0            0  1        1
13  5    0  71 Placebo           0           0            0  0        0
14  5    1  71 Placebo           0           0            0  0        1
15  5    2  71 Placebo           0           0            0  0        0
16  6    0  48 Placebo           0           0            0  0        1
```

17	6	1	48	Placebo	0	0	0	1	0
18	6	2	48	Placebo	0	0	0	1	1
19	7	0	59	Active	1	1	1	0	0
20	7	1	59	Active	1	1	0	0	1
21	7	2	59	Active	1	1	1	0	0
22	8	0	60	Placebo	0	0	0	0	1
23	8	1	60	Placebo	0	0	0	0	0
24	8	2	60	Placebo	0	0	0	0	1
25	9	0	61	Active	1	1	1	0	0
26	9	1	61	Active	0	1	1	0	1
27	9	2	61	Active	1	0	0	0	0
28	10	0	39	Active	1	0	1	0	1
29	10	1	39	Active	1	0	0	0	0
30	10	2	39	Active	1	1	1	0	1
31	11	0	61	Placebo	0	0	0	0	0
32	11	1	61	Placebo	0	0	0	1	1
33	11	2	61	Placebo	0	0	0	0	0
34	12	0	62	Placebo	1	0	1	0	1
35	12	1	62	Placebo	0	0	0	0	0
36	12	2	62	Placebo	0	0	0	0	1
37	13	0	43	Placebo	0	0	0	0	0
38	13	1	43	Placebo	0	0	0	0	1
39	13	2	43	Placebo	0	0	0	0	0
40	14	0	63	Placebo	0	0	0	0	1
41	14	1	63	Placebo	0	0	0	0	0
42	14	2	63	Placebo	0	0	0	0	1
43	15	0	69	Active	1	1	1	0	0
44	15	1	69	Active	1	0	1	0	1
45	15	2	69	Active	1	1	1	0	0
46	16	0	42	Placebo	0	0	0	0	1
47	16	1	42	Placebo	0	0	1	0	0
48	16	2	42	Placebo	0	0	0	1	1
49	17	0	60	Placebo	0	0	0	0	0
50	17	1	60	Placebo	0	0	0	0	1
51	17	2	60	Placebo	1	0	0	0	0
52	18	0	41	Active	1	1	1	0	1
53	18	1	41	Active	1	1	1	0	0
54	18	2	41	Active	1	1	0	1	1
55	19	0	43	Placebo	0	0	0	0	0
56	19	1	43	Placebo	0	0	0	0	1
57	19	2	43	Placebo	0	0	0	0	0
58	20	0	53	Placebo	0	0	0	0	1
59	20	1	53	Placebo	0	0	0	0	0

```
60 20    2  53 Placebo           0           0           0  0        1
```

As you can see, the values 0 and 1 are now recycled as expected. Because 30 * 2 = 60, the length of the longer vector (60) is an integer multiple (30) of the length of the shorter vector (2). Now, what happens in a situation where the length of the longer vector is *not* an integer multiple of the length of the shorter vector.

```
drug_trial_df$complete <- c(0, 1, 2, 3, 4, 5, 6) # 7 values
```

```
Error in `$<-.data.frame`(`*tmp*`, complete, value = c(0, 1, 2, 3, 4, : replacement has 7 rows
```

60 / 7 = 8.571429 – not an integer. Because there is no integer value that we can multiply by 7 to get the number 60, R throws us an error telling us that it isn't able to use the recycling rules.

Finally, the recycling rules don't only apply to creating new data frame columns. It applies in all cases where R is using two vectors to perform an operation. For example, R uses the recycling rules in mathematical operations.

```
nums <- 1:10
nums
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

To demonstrate, we create a simple numeric vector above. This vector just contains the numbers 1 through 10. Now, we can add 1 to each of those numbers like so:

```
nums + 1
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

Notice how R used the recycling rules to add 1 to every number in the `nums` vector. We didn't have to explicitly tell R to add 1 to each number. This is sometimes referred to as **vectorization**. Functions that perform an action on *all* elements of a vector, rather than having to be explicitly programmed to perform an action on *each* element of a vector, is a **vectorized** function. Remember, that mathematical operators – including `+` – *are functions* in R. More specifically, `+` is a **vectorized** function. In fact, most built-in R functions are vectorized. Why are we telling you this? It isn't intended to confuse you, but when I was learning R I came across this term all the time in R resources and help pages, and I had no idea what it meant. We hope that this very simple example above makes it easy to understand

what vectorization means, and you won't be intimidated when it pops up while you're trying to get help with your R programs.

Ok, so what happens when we add a longer vector and a shorter vector?

```
nums + c(1, 2)
```

```
 [1]  2  4  4  6  6  8  8 10 10 12
```

As expected, R uses the recycling rules to change the length of the short vector to match the length of the longer vector, and then performs the operation – in this case, addition. So, the net result is $1 + 1 = 2$, $2 + 2 = 4$, $3 + 1 = 4$, $4 + 2 = 6$, etc. You probably already guessed what's going to happen if we try to add a length 3 vector to `nums`, but let's go ahead and take a look for the sake of completeness:

```
nums + c(1, 2, 3)
```

```
Warning in nums + c(1, 2, 3): longer object length is not a multiple of shorter
object length
```

```
 [1]  2  4  6  5  7  9  8 10 12 11
```

Yep, we get an error. $10 / 3 = 3.333333$ – not an integer. Because there is no integer value that we can multiply by 3 to get the number 10, R throws us an error telling us that it isn't able to use the recycling rules.

Now that you understand R's recycling rules, let's return to our motivating example.

```
drug_trial %>%
  mutate(complete = 0)
```

```
# A tibble: 60 x 9
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi complete
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>    <dbl>
 1     1     0    65 Active            0           1            1     0        0
 2     1     1    65 Active            1           1            1     0        0
 3     1     2    65 Active            1           1            0     0        0
 4     2     0    49 Active            1           1            1     0        0
 5     2     1    49 Active            0           0            1     0        0
 6     2     2    49 Active            1           1            1     0        0
 7     3     0    48 Placebo           0           0            0     0        0
```

```
 8     3     1    48 Placebo           0           0           0     0        0
 9     3     2    48 Placebo           0           0           0     0        0
10     4     0    37 Placebo           0           0           0     0        0
# i 50 more rows
```

This method works, but not always. And, it can sometimes give us intended results. You may have originally thought to yourself, "we've already learned the `rep()` function. Let's use that." In fact, that's a great idea!

```
drug_trial %>%
  mutate(complete = rep(0, 60))
```

```
# A tibble: 60 x 9
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi complete
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>    <dbl>
 1     1     0    65 Active            0           1            1     0        0
 2     1     1    65 Active            1           1            1     0        0
 3     1     2    65 Active            1           1            0     0        0
 4     2     0    49 Active            1           1            1     0        0
 5     2     1    49 Active            0           0            1     0        0
 6     2     2    49 Active            1           1            1     0        0
 7     3     0    48 Placebo           0           0            0     0        0
 8     3     1    48 Placebo           0           0            0     0        0
 9     3     2    48 Placebo           0           0            0     0        0
10     4     0    37 Placebo           0           0            0     0        0
# i 50 more rows
```

That's a lot less typing than the first method we tried, and it also has the added benefit of providing code that is easier for humans to read. We can both look at the code we used in the first method and tell that there are a bunch of zeros, but it's hard to guess exactly how many, and it's hard to feel completely confident that there isn't a 1 in there somewhere that our eyes are missing. By contrast, it's easy to look at `rep(0, 60)` and know that there are exactly 60 zeros, and only 60 zeros.

26.5.3 Using existing variables in name-value pairs

In the example above, we create a new column called `complete` by directly supplying values for that column in the name-value pair. In our experience, it is probably more common to create new columns in our data frames by combining or transforming the values of columns that already exist in our data frame. You've already seen an example of doing so when we

created factor versions of variables. As an additional example, we could create a factor version of our `mi` variable like this:

```
drug_trial %>%
  mutate(mi_f = factor(mi, c(0, 1), c("No", "Yes")))
```

```
# A tibble: 60 x 9
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi mi_f
   <int> <int> <int> <chr>         <int>       <int>        <int> <int> <fct>
 1     1     0    65 Active            0           1            1     0 No
 2     1     1    65 Active            1           1            1     0 No
 3     1     2    65 Active            1           1            0     0 No
 4     2     0    49 Active            1           1            1     0 No
 5     2     1    49 Active            0           0            1     0 No
 6     2     2    49 Active            1           1            1     0 No
 7     3     0    48 Placebo           0           0            0     0 No
 8     3     1    48 Placebo           0           0            0     0 No
 9     3     2    48 Placebo           0           0            0     0 No
10     4     0    37 Placebo           0           0            0     0 No
# i 50 more rows
```

Notice that in the code above, we didn't tell R what values to use for `mi_f` by typing them explicitly in the name-value pair. Instead, we told R to go get the values of the column `mi`, do some stuff to those values, and then assign those modified values to a column in the data frame and name that column `mi_f`.

Here's another example. It's common to mean-center numeric values for many different kinds of analyses. For example, this is often done in regression analysis to aid in the interpretation of regression coefficients. We can easily mean-center numeric variables inside our `mutate()` function like so:

```
drug_trial %>%
  mutate(age_center = age - mean(age))
```

```
# A tibble: 60 x 9
      id  year   age drug  se_headache se_diarrhea se_dry_mouth    mi age_center
   <int> <int> <int> <chr>       <int>       <int>        <int> <int>      <dbl>
 1     1     0    65 Acti~           0           1            1     0       11.3
 2     1     1    65 Acti~           1           1            1     0       11.3
 3     1     2    65 Acti~           1           1            0     0       11.3
 4     2     0    49 Acti~           1           1            1     0       -4.7
 5     2     1    49 Acti~           0           0            1     0       -4.7
```

```
 6     2     2    49 Acti~           1           1           1     0       -4.7
 7     3     0    48 Plac~           0           0           0     0       -5.7
 8     3     1    48 Plac~           0           0           0     0       -5.7
 9     3     2    48 Plac~           0           0           0     0       -5.7
10     4     0    37 Plac~           0           0           0     0      -16.7
# i 50 more rows
```

Notice how succinctly we were able to express this fairly complicated task. We had to figure out the find the mean of the variable `age` in the `drug_trial` data frame, subtract that value from the value for `age` in each row of the data frame, and then create a new column in the data frame containing the mean-centered values. Because of the fact that `mutate()`'s name-value pairs can accept complex expressions a value, and because all of the functions used in the code above are vectorized, we can perform this task using only a single, easy-to-read line of code (`age_center = age - mean(age)`).

26.5.4 Adding or modifying multiple columns

In all of the examples above, we passed a single name-value pair to the `...` argument of the `mutate()` function. If we want to create or modify multiple columns, we don't need to keep typing the `mutate()` function over and over. We can simply pass multiple name-value pairs, separated by columns, to the `...` argument. And, there is no limit to the number of pairs we can pass. This is part of the beauty of the `...` argument in R. For example, we have three variables in `drug_trial` that capture information about whether or not the participant reported side effects including headache, diarrhea, and dry mouth. Currently, those are all stored as integer vectors that can take the values `0` and `1`. Let's say that we want to also create factor versions of those vectors:

```
drug_trial %>%
  mutate(
    se_headache_f  = factor(se_headache, c(0, 1), c("No", "Yes")),
    se_diarrhea_f  = factor(se_diarrhea, c(0, 1), c("NO", "Yes")),
    se_dry_mouth_f = factor(se_dry_mouth, c(0, 1), c("No", "Yes"))
  )
```

```
# A tibble: 60 x 11
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>
 1     1     0    65 Active            0           1            1     0
 2     1     1    65 Active            1           1            1     0
 3     1     2    65 Active            1           1            0     0
 4     2     0    49 Active            1           1            1     0
```

```
 5     2     1    49 Active            0           0           1     0
 6     2     2    49 Active            1           1           1     0
 7     3     0    48 Placebo           0           0           0     0
 8     3     1    48 Placebo           0           0           0     0
 9     3     2    48 Placebo           0           0           0     0
10     4     0    37 Placebo           0           0           0     0
# i 50 more rows
# i 3 more variables: se_headache_f <fct>, se_diarrhea_f <fct>,
#   se_dry_mouth_f <fct>
```

Here's what we did above:

- we created three new factor columns in the `drug_trial` data called `se_headache_f`, `se_diarrhea_f`, and `se_dry_mouth_f`.
- we created all columns inside a single `mutate()` function.
- Notice that we created one variable per line. We suggest you do the same. It just makes your code much easier to read.

So, adding or modifying multiple columns is really easy with `mutate()`. But, did any of you notice an error? Take a look at the structure of the data the line of code that creates `se_diarrhea_f`. Instead of writing the "No" label with an "N" and an "o", we accidently wrote it with an "N" and a zero. We find that when we have to type something over and over like this, we are more likely to make a mistake. Further, if we ever need to change the levels or labels, we will have to change them in every `factor()` function in the code above.

For these reasons (and others), programmers of many languages – including R – are taught the DRY principle. DRY is an acronym for don't repeat yourself. We will discuss the DRY principle again in Chapter 33, but for now, it just means that you typically don't want to type code that is the same (or nearly the same) over and over in your programs. Here's one way we could reduce the repetition in the code above:

```
# Create a vector of 0/1 levels that can be reused below.
yn_levs <- c(0, 1)
# Create a vector of "No"/"Yes" labels that can be reused below.
yn_labs <- c("No", "Yes")

drug_trial %>%
  mutate(
    se_headache_f  = factor(se_headache, yn_levs, yn_labs),
    se_diarrhea_f  = factor(se_diarrhea, yn_levs, yn_labs),
    se_dry_mouth_f = factor(se_dry_mouth, yn_levs, yn_labs)
  )
```

```
# A tibble: 60 x 11
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>
 1     1     0    65 Active            0           1            1     0
 2     1     1    65 Active            1           1            1     0
 3     1     2    65 Active            1           1            0     0
 4     2     0    49 Active            1           1            1     0
 5     2     1    49 Active            0           0            1     0
 6     2     2    49 Active            1           1            1     0
 7     3     0    48 Placebo           0           0            0     0
 8     3     1    48 Placebo           0           0            0     0
 9     3     2    48 Placebo           0           0            0     0
10     4     0    37 Placebo           0           0            0     0
# i 50 more rows
# i 3 more variables: se_headache_f <fct>, se_diarrhea_f <fct>,
#   se_dry_mouth_f <fct>
```

Notice that in the code above we type `c(0, 1)` and `c("No", "Yes")` once each instead of 3 times each. In the chapter on repeated operations we will learn techniques for removing even more repetition from the code above.

26.5.5 Rowwise mutations

In all the examples above we used the values from *a single* already existing variable in our name-value pair. However, we can also use the values from *multiple* variables in our name-value pairs.

For example, we have three variables in our `drug_trial` data that capture information about whether or not the participant reported side effects including headache, diarrhea, and dry mouth (sounds like every drug commercial that exists). What if we want to know if our participants reported *any* side effect at each follow-up? That requires us to combine and transform data from across three different columns! This is one of those situations where there are many different ways we could accomplish this task, but we're going to use `dplyr`'s `rowwise()` function to do so in the following code:

```
drug_trial %>%
  rowwise() %>%
  mutate(any_se_year = sum(se_headache, se_diarrhea, se_dry_mouth) > 0)
```

```
# A tibble: 60 x 9
# Rowwise:
```

```
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>
 1     1     0    65 Active            0           1            1     0
 2     1     1    65 Active            1           1            1     0
 3     1     2    65 Active            1           1            0     0
 4     2     0    49 Active            1           1            1     0
 5     2     1    49 Active            0           0            1     0
 6     2     2    49 Active            1           1            1     0
 7     3     0    48 Placebo           0           0            0     0
 8     3     1    48 Placebo           0           0            0     0
 9     3     2    48 Placebo           0           0            0     0
10     4     0    37 Placebo           0           0            0     0
# i 50 more rows
# i 1 more variable: any_se_year <lgl>
```

Here's what we did above:

- we created a new column in the `drug_trial` data called `any_se_year` using the `mutate()` function.
- we used the `rowwise()` function to tell R to group the data frame by rows. Said another way `rowwise()` tells R to do any calculations that follow *across* columns instead *within* columns. Don't worry, there are more examples below.
- The value we passed to the name-value pair inside `mutate()` was actually the result of two calculations.
  - First, R summed the values of se_headache, se_diarrhea, and se_dry_mouth (i.e., `sum(se_headache, se_diarrhea, se_dry_mouth)`).
  - Next, R compared that the summed value to `0`. If the summed value was greater than 0, then the value assigned to `any_se_year` was `TRUE`. Otherwise, the value assigned to `any_se_year` was `FALSE`.

Because there is some new stuff in the code above, I'm going break it down a little bit further. We'll start with `rowwise()`. And, to reduce distractions a much as possible, I'm going to create a new data frame with only the columns we need for this example (sneak peek at the next chapter):

```
drug_trial_sub <- drug_trial %>%
  select(id, year, starts_with("se")) %>%
  print()
```

```
# A tibble: 60 x 5
      id  year se_headache se_diarrhea se_dry_mouth
   <int> <int>       <int>       <int>        <int>
 1     1     0           0           1            1
 2     1     1           1           1            1
 3     1     2           1           1            0
 4     2     0           1           1            1
 5     2     1           0           0            1
 6     2     2           1           1            1
 7     3     0           0           0            0
 8     3     1           0           0            0
 9     3     2           0           0            0
10     4     0           0           0            0
# i 50 more rows
```

Let's start by discussing what `rowwise()` does. As we discussed above, most built-in R functions are vectorized. They *do things* to entire vectors, and data frame columns *are* vectors. So, without using `rowwise()` the `sum()` function would have returned the value `54`:

```
drug_trial_sub %>%
  mutate(any_se_year = sum(se_headache, se_diarrhea, se_dry_mouth))
```

```
# A tibble: 60 x 6
      id  year se_headache se_diarrhea se_dry_mouth any_se_year
   <int> <int>       <int>       <int>        <int>       <int>
 1     1     0           0           1            1          54
 2     1     1           1           1            1          54
 3     1     2           1           1            0          54
 4     2     0           1           1            1          54
 5     2     1           0           0            1          54
 6     2     2           1           1            1          54
 7     3     0           0           0            0          54
 8     3     1           0           0            0          54
 9     3     2           0           0            0          54
10     4     0           0           0            0          54
# i 50 more rows
```

Any guesses why it returns 54? Here's a hint:

```
sum(c(0, 1, 0))
```

```
[1] 1
```

```
sum(c(1, 1, 0))
```

```
[1] 2
```

```
sum(
  c(0, 1, 0),
  c(1, 1, 0)
)
```

```
[1] 3
```

When we pass a single numeric vector to the `sum()` function, it adds together all the numbers in that function. When we pass two or more numeric vectors to the `sum()` function, it adds together all the numbers in all the vectors combined. Our data frame columns are no different:

```
sum(drug_trial_sub$se_headache)
```

```
[1] 20
```

```
sum(drug_trial_sub$se_diarrhea)
```

```
[1] 16
```

```
sum(drug_trial_sub$se_dry_mouth)
```

```
[1] 18
```

```
sum(
  drug_trial_sub$se_headache,
  drug_trial_sub$se_diarrhea,
  drug_trial_sub$se_dry_mouth
)
```

```
[1] 54
```

Hopefully, you see that the `sum()` function is taking the total of all three vectors added together, which is a single number (`54`), and then using recycling rules to assign that value to every row of `any_se_year`.

Using `rowwise()` tells R to add *across* the columns instead of *within* the columns. So, add the first value for `se_headache` to the first value for `se_diarrhea` to the first value for `se_dry_mouth`, assign that value to the first value of `any_se_year`, and then repeat for each subsequent row. This is what that result looks like:

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(any_se_year = sum(se_headache, se_diarrhea, se_dry_mouth))
```

```
# A tibble: 60 x 6
# Rowwise:
      id  year se_headache se_diarrhea se_dry_mouth any_se_year
   <int> <int>       <int>       <int>        <int>       <int>
 1     1     0           0           1            1           2
 2     1     1           1           1            1           3
 3     1     2           1           1            0           2
 4     2     0           1           1            1           3
 5     2     1           0           0            1           1
 6     2     2           1           1            1           3
 7     3     0           0           0            0           0
 8     3     1           0           0            0           0
 9     3     2           0           0            0           0
10     4     0           0           0            0           0
# i 50 more rows
```

Because the value for each side effect could only be 0 (if not reported) or 1 (if reported) then the rowwise sum of those numbers is a count of the number of side effects reported in each row. For example, person 1 reported not having headaches (`0`), having diarrhea (`1`), and having dry mouth (`1`) at baseline (`year == 0`). And, $0 + 1 + 1 = 2$ – the same value you see for `any_se_year` in that row. For instructional purposes, let's run the code above again, but change the name of the variable to `n_se_year` (i.e., the count of side effects a participant reported in a given year).

This may be a useful result in and of itself. However, we said we wanted a variable that captured whether a participant reported *any* side effect at each follow-up. Well, because `any_se_year` is currently a count of side effects reported for that participant in that year, then where the value of `any_se_year` is `0` no side effects were reported. If the current value of `any_se_year` is greater than `0`, then one or more side effects were reported. Generally, we can test inequalities like this in the following way:

```
# Is 0 greater than 0?
0 > 0
```

```
[1] FALSE
```

```
# Is 2 greater than 0?
2 > 0
```

```
[1] TRUE
```

In our specific situation, instead of using a number on the left side of the inequality, we can use our calculated `n_se_year` variable values on the left side of the inequality:

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(
    n_se_year   = sum(se_headache, se_diarrhea, se_dry_mouth),
    any_se_year = n_se_year > 0
  )
```

```
# A tibble: 60 x 7
# Rowwise:
      id  year se_headache se_diarrhea se_dry_mouth n_se_year any_se_year
   <int> <int>       <int>       <int>        <int>     <int> <lgl>
 1     1     0           0           1            1         2 TRUE
 2     1     1           1           1            1         3 TRUE
 3     1     2           1           1            0         2 TRUE
 4     2     0           1           1            1         3 TRUE
 5     2     1           0           0            1         1 TRUE
 6     2     2           1           1            1         3 TRUE
 7     3     0           0           0            0         0 FALSE
 8     3     1           0           0            0         0 FALSE
 9     3     2           0           0            0         0 FALSE
10     4     0           0           0            0         0 FALSE
# i 50 more rows
```

In this way, `any_se_year` is TRUE if the participant reported any side effect in that year and false if they reported no side effects in that year. We could write the code more succinctly like this:

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(any_se_year = sum(se_headache, se_diarrhea, se_dry_mouth) > 0)
```

```
# A tibble: 60 x 6
# Rowwise:
      id  year se_headache se_diarrhea se_dry_mouth any_se_year
   <int> <int>       <int>       <int>        <int> <lgl>
 1     1     0           0           1            1 TRUE
 2     1     1           1           1            1 TRUE
 3     1     2           1           1            0 TRUE
 4     2     0           1           1            1 TRUE
 5     2     1           0           0            1 TRUE
 6     2     2           1           1            1 TRUE
 7     3     0           0           0            0 FALSE
 8     3     1           0           0            0 FALSE
 9     3     2           0           0            0 FALSE
10     4     0           0           0            0 FALSE
# i 50 more rows
```

But, is that really what we want to do? The answer is it depends. If we are going to stop here, then the succinct code may be what we want. But, what if we want to also know if the participant reported *all* side effects in each year. Perhaps, you've already worked out what that code would look like. Perhaps you're thinking something like:

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(
    any_se_year = sum(se_headache, se_diarrhea, se_dry_mouth) > 0,
    all_se_year = sum(se_headache, se_diarrhea, se_dry_mouth) == 3
  )
```

```
# A tibble: 60 x 7
# Rowwise:
      id  year se_headache se_diarrhea se_dry_mouth any_se_year all_se_year
   <int> <int>       <int>       <int>        <int> <lgl>       <lgl>
 1     1     0           0           1            1 TRUE        FALSE
 2     1     1           1           1            1 TRUE        TRUE
 3     1     2           1           1            0 TRUE        FALSE
 4     2     0           1           1            1 TRUE        TRUE
 5     2     1           0           0            1 TRUE        FALSE
```

```
 6     2     2           1           1           1 TRUE        TRUE
 7     3     0           0           0           0 FALSE       FALSE
 8     3     1           0           0           0 FALSE       FALSE
 9     3     2           0           0           0 FALSE       FALSE
10     4     0           0           0           0 FALSE       FALSE
# i 50 more rows
```

That works, and hopefully, you're able to reason out why it works. But, there we go repeating code again! So, in this case, we have to choose between more succinct code and the DRY principle. When presented with that choice, I will typically favor the DRY principle. Therefore, my code would look like this:

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(
    n_se_year   = sum(se_headache, se_diarrhea, se_dry_mouth),
    any_se_year = n_se_year > 0,
    all_se_year = n_se_year == 3
  )
```

```
# A tibble: 60 x 8
# Rowwise:
      id  year se_headache se_diarrhea se_dry_mouth n_se_year any_se_year
   <int> <int>       <int>       <int>        <int>     <int> <lgl>
 1     1     0           0           1            1         2 TRUE
 2     1     1           1           1            1         3 TRUE
 3     1     2           1           1            0         2 TRUE
 4     2     0           1           1            1         3 TRUE
 5     2     1           0           0            1         1 TRUE
 6     2     2           1           1            1         3 TRUE
 7     3     0           0           0            0         0 FALSE
 8     3     1           0           0            0         0 FALSE
 9     3     2           0           0            0         0 FALSE
10     4     0           0           0            0         0 FALSE
# i 50 more rows
# i 1 more variable: all_se_year <lgl>
```

Not only am I less like to make a typing error in this code, but I think the differences between each line of code (i.e., what that line of code is doing) stands out more. In other words, the intent of the code isn't buried in unneeded words.

Before moving on, I also want to point out that the method above would not have worked on factors. For example:

```
drug_trial_sub %>%
  mutate(
    se_headache  = factor(se_headache, yn_levs, yn_labs),
    se_diarrhea  = factor(se_diarrhea, yn_levs, yn_labs),
    se_dry_mouth = factor(se_dry_mouth, yn_levs, yn_labs)
  ) %>%
  rowwise() %>%
  mutate(
    n_se_year   = sum(se_headache, se_diarrhea, se_dry_mouth),
    any_se_year = n_se_year > 0,
    all_se_year = n_se_year == 3
  )
```

```
Error in `mutate()`:
i In argument: `n_se_year = sum(se_headache, se_diarrhea,
  se_dry_mouth)`.
i In row 1.
Caused by error in `Summary.factor()`:
! 'sum' not meaningful for factors
```

The `sum()` function cannot add factors. Back when I first introduced factors in this book, I suggested that you keep the numeric version of your variables in your data frames and create factors as new variables. I said that I thought this was a good idea because I often find that it can be useful to have both versions of the variable hanging around during the analysis process. The situation above is an example of what I was talking about.

26.5.6 Group_by mutations

So far, we've created variables that tell us if our participants reported *any* side effects in a given and if they reported *all 3* side effects in a given year. The next logical question might be to ask if each participant experienced *any* side effect in *any year*. For that, we will need `dplyr`'s `group_by()` function. Before discussing `group_by()`, I'm going to show you the code I would use to accomplish this task:

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(
    n_se_year   = sum(se_headache, se_diarrhea, se_dry_mouth),
    any_se_year = n_se_year > 0,
    all_se_year = n_se_year == 3
```

```
  ) %>%
  group_by(id) %>%
  mutate(any_se = sum(any_se_year) > 0)
```

```
# A tibble: 60 x 9
# Groups:   id [20]
      id  year se_headache se_diarrhea se_dry_mouth n_se_year any_se_year
   <int> <int>       <int>       <int>        <int>     <int> <lgl>
 1     1     0           0           1            1         2 TRUE
 2     1     1           1           1            1         3 TRUE
 3     1     2           1           1            0         2 TRUE
 4     2     0           1           1            1         3 TRUE
 5     2     1           0           0            1         1 TRUE
 6     2     2           1           1            1         3 TRUE
 7     3     0           0           0            0         0 FALSE
 8     3     1           0           0            0         0 FALSE
 9     3     2           0           0            0         0 FALSE
10     4     0           0           0            0         0 FALSE
# i 50 more rows
# i 2 more variables: all_se_year <lgl>, any_se <lgl>
```

Here's what we did above:

- We created a new column in the `drug_trial_sub` data called `any_se` using the `mutate()` function. The `any_se` column is TRUE if the participant reported *any* side effect in *any year* and FALSE if they *never* reported a side effect in *any year*.
- We first grouped the data by `id` using the `group_by()` function. Note that grouping the data by `id` with `group_by()` overrides grouping the data by row with `rowwise()` as soon as R gets to that point in the code. In other words, the data is grouped by row from `rowwise() %>%` to `group_by(id) %>%` and grouped by `id` after.

Side Note: You can use `dplyr::ungroup()` to ungroup your data frames. This works regardless of whether you grouped them with `rowwise()` or `group_by()`.

I already introduced `group_by()` in the chapter on numerical descriptions of categorical variables. I also said that `group_by()` operationalizes the **Split - Apply - Combine** strategy for data analysis. That 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 `drug_trial_sub` data frame was split into twenty separate little data frames (i.e., one for each study id). Because there are 3 rows for each study id, each of these 20 little data frames had three rows.

Each of those 20 little data frames was then passed to the `mutate()` function. The name-value pair inside the `mutate()` function `any_se = sum(any_se_year) > 0` told R to add up all the values for the column `any_se_year` (i.e., `sum(any_se_year)`), compare that summed value to `0` (i.e., `sum(any_se_year) > 0`), and then assign `TRUE` to `any_se` if the summed value is greater than zero and `FALSE` otherwise. Then, all 20 of the little data frames are combined back together and returned to us as a single data frame.

```
drug_trial_sub %>%
  rowwise() %>%
  mutate(
    n_se_year   = sum(se_headache, se_diarrhea, se_dry_mouth),
    any_se_year = n_se_year > 0,
    all_se_year = n_se_year == 3
  ) %>%
  mutate(any_se = sum(any_se_year) > 0)
```

```
# A tibble: 60 x 9
# Rowwise:
      id  year se_headache se_diarrhea se_dry_mouth n_se_year any_se_year
   <int> <int>       <int>       <int>        <int>     <int> <lgl>
 1     1     0           0           1            1         2 TRUE
 2     1     1           1           1            1         3 TRUE
 3     1     2           1           1            0         2 TRUE
 4     2     0           1           1            1         3 TRUE
 5     2     1           0           0            1         1 TRUE
 6     2     2           1           1            1         3 TRUE
 7     3     0           0           0            0         0 FALSE
 8     3     1           0           0            0         0 FALSE
 9     3     2           0           0            0         0 FALSE
10     4     0           0           0            0         0 FALSE
# i 50 more rows
# i 2 more variables: all_se_year <lgl>, any_se <lgl>
```

You may be wondering why I used the `sum()` function when the values for `any_se_year` are not numbers. The way R treats logical vectors can actually be pretty useful in situations like this. That is, when mathematical operations are applied to logical vectors, R treats FALSE as a 0 and TRUE as a 1. So, for participant 1, R calculated the value for `any_se` something like this:

```
any_se_year <- c(TRUE, TRUE, TRUE)
any_se_year
```

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

```
sum_any_se_year <- sum(any_se_year)
sum_any_se_year
```

```
[1] 3
```

```
any_se <- sum_any_se_year > 0
any_se
```

```
[1] TRUE
```

R used the recycling rules to copy that result to the other two rows of data from participant 1. R then repeated that process for every other participant, and then returned the combined data frame to us.

I hope you found the example above useful. I think it's fairly representative of the kinds of data management stuff I tend to do on a day-to-day basis. Of course, missing data always complicates things (more to come on that!). In the next chapter, we will round out our introduction to the basics of data management by learning how to subset rows and columns of a data frame.

27 Subsetting Data Frames

Subsetting data frames is another one of the most common data management tasks we carryout in our data analysis projects. Subsetting data frames just refers to the process of deciding which columns and rows to keep in your data frame and which to drop.

For example, we may need to subset the rows of a data frame because we're interested in understanding a subpopulation in our sample. Below, we only want to analyze the rows that correspond to participants from Texas.

ID	State	Height	Weight
001	TX	71	190
002	TX	69	176
003	CA	64	130
004	CA	65	154

Or, perhaps we're only interested in a subset of the statistics returned to me in a data frame of analysis results. Below, we only want to view and present the variable name, variable category, count, and percent.

var	cat	n	n_total	percent	se	t_crit	lcl	ucl
cvd	No	19	32	59.375	8.820997	2.039513	40.94225	75.49765
cvd	Yes	13	32	40.625	8.820997	2.039513	24.50235	59.05775

Fortunately, the `dplyr` package includes functions that make it really easy for us to subset our data frames – even in some fairly complicated ways. Let's start by simulating the same drug trial data we simulated in the last chapter and use it to work through some examples.

```
# Load dplyr
library(dplyr)
```

```
set.seed(123)

drug_trial <- tibble(
  # Follow-up year, 0 = baseline, 1 = year one, 2 = year two.
  year = rep(0:2, times = 20),
  # Participant age a baseline. Must be between the ages of 35 and 75 at
  # baseline to be eligible for the study
  age = sample(35:75, 20, TRUE) %>% rep(each = 3),
  # Drug the participant received, Placebo or active
  drug = sample(c("Placebo", "Active"), 20, TRUE) %>%
    rep(each = 3),
  # Reported headaches side effect, Y/N
  se_headache = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.95,.05)),
    sample(0:1, 60, TRUE, c(.10, .90))
```

```
  ),
  # Report diarrhea side effect, Y/N
  se_diarrhea = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.98,.02)),
    sample(0:1, 60, TRUE, c(.20, .80))
  ),
  # Report dry mouth side effect, Y/N
  se_dry_mouth = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.97,.03)),
    sample(0:1, 60, TRUE, c(.30, .70))
  ),
  # Participant had myocardial infarction in study year, Y/N
  mi = if_else(
    drug == "Placebo",
    sample(0:1, 60, TRUE, c(.85, .15)),
    sample(0:1, 60, TRUE, c(.80, .20))
  )
)
```

As a reminder, we are simulating some drug trial data that includes the following variables:

- id: Study id, there are 20 people enrolled in the trial.
- year: Follow-up year, 0 = baseline, 1 = year one, 2 = year two.
- age: Participant age a baseline. Must be between the ages of 35 and 75 at baseline to be eligible for the study.
- drug: Drug the participant received, Placebo or active.
- se_headache: Reported headaches side effect, Y/N.
- se_diarrhea: Report diarrhea side effect, Y/N.
- se_dry_mouth: Report dry mouth side effect, Y/N.
- mi: Participant had myocardial infarction in study year, Y/N.

Actually, this data is slightly different than the data we used in the last chapter. Did you catch the difference? Take another look:

```
drug_trial
```

```
# A tibble: 60 x 7
    year   age drug    se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <chr>         <int>       <int>        <int> <int>
 1     0    65 Active            0           1            1     0
 2     1    65 Active            1           1            1     0
 3     2    65 Active            1           1            0     0
 4     0    49 Active            1           1            1     0
 5     1    49 Active            0           0            1     0
 6     2    49 Active            1           1            1     0
 7     0    48 Placebo           0           0            0     0
 8     1    48 Placebo           0           0            0     0
 9     2    48 Placebo           0           0            0     0
10     0    37 Placebo           0           0            0     0
# i 50 more rows
```

we forgot to put a study id in our data. Because we simulated this data above, the best way to fix this oversite is to make the necessary change to the simulation code above. But, let's pretend that someone sent us this data instead, and we have to add a new study id column to it. Well, we now know how to use the `mutate()` function to columns to our data frame. We can do so like this:

```
drug_trial <- drug_trial %>%
  mutate(
    # Study id, there are 20 people enrolled in the trial.
    id = rep(1:20, each = 3)
  ) %>%
  print()
```

```
# A tibble: 60 x 8
    year   age drug    se_headache se_diarrhea se_dry_mouth    mi    id
   <int> <int> <chr>         <int>       <int>        <int> <int> <int>
 1     0    65 Active            0           1            1     0     1
 2     1    65 Active            1           1            1     0     1
 3     2    65 Active            1           1            0     0     1
 4     0    49 Active            1           1            1     0     2
 5     1    49 Active            0           0            1     0     2
 6     2    49 Active            1           1            1     0     2
 7     0    48 Placebo           0           0            0     0     3
 8     1    48 Placebo           0           0            0     0     3
 9     2    48 Placebo           0           0            0     0     3
10     0    37 Placebo           0           0            0     0     4
# i 50 more rows
```

And now we have the study id in our data. But, by default R adds new columns as the rightmost column of the data frame. In terms of analysis, it doesn't really matter where this column is located in our data. R couldn't care less. However, when humans look at this data, they typically expect the study id (or some other identifier) to be the first column in the data frame. That is a job for `select()`.

27.1 The select() function

```
drug_trial %>%
  select(id, year, age, se_headache, se_diarrhea, se_dry_mouth, mi)
```

```
# A tibble: 60 x 7
      id  year   age se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <int>       <int>       <int>        <int> <int>
 1     1     0    65           0           1            1     0
 2     1     1    65           1           1            1     0
 3     1     2    65           1           1            0     0
 4     2     0    49           1           1            1     0
 5     2     1    49           0           0            1     0
 6     2     2    49           1           1            1     0
 7     3     0    48           0           0            0     0
 8     3     1    48           0           0            0     0
 9     3     2    48           0           0            0     0
10     4     0    37           0           0            0     0
# i 50 more rows
```

Here's what we did above:

- we used the `select()` function to change the order of the columns in the `drug_trial` data frame so that `id` would be the first variable in the data frame when reading from left to right.
- You can type `?select` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `select()` function is `.data`. The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% select()`).
- The second argument to the `select()` function is `....` The value passed to the `...` argument should column names or expressions that return column positions. We'll dive deeper into this soon.

More generally, the `select()` function tells R which variables in your data frame to keep (or drop) and in what order.

The code above gave us the result we wanted. But, it can be tedious and error prone to manually type every variable name inside the `select()` function. Did you notice that we forgot the `drug` column "by accident"?

Thankfully, the `select()` function is one of several `dplyr` functions that accept tidy-select argument modifiers (i.e., functions and operators). In this chapter, we will show you some of the tidy-select argument modifiers we regularly use, but you can always type `?dplyr_tidy_select` into your console to see a complete list.

In our little example above, we could have used the tidy-select `everything()` function to make our code easier to write and we wouldn't have accidently missed the `drug` column. We can do so like this:

```
drug_trial <- drug_trial %>%
  select(id, everything()) %>%
  print()
```

```
# A tibble: 60 x 8
      id  year   age drug    se_headache se_diarrhea se_dry_mouth    mi
   <int> <int> <int> <chr>         <int>       <int>        <int> <int>
 1     1     0    65 Active            0           1            1     0
 2     1     1    65 Active            1           1            1     0
 3     1     2    65 Active            1           1            0     0
 4     2     0    49 Active            1           1            1     0
 5     2     1    49 Active            0           0            1     0
 6     2     2    49 Active            1           1            1     0
 7     3     0    48 Placebo           0           0            0     0
 8     3     1    48 Placebo           0           0            0     0
 9     3     2    48 Placebo           0           0            0     0
10     4     0    37 Placebo           0           0            0     0
# i 50 more rows
```

Here's what we did above:

- we used the `select()` function to change the order of the columns in the `drug_trial` data frame so that `id` would be the first variable in the data frame when reading from left to right.
- Rather than explicitly typing the other column names, we used the `everything()` tidy-select function. As you may have guessed, `everything()` tells R to do X (in this keep) to all the other variables not explicitly mentioned.

For our next example, let's go ahead and add our mean-centered age variable to our `drug_trial` data again. We did this for the first time in the last chapter, in case you missed.

```
drug_trial <- drug_trial %>%
  mutate(age_center = age - mean(age)) %>%
  print()
```

```
# A tibble: 60 x 9
      id  year   age drug  se_headache se_diarrhea se_dry_mouth    mi age_center
   <int> <int> <int> <chr>       <int>       <int>        <int> <int>      <dbl>
 1     1     0    65 Acti~           0           1            1     0       11.3
 2     1     1    65 Acti~           1           1            1     0       11.3
 3     1     2    65 Acti~           1           1            0     0       11.3
 4     2     0    49 Acti~           1           1            1     0       -4.7
 5     2     1    49 Acti~           0           0            1     0       -4.7
 6     2     2    49 Acti~           1           1            1     0       -4.7
 7     3     0    48 Plac~           0           0            0     0       -5.7
 8     3     1    48 Plac~           0           0            0     0       -5.7
 9     3     2    48 Plac~           0           0            0     0       -5.7
10     4     0    37 Plac~           0           0            0     0      -16.7
# i 50 more rows
```

One way we will often use `select()` is for performing quick little **data checks**. For example, let's say that we wanted to make sure the code we wrote above actually *did* what we *intended* it to do. If we print the entire data frame to the screen, `age` and `age_center` aren't directly side-by-side, and there's a lot of other visual clutter from the other variables. In a case like this, we would use `select()` to get a clearer picture:

```
drug_trial %>%
  select(age, age_center)
```

```
# A tibble: 60 x 2
     age age_center
   <int>      <dbl>
 1    65       11.3
 2    65       11.3
 3    65       11.3
 4    49       -4.7
 5    49       -4.7
 6    49       -4.7
```

```
 7    48       -5.7
 8    48       -5.7
 9    48       -5.7
10    37      -16.7
# i 50 more rows
```

Here's what we did above:

- we used the `select()` function to view the `age` and `age_center` columns *only*.
- we can type individual column names, separated by commas, into `select()` to return a data frame containing only those columns, and in that order.

Warning: Notice that we didn't assign our result above to anything (i.e., there's no `drug_trial <-`). If we had done so, the `drug_trial` data would have contained these two columns only. We didn't want to drop the other columns. We could have assigned the result of the code to a different R object (e.g., `check_age <-`, but it wasn't really necessary. We just wanted to quickly view `age` and `age_center` side-by-side for data checking purposes. When we're satisfied that we coded it correctly, we can move on. There's no need to save those results to an R object.

You may also recall that we wanted to subset the `drug_trial` data to include only the columns we needed for the rowwise demonstrations. Here is the code we used to do so:

```
drug_trial %>%
  select(id, year, starts_with("se"))
```

```
# A tibble: 60 x 5
      id  year se_headache se_diarrhea se_dry_mouth
   <int> <int>       <int>       <int>        <int>
 1     1     0           0           1            1
 2     1     1           1           1            1
 3     1     2           1           1            0
 4     2     0           1           1            1
 5     2     1           0           0            1
 6     2     2           1           1            1
 7     3     0           0           0            0
 8     3     1           0           0            0
 9     3     2           0           0            0
10     4     0           0           0            0
# i 50 more rows
```

Here's what we did above:

- we used the `select()` function to view the `id year`, `se_headache`, `se_diarrhea`, and `se_dry_mouth` columns *only*.
- we used the tidy-select `starts_with()` function to select all the side effect variables.

we already know that we can use `everything()` to select *all* of the other variables in a data frame, but what if we just want to grab a *range* or *group* of other variables in a data frame? tidy-select makes it easy for us. Above, we used the `starts_with()` function to select all the columns with names that literally start with the letters "se". Because all of the side effect columns are directly next to each other (i.e., no columns in between them) we could have also used the colon operator `:` like this:

```
drug_trial %>%
  select(id, year, se_headache:se_dry_mouth)
```

```
# A tibble: 60 x 5
      id  year se_headache se_diarrhea se_dry_mouth
   <int> <int>       <int>       <int>        <int>
 1     1     0           0           1            1
 2     1     1           1           1            1
 3     1     2           1           1            0
 4     2     0           1           1            1
 5     2     1           0           0            1
 6     2     2           1           1            1
 7     3     0           0           0            0
 8     3     1           0           0            0
 9     3     2           0           0            0
10     4     0           0           0            0
# i 50 more rows
```

While either method gets us the same result, we tend to prefer using `starts_with()` when possible. We think it makes your code easier to read (i.e., "Oh, he's selecting all the side effect columns here.").

In addition to `starts_with()`, there is also an `ends_with()` tidy-select function that can also be useful. For example, we've named factors with the `_f` naming convention throughout the book. We could use that, along with the `ends-with()` function to create a subset of our data that includes only the factor versions of our side effects columns.

```
# Add the side effect factor columns to our data frame again...
yn_levs <- c(0, 1)
yn_labs <- c("No", "Yes")
```

```
drug_trial <- drug_trial %>%
  mutate(
    se_headache_f  = factor(se_headache, yn_levs, yn_labs),
    se_diarrhea_f  = factor(se_diarrhea, yn_levs, yn_labs),
    se_dry_mouth_f = factor(se_dry_mouth, yn_levs, yn_labs)
  )
```

```
drug_trial %>%
  select(id, year, ends_with("_f"))
```

```
# A tibble: 60 x 5
      id  year se_headache_f se_diarrhea_f se_dry_mouth_f
   <int> <int> <fct>         <fct>         <fct>
 1     1     0 No            Yes           Yes
 2     1     1 Yes           Yes           Yes
 3     1     2 Yes           Yes           No
 4     2     0 Yes           Yes           Yes
 5     2     1 No            No            Yes
 6     2     2 Yes           Yes           Yes
 7     3     0 No            No            No
 8     3     1 No            No            No
 9     3     2 No            No            No
10     4     0 No            No            No
# i 50 more rows
```

> **Note**
>
> Variable names are important! Throughout this book, I've tried to repeatedly emphasize the importance of coding style – including the way we name our R objects. Many people who are new to data management and analysis (and some who aren't, **MDL**) don't fully appreciate the importance of such things. We hope that the preceding two examples are helping you to see why the little details, like variable names, are important. Using consistent variable naming conventions, for example, allows us to write code that requires less typing, is easier for humans to skim and understand, and is less prone to typos and other related errors.

we can also select columns we want to keep by position instead of name. We don't do this often. We think it's generally better to use column names or tidy-select argument modifiers when subsetting columns in your data frame. However, we do sometimes select columns by position when we're writing our own functions. Therefore, we want to quickly show you what this looks like:

```
drug_trial %>%
  select(1:2, 4)
```

```
# A tibble: 60 x 3
      id  year drug
   <int> <int> <chr>
 1     1     0 Active
 2     1     1 Active
 3     1     2 Active
 4     2     0 Active
 5     2     1 Active
 6     2     2 Active
 7     3     0 Placebo
 8     3     1 Placebo
 9     3     2 Placebo
10     4     0 Placebo
# i 50 more rows
```

Here's what we did above:

- we passed column numbers to the `select()` function to keep the 1st, 2nd, and 4th columns from our `drug_trial` data frame.

Finally, in addition to using `select()` to *keep* columns in our data frame, we can also use `select()` to explicitly *drop* columns from our data frame. To do so, we just need to use either the subtraction symbol (`-`) or the Not operator (`!`).

Think back to our example from the previous chapter. There we created some new variables that captured information about participants reporting *any* and *all* side effects. During that process we created a column that contained a count of the side effects experienced in each year – `n_se_year`.

```
drug_trial_sub <- drug_trial %>%
  rowwise() %>%
  mutate(
    n_se_year   = sum(se_headache, se_diarrhea, se_dry_mouth),
    any_se_year = n_se_year > 0,
    all_se_year = n_se_year == 3
  ) %>%
  group_by(id) %>%
  mutate(any_se = sum(any_se_year) > 0) %>%
  ungroup() %>%
```

```
  select(id:year, n_se_year:any_se) %>%
  print()
```

```
# A tibble: 60 x 6
      id  year n_se_year any_se_year all_se_year any_se
   <int> <int>     <int> <lgl>       <lgl>       <lgl>
 1     1     0         2 TRUE        FALSE       TRUE
 2     1     1         3 TRUE        TRUE        TRUE
 3     1     2         2 TRUE        FALSE       TRUE
 4     2     0         3 TRUE        TRUE        TRUE
 5     2     1         1 TRUE        FALSE       TRUE
 6     2     2         3 TRUE        TRUE        TRUE
 7     3     0         0 FALSE       FALSE       FALSE
 8     3     1         0 FALSE       FALSE       FALSE
 9     3     2         0 FALSE       FALSE       FALSE
10     4     0         0 FALSE       FALSE       FALSE
# i 50 more rows
```

Let's say we decided we don't need `n_se_year` column now that we created `any_se_year`, `all_se_year`, and `any_se`. We can easily drop it from the data frame in a couple of ways:

```
drug_trial_sub %>%
  select(-n_se_year)
```

```
# A tibble: 60 x 5
      id  year any_se_year all_se_year any_se
   <int> <int> <lgl>       <lgl>       <lgl>
 1     1     0 TRUE        FALSE       TRUE
 2     1     1 TRUE        TRUE        TRUE
 3     1     2 TRUE        FALSE       TRUE
 4     2     0 TRUE        TRUE        TRUE
 5     2     1 TRUE        FALSE       TRUE
 6     2     2 TRUE        TRUE        TRUE
 7     3     0 FALSE       FALSE       FALSE
 8     3     1 FALSE       FALSE       FALSE
 9     3     2 FALSE       FALSE       FALSE
10     4     0 FALSE       FALSE       FALSE
# i 50 more rows
```

```
drug_trial_sub %>%
  select(!n_se_year)
```

```
# A tibble: 60 x 5
      id  year any_se_year all_se_year any_se
   <int> <int> <lgl>       <lgl>       <lgl>
 1     1     0 TRUE        FALSE       TRUE
 2     1     1 TRUE        TRUE        TRUE
 3     1     2 TRUE        FALSE       TRUE
 4     2     0 TRUE        TRUE        TRUE
 5     2     1 TRUE        FALSE       TRUE
 6     2     2 TRUE        TRUE        TRUE
 7     3     0 FALSE       FALSE       FALSE
 8     3     1 FALSE       FALSE       FALSE
 9     3     2 FALSE       FALSE       FALSE
10     4     0 FALSE       FALSE       FALSE
# i 50 more rows
```

Note that we could have also dropped it indirectly by selecting everything else:

```
drug_trial_sub %>%
  select(id:year, any_se_year:any_se)
```

```
# A tibble: 60 x 5
      id  year any_se_year all_se_year any_se
   <int> <int> <lgl>       <lgl>       <lgl>
 1     1     0 TRUE        FALSE       TRUE
 2     1     1 TRUE        TRUE        TRUE
 3     1     2 TRUE        FALSE       TRUE
 4     2     0 TRUE        TRUE        TRUE
 5     2     1 TRUE        FALSE       TRUE
 6     2     2 TRUE        TRUE        TRUE
 7     3     0 FALSE       FALSE       FALSE
 8     3     1 FALSE       FALSE       FALSE
 9     3     2 FALSE       FALSE       FALSE
10     4     0 FALSE       FALSE       FALSE
# i 50 more rows
```

But, we think this is generally a bad idea. Not only is it more typing, but skimming through your code doesn't really tell us (or future you) what you were trying to accomplish there.

27.2 The rename() function

Sometimes, we want to change the names of some, or all, of the columns in our data frame. For me, this most commonly comes up with data I've imported from someone else. For example, let's say I'm importing data that uses column names that aren't super informative. We saw column names like that when we imported NHANES data. It looked something like this:

```
nhanes <- tibble(
  SEQN = c(1:4),
  ALQ101 = c(1, 2, 1, 2),
  ALQ110 = c(2, 2, 2, 1)
) %>%
  print()
```

```
# A tibble: 4 x 3
   SEQN ALQ101 ALQ110
  <int>  <dbl>  <dbl>
1     1      1      2
2     2      2      2
3     3      1      2
4     4      2      1
```

we previously learned how to change these column names on import (i.e., `col_names`), but let's say we didn't do that for whatever reason. We can rename columns in our data frame using the `rename()` function like so:

```
nhanes %>%
  rename(
    id = SEQN,
    drinks_12_year = ALQ101,
    drinks_12_life = ALQ110
  )
```

```
# A tibble: 4 x 3
     id drinks_12_year drinks_12_life
  <int>          <dbl>          <dbl>
1     1              1              2
2     2              2              2
3     3              1              2
4     4              2              1
```

Here's what we did above:

- we used the `rename()` function to change the name of each column in the `drug_trial` data frame to be more informative.
- You can type `?rename` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `rename()` function is `.data`. The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% rename()`).
- The second argument to the `rename()` function is `....` The value passed to the `...` argument should be a name value pair, or series of name-value pairs separated by columns. The name-value pairs should be in the format `new name = original name`.

we think these names are much better, but for the sake of argument let's say that we wanted to keep the original names – just coerce them to lowercase. We can do that using the `rename_with()` variation of the `rename()` function in combination with the `tolower()` function:

```
nhanes %>%
  rename_with(tolower)
```

```
# A tibble: 4 x 3
   seqn alq101 alq110
  <int>  <dbl>  <dbl>
1     1      1      2
2     2      2      2
3     3      1      2
4     4      2      1
```

Here's what we did above:

- we used the `rename_with()` function to coerce all column names in the `drug_trial` data frame to lowercase.
- You can type `?rename` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `rename_with()` function is `.data`. The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% rename_with()`).

- The second argument to the `rename_with()` function is `.fn`. The value passed to the `.fn` argument should be a function that you want to apply to all the columns selected in the `.cols` argument (see below).
- The third argument to the `rename_with()` function is `.cols`. The value passed to the `.cols` argument should be the columns you want to apply the function passed to the `.fn` argument to. You can select the columns using tidy-select argument modifiers.

27.3 The filter() function

we just saw how to keep and drop *columns* in our data frame using the `select()` function. We can keep and drop *rows* in our data frame using the *filter()* function or the *slice()* function.

Similar to selecting columns by position instead of name:

```
drug_trial %>%
  select(1:2, 4)
```

```
# A tibble: 60 x 3
      id  year drug
   <int> <int> <chr>
 1     1     0 Active
 2     1     1 Active
 3     1     2 Active
 4     2     0 Active
 5     2     1 Active
 6     2     2 Active
 7     3     0 Placebo
 8     3     1 Placebo
 9     3     2 Placebo
10     4     0 Placebo
# i 50 more rows
```

we can also select rows we want to keep by position. Again, we don't do this often, but it is sometimes useful when we're writing our own functions. Therefore, we want to quickly show you what this looks like:

```
drug_trial %>%
  slice(1:5)
```

```
# A tibble: 5 x 12
     id  year   age drug   se_headache se_diarrhea se_dry_mouth    mi age_center
  <int> <int> <int> <chr>        <int>       <int>        <int> <int>      <dbl>
1     1     0    65 Active           0           1            1     0       11.3
2     1     1    65 Active           1           1            1     0       11.3
3     1     2    65 Active           1           1            0     0       11.3
4     2     0    49 Active           1           1            1     0       -4.7
5     2     1    49 Active           0           0            1     0       -4.7
# i 3 more variables: se_headache_f <fct>, se_diarrhea_f <fct>,
#   se_dry_mouth_f <fct>
```

Here's what we did above:

- we used the `slice()` function to keep only the first 5 rows in the `drug_trial` data frame.
- You can type `?slice` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `slice()` function is `.data`. The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% slice()`).
- The second argument to the `slice()` function is `....` The value passed to the `...` argument should be a row numbers you want returned to you.

Generally speaking, we're far more likely to use the `filter()` function to select only a subset of rows from our data frame. Two of the most common scenarios, of many possible scenarios, where want to subset rows include:

- Performing a subgroup analysis. This is a situation where we want our analysis to include only some of the people (or places, or things) in our data frame.
- Performing a complete case analysis. This is a situation where we want to remove rows that contain missing values from our data frame before performing an analysis.

27.3.1 Subgroup analysis

Let's say that we want to count the number of people in the drug trial who reported having headaches in the baseline year by drug status (active vs. placebo). We would first use `filter()` to keep only the rows that contain data from the baseline year:

```
drug_trial %>%
  filter(year == 0)
```

```
# A tibble: 20 x 12
      id  year   age drug  se_headache se_diarrhea se_dry_mouth    mi age_center
   <int> <int> <int> <chr>       <int>       <int>        <int> <int>      <dbl>
 1     1     0    65 Acti~           0           1            1     0     11.3
 2     2     0    49 Acti~           1           1            1     0     -4.7
 3     3     0    48 Plac~           0           0            0     0     -5.7
 4     4     0    37 Plac~           0           0            0     0    -16.7
 5     5     0    71 Plac~           0           0            0     0     17.3
 6     6     0    48 Plac~           0           0            0     0     -5.7
 7     7     0    59 Acti~           1           1            1     0      5.3
 8     8     0    60 Plac~           0           0            0     0      6.3
 9     9     0    61 Acti~           1           1            1     0      7.3
10    10     0    39 Acti~           1           0            1     0    -14.7
11    11     0    61 Plac~           0           0            0     0      7.3
12    12     0    62 Plac~           1           0            1     0      8.3
13    13     0    43 Plac~           0           0            0     0    -10.7
14    14     0    63 Plac~           0           0            0     0      9.3
15    15     0    69 Acti~           1           1            1     0     15.3
16    16     0    42 Plac~           0           0            0     0    -11.7
17    17     0    60 Plac~           0           0            0     0      6.3
18    18     0    41 Acti~           1           1            1     0    -12.7
19    19     0    43 Plac~           0           0            0     0    -10.7
20    20     0    53 Plac~           0           0            0     0     -0.700
# i 3 more variables: se_headache_f <fct>, se_diarrhea_f <fct>,
#   se_dry_mouth_f <fct>
```

Here's what we did above:

- we used the `filter()` function to keep only the rows in the `drug_trial` data frame that contain data from the baseline year.
- You can type `?filter` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `filter()` function is `.data`. The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% filter()`).
- The second argument to the `filter()` function is `....` The value passed to the `...` argument should be a name-value pair or multiple name value pairs separated by commas. The `...` argument is where you will tell `filter()` how to decide which rows to keep.

> **Warning**
>
> Remember, that in the R language `=` (i.e., one equal sign) and `==` (i.e., two equal signs) are different things. The `=` operator *tells* R to *make* the thing on the left equal to the thing on the right. In other words, it *assigns* values. The `==` *asks* R if the thing on the left is equal to the thing on the right. In other words, it *test the equality* of values.

Now, we can use the descriptive analysis techniques we've already learned to answer our research question:

```
drug_trial %>%
  filter(year == 0) %>%
  group_by(drug, se_headache_f) %>%
  summarise(n = n())
```

```
# A tibble: 4 x 3
# Groups:   drug [2]
  drug    se_headache_f     n
  <chr>   <fct>         <int>
1 Active  No                1
2 Active  Yes               6
3 Placebo No               12
4 Placebo Yes               1
```

So, 6 out of 7 ($\sim 86\%$) of the people in our active drug group reported headaches in the baseline year. Now, let's say that we have reason to suspect that the drug affects people differently based on their age. Let's go ahead and repeat this analysis, but only in a subgroup of people who are below age 65. Again, we can use the `filter()` function to do this:

```
drug_trial %>%
  filter(year == 0) %>%
  filter(age < 65) %>%
  group_by(drug, se_headache_f) %>%
  summarise(n = n())
```

```
# A tibble: 3 x 3
# Groups:   drug [2]
  drug    se_headache_f     n
  <chr>   <fct>         <int>
1 Active  Yes               5
2 Placebo No               11
3 Placebo Yes               1
```

Wow! It looks like everyone under age 65 who received active drug also reported headaches!

we can show this more explicitly by using passing the value `FALSE` to the `.drop` argument of `group_by()`. This tells R to keep all factor levels in the output, even if they were *observed* in the data zero times.

```
drug_trial %>%
  filter(year == 0) %>%
  filter(age < 65) %>%
  group_by(drug, se_headache_f, .drop = FALSE) %>%
  summarise(n = n())
```

```
# A tibble: 4 x 3
# Groups:   drug [2]
  drug    se_headache_f     n
  <chr>   <fct>         <int>
1 Active  No                0
2 Active  Yes               5
3 Placebo No               11
4 Placebo Yes               1
```

Finally, we could make our code above more succinct by combining our two filter functions into one:

```
drug_trial %>%
  filter(year == 0 & age < 65) %>%
  group_by(drug, se_headache_f, .drop = FALSE) %>%
  summarise(n = n())
```

```
# A tibble: 4 x 3
# Groups:   drug [2]
  drug    se_headache_f     n
  <chr>   <fct>         <int>
1 Active  No                0
2 Active  Yes               5
3 Placebo No               11
4 Placebo Yes               1
```

Here's what we did above:

- we used the `filter()` function to keep only the rows in the `drug_trial` data frame that contain data from the baseline year *AND* (`&`) contain data from rows with a value that is less than 65 in the `age` column. The *AND* (`&`) here is important. A row must satisfy both of these conditions in order for R to keep it in the returned data frame. If we had used *OR* instead (`filter(year == 0 | age < 65)`), then only one condition *OR* the other would need to be met for R to keep the row in the returned data frame.

> **Note**
>
> In the R language, we use the pipe operator to create *OR* conditions. The pipe operator looks like `|` and is probably the key immediately to the right of your enter/return key on your keyboard.

27.3.2 Complete case analysis

Now let's say that we want to compare age at baseline by drug status (active vs. placebo). Additionally, let's say that we have some missing values in our data.

Let's first simulate some new data with missing values:

```
drug_trial_short <- drug_trial %>%
  filter(year == 0) %>%
  slice(1:10) %>%
  mutate(
    age  = replace(age, 1, NA),
    drug = replace(drug, 4, NA)
  ) %>%
  print()
```

```
# A tibble: 10 x 12
      id  year   age drug  se_headache se_diarrhea se_dry_mouth    mi age_center
   <int> <int> <int> <chr>       <int>       <int>        <int> <int>      <dbl>
 1     1     0    NA Acti~           0           1            1     0       11.3
 2     2     0    49 Acti~           1           1            1     0       -4.7
 3     3     0    48 Plac~           0           0            0     0       -5.7
 4     4     0    37 <NA>            0           0            0     0      -16.7
 5     5     0    71 Plac~           0           0            0     0       17.3
 6     6     0    48 Plac~           0           0            0     0       -5.7
 7     7     0    59 Acti~           1           1            1     0        5.3
 8     8     0    60 Plac~           0           0            0     0        6.3
 9     9     0    61 Acti~           1           1            1     0        7.3
10    10     0    39 Acti~           1           0            1     0      -14.7
```

```
# i 3 more variables: se_headache_f <fct>, se_diarrhea_f <fct>,
#   se_dry_mouth_f <fct>
```

Here's what we did above:

- we used the `filter()` and `slice()` functions to create a new data frame that contains only a subset of our original `drug_trial` data frame. The subset includes only the first 10 rows of the data frame remaining after selecting only the baseline year rows from the original data frame.
- we used the `replace()` function to replace the first value of age with `NA` and the fourth value of `drug` with `NA`.
- You can type `?replace` into your R console to view the help documentation for this function.

If we try to answer our research question above without dealing with the missing data, we get the following undesirable results:

```
drug_trial_short %>%
  group_by(drug) %>%
  summarise(mean_age = mean(age))
```

```
# A tibble: 3 x 2
  drug    mean_age
  <chr>      <dbl>
1 Active      NA
2 Placebo     56.8
3 <NA>        37
```

One way we can improve our result is by adding the `na.rm` argument to the `mean()` function.

```
drug_trial_short %>%
  group_by(drug) %>%
  summarise(mean_age = mean(age, na.rm = TRUE))
```

```
# A tibble: 3 x 2
  drug    mean_age
  <chr>      <dbl>
1 Active      52
2 Placebo     56.8
3 <NA>        37
```

But, we previously saw how it can sometimes be more efficient to drop the row with missing data from the data frame explicitly. This is called a **complete case analysis** or **list-wise deletion**.

```
drug_trial_short %>%
  filter(!is.na(age)) %>%
  group_by(drug) %>%
  summarise(mean_age = mean(age))
```

```
# A tibble: 3 x 2
  drug    mean_age
  <chr>      <dbl>
1 Active      52
2 Placebo     56.8
3 <NA>        37
```

However, we still have that missing value for `drug`. We can easily drop the row with the missing value by adding an additional value to the `...` argument of our `filter()` function:

```
drug_trial_short %>%
  filter(!is.na(age) & !is.na(drug)) %>%
  group_by(drug) %>%
  summarise(mean_age = mean(age))
```

```
# A tibble: 2 x 2
  drug    mean_age
  <chr>      <dbl>
1 Active      52
2 Placebo     56.8
```

27.4 Deduplication

Another common data management task that we want to discuss in this chapter is deduplicating data. Let's go ahead and simulate some data to illustrate what we mean:

```
df <- tribble(
  ~id, ~day, ~x,
    1, 1, 1,
    1, 2, 11,
```

```
    2, 1, 12,
    2, 2, 13,
    2, 2, 14,
    3, 1, 12,
    3, 1, 12,
    3, 2, 13,
    4, 1, 13,
    5, 1, 10,
    5, 2, 11,
    5, 1, 10
) %>%
  print()
```

```
# A tibble: 12 x 3
      id   day     x
   <dbl> <dbl> <dbl>
 1     1     1     1
 2     1     2    11
 3     2     1    12
 4     2     2    13
 5     2     2    14
 6     3     1    12
 7     3     1    12
 8     3     2    13
 9     4     1    13
10     5     1    10
11     5     2    11
12     5     1    10
```

- All id's but 4 have multiple observations.
- ID 2 has row with duplicate values for `id` and `day`, but a non-duplicate value for `x`. These rows are partial duplicates.
- ID 3 has a row with duplicate values for all three columns (i.e., `3, 1, 12`). These rows are complete duplicates.
- ID 5 has a row with duplicate values for all three columns (i.e., `5, 1, 10`). These rows are complete duplicates. However, they are not in sequential order in the dataset.

27.4.1 The distinct() function

we can use `dplyr`'s `distinct()` function to remove all complete duplicates from the data frame:

```
df %>%
  distinct()
```

```
# A tibble: 10 x 3
      id   day     x
   <dbl> <dbl> <dbl>
 1     1     1     1
 2     1     2    11
 3     2     1    12
 4     2     2    13
 5     2     2    14
 6     3     1    12
 7     3     2    13
 8     4     1    13
 9     5     1    10
10     5     2    11
```

Here's what we did above:

- we used the `distinct()` function to keep only one row from a group of complete duplicate rows in the `df` data frame.
- You can type `?distinct` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `distinct()` function is `.data`. The value passed to `.data` should always be a data frame. In this book, we will often pass data frames to the `.data` argument using the pipe operator (e.g., `df %>% distinct()`).
- The second argument to the `distinct()` function is `....` The value passed to the `...` argument should be the variables to use when determining uniqueness. Passing no variables to the `...` argument is equivalent to pass all variables to the `...` argument.

27.4.2 Complete duplicate row add tag

If want to identify the complete duplicate rows, without immediately dropping them, we can use the `duplicated()` function inside the `mutate()` function. This creates a new column in

our data frame that has the value `TRUE` when the row is a complete duplicate and the value `FALSE` otherwise.

```
df %>%
  mutate(dup = duplicated(df))
```

```
# A tibble: 12 x 4
      id   day     x dup
   <dbl> <dbl> <dbl> <lgl>
 1     1     1     1 FALSE
 2     1     2    11 FALSE
 3     2     1    12 FALSE
 4     2     2    13 FALSE
 5     2     2    14 FALSE
 6     3     1    12 FALSE
 7     3     1    12 TRUE
 8     3     2    13 FALSE
 9     4     1    13 FALSE
10     5     1    10 FALSE
11     5     2    11 FALSE
12     5     1    10 TRUE
```

Alternatively, we could get the same result using:

```
df %>%
  group_by_all() %>%
  mutate(
    n_row = row_number(),
    dup   = n_row > 1
  )
```

```
# A tibble: 12 x 5
# Groups:   id, day, x [10]
      id   day     x n_row dup
   <dbl> <dbl> <dbl> <int> <lgl>
 1     1     1     1     1 FALSE
 2     1     2    11     1 FALSE
 3     2     1    12     1 FALSE
 4     2     2    13     1 FALSE
 5     2     2    14     1 FALSE
 6     3     1    12     1 FALSE
```

```
 7     3     1    12     2 TRUE
 8     3     2    13     1 FALSE
 9     4     1    13     1 FALSE
10     5     1    10     1 FALSE
11     5     2    11     1 FALSE
12     5     1    10     2 TRUE
```

Here's what we did above:

- we used the `group_by_all()` function to split our data frame into multiple data frames grouped by all the columns in `df`.
- we used the `row_number()` to sequentially count every row in each of the little data frames created by `group_by_all()`. We assigned the sequential count to a new column named `n_row`.
- we created a new column named `dup` that has a value of `TRUE` when the value of `n_row` is greater than 1 and `FALSE` otherwise.

Notice that R only tags the second in a set of duplicate rows as a duplicate. Below we tag both rows with complete duplicate values.

```
df %>%
  mutate(dup = duplicated(.) | duplicated(., fromLast = TRUE))
```

```
# A tibble: 12 x 4
      id   day     x dup
   <dbl> <dbl> <dbl> <lgl>
 1     1     1     1 FALSE
 2     1     2    11 FALSE
 3     2     1    12 FALSE
 4     2     2    13 FALSE
 5     2     2    14 FALSE
 6     3     1    12 TRUE
 7     3     1    12 TRUE
 8     3     2    13 FALSE
 9     4     1    13 FALSE
10     5     1    10 TRUE
11     5     2    11 FALSE
12     5     1    10 TRUE
```

27.4.3 Partial duplicate rows

```
df %>%
  distinct(id, day, .keep_all = TRUE)
```

```
# A tibble: 9 x 3
     id   day     x
  <dbl> <dbl> <dbl>
1     1     1     1
2     1     2    11
3     2     1    12
4     2     2    13
5     3     1    12
6     3     2    13
7     4     1    13
8     5     1    10
9     5     2    11
```

Here's what we did above:

- we used the `distinct()` function to keep only one row from a group of duplicate rows in the `df` data frame.
- You can type `?distinct` into your R console to view the help documentation for this function and follow along with the explanation below.
- This time we passed the column names `id` and `day` to the `...` argument. This tells R to consider any rows that have the same value of `id` *AND* `day` to be duplicates – even if they have different values in their other columns.
- The `.keep_all` argument tells R to return all of the columns in `df` to us – not just the columns that we are testing for uniqueness (i.e., `id` and `day`).

27.4.4 Partial duplicate rows - add tag

we can tag partial duplicate rows in a similar fashion to the way we tagged complete duplicate rows above:

```r
df %>%
  group_by(id, day) %>%
  mutate(
    count = row_number(), # Counts rows by group
    dup   = count > 1     # TRUE if there is more than one row per group
  )
```

```
# A tibble: 12 x 5
# Groups:   id, day [9]
      id   day     x count dup
   <dbl> <dbl> <dbl> <int> <lgl>
 1     1     1     1     1 FALSE
 2     1     2    11     1 FALSE
 3     2     1    12     1 FALSE
 4     2     2    13     1 FALSE
 5     2     2    14     2 TRUE
 6     3     1    12     1 FALSE
 7     3     1    12     2 TRUE
 8     3     2    13     1 FALSE
 9     4     1    13     1 FALSE
10     5     1    10     1 FALSE
11     5     2    11     1 FALSE
12     5     1    10     2 TRUE
```

27.4.5 Count the number of duplicates

Finally, sometimes it can be useful to get a count of the number of duplicate rows. The code below returns a data frame that summarizes the number of rows that contain duplicate values for `id` and `day`, and what those duplicate values are.

```r
df %>%
  group_by(id, day) %>%
  filter(n() > 1) %>%
  count()
```

```
# A tibble: 3 x 3
# Groups:   id, day [3]
     id   day     n
  <dbl> <dbl> <int>
1     2     2     2
```

```
2     3     1     2
3     5     1     2
```

27.4.6 What to do about duplicates

Finding duplicates is only half the battle. After finding them, you have to decide what to do about them. In some ways it's hard to give clear-cut advice on this because different situations require different decisions. However, here are some things you may want to consider:

- If two or more rows are complete duplicates, then the additional rows provide no additional information. I have a hard time thinking of a scenario where dropping them would be a problem. Additionally, because they are completely identical, it doesn't matter which row you drop.
- If have two more rows that are partial duplicates, then you will want to look for obvious errors in the other variables. When you have two rows that are partial duplicates, and one row has very obvious errors in it, then keeping the row without the obvious errors is *usually* the correct decision. Having said that, you should meticulously document which rows you dropped and why, and make that information known to anyone consuming the results of your analysis.
- When there are no obvious errors, deciding which rows to keep and which to drop can be really tricky. In this situation the best advice I can give is to be systematic in your approach. What I mean by that is to choose a strategy that seems least likely to introduce bias into your data and then apply that strategy consistently throughout your data. So, something like always keeping the first row among a group of duplicate rows. However, keep in mind that if rows are ordered by data, this strategy could easily introduce bias. In that case, some other strategy may be more appropriate. And again, you should meticulously document which rows you dropped and why, and make that information known to anyone consuming the results of your analysis.
- Finally, I can definitively tell you a strategy that you should *never* use. That is, you should never pick and choose, or even give the appearance of picking and choosing, rows with values that are aligned with the results you want to see. I hope the unethical nature of this strategy is blatantly obvious to you.

Congratulations! At this point, you are well-versed in all of the `dplyr` verbs. More importantly, you now have a foundation of tools you can call upon to complete the many of basic data management tasks that you will encounter. In the rest of the data management part of the book we will build on these tools, and learn some new tools, we can use to solve more complex data management problems.

28 Working with Dates

In epidemiology, it isn't uncommon at all for the data we are analyzing to include important date values. Some common examples include date of birth, hospital admission date, date of symptom onset, and follow-up dates in longitudinal studies. In this chapter, we will learn about two new vector types that we can use to work with date and date-time data. Additionally, we will learn about a new package, `lubridate`, which provides a robust set of functions designed specifically for working with date and date-time data in R.

28.1 Date vector types

In R, there are two different vector types that we can use to store, and work with, dates. They are:

`date` vectors for working with date values. By default, R will display dates in this format: 4-digit year, a dash, 2-digit month, a dash, and 2-digit day. For example, the date that the University of Florida won its last national football championship, January 8, 2009, looks like this as a date in R: `2009-01-08`. It's about time for another championship!

`POSIXct` vectors for working with date-time values. Date-time values are just dates with time values added to them. By default, R will display date-times in this format: 4-digit year, a dash, 2-digit month, a dash, 2-digit day, a space, 2-digit hour value, a colon, 2-digit minute value, a colon, and 2-digit second value. So, let's say that kickoff for the previously mentioned national championship game was at 8:00 PM local time. In R, that looks like this: `2009-01-08 20:00:00`.

> **Note**
>
> You were probably pretty confused when you saw the 20:00:00 above if you've never used 24-hour clock time (also called military time) before. We'll let you read the details on Wikipedia, but here's a couple of simple tips to get you started working with 24-hour time. Any time before noon is written the same as you would write it if you were using 12-hour (AM/PM) time. So, 8:00 AM would be 8:00 in 24-hour time. After noon, just add 12 to whatever time you want to write. So, 1:00 PM is 13:00 (1 + 12 = 13) and 8:00 PM is 20:00 (8 + 12 = 20).

> **Note**
>
> Base R does not have a built-in vector type for working with pure time (as opposed to date-time) values. If you need to work with pure time values only, then the hms package is what you want to try first.

In general, we try to work with date values, rather than date-time values, whenever possible. Working with date-time values is slightly more complicated than working with date values, and we rarely have time data anyway. However, that doesn't stop some R functions from trying to store dates as POSIXct vectors by default, which can sometimes cause unexpected errors in our R code. But, don't worry. We are going to show you how to coerce POSIXct vectors to date vectors below.

Before we go any further, let's go ahead and look at some data that we can use to help us learn to work with dates in R.

You can click here to download the data and import it into your R session, if you want to follow along.

```
Rows: 10 Columns: 6
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (4): name_first, name_last, dob_typical, dob_long
dttm (1): dob_actual
date (1): dob_default

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: 10 x 6
   name_first name_last dob_actual          dob_default dob_typical dob_long
   <chr>      <chr>     <dttm>              <date>      <chr>       <chr>
 1 Nathaniel  Watts     1996-03-04 16:59:18 1996-03-04  03/04/1996  March 04, 1~
 2 Sophia     Gomez     1998-11-21 21:52:08 1998-11-21  11/21/1998  November 21~
 3 Emmett     Steele    1994-09-03 23:26:19 1994-09-03  09/03/1994  September 0~
 4 Levi       Sanchez   1996-08-03 17:18:50 1996-08-03  08/03/1996  August 03, ~
 5 August     Murray    1980-06-13 18:27:13 1980-06-13  06/13/1980  June 13, 19~
 6 Juan       Clark     1996-12-09 05:33:24 1996-12-08  12/08/1996  December 08~
 7 Lilly      Levy      1992-11-27 17:36:43 1992-11-27  11/27/1992  November 27~
 8 Natalie    Rogers    1983-04-27 23:31:56 1983-04-27  04/27/1983  April 27, 1~
 9 Solomon    Harding   1988-06-28 16:13:46 1988-06-28  06/28/1988  June 28, 19~
10 Olivia     House     1997-08-02 22:09:50 1997-08-02  08/02/1997  August 02, ~
```

Here's what we did above:

- we used the `read_csv()` function to import a csv file containing simulated data into R.
- The simulated data contains the first name, last name, and date of birth for 10 fictitious people.
- In this data, date of birth is recorded in the four most common formats that we typically come across.

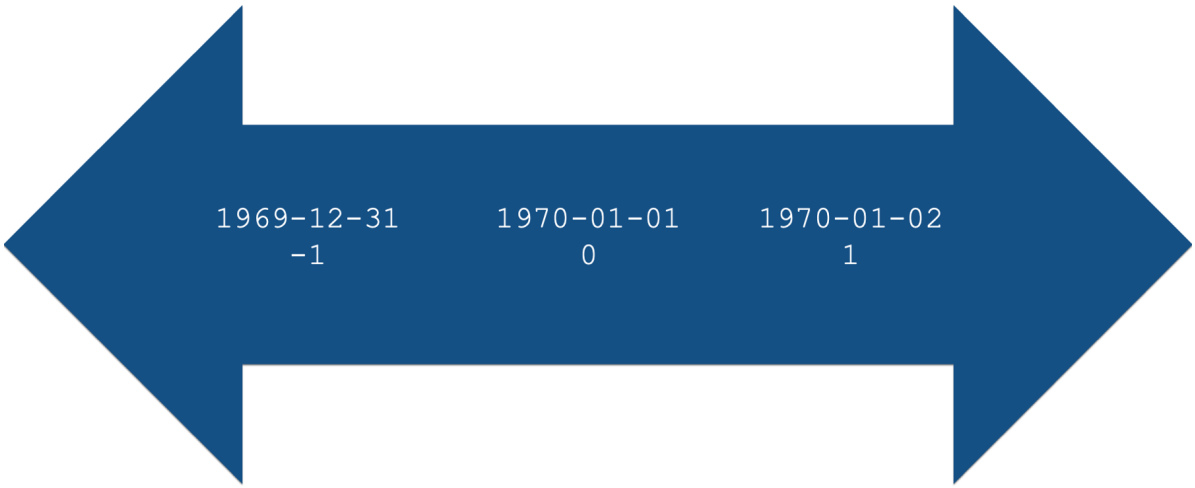
1. `dob_actual` is each person's *actual* date of birth measured down to the second. Notice that this column's type is `<S3: POSIXct>`. Again, that means that this vector contains date-time values. Also, notice that the format of these values matches the format we discussed for date-time vectors above: 4-digit year, a dash, 2-digit month, a dash, 2-digit day, a space, 2-digit hour value, a colon, 2-digit minute value, a colon, and 2-digit second value.
2. `dob_default` is each person's date of birth without their time of birth included. Notice that this column's type is `<date>`. Also, notice that the format of these values matches the format we discussed for date vectors above: 4-digit year, a dash, 2-digit month, a dash, and 2-digit day.
3. `dob_typical` is each person's date of birth written in the format that is probably most often used in the United States: 2-digit month, a forward slash, 2-digit day, a forward slash, and 4-digit year.
4. `dob_long` is each person's date of birth written out in a sometimes-used long format. That is, the month name written out, 2-digit day, a comma, and 4-digit year.

- Notice that `readr` did a good job of importing `dob_actual` and `dob_default` as date-time and date values respectively. It did so because the values were stored in the csv file in the default format that R expects to see date-time and date values have.
- Notice that `readr` imported `dob_typical` and `dob_long` as character strings. It does so because the values in these columns were not stored in a format that R recognizes as a date or date-time.

28.2 Dates under the hood

Under the hood, R actually stores dates as numbers. Specifically, the number of days before or after January 1st, 1970, 00:00:00 UTC.

> **Note**
>
> Why January 1st, 1970, 00:00:00 UTC? Well, it's not really important to know the answer for the purposes of this book, or for programming in R, but Kristina Hill (a former student) figured out the answer for those of you who are curious. New Year's Day in 1970 was an easy date for early Unix developers to use as a uniform date for the start of time. So, January 1st, 1970 at 00:00:00 UTC is referred to as the "Unix epoch", and it's a popular epoch used by many (but not all) software platforms. The use of any epoch date is mostly arbitrary, and this one leads to some interesting situations (like the Year 2038 Problem and this little issue that Apple had a few years ago (yikes!). Generally speaking, though, this is in no way likely to impact your day-to-day programming in R, or your life at all (unless you happen to also be a software developer in a platform that uses this epoch date).



For example, let's use base R's `as.Date()` function to create a date value from the string "2000-01-01".

```
as.Date("2000-01-01")
```

```
[1] "2000-01-01"
```

On the surface, it doesn't look like anything happened. However, we can use base R's `unclass()` function to see R's internal integer representation of the date.

```
unclass(as.Date("2000-01-01"))
```

```
[1] 10957
```

Specifically, January 1st, 2000 is apparently 10,957 days after January 1st, 1970. What number would you expect to be returned if we used the date "1970-01-01"?

```
unclass(as.Date("1970-01-01"))
```

```
[1] 0
```

What number would you expect to be returned if we used the date "1970-01-02"?

```
unclass(as.Date("1970-01-02"))
```

```
[1] 1
```

And finally, what number would you expect to be returned if we used the date "1969-12-31"?

```
unclass(as.Date("1969-12-31"))
```

```
[1] -1
```

This numeric representation of dates also works in the other direction. For example, we can pass the number 10,958 to the `as.Date()` function, along with the date origin, and R will return a human-readable date.

```
as.Date(10958, origin = "1970-01-01")
```

```
[1] "2000-01-02"
```

You may be wondering why we had to tell R the date origin. After all, didn't we already say that the origin is January 1st, 1970? Well, not all programs and programming languages use the same date origin. For example, SAS uses the date January 1st, 1960 as its origin. In our experience, this differing origin value can occasionally give us incorrect dates. When that happens, one option is to strip the date value down to its numeric representation, and then tell R what the origin was for that numeric representation in the program you are importing the data from.

For example, if we imported a data set from SAS, we could correctly produce human-readable dates in the manner shown below:

```
from_sas <- tibble(
  date = c(10958, 10959, 10960)
)
```

```
from_sas %>%
  mutate(new_date = as.Date(date, origin = "1960-01-01"))
```

```
# A tibble: 3 x 2
   date new_date
  <dbl> <date>
1 10958 1990-01-01
2 10959 1990-01-02
3 10960 1990-01-03
```

Hopefully, you now have a good intuition about how R stores dates under the hood. This numeric representation of dates is what will allow us to perform calculations with dates later in the chapter.

28.3 Coercing date-times to dates

As we said above, it's usually preferable to work with date values instead of date-time values. Fortunately, converting date-time values to dates is usually really easy. All we need to do is pass those values to the same `as.Date()` function we already saw above. For example:

```
birth_dates %>%
  mutate(posix_to_date = as.Date(dob_actual)) %>%
  select(dob_actual, posix_to_date)
```

```
# A tibble: 10 x 2
   dob_actual          posix_to_date
   <dttm>              <date>
 1 1996-03-04 16:59:18 1996-03-04
 2 1998-11-21 21:52:08 1998-11-21
 3 1994-09-03 23:26:19 1994-09-03
 4 1996-08-03 17:18:50 1996-08-03
 5 1980-06-13 18:27:13 1980-06-13
 6 1996-12-09 05:33:24 1996-12-09
 7 1992-11-27 17:36:43 1992-11-27
 8 1983-04-27 23:31:56 1983-04-27
```

```
 9 1988-06-28 16:13:46 1988-06-28
10 1997-08-02 22:09:50 1997-08-02
```

Here's what we did above:

- we created a new column in the `birth_dates` data frame called `posix_to_date`.
- we used the `as.Date()` function to coerce the date-time values in `dob_actual` to dates. In other words, we dropped the time part of the date-time. Make sure to capitalize the "D" in `as.Date()`.
- we used the `select()` function to keep only the columns we are interested in comparing side-by-side in our output.
- Notice that `dob_actual`'s column type is still `<S3: POSIXct>`, but `posix_to_date`'s column type is `<date>`.

28.4 Coercing character strings to dates

Converting character strings to dates can be slightly more complicated than converting date-times to dates. This is because we have to explicitly tell R which characters in the character string correspond to each date component. For example, let's say we have a date value of `04-05-06`. Is that April 5th, 2006? Is it April 5th, 1906? Or perhaps it's May 6th, 2004?

we need to use a series of special symbols to tell R which characters in the character string correspond to each date component. We'll list some of the most common ones first and then show you how to use them. The examples below assume that date each symbol is being applied to is `2000-01-15`.

```
tribble(
  ~Symbol, ~Description, ~Example,
  "%a", "Abbreviated weekday name", "Sat",
  "%A", "Full weekday name", "Saturday",
  "%b", "Abbreviated month name", "Jan",
  "%B", "Full month name", "January",
  "%d", "Day of the month as a number (01-31)", "15",
  "%m", "Month as a number", "01",
  "%u", "Weekday as a number (1-7, Monday is 1)", "6",
  "%U", "Week of the year as a number (00-53) using Sunday as the first day 1 of the week",
  "%y", "Year without century (00-99)", "00",
  "%Y", "Year with century", "2000"
) %>%
  knitr::kable()
```

Symbol	Description	Example
%a	Abbreviated weekday name	Sat
%A	Full weekday name	Saturday
%b	Abbreviated month name	Jan
%B	Full month name	January
%d	Day of the month as a number (01–31)	15
%m	Month as a number	01
%u	Weekday as a number (1–7, Monday is 1)	6
%U	Week of the year as a number (00–53) using Sunday as the first day 1 of the week	02
%y	Year without century (00-99)	00
%Y	Year with century	2000

Now that we have a list of useful symbols that we can use to communicate with R, let's take another look at our birth date data.

```
birth_dates
```

```
# A tibble: 10 x 6
   name_first name_last dob_actual          dob_default dob_typical dob_long
   <chr>      <chr>     <dttm>              <date>      <chr>       <chr>
 1 Nathaniel  Watts     1996-03-04 16:59:18 1996-03-04  03/04/1996  March 04, 1~
 2 Sophia     Gomez     1998-11-21 21:52:08 1998-11-21  11/21/1998  November 21~
 3 Emmett     Steele    1994-09-03 23:26:19 1994-09-03  09/03/1994  September 0~
 4 Levi       Sanchez   1996-08-03 17:18:50 1996-08-03  08/03/1996  August 03, ~
 5 August     Murray    1980-06-13 18:27:13 1980-06-13  06/13/1980  June 13, 19~
 6 Juan       Clark     1996-12-09 05:33:24 1996-12-08  12/08/1996  December 08~
 7 Lilly      Levy      1992-11-27 17:36:43 1992-11-27  11/27/1992  November 27~
 8 Natalie    Rogers    1983-04-27 23:31:56 1983-04-27  04/27/1983  April 27, 1~
 9 Solomon    Harding   1988-06-28 16:13:46 1988-06-28  06/28/1988  June 28, 19~
10 Olivia     House     1997-08-02 22:09:50 1997-08-02  08/02/1997  August 02, ~
```

For our first example, let's try converting the character strings stored in the `dob_typical` to date values. Let' start by passing the values to `as.Date()` exactly as we did above and see what happens:

```
birth_dates %>%
  mutate(dob_typical_to_date = as.Date(dob_typical)) %>%
  select(dob_typical, dob_typical_to_date)
```

```
# A tibble: 10 x 2
   dob_typical dob_typical_to_date
   <chr>       <date>
 1 03/04/1996  0003-04-19
 2 11/21/1998  NA
 3 09/03/1994  0009-03-19
 4 08/03/1996  0008-03-19
 5 06/13/1980  NA
 6 12/08/1996  0012-08-19
 7 11/27/1992  NA
 8 04/27/1983  NA
 9 06/28/1988  NA
10 08/02/1997  0008-02-19
```

This is definitely not the result we wanted, right? Why didn't it work? Well, R was looking for the values in `dob_typical` to have the format 4-digit year, a dash, 2-digit month, a dash, and 2-digit day. In reality, `dob_typical` has the format 2-digit month, a forward slash, 2-digit day, a forward slash, and 4-digit year. Now, all we have to do is tell R how to read this character string as a date using some of the symbols we learned about in the table above.

Let's try again:

```
birth_dates %>%
  mutate(dob_typical_to_date = as.Date(dob_typical, format = "%m %d %Y")) %>%
  select(dob_typical, dob_typical_to_date)
```

```
# A tibble: 10 x 2
   dob_typical dob_typical_to_date
   <chr>       <date>
 1 03/04/1996  NA
 2 11/21/1998  NA
 3 09/03/1994  NA
 4 08/03/1996  NA
 5 06/13/1980  NA
 6 12/08/1996  NA
 7 11/27/1992  NA
 8 04/27/1983  NA
 9 06/28/1988  NA
10 08/02/1997  NA
```

Wait, what? We told R that the values were 2-digit month (`%m`), 2-digit day (`%d`), and 4-digit year (`%Y`). Why didn't it work this time? It didn't work because we didn't pass the forward

slashes to the format argument. Yes, it's that literal. We even have to tell R that there are symbols mixed in with our date values in the character string we want to convert to a date.

Let's try one more time:

```
birth_dates %>%
  mutate(dob_typical_to_date = as.Date(dob_typical, format = "%m/%d/%Y")) %>%
  select(dob_typical, dob_typical_to_date)
```

```
# A tibble: 10 x 2
   dob_typical dob_typical_to_date
   <chr>       <date>
 1 03/04/1996  1996-03-04
 2 11/21/1998  1998-11-21
 3 09/03/1994  1994-09-03
 4 08/03/1996  1996-08-03
 5 06/13/1980  1980-06-13
 6 12/08/1996  1996-12-08
 7 11/27/1992  1992-11-27
 8 04/27/1983  1983-04-27
 9 06/28/1988  1988-06-28
10 08/02/1997  1997-08-02
```

Here's what we did above:

- we created a new column in the `birth_dates` data frame called `dob_typical_to_date`.
- we used the `as.Date()` function to coerce the character string values in `dob_typical` to dates.
- we did so by passing the value `"%m/%d/%Y"` to the `format` argument of the `as.Date()` function. These symbols tell R to read the character strings in `dob_typical` as 2-digit month (`%m`), a forward slash (`/`), 2-digit day (`%d`), a forward slash (`/`), and 4-digit year (`%Y`).
- we used the `select()` function to keep only the columns we are interested in comparing side-by-side in our output.
- Notice that `dob_typical`'s column type is still character (`<chr>`), but `dob_typical_to_date`'s column type is `<date>`.

Let's try one more example, just to make sure we've got this down. Take a look at the `dob_long` column. What value will we need to pass to `as.Date()`'s format argument in order to convert these character strings to dates?

```
select(birth_dates, dob_long)
```

```
# A tibble: 10 x 1
   dob_long
   <chr>
 1 March 04, 1996
 2 November 21, 1998
 3 September 03, 1994
 4 August 03, 1996
 5 June 13, 1980
 6 December 08, 1996
 7 November 27, 1992
 8 April 27, 1983
 9 June 28, 1988
10 August 02, 1997
```

Did you figure it out? The solution is below:

```
birth_dates %>%
  mutate(dob_long_to_date = as.Date(dob_long, format = "%B %d, %Y")) %>%
  select(dob_long, dob_long_to_date)
```

```
# A tibble: 10 x 2
   dob_long           dob_long_to_date
   <chr>              <date>
 1 March 04, 1996     1996-03-04
 2 November 21, 1998  1998-11-21
 3 September 03, 1994 1994-09-03
 4 August 03, 1996    1996-08-03
 5 June 13, 1980      1980-06-13
 6 December 08, 1996  1996-12-08
 7 November 27, 1992  1992-11-27
 8 April 27, 1983     1983-04-27
 9 June 28, 1988      1988-06-28
10 August 02, 1997    1997-08-02
```

Here's what we did above:

- we created a new column in the `birth_dates` data frame called `dob_long_to_date`.

- we used the `as.Date()` function to coerce the character string values in `dob_long` to dates.
- we did so by passing the value `"%B %d, %Y"` to the `format` argument of the `as.Date()` function. These symbols tell R to read the character strings in `dob_long` as full month name (`%B`), 2-digit day (`%d`), a comma (`,`), and 4-digit year (`%Y`).
- we used the `select()` function to keep only the columns we are interested in comparing side-by-side in our output.
- Notice that `dob_long`'s column type is still character (`<chr>`), but `dob_long_to_date`'s column type is `<date>`.

28.5 Change the appearance of dates with format()

So, far we've talked about transforming character strings into dates. However, the reverse is also possible. Meaning, we can transform date values into character strings that we can style (i.e., format) in just about any way you could possibly want to style a date. For example:

```
birth_dates %>%
  mutate(dob_abbreviated = format(dob_actual, "%d %b %y")) %>%
  select(dob_actual, dob_abbreviated)
```

```
# A tibble: 10 x 2
   dob_actual          dob_abbreviated
   <dttm>              <chr>
 1 1996-03-04 16:59:18 04 Mar 96
 2 1998-11-21 21:52:08 21 Nov 98
 3 1994-09-03 23:26:19 03 Sep 94
 4 1996-08-03 17:18:50 03 Aug 96
 5 1980-06-13 18:27:13 13 Jun 80
 6 1996-12-09 05:33:24 09 Dec 96
 7 1992-11-27 17:36:43 27 Nov 92
 8 1983-04-27 23:31:56 27 Apr 83
 9 1988-06-28 16:13:46 28 Jun 88
10 1997-08-02 22:09:50 02 Aug 97
```

Here's what we did above:

- we created a new column in the `birth_dates` data frame called `dob_abbreviated`.
- we used the `format()` function to coerce the date values in `dob_actual` to character string values in `dob_abbreviated`.

- we did so by passing the value `"%d %b %y"` to the `...` argument of the `format()` function. These symbols tell R to create a character string as 2-digit day (`%d`), a space (`" "`), abbreviated month name (`%b`), a space (`" "`), and 2-digit year (`%y`).
- we used the `select()` function to keep only the columns we are interested in comparing side-by-side in our output.
- Notice that `dob_actual`'s column type is still date_time (`<S3: POSIXct>`), but `dob_abbreviated`'s column type is character (`<chr>`). So, while `dob_abbreviated` *looks* like a date to us, **it is no longer a date value to R.** In other words, `dob_abbreviated` doesn't have an integer representation under the hood. It is simply a character string.

28.6 Some useful built-in dates

Base R actually includes a few useful built-in dates that we can use. They can often be useful when doing calculations with dates. Here are a few examples:

28.6.1 Today's date

```
Sys.Date()
```

```
[1] "2025-06-12"
```

```
lubridate::today()
```

```
[1] "2025-06-12"
```

These functions can be useful for calculating any length of time up to today. For example, your age today is just the length of time that spans between your birth date and today.

28.6.2 Today's date-time

```
Sys.time()
```

```
[1] "2025-06-12 20:14:25 CDT"
```

```
lubridate::now()
```

```
[1] "2025-06-12 20:14:25 CDT"
```

Because these functions also return the current time, they can be useful for timing how long it takes your R code to run. As we've said many times, there is typically multiple ways to accomplish a given task in R. Sometimes, the difference between any to ways to accomplish the task is basically just a matter of preference. However, sometimes one way can be much faster than another way. All the examples we've seen so far in this book take a trivial amount of time to run – usually less than a second. However, we have written R programs that took several minutes to several hours to complete. For example, complex data simulations and multiple imputation procedures can both take a long time to run. In such cases, we will sometimes check to see if there any significant performance differences between two different approaches to accomplishing the coding task.

As a silly example to show you how this works, let's generate 1,000,000 random numbers.

```
set.seed(703)
rand_mill <- rnorm(1000000)
```

Now, let's find the mean value of those numbers two different ways, and check to see if there is any time difference between the two:

```
# Save the start time
start  <- lubridate::now()
sum    <- sum(rand_mill)
length <- length(rand_mill)
mean   <- sum / length
mean
```

```
[1] 0.0009259691
```

```
# Save the stop time
stop   <- lubridate::now()
```

```
stop - start
```

```
Time difference of 0.002154112 secs
```

```
rm(mean)
```

So, finding the mean this way took less than a second. Let's see how long using the `mean()` function takes:

```
# Save the start time
start  <- lubridate::now()
mean(rand_mill)
```

```
[1] 0.0009259691
```

```
# Save the stop time
stop   <- lubridate::now()
```

```
stop - start
```

```
Time difference of 0.001874924 secs
```

Although both methods above took less than a second to complete the calculations we were interested in, the second method (i.e., using the `mean()` function) took only about a third as as much time as the first. Again, it obviously doesn't matter in this scenario, but doing these kinds of checks can be useful when calculations take much longer. For example, that time savings we saw above would be pretty important if we were comparing two methods to accomplish a task where the longer method took an hour to complete and the shorter method took a third as much time (About 20 minutes).

28.6.3 Character vector of full month names

```
month.name
```

```
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

28.6.4 Character vector of abbreviated month names

```
month.abb
```

```
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

`month.name` and `month.abb` aren't functions. They don't *do* anything. Rather, they are just saved values that can save us some typing if you happen to be working with data that requires you create variables, or perform calculations, by month.

28.6.5 Creating a vector containing a sequence of dates

In the same way that we can simulate a sequence of numbers using the `seq()` function, we can simulate a sequence of dates using the `seq.Date()` function. We sometimes find this function useful for simulating data (including some of the data used in this book), and for filling in missing dates in longitudinal data. For example, we can use the `seq.Date()` function to return a vector of dates that includes all days between January 1st, 2020 and January 15th, 2020 like this:

```
seq.Date(
  from = as.Date("2020-01-01"),
  to   = as.Date("2020-01-15"),
  by   = "days"
)
```

```
 [1] "2020-01-01" "2020-01-02" "2020-01-03" "2020-01-04" "2020-01-05"
 [6] "2020-01-06" "2020-01-07" "2020-01-08" "2020-01-09" "2020-01-10"
[11] "2020-01-11" "2020-01-12" "2020-01-13" "2020-01-14" "2020-01-15"
```

28.7 Calculating date intervals

So far, we've learned how to create and format dates in R. However, the real value in being able to coerce character strings to date values is that doing so allows us to perform *calculations* with the dates that we could not perform with the character strings. In our experience, calculating intervals of time between dates is probably the most common type of calculation we will want to perform.

Before we get into some examples, we are going to drop some of the columns from our `birth_dates` data frame because we won't need them anymore.

```
ages <- birth_dates %>%
  select(name_first, dob = dob_default) %>%
  print()
```

```
# A tibble: 10 x 2
   name_first dob
   <chr>      <date>
 1 Nathaniel  1996-03-04
 2 Sophia     1998-11-21
 3 Emmett     1994-09-03
 4 Levi       1996-08-03
 5 August     1980-06-13
 6 Juan       1996-12-08
 7 Lilly      1992-11-27
 8 Natalie    1983-04-27
 9 Solomon    1988-06-28
10 Olivia     1997-08-02
```

Here's what we did above:

- we created a new data frame called `ages` by subsetting the `birth_dates` data frame.
- we used the `select()` function to keep only the `name_first` and `dob_default` columns from `birth_dates`. We used a name-value pair (`dob = dob_default`) inside the `select()` function to rename `dob_default` to `dob`.

Next, let's create a variable in our data frame that is equal to today's date. In reality, this would be a great time to use `Sys.Date()` to ask R to return today's date.

```
ages %>%
  mutate(today = Sys.Date())
```

However, we are not going to do that here, because it would cause the value of the `today` variable to update every time we update the book. That would make it challenging to write about the results we get. So, we're going to pretend that today is May 7th, 2020. We'll add that to our data frame like so:

```
ages <- ages %>%
  mutate(today = as.Date("2020-05-07")) %>%
  print()
```

```
# A tibble: 10 x 3
   name_first dob        today
   <chr>      <date>     <date>
 1 Nathaniel  1996-03-04 2020-05-07
 2 Sophia     1998-11-21 2020-05-07
 3 Emmett     1994-09-03 2020-05-07
 4 Levi       1996-08-03 2020-05-07
 5 August     1980-06-13 2020-05-07
 6 Juan       1996-12-08 2020-05-07
 7 Lilly      1992-11-27 2020-05-07
 8 Natalie    1983-04-27 2020-05-07
 9 Solomon    1988-06-28 2020-05-07
10 Olivia     1997-08-02 2020-05-07
```

Here's what we did above:

- we created a new column in the `ages` data frame called `today`.
- we made set the value of the `today` column to May 7th, 2020 by passing the value `"2020-05-07"` to the `as.Date()` function.

28.7.1 Calculate age as the difference in time between dob and today

Calculating age from date of birth is a pretty common data management task. While you know what ages are, you probably don't think much about their calculation. Age is just the difference between two points in time. The starting point is always the date of birth. However, because age is constantly changing the end point changes as well. For example, you're one day older today than you were yesterday. So, to calculate age, we must always have a start date (i.e., date of birth) and an end date. In the example below, our end date will be May 7th, 2020.

Once we have those two pieces of information, we can ask R to calculate age for us in a few different ways. We are going to suggest that you use the method below that uses functions from the `lubridate` package. We will show you why soon. However, we want to show you the base R way of calculating time intervals for comparison, and because a lot of the help documentation we've seen online uses the base R methods shown below.

Let's go ahead and load the `lubridate` package now.

```
library(lubridate)
```

Next, let's go ahead and calculate age 3 different ways:

```r
ages %>%
  mutate(
    age_subtraction = today - dob,
    age_difftime    = difftime(today, dob),
    age_lubridate   = dob %--% today # lubridate's %--% operator creates a time interval
  )
```

```
# A tibble: 10 x 6
   name_first dob        today      age_subtraction age_difftime
   <chr>      <date>     <date>     <drtn>          <drtn>
 1 Nathaniel  1996-03-04 2020-05-07  8830 days       8830 days
 2 Sophia     1998-11-21 2020-05-07  7838 days       7838 days
 3 Emmett     1994-09-03 2020-05-07  9378 days       9378 days
 4 Levi       1996-08-03 2020-05-07  8678 days       8678 days
 5 August     1980-06-13 2020-05-07 14573 days      14573 days
 6 Juan       1996-12-08 2020-05-07  8551 days       8551 days
 7 Lilly      1992-11-27 2020-05-07 10023 days      10023 days
 8 Natalie    1983-04-27 2020-05-07 13525 days      13525 days
 9 Solomon    1988-06-28 2020-05-07 11636 days      11636 days
10 Olivia     1997-08-02 2020-05-07  8314 days       8314 days
# i 1 more variable: age_lubridate <Interval>
```

Here's what we did above:

- we created three new columns in the `ages` data frame called `age_subtraction`, `age_difftime`, and `age_lubridate`.
  - we created `age_subtraction` using the subtraction operator (`-`). Remember, R stores dates values as numbers under the hood. So, we literally just asked R to subtract the value for `dob` from the value for `today`. The value returned to us was a vector of time differences measured in days.
  - we created `age_difftime` base R's `difftime()` function. The value returned to us was a vector of time differences measured in days. As you can see, the results returned by `today - dob` and `difftime(today, dob)` are identical.
  - we created `age_lubridate` using `lubridate`'s time interval operator (`%--%`). Notice that the order of `dob` and `today` are switched here compared to the previous two methods. By itself, the `%--%` operator doesn't return a time difference value. It returns a time interval value.

Here is how we can convert the time difference and time interval values to age in years:

```r
ages %>%
  mutate(
    age_subtraction = as.numeric(today - dob) / 365.25,
    age_difftime    = as.numeric(difftime(today, dob)) / 365.25,
    age_lubridate   = (dob %--% today) / years(1)
  )
```

```
# A tibble: 10 x 6
   name_first dob        today      age_subtraction age_difftime age_lubridate
   <chr>      <date>     <date>               <dbl>        <dbl>          <dbl>
 1 Nathaniel  1996-03-04 2020-05-07            24.2         24.2           24.2
 2 Sophia     1998-11-21 2020-05-07            21.5         21.5           21.5
 3 Emmett     1994-09-03 2020-05-07            25.7         25.7           25.7
 4 Levi       1996-08-03 2020-05-07            23.8         23.8           23.8
 5 August     1980-06-13 2020-05-07            39.9         39.9           39.9
 6 Juan       1996-12-08 2020-05-07            23.4         23.4           23.4
 7 Lilly      1992-11-27 2020-05-07            27.4         27.4           27.4
 8 Natalie    1983-04-27 2020-05-07            37.0         37.0           37.0
 9 Solomon    1988-06-28 2020-05-07            31.9         31.9           31.9
10 Olivia     1997-08-02 2020-05-07            22.8         22.8           22.8
```

Here's what we did above:

- we created three new columns in the `ages` data frame called `age_subtraction`, `age_difftime`, and `age_lubridate`.
  - we used the `as.numeric()` function to convert the values of `age_subtraction` from a time differences to a number – the number of days. We then divided the number of days by 365.25 – roughly the number of days in a year. The result is age in years.
  - we used the `as.numeric()` function to convert the values of `age_difftime` from a time differences to a number – the number of days. We then divided the number of days by 365.25 – roughly the number of days in a year. The result is age in years.
  - Again, the results of the first two methods are identical.
  - we asked R to show us the time interval values we created `age_lubridate` using `lubridate`'s time interval operator (`%--%`) as years of time. We did so by dividing the time interval into years. Specifically, we used the division operator (`/`) and `lubridate`'s `years()` function. The value we passed to the `years()` function was `1`. In other words, we asked R to tell us how many 1-year periods are in each time interval we created with `dob %--% today`.
  - In case you're wondering, here's the value returned by the `years()` function alone:

```
years(1)
```

```
[1] "1y 0m 0d 0H 0M 0S"
```

So, why did the results of the first two methods differ from the results of the third method? Well, dates are much more complicated to work with than they may seem on the surface. Specifically, each day doesn't have exactly 24 hours and each year doesn't have exactly 365 days. Some have more and some have less – so called, leap years. You can find more details on the lubridate website, but the short answer is that `lubridate`'s method gives us a more precise answer than the first two methods do because it accounts for date complexities in a different way.

Here's an example to quickly illustrate what we mean:

Say we want to calculate the number of years between "2017-03-01" and "2018-03-01".

```
start <- as.Date("2017-03-01")
end   <- as.Date("2018-03-01")
```

The most meaningful result in this situation is obviously 1 year.

```
# The base R way
as.numeric(difftime(end, start)) / 365.25
```

```
[1] 0.9993155
```

```
# The lubridate way
(start %--% end) / years(1)
```

```
[1] 1
```

Notice that `lubridate`'s method returns exactly one year, but the base R method returns an approximation of a year.

To further illustrate this point, let's look at what happens when the time interval includes a leap year. The year 2020 is a leap year, so let's calculate the number of years between "2019-03-01" and "2020-03-01". Again, a meaningful result here should be a year.

```
start <- as.Date("2019-03-01")
end   <- as.Date("2020-03-01")
```

```
# The base R way
as.numeric(difftime(end, start)) / 36
```

```
[1] 10.16667
```

```
# The lubridate way
(start %--% end) / years(1)
```

```
[1] 1
```

Notice that `lubridate`'s method returns exactly one year, but the base R method returns an approximation of a year.

To further illustrate this point, let's look at what happens when the time interval includes a leap year. The year 2020 is a leap year, so let's calculate the number of years between "2019-03-01" and "2020-03-01". Again, a meaningful result here should be a year.

```
start <- as.Date("2019-03-01")
end   <- as.Date("2020-03-01")
```

```
# The base R way
as.numeric(difftime(end, start)) / 365.25
```

```
[1] 1.002053
```

```
# The lubridate way
(start %--% end) / years(1)
```

```
[1] 1
```

Once again, the `lubridate` method returns exactly one year, while the base R method returns an approximation of a year.

28.7.2 Rounding time intervals

Okay, so now we know how to get age in years, and hopefully I convinced you that using functions from the `lubridate` package can help us do so in the most precise way possible. However, in most situations we would want to take our calculations one step further and round to whole years. There are actually a couple different ways to do so. For example:

```r
ages %>%
  mutate(
    age_years = (dob %--% today) / years(1),
    # If you want the age (in years) as of the person's last birthday
    age_last  = trunc(age_years),
    # If you want to round the age to the nearest year
    age_near  = round(age_years)
  )
```

```
# A tibble: 10 x 6
   name_first dob        today      age_years age_last age_near
   <chr>      <date>     <date>         <dbl>    <dbl>    <dbl>
 1 Nathaniel  1996-03-04 2020-05-07      24.2       24       24
 2 Sophia     1998-11-21 2020-05-07      21.5       21       21
 3 Emmett     1994-09-03 2020-05-07      25.7       25       26
 4 Levi       1996-08-03 2020-05-07      23.8       23       24
 5 August     1980-06-13 2020-05-07      39.9       39       40
 6 Juan       1996-12-08 2020-05-07      23.4       23       23
 7 Lilly      1992-11-27 2020-05-07      27.4       27       27
 8 Natalie    1983-04-27 2020-05-07      37.0       37       37
 9 Solomon    1988-06-28 2020-05-07      31.9       31       32
10 Olivia     1997-08-02 2020-05-07      22.8       22       23
```

Here's what we did above:

- We created two new columns in the `ages` data frame called `age_last`, and `age_near`.
  - We created `age_last` using the `trunc()` (for truncate) function. The value returned by the `trunc()` function can be interpreted as each person's age in years at their last birthday.
  - We created `age_near` using the `round()` function. The value returned by the `round()` function can be interpreted as each person's age in years at their nearest birthday – which may not have occurred yet. This is probably not the value that you will typically be looking for. So, just make sure you choose the correct function for the type of rounding you want to do.

As a shortcut, we can use the integer division operator (`%/%`) to calculate each person's age in years at their nearest birthday without the `trunc()` function.

```
ages %>%
  mutate(
    # If you want the age (in years) as of the person's last birthday
    age_years = (dob %--% today) %/% years(1)
  )
```

```
# A tibble: 10 x 4
   name_first dob        today      age_years
   <chr>      <date>     <date>         <dbl>
 1 Nathaniel  1996-03-04 2020-05-07        24
 2 Sophia     1998-11-21 2020-05-07        21
 3 Emmett     1994-09-03 2020-05-07        25
 4 Levi       1996-08-03 2020-05-07        23
 5 August     1980-06-13 2020-05-07        39
 6 Juan       1996-12-08 2020-05-07        23
 7 Lilly      1992-11-27 2020-05-07        27
 8 Natalie    1983-04-27 2020-05-07        37
 9 Solomon    1988-06-28 2020-05-07        31
10 Olivia     1997-08-02 2020-05-07        22
```

28.8 Extracting out date parts

Sometimes it can be useful to store parts of a date in separate columns. For example, it is common to break date values up into their component parts when linking records across multiple data frames. We will learn how to link data frames a little later in the book. For now, we're just going to learn how separate dates into their component parts.

We won't need the `today` column anymore, so I'll go ahead a drop it here.

```
ages <- ages %>%
  select(-today) %>%
  print()
```

```
# A tibble: 10 x 2
   name_first dob
   <chr>      <date>
 1 Nathaniel  1996-03-04
 2 Sophia     1998-11-21
 3 Emmett     1994-09-03
 4 Levi       1996-08-03
```

```
 5 August     1980-06-13
 6 Juan       1996-12-08
 7 Lilly      1992-11-27
 8 Natalie    1983-04-27
 9 Solomon    1988-06-28
10 Olivia     1997-08-02
```

Typically, separating the date will include creating separate columns for the day, the month, and the year. Fortunately, lubridate includes intuitively named functions that make this really easy:

```
ages %>%
  mutate(
    day   = day(dob),
    month = month(dob),
    year  = year(dob)
  )
```

```
# A tibble: 10 x 5
   name_first dob          day month  year
   <chr>      <date>     <int> <dbl> <dbl>
 1 Nathaniel  1996-03-04     4     3  1996
 2 Sophia     1998-11-21    21    11  1998
 3 Emmett     1994-09-03     3     9  1994
 4 Levi       1996-08-03     3     8  1996
 5 August     1980-06-13    13     6  1980
 6 Juan       1996-12-08     8    12  1996
 7 Lilly      1992-11-27    27    11  1992
 8 Natalie    1983-04-27    27     4  1983
 9 Solomon    1988-06-28    28     6  1988
10 Olivia     1997-08-02     2     8  1997
```

Here's what we did above:

- We created three new columns in the `ages` data frame called `day`, `month`, and `year`. We created them by passing the `dob` column to the `x` argument of `lubridate`'s `day()`, `month()`, and `year()` functions respectively.

`lubridate` also includes functions for extracting other information from date values. For example:

```r
ages %>%
  mutate(
    wday         = wday(dob),
    day_full     = wday(dob, label = TRUE, abbr = FALSE),
    day_abb      = wday(dob, label = TRUE, abbr = TRUE),
    week_of_year = week(dob),
    week_cdc     = epiweek(dob)
  )
```

```
# A tibble: 10 x 7
   name_first dob         wday day_full  day_abb week_of_year week_cdc
   <chr>      <date>     <dbl> <ord>     <ord>          <dbl>    <dbl>
 1 Nathaniel  1996-03-04     2 Monday    Mon               10       10
 2 Sophia     1998-11-21     7 Saturday  Sat               47       46
 3 Emmett     1994-09-03     7 Saturday  Sat               36       35
 4 Levi       1996-08-03     7 Saturday  Sat               31       31
 5 August     1980-06-13     6 Friday    Fri               24       24
 6 Juan       1996-12-08     1 Sunday    Sun               49       50
 7 Lilly      1992-11-27     6 Friday    Fri               48       48
 8 Natalie    1983-04-27     4 Wednesday Wed               17       17
 9 Solomon    1988-06-28     3 Tuesday   Tue               26       26
10 Olivia     1997-08-02     7 Saturday  Sat               31       31
```

Here's what we did above:

- We created five new columns in the `ages` data frame called `wday`, `day_abb`,`day_full`, `week_of_year`, and `week_cdc`. We created them by passing the `dob` column to the `x` argument of `lubridate`'s `wday()`, `week()`, and `epiweek()` functions respectively.
- The `wday()` function returns the day of the week the given date falls on. By default, the `wday()` returns an integer value between 1 and 7. We can adjust the values passed to `wday()`'s `label` and `abbr` arguments to return full day names (`day_full`) and abbreviated day names (`day_abb`).
- The `week()` function returns the week of the year the given date falls in. More formally, the `week()` function "returns the number of complete seven-day periods that have occurred between the date and January 1st, plus one." You can see this information by typing `?week` in your console.
- The `epiweek()` function also returns the week of the year the given date falls in. However, it calculates the week in a slightly different way. Specifically, "it uses the US CDC version of epidemiological week. It follows same rules as isoweek() but starts on Sunday. In other parts of the world the convention is to start epidemiological weeks on Monday, which

is the same as isoweek." Again, you can see this information by typing `?week` in your console.

28.9 Sorting dates

Another really common thing we might want to do with date values is sort them chronologically. Fortunately, this is really easy to do with `dplyr`'s `arrange()` function. If we want to sort our dates in ascending order (i.e., oldest to most recent), we just pass the date column to the `...` argument of the `arrange()` function like so:

```
# Oldest (top) to most recent (bottom)
# Ascending order
ages %>%
  arrange(dob)
```

```
# A tibble: 10 x 2
   name_first dob
   <chr>      <date>
 1 August     1980-06-13
 2 Natalie    1983-04-27
 3 Solomon    1988-06-28
 4 Lilly      1992-11-27
 5 Emmett     1994-09-03
 6 Nathaniel  1996-03-04
 7 Levi       1996-08-03
 8 Juan       1996-12-08
 9 Olivia     1997-08-02
10 Sophia     1998-11-21
```

If we want to sort our dates in descending order (i.e., most recent to oldest), we just pass the date column to the `desc()` function before passing it to the `...` argument of the `arrange()` function.

```
# Most recent (top) to oldest (bottom)
# Descending order
ages %>%
  arrange(desc(dob))
```

```
# A tibble: 10 x 2
   name_first dob
```

```
   <chr>      <date>
 1 Sophia     1998-11-21
 2 Olivia     1997-08-02
 3 Juan       1996-12-08
 4 Levi       1996-08-03
 5 Nathaniel  1996-03-04
 6 Emmett     1994-09-03
 7 Lilly      1992-11-27
 8 Solomon    1988-06-28
 9 Natalie    1983-04-27
10 August     1980-06-13
```

Much of the data we work with in epidemiology includes dates. In fact, it isn't uncommon for the length of time that passes between to events to be the primary outcome that we are trying to understand. Hopefully, the tools we've learned in this chapter will give you a solid foundation for working with dates in R. For more information on dates, including a handy cheat sheet, I recommend visiting the lubridate website.

29 Working with Character Strings

In previous chapters, we learned how to create character vectors, which can be useful on their own. We also learned how to coerce character vectors to factor vectors that we can use for categorical data analysis. However, up to this point, we haven't done a lot of manipulation of the values stored inside of the character strings themselves. Sometimes, however, we will need to manipulate the character string before we can complete other data management tasks or analysis. Some common examples from my projects include separating character strings into multiple parts and creating dummy variables from character strings that can take multiple values. In this chapter, we'll see some specific example of both, and we'll learn a few new tools for working with character strings along the way.

To get started, feel free to download the simulated electronic health record that we will use in the following examples. Additionally, we will use the `readr`, `dplyr`, and `stringr` packages in the code below. You will be able to recognize functions from the `stringr` package because they will all begin with `str_`.

```
library(readr)
library(dplyr)
library(stringr) # All stringr functions begin with "str_"
```

```
ehr <- read_rds("ehr.Rds")
```

```
ehr
```

```
# A tibble: 15 x 6
   admit_date          name               dob        address      city  symptoms
   <dttm>              <chr>              <date>     <chr>        <chr> <chr>
 1 2017-02-01 05:22:30 "Zariah Hernandez" 1944-09-27 3201 ORANGE~ FORT~ "\"Pain~
 2 2017-04-08 09:17:17 "Tatum Chavez"     1952-06-12 1117 richmo~ Fort~ "Pain"
 3 2017-04-18 09:17:17 "Tatum S Chavez"   1952-06-12 1117 richmo~ Fort~ "Pain"
 4 2017-08-31 18:29:34 "Arabella George"  1966-06-15 357 Angle    FORT~ "\"Naus~
 5 2017-09-13 06:27:07 "Jasper Decker"    1954-05-11 3612 LAURA ~ FORT~ "\"Pain~
 6 2017-09-15 18:29:34 "ARABELLA GEORGE"  1966-06-15 357 Angle    FORT~ "\"Naus~
 7 2017-10-07 06:31:18 "Weston Fox"       2009-08-21 6433 HATCHE~ City~ "Pain"
 8 2017-10-08 23:17:18 "Ryan Edwards"     1917-12-10 3201 HORIZO~ City~  <NA>
```

```
 9 2017-10-16 06:31:18 "Weston Fox,"      2009-08-21 6433 HATCHE~ City~ "Pain"
10 2017-10-26 23:17:18 "Ryan Edwards  "   1917-12-10 3201 HORIZO~ City~  <NA>
11 2017-10-27 18:37:00 "Emma Medrano"     1975-05-01 6301 BEECHC~ KELL~ "\"Naus~
12 2017-12-18 20:47:48 "Ivy Mccann"       1911-06-21 5426 CHILDR~ FORT~ "\"Head~
13 2017-12-20 13:40:04 "Charlee Carroll"  1908-07-22 8190 DUCK C~ City~ "Headac~
14 2017-12-26 20:47:48 "Ivy   Mccann"     1911-06-21 5426 CHILDR~ FORT~ "\"Head~
15 2018-01-28 08:49:38 "Kane Martin"      1939-10-27 4929 asbury  FORT~  <NA>
```

Here's what we did above:

- we used the `read_csv()` function to import a .Rds file containing simulated data into R.
- The simulated data contains admission date (`admit_date`), the patient's name (`name`), the patient's date of birth (`dob`), the patient's address (`address`), the city the patient lives in (`city`), and column that contains the symptoms each patient was experiencing at admission (`symptoms`).
- In this data, date of birth is recorded in the four most common formats that I typically come across.

A common initial question we may need to ask of this kind of data is, *"how many unique people are represented in this data?"* Well, there are 15 rows, so a good first guess might be 15 unique people. However, let's arrange the data by the `name` column and see if that guess still looks reasonable.

```
ehr %>%
  group_by(name) %>%
  mutate(dup = row_number() > 1) %>%
  arrange(name) %>%
  select(name, dup, dob, address, city)
```

```
# A tibble: 15 x 5
# Groups:   name [15]
   name               dup   dob        address             city
   <chr>              <lgl> <date>     <chr>               <chr>
 1 "ARABELLA GEORGE"  FALSE 1966-06-15 357 Angle           FORT WORTH
 2 "Arabella George"  FALSE 1966-06-15 357 Angle           FORT WORTH
 3 "Charlee Carroll"  FALSE 1908-07-22 8190 DUCK CREEK CT  City of Fort Worth
 4 "Emma Medrano"     FALSE 1975-05-01 6301 BEECHCREEK DR  KELLER
 5 "Ivy   Mccann"     FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 6 "Ivy Mccann"       FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 7 "Jasper Decker"    FALSE 1954-05-11 3612 LAURA ANNE CT. FORT WORTH
 8 "Kane Martin"      FALSE 1939-10-27 4929 asbury         FORT WORTH
```

```
 9 "Ryan Edwards"     FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
10 "Ryan Edwards  "   FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
11 "Tatum Chavez"     FALSE 1952-06-12 1117 richmond ave   Fort Worth
12 "Tatum S Chavez"   FALSE 1952-06-12 1117 richmond ave   Fort Worth
13 "Weston Fox"       FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
14 "Weston Fox,"      FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
15 "Zariah Hernandez" FALSE 1944-09-27 3201 ORANGE AVE     FORT WORTH
```

Clearly, some of these people are the same. However, little data entry discrepancies in their name values would prevent us from calculating the number of unique people in a programmatic way. Let's take a closer look at the values in the `name` column and see if we can figure out exactly what these data entry discrepancies are:

```
ehr %>%
  arrange(name) %>%
  pull(name)
```

```
 [1] "ARABELLA GEORGE"  "Arabella George"  "Charlee Carroll"  "Emma Medrano"
 [5] "Ivy   Mccann"     "Ivy Mccann"       "Jasper Decker"    "Kane Martin"
 [9] "Ryan Edwards"     "Ryan Edwards  "   "Tatum Chavez"     "Tatum S Chavez"
[13] "Weston Fox"       "Weston Fox,"      "Zariah Hernandez"
```

Here's what we did above:

- we `dplyr`'s `pull()` function to return the `name` column as a character vector. Doing so makes it easier to see some of the discrepancies in the way the patient's names were entered into the ehr.
- Notice that Arabella George's name is written in title case one time and written in all caps another time. Remember that R is case sensitive. So, these two values – "Arabella George" and "ARABELLA GEORGE" – are different values to R.
- Notice that in one instance of Ivy Mccann's name someone accidently typed two spaces between her first and last name. These two values – "Ivy Mccann" and "Ivy Mccann" – are different values to R.
- Notice that in one instance of Ryan Edwards' name someone accidently typed an extra space after his last name. These two values – "Ryan Edwards" and "Ryan Edwards" – are different values to R.
- Notice that in one instance of Tatum Chavez's name was entered into the ehr *with* his middle initial on one instance. These two values – "Tatum Chavez" and "Tatum S Chavez" – are different values to R.

- Notice that Weston Fox's name was entered into the ehr with a comma immediately following his last name on one instance. These two values – "Weston Fox" and "Weston Fox," – are different values to R.

29.1 Coerce to lowercase

A good place to start cleaning these character strings is by coercing them all to lowercase. We've already used base R's `tolower()` function a couple of times before. So, you may have already guessed how to complete this task. However, before moving on to coercing all the names in our ehr data to lowercase, we want to show you some of the other functions that the `stringr` package contains for changing the case of character strings. For example:

29.1.1 Lowercase

```
ehr %>%
  arrange(name) %>%
  pull(name) %>%
  str_to_lower()
```

```
 [1] "arabella george"  "arabella george"  "charlee carroll"  "emma medrano"
 [5] "ivy   mccann"     "ivy mccann"       "jasper decker"    "kane martin"
 [9] "ryan edwards"     "ryan edwards  "   "tatum chavez"     "tatum s chavez"
[13] "weston fox"       "weston fox,"      "zariah hernandez"
```

29.1.2 Upper case

```
ehr %>%
  arrange(name) %>%
  pull(name) %>%
  str_to_upper()
```

```
 [1] "ARABELLA GEORGE"  "ARABELLA GEORGE"  "CHARLEE CARROLL"  "EMMA MEDRANO"
 [5] "IVY   MCCANN"     "IVY MCCANN"       "JASPER DECKER"    "KANE MARTIN"
 [9] "RYAN EDWARDS"     "RYAN EDWARDS  "   "TATUM CHAVEZ"     "TATUM S CHAVEZ"
[13] "WESTON FOX"       "WESTON FOX,"      "ZARIAH HERNANDEZ"
```

29.1.3 Title case

```
ehr %>%
  arrange(name) %>%
  pull(name) %>%
  str_to_title()
```

```
 [1] "Arabella George"  "Arabella George"  "Charlee Carroll"  "Emma Medrano"
 [5] "Ivy   Mccann"     "Ivy Mccann"       "Jasper Decker"    "Kane Martin"
 [9] "Ryan Edwards"     "Ryan Edwards  "   "Tatum Chavez"     "Tatum S Chavez"
[13] "Weston Fox"       "Weston Fox,"      "Zariah Hernandez"
```

29.1.4 Sentence case

```
ehr %>%
  arrange(name) %>%
  pull(name) %>%
  str_to_sentence()
```

```
 [1] "Arabella george"  "Arabella george"  "Charlee carroll"  "Emma medrano"
 [5] "Ivy   mccann"     "Ivy mccann"       "Jasper decker"    "Kane martin"
 [9] "Ryan edwards"     "Ryan edwards  "   "Tatum chavez"     "Tatum s chavez"
[13] "Weston fox"       "Weston fox,"      "Zariah hernandez"
```

Each of the function above can come in handy from time-to-time. So, you may just want to keep them in your back pocket. Let's go ahead and use the `str_to_lower()` function now as the first step in cleaning our data:

```
ehr <- ehr %>%
  mutate(name = str_to_lower(name)) %>%
  print()
```

```
# A tibble: 15 x 6
   admit_date          name               dob        address      city  symptoms
   <dttm>              <chr>              <date>     <chr>        <chr> <chr>
 1 2017-02-01 05:22:30 "zariah hernandez" 1944-09-27 3201 ORANGE~ FORT~ "\"Pain~
 2 2017-04-08 09:17:17 "tatum chavez"     1952-06-12 1117 richmo~ Fort~ "Pain"
 3 2017-04-18 09:17:17 "tatum s chavez"   1952-06-12 1117 richmo~ Fort~ "Pain"
```

```
 4 2017-08-31 18:29:34 "arabella george"  1966-06-15 357 Angle    FORT~ "\"Naus~
 5 2017-09-13 06:27:07 "jasper decker"    1954-05-11 3612 LAURA ~ FORT~ "\"Pain~
 6 2017-09-15 18:29:34 "arabella george"  1966-06-15 357 Angle    FORT~ "\"Naus~
 7 2017-10-07 06:31:18 "weston fox"       2009-08-21 6433 HATCHE~ City~ "Pain"
 8 2017-10-08 23:17:18 "ryan edwards"     1917-12-10 3201 HORIZO~ City~  <NA>
 9 2017-10-16 06:31:18 "weston fox,"      2009-08-21 6433 HATCHE~ City~ "Pain"
10 2017-10-26 23:17:18 "ryan edwards  "   1917-12-10 3201 HORIZO~ City~  <NA>
11 2017-10-27 18:37:00 "emma medrano"     1975-05-01 6301 BEECHC~ KELL~ "\"Naus~
12 2017-12-18 20:47:48 "ivy mccann"       1911-06-21 5426 CHILDR~ FORT~ "\"Head~
13 2017-12-20 13:40:04 "charlee carroll"  1908-07-22 8190 DUCK C~ City~ "Headac~
14 2017-12-26 20:47:48 "ivy   mccann"     1911-06-21 5426 CHILDR~ FORT~ "\"Head~
15 2018-01-28 08:49:38 "kane martin"      1939-10-27 4929 asbury  FORT~  <NA>
```

Here's what we did above:

- we used `stringr`'s `str_to_lower()` function to coerce all the letters in the `name` column to lowercase.

Now, let's check and see how many unique people R finds in our data?

```
ehr %>%
  group_by(name) %>%
  mutate(dup = row_number() > 1) %>%
  arrange(name) %>%
  select(name, dup, dob, address, city)
```

```
# A tibble: 15 x 5
# Groups:   name [14]
   name               dup   dob        address             city
   <chr>              <lgl> <date>     <chr>               <chr>
 1 "arabella george"  FALSE 1966-06-15 357 Angle           FORT WORTH
 2 "arabella george"  TRUE  1966-06-15 357 Angle           FORT WORTH
 3 "charlee carroll"  FALSE 1908-07-22 8190 DUCK CREEK CT  City of Fort Worth
 4 "emma medrano"     FALSE 1975-05-01 6301 BEECHCREEK DR  KELLER
 5 "ivy   mccann"     FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 6 "ivy mccann"       FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 7 "jasper decker"    FALSE 1954-05-11 3612 LAURA ANNE CT. FORT WORTH
 8 "kane martin"      FALSE 1939-10-27 4929 asbury         FORT WORTH
 9 "ryan edwards"     FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
10 "ryan edwards  "   FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
11 "tatum chavez"     FALSE 1952-06-12 1117 richmond ave   Fort Worth
12 "tatum s chavez"   FALSE 1952-06-12 1117 richmond ave   Fort Worth
```

```
13 "weston fox"       FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
14 "weston fox,"      FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
15 "zariah hernandez" FALSE 1944-09-27 3201 ORANGE AVE     FORT WORTH
```

In the output above, there are 15 rows. R has identified 1 row with a duplicate name (dup == TRUE), which results in a count of 14 unique people. So, simply coercing all the letters to lower case alone helped R figure out that there was a duplicate name value for arabella george. Next, let's go ahead and remove the trailing space from Ryan Edwards' name.

29.2 Trim white space

we can use `stringr`'s `str_trim()` function to "trim" white space from the beginning and end of character strings. For example:

```
str_trim("Ryan Edwards  ")
```

```
[1] "Ryan Edwards"
```

Let's go ahead and use the `str_trim()` function now as the next step in cleaning our data:

```
ehr <- ehr %>%
  mutate(name = str_trim(name))
```

Now, let's check and see how many unique people R finds in our data?

```
ehr %>%
  group_by(name) %>%
  mutate(dup = row_number() > 1) %>%
  arrange(name) %>%
  select(name, dup, dob, address, city)
```

```
# A tibble: 15 x 5
# Groups:   name [13]
   name             dup   dob        address             city
   <chr>            <lgl> <date>     <chr>               <chr>
 1 arabella george  FALSE 1966-06-15 357 Angle           FORT WORTH
 2 arabella george  TRUE  1966-06-15 357 Angle           FORT WORTH
 3 charlee carroll  FALSE 1908-07-22 8190 DUCK CREEK CT  City of Fort Worth
 4 emma medrano     FALSE 1975-05-01 6301 BEECHCREEK DR  KELLER
```

```
 5 ivy   mccann     FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 6 ivy mccann       FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 7 jasper decker    FALSE 1954-05-11 3612 LAURA ANNE CT. FORT WORTH
 8 kane martin      FALSE 1939-10-27 4929 asbury         FORT WORTH
 9 ryan edwards     FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
10 ryan edwards     TRUE  1917-12-10 3201 HORIZON PL     City of Saginaw
11 tatum chavez     FALSE 1952-06-12 1117 richmond ave   Fort Worth
12 tatum s chavez   FALSE 1952-06-12 1117 richmond ave   Fort Worth
13 weston fox       FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
14 weston fox,      FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
15 zariah hernandez FALSE 1944-09-27 3201 ORANGE AVE     FORT WORTH
```

In the output above, there are 15 rows. R has identified 2 rows with a duplicate name (dup == TRUE), which results in a count of 13 unique people. We're getting closer. However, the rest of the discrepancies in the `name` column that we want to address are a little more complicated. There isn't a pre-made base R or `stringr` function that will fix them. Instead, we'll need to learn how to use something called **regular expressions**.

29.3 Regular expressions

Regular expressions, also called **regex** or **regexps**, can be really intimidating at first. In fact, I debated whether or not to even include a discussion of regular expressions at this point in the book. However, regular expressions are *the* most powerful and flexible tool for manipulating character strings that I are aware of. So, I think it's important for you to get a little exposure to regular expressions, even if you aren't a regular expressions expert by the end of this chapter.

The first time you see regular expressions, you will probably think they look like gibberish. For example, here's a regular expression that I recently used to clean a data set `(\d{1,2}\/\d{1,2}\/\d{2})`. You can think of regular expressions as an entirely different programming language that the R interpreter can also understand. Regular expressions aren't unique to R. Many programming languages can accept regular expressions as a way to manipulate character strings.

In the examples that follow, we hope
1. To give you a feel for how regular expression can be useful.
2. Provide you with some specific regular expressions that you may want to save for your epi work (or your class assignments).
3. Provide you with some resources to help you take your regular expression skills to the next level when you are ready.

29.3.1 Remove the comma

For our first example, let's remove the comma from Weston Fox's last name.

```
str_replace(
  string      = "weston fox,",
  pattern     = ",",
  replacement = ""
)
```

```
[1] "weston fox"
```

Here's what we did above:

- we used `stringr`'s `str_replace()` function remove the comma from the character string "weston fox,".
- The first argument to the `str_replace()` function is `string`. The value passed the `string` argument should be the character string, or vector of character strings, we want to manipulate.
- The second argument to the `str_replace()` function is `pattern`. The value passed the `pattern` argument should be regular expression. It should tell the `str_replace()` function what part of the character string we want to replace. In this case, it is a comma (`","`). We are telling the `str_replace()` function that we want it to replace the first comma it sees in the character string "weston fox," with the value we pass to the `replacement` argument.
- The third argument to the `str_replace()` function is `replacement`. The value passed the `replacement` argument should also be regular expression. It should tell the `str_replace()` function to what replace the value identified in the `pattern` argument with. In this case, it is nothing (`""`) – two double quotes with nothing in-between. We are telling the `str_replace()` function that we want it to replace the first comma it sees in the character string "weston fox," with nothing. This is sort of a long-winded way of saying, "delete the comma."

> ⚠ Warning
>
> Notice that our regular expressions above are wrapped in quotes. Regular expressions should always be wrapped in quotes.

Let's go ahead and use the `str_replace()` function now as the next step in cleaning our data:

```
ehr <- ehr %>%
  mutate(name = str_replace(name, ",", ""))
```

Now, let's check and see how many unique people R finds in our data?

```
ehr %>%
  group_by(name) %>%
  mutate(dup = row_number() > 1) %>%
  arrange(name) %>%
  select(name, dup, dob, address, city)
```

```
# A tibble: 15 x 5
# Groups:   name [12]
   name             dup   dob        address             city
   <chr>            <lgl> <date>     <chr>               <chr>
 1 arabella george  FALSE 1966-06-15 357 Angle           FORT WORTH
 2 arabella george  TRUE  1966-06-15 357 Angle           FORT WORTH
 3 charlee carroll  FALSE 1908-07-22 8190 DUCK CREEK CT  City of Fort Worth
 4 emma medrano     FALSE 1975-05-01 6301 BEECHCREEK DR  KELLER
 5 ivy   mccann     FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 6 ivy mccann       FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 7 jasper decker    FALSE 1954-05-11 3612 LAURA ANNE CT. FORT WORTH
 8 kane martin      FALSE 1939-10-27 4929 asbury         FORT WORTH
 9 ryan edwards     FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
10 ryan edwards     TRUE  1917-12-10 3201 HORIZON PL     City of Saginaw
11 tatum chavez     FALSE 1952-06-12 1117 richmond ave   Fort Worth
12 tatum s chavez   FALSE 1952-06-12 1117 richmond ave   Fort Worth
13 weston fox       FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
14 weston fox       TRUE  2009-08-21 6433 HATCHER ST     City of Fort Worth
15 zariah hernandez FALSE 1944-09-27 3201 ORANGE AVE     FORT WORTH
```

In the output above, there are 15 rows. R has identified 3 rows with a duplicate name (dup == TRUE), which results in a count of 12 unique people.

29.3.2 Remove middle initial

Next, let's remove the middle initial from Tatum Chavez's name.

```
str_replace(
  string      = "tatum s chavez",
  pattern     = " \\w ",
  replacement = " "
)
```

```
[1] "tatum chavez"
```

Here's what we did above:

- we used `stringr`'s `str_replace()` function remove the "s" from the character string "tatum s chavez".
- The first argument to the `str_replace()` function is `string`. The value passed the `string` argument should be the character string, or vector of character strings, we want to manipulate.
- The second argument to the `str_replace()` function is `pattern`. The value passed the `pattern` argument should be regular expression. It should tell the `str_replace()` function what part of the character string we want to replace. In this case, it is `" \\w "`. That is a space, two backslashes, a "w," and a space. This regular expression looks a little stranger than the last one we saw.
  - The `\w` is called a **token** in regular expression lingo. The `\w` token means "Any word character." Any word character includes all the letters of the alphabet upper and lowercase (i.e., `[a-zA-Z]`), all numbers (i.e., `[0-9]`), and the underscore character (_).
  - When passing regular expression to R, we must always add an additional backslash in front of any other backslash in the regular expression. In this case, `\\w` instead of `\w`.
  - If we had stopped here (`"\\w"`), this regular expression would have told the `str_replace()` function that we want it to replace the first word character it sees in the character string "tatum s chavez" with the value we pass to the `replacement` argument. In this case, that would have been the "t" at the beginning of "tatum s chavez".
  - The final component of the regular expression we passed to the `pattern` argument is spaces on both sides of the `\\w` token. The complete regular expression, `" \\w "`, tells the `str_replace()` function that we want it to replace the first time it sees a space, followed by any word character, followed by another space in the character string "tatum s chavez" with the value we pass to the `replacement` argument. The first section of the character string above that matches that pattern is the " s " in "tatum s chavez".

- The third argument to the `str_replace()` function is `replacement`. The value passed the `replacement` argument should also be regular expression. It should tell the `str_replace()` function what to replace the value identified in the `pattern` argument with. In this case, it is a single space (`" "`).

Let's go ahead and use the `str_replace()` function now as the next step in cleaning our data:

```
ehr <- ehr %>%
  mutate(name = str_replace(name, " \\w ", " "))
```

And, let's once again check and see how many unique people R finds in our data?

```
ehr %>%
  group_by(name) %>%
  mutate(dup = row_number() > 1) %>%
  arrange(name) %>%
  select(name, dup, dob, address, city)
```

```
# A tibble: 15 x 5
# Groups:   name [11]
   name             dup   dob        address             city
   <chr>            <lgl> <date>     <chr>               <chr>
 1 arabella george  FALSE 1966-06-15 357 Angle           FORT WORTH
 2 arabella george  TRUE  1966-06-15 357 Angle           FORT WORTH
 3 charlee carroll  FALSE 1908-07-22 8190 DUCK CREEK CT  City of Fort Worth
 4 emma medrano     FALSE 1975-05-01 6301 BEECHCREEK DR  KELLER
 5 ivy   mccann     FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 6 ivy mccann       FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 7 jasper decker    FALSE 1954-05-11 3612 LAURA ANNE CT. FORT WORTH
 8 kane martin      FALSE 1939-10-27 4929 asbury         FORT WORTH
 9 ryan edwards     FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
10 ryan edwards     TRUE  1917-12-10 3201 HORIZON PL     City of Saginaw
11 tatum chavez     FALSE 1952-06-12 1117 richmond ave   Fort Worth
12 tatum chavez     TRUE  1952-06-12 1117 richmond ave   Fort Worth
13 weston fox       FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
14 weston fox       TRUE  2009-08-21 6433 HATCHER ST     City of Fort Worth
15 zariah hernandez FALSE 1944-09-27 3201 ORANGE AVE     FORT WORTH
```

In the output above, there are 15 rows. R has identified 4 rows with a duplicate name (dup == TRUE), which results in a count of 11 unique people.

29.3.3 Remove double spaces

Finally, let's remove the double space from Ivy Mccann's name.

```
str_replace(
  string      = "Ivy   Mccann",
  pattern     = "\\s{2,}",
  replacement = " "
)
```

```
[1] "Ivy Mccann"
```

Here's what we did above:

- we used `stringr`'s `str_replace()` function remove the double space from the character string "Ivy Mccann".
- The first argument to the `str_replace()` function is `string`. The value passed the `string` argument should be the character string, or vector of character strings, we want to manipulate.
- The second argument to the `str_replace()` function is `pattern`. The value passed the `pattern` argument should be regular expression. It should tell the `str_replace()` function what part of the character string we want to replace. In this case, it is `\\s{2,}`. This regular expression looks even more strange than the last one we saw.
  - The `\s` is another token. The `\s` token means "Any whitespace character."
  - When passing regular expression to R, we must always add an additional backslash in front of any other backslash in the regular expression. In this case, `\\s` instead of `\s`.
  - The curly braces with numbers inside is called a **quantifier** in regular expression lingo. The first number inside the curly braces tells `str_replace()` to look for *at least* this many occurrences of whatever is immediately before the curly braces in the regular expression. The second number inside the curly braces tells `str_replace()` to look for *no more than* this many occurrences of whatever is immediately before the curly braces in the regular expression. When there is no number in the first position, that means that there is no minimum number of occurrences that count. When there is no number is the second position, that means that there is no upper limit of occurrences that count. In this case, the thing immediately before the curly braces in the regular expression was a whitespace (`\\s`), and the `{2,}` tells `str_replace()` to look for between 2 and unlimited consecutive occurrences of whitespace.

- The third argument to the `str_replace()` function is `replacement`. The value passed the `replacement` argument should also be regular expression. It should tell the `str_replace()` function what to replace the value identified in the `pattern` argument with. In this case, it is a single space (`" "`).

Let's go ahead and use the `str_replace()` function now as the final step in cleaning our `name` column:

```
ehr <- ehr %>%
  mutate(name = str_replace(name, "\\s{2,}", " "))
```

Let's check one final time to see how many unique people R finds in our data.

```
ehr %>%
  group_by(name) %>%
  mutate(dup = row_number() > 1) %>%
  arrange(name) %>%
  select(name, dup, dob, address, city)
```

```
# A tibble: 15 x 5
# Groups:   name [10]
   name             dup   dob        address             city
   <chr>            <lgl> <date>     <chr>               <chr>
 1 arabella george  FALSE 1966-06-15 357 Angle           FORT WORTH
 2 arabella george  TRUE  1966-06-15 357 Angle           FORT WORTH
 3 charlee carroll  FALSE 1908-07-22 8190 DUCK CREEK CT  City of Fort Worth
 4 emma medrano     FALSE 1975-05-01 6301 BEECHCREEK DR  KELLER
 5 ivy mccann       FALSE 1911-06-21 5426 CHILDRESS ST   FORT WORTH
 6 ivy mccann       TRUE  1911-06-21 5426 CHILDRESS ST   FORT WORTH
 7 jasper decker    FALSE 1954-05-11 3612 LAURA ANNE CT. FORT WORTH
 8 kane martin      FALSE 1939-10-27 4929 asbury         FORT WORTH
 9 ryan edwards     FALSE 1917-12-10 3201 HORIZON PL     City of Saginaw
10 ryan edwards     TRUE  1917-12-10 3201 HORIZON PL     City of Saginaw
11 tatum chavez     FALSE 1952-06-12 1117 richmond ave   Fort Worth
12 tatum chavez     TRUE  1952-06-12 1117 richmond ave   Fort Worth
13 weston fox       FALSE 2009-08-21 6433 HATCHER ST     City of Fort Worth
14 weston fox       TRUE  2009-08-21 6433 HATCHER ST     City of Fort Worth
15 zariah hernandez FALSE 1944-09-27 3201 ORANGE AVE     FORT WORTH
```

In the output above, there are 15 rows. R has identified 5 rows with a duplicate name (dup == TRUE), which results in a count of 10 unique people. This is the answer we wanted!

If our data frame was too big to count unique people manually, we could have R calculate the number of unique people for us like this:

```
ehr %>%
  group_by(name) %>%
  filter(row_number() == 1) %>%
  ungroup() %>%
  summarise(`Unique People` = n())
```

```
# A tibble: 1 x 1
  `Unique People`
            <int>
1              10
```

Here's what we did above:

- With the exception of `filter(row_number() == 1)`, you should have seen all of the elements in the code above before.
- we saw the `row_number()` function used before inside of `mutate()` to sequentially count the number of rows that belong to each group created with `group_by()`. We could have done that in the code above. The `filter(row_number() == 1)` code is really just a shorthand way to write `mutate(row = row_number()) %>% filter(row == 1)`. It has the effect of telling R to just keep the first row for each group created by `group_by()`. In this case, just keep the first row for each name in the data frame.

Now that we know how many unique people are in our data, let's say we want to know how many of them live in each city that our data contains.

First, we will subset our data to include one row only for each person:

```
ehr_unique <- ehr %>%
  group_by(name) %>%
  filter(row_number() == 1) %>%
  ungroup() %>%
  print()
```

```
# A tibble: 10 x 6
   admit_date          name             dob        address         city  symptoms
   <dttm>              <chr>            <date>     <chr>           <chr> <chr>
 1 2017-02-01 05:22:30 zariah hernandez 1944-09-27 3201 ORANGE A~ FORT~ "\"Pain~
 2 2017-04-08 09:17:17 tatum chavez     1952-06-12 1117 richmond~ Fort~ "Pain"
```

```
 3 2017-08-31 18:29:34 arabella george  1966-06-15 357 Angle      FORT~ "\"Naus~
 4 2017-09-13 06:27:07 jasper decker    1954-05-11 3612 LAURA AN~ FORT~ "\"Pain~
 5 2017-10-07 06:31:18 weston fox       2009-08-21 6433 HATCHER ~ City~ "Pain"
 6 2017-10-08 23:17:18 ryan edwards     1917-12-10 3201 HORIZON ~ City~  <NA>
 7 2017-10-27 18:37:00 emma medrano     1975-05-01 6301 BEECHCRE~ KELL~ "\"Naus~
 8 2017-12-18 20:47:48 ivy mccann       1911-06-21 5426 CHILDRES~ FORT~ "\"Head~
 9 2017-12-20 13:40:04 charlee carroll  1908-07-22 8190 DUCK CRE~ City~ "Headac~
10 2018-01-28 08:49:38 kane martin      1939-10-27 4929 asbury    FORT~  <NA>
```

Let's go ahead and get an initial count of how many people live in each city:

```
ehr %>%
  group_by(city) %>%
  summarise(n = n())
```

```
# A tibble: 5 x 2
  city                   n
  <chr>              <int>
1 City of Fort Worth     3
2 City of Saginaw        2
3 FORT WORTH             7
4 Fort Worth             2
5 KELLER                 1
```

I'm sure you saw this coming, but we have more data entry discrepancies that are preventing us from completing our analysis. Now that you've gotten your feet wet with character string manipulation and regular expressions, what do we need to do in order to complete our analysis?

Hopefully, your first instinct by now is to coerce all the letters to lowercase. In fact, one of the first things we typically do is coerce all character columns to lowercase. Let's do that now.

```
ehr <- ehr %>%
  mutate(
    address = tolower(address),
    city    = tolower(city)
  )
```

Now how many people live in each city?

```
ehr %>%
  group_by(city) %>%
  summarise(n = n())
```

```
# A tibble: 4 x 2
  city                   n
  <chr>              <int>
1 city of fort worth     3
2 city of saginaw        2
3 fort worth             9
4 keller                 1
```

we're getting closer to the right answer, but we still need to remove "city of" from some of the values. This sounds like another job for `str_replace()`.

```
str_replace(
  string      = "city of fort worth",
  pattern     = "city of ",
  replacement = ""
)
```

```
[1] "fort worth"
```

That regular expression looks like it will work. Let's go ahead and use it to remove "city of" from the values in the `address_city` column now.

```
ehr <- ehr %>%
  mutate(city = str_replace(city, "city of ", ""))
```

One last time, how many people live in each city?

```
ehr %>%
  group_by(city) %>%
  summarise(n = n())
```

```
# A tibble: 3 x 2
  city           n
  <chr>      <int>
1 fort worth    12
2 keller         1
3 saginaw        2
```

29.4 Separate values into component parts

Another common task that I perform on character strings is to separate the strings into multiple parts. For example, sometimes we may want to separate full names into two columns. One for fist name and one for last name. To complete this task, we will once again use regular expressions. We will also learn how to use the `str_extract()` function to pull values out of a character string when the match a pattern we create with a regular expression.

```
str_extract("zariah hernandez", "^\\w+")
```

```
[1] "zariah"
```

Here's what we did above:

- we used `stringr`'s `str_extract()` function pull the first name out of the full name "zariah hernandez".
- The first argument to the `str_extract()` function is `string`. The value passed the `string` argument should be the character string, or vector of character strings, we want to manipulate.
- The second argument to the `str_extract()` function is `pattern`. The value passed the `pattern` argument should be regular expression. It should tell the `str_extract()` function what part of the character string we want to pull out of the character string. In this case, it is `^\\w+`.
  - we've already seen that the `\w` token means "Any word character."
  - When passing regular expression to R, we must always add an additional backslash in front of any other backslash in the regular expression. In this case, `\\w` instead of `\w`.
  - The carrot (`^`) is a type of **anchor** in regular expression lingo. It tells the `str_extract()` function to look for the pattern at the start of the character sting only.
  - The plus sign (`+`) is another quantifier. It means, "match the pattern one or more times."
  - Taken together, `^\\w+` tells the `str_extract()` function to look for one or more consecutive word characters beginning at the start of the character string and extract them.
  - The first word character at the start of the string is "z", then "a", then "riah". Finally, R gets to the space between "zariah" and "hernandez", which isn't a word character, and stops the extraction. The result is "zariah".

we can pull the last name from the character string in a similar way:

```
str_extract("zariah hernandez", "\\w+$")
```

```
[1] "hernandez"
```

Here's what we did above:

- we used `stringr`'s `str_extract()` function pull the last name out of the full name "zariah hernandez".
- The first argument to the `str_extract()` function is `string`. The value passed the `string` argument should be the character string, or vector of character strings, we want to manipulate.
- The second argument to the `str_extract()` function is `pattern`. The value passed the `pattern` argument should be regular expression. It should tell the `str_extract()` function what part of the character string we want to pull out of the character string. In this case, it is `\\w+$`.
  - we've already seen that the `\w` token means "Any word character."
  - When passing regular expression to R, we must always add an additional backslash in front of any other backslash in the regular expression. In this case, `\\w` instead of `\w`.
  - The dollar sign (`$`) is another type of anchor. It tells the `str_extract()` function to look for the pattern at the end of the string only.
  - we've already seen that the plus sign (`+`) is a quantifier that means, "match the pattern one or more times."

  -Taken together, `\\w+$` tells the `str_extract()` function to look for one or more consecutive word characters beginning at the end of the string and extract them.
  - The first word character at the end of the string is "z", then "e", then "dnanreh". Finally, R gets to the space between "zariah" and "hernandez", which isn't a word character, and stops the extraction. The result is "hernandez".

Now, let's use `str_extract()` to separate full name into `name_first` and `name_last`.

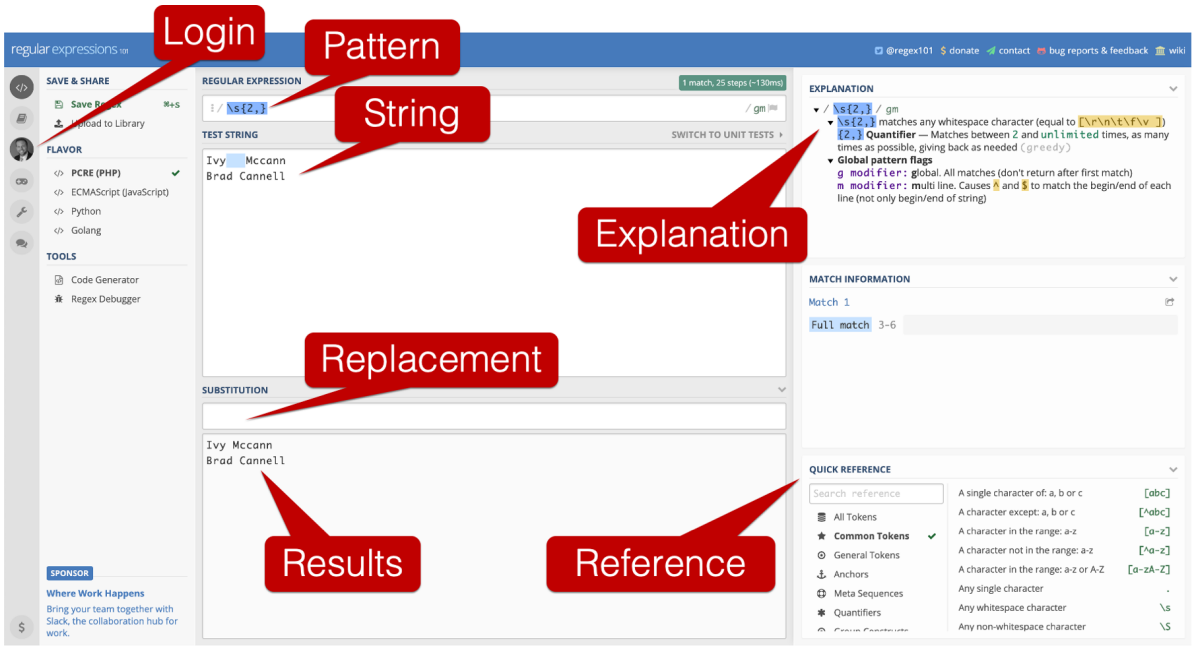
```
ehr <- ehr %>%
  mutate(
    # Separate name into first name and last name
    name_first = str_extract(name, "^\\w+"),
    name_last  = str_extract(name, "\\w+$")
  )
```

```
ehr %>%
  select(name, name_first, name_last)
```

```
# A tibble: 15 x 3
   name             name_first name_last
   <chr>            <chr>      <chr>
 1 zariah hernandez zariah     hernandez
 2 tatum chavez     tatum      chavez
 3 tatum chavez     tatum      chavez
 4 arabella george  arabella   george
 5 jasper decker    jasper     decker
 6 arabella george  arabella   george
 7 weston fox       weston     fox
 8 ryan edwards     ryan       edwards
 9 weston fox       weston     fox
10 ryan edwards     ryan       edwards
11 emma medrano     emma       medrano
12 ivy mccann       ivy        mccann
13 charlee carroll  charlee    carroll
14 ivy mccann       ivy        mccann
15 kane martin      kane       martin
```

The regular expressions we used in the examples above weren't super complex. We hope that leaves you feeling like you can use regular expression to complete data cleaning tasks that are actually useful, even if you haven't totally mastered them yet (I haven't totally mastered them either).

Before moving on, we want to introduce you to a free tool I use when I have to do more complex character string manipulations with regular expressions. It is the regular expressions 101 online regex tester and debugger.



In the screenshot above, I highlight some of the really cool features of the regex tester and debugger.

- First, you can use the regex tester without logging in. However, I typically do log in because that allows me to save regular expressions and use them again later.
- The top input box on the screen corresponds to what you would type into the `pattern` argument of the `str_replace()` function.
- The middle input box on the screen corresponds to what you would type into the `string` argument of the `str_replace()` function.
- The third input box on the screen corresponds to what you would type into the `replacement` argument of the `str_replace()` function, and the results are presented below.
- In addition, the regex tester and debugger has a quick reference pane that allows you to lookup different elements you might want to use in your regular expression. It also has an explanation pane that tells you what each of the elements in the current regular expression you typed out mean.

29.5 Dummy variables

Data collection tools in epidemiology often include "check all that apply" questions. In our `ehr` example data, patients were asked about what symptoms they were experiencing at admission. The choices were pain, headache, and nausea. They were allowed to check any combination of the three that they wanted. That results in a `symptoms` column in our data frame that looks like this:

> **Note**
>
> Any categorical variable can be transformed into dummy variables. Not just the variables that result from "check all that apply" survey questions. However, the "check all that apply" survey questions often require extra data cleaning steps relative to categorical variables that can only take a single value in each row.

```
ehr %>%
  select(name_first, name_last, symptoms)
```

```
# A tibble: 15 x 3
   name_first name_last symptoms
   <chr>      <chr>     <chr>
 1 zariah     hernandez "\"Pain\", \"Headache\", \"Nausea\""
 2 tatum      chavez    "Pain"
 3 tatum      chavez    "Pain"
 4 arabella   george    "\"Nausea\", \"Headache\""
 5 jasper     decker    "\"Pain\", \"Headache\""
 6 arabella   george    "\"Nausea\", \"Headache\""
 7 weston     fox       "Pain"
 8 ryan       edwards    <NA>
 9 weston     fox       "Pain"
10 ryan       edwards    <NA>
11 emma       medrano   "\"Nausea\", \"Headache\""
12 ivy        mccann    "\"Headache\", \"Pain\", \"Nausea\""
13 charlee    carroll   "Headache"
14 ivy        mccann    "\"Headache\", \"Pain\", \"Nausea\""
15 kane       martin     <NA>
```

Notice that some people didn't report their symptoms (`NA`), some people reported only one symptom, and some people reported multiple symptoms. The way the data is currently formatted is not ideal for analysis. For example, if I asked you to tell me how many people ever came in complaining of headache, how would you do that? Maybe like this:

```
ehr %>%
  group_by(symptoms) %>%
  summarise(n = n())
```

```
# A tibble: 7 x 2
  symptoms                                  n
  <chr>                                 <int>
1 "\"Headache\", \"Pain\", \"Nausea\""      2
2 "\"Nausea\", \"Headache\""                3
3 "\"Pain\", \"Headache\""                  1
4 "\"Pain\", \"Headache\", \"Nausea\""      1
5 "Headache"                                1
6 "Pain"                                    4
7  <NA>                                     3
```

In this case, you could probably count manually and get the right answer. But what if we had many more possible symptoms and many more rows. Counting would quickly become tedious and error prone. The solution is to create dummy variables. We can create dummy variables like this:

```
ehr <- ehr %>%
  mutate(
    pain     = str_detect(symptoms, "Pain"),
    headache = str_detect(symptoms, "Headache"),
    nausea   = str_detect(symptoms, "Nausea")
  )
```

```
ehr %>%
  select(symptoms, pain, headache, nausea)
```

```
# A tibble: 15 x 4
   symptoms                                 pain  headache nausea
   <chr>                                    <lgl> <lgl>    <lgl>
 1 "\"Pain\", \"Headache\", \"Nausea\""     TRUE  TRUE     TRUE
 2 "Pain"                                   TRUE  FALSE    FALSE
 3 "Pain"                                   TRUE  FALSE    FALSE
 4 "\"Nausea\", \"Headache\""               FALSE TRUE     TRUE
 5 "\"Pain\", \"Headache\""                 TRUE  TRUE     FALSE
 6 "\"Nausea\", \"Headache\""               FALSE TRUE     TRUE
 7 "Pain"                                   TRUE  FALSE    FALSE
 8  <NA>                                    NA    NA       NA
```

```
 9 "Pain"                                TRUE  FALSE    FALSE
10  <NA>                                 NA    NA       NA
11 "\"Nausea\", \"Headache\""            FALSE TRUE     TRUE
12 "\"Headache\", \"Pain\", \"Nausea\"" TRUE  TRUE     TRUE
13 "Headache"                           FALSE TRUE     FALSE
14 "\"Headache\", \"Pain\", \"Nausea\"" TRUE  TRUE     TRUE
15  <NA>                                 NA    NA       NA
```

Here's what we did above:

- we used `stringr`'s `str_detect()` function create three new dummy variables in our data frame.
- The first argument to the `str_detect()` function is `string`. The value passed the `string` argument should be the character string, or vector of character stings, we want to manipulate.
- The second argument to the `str_detect()` function is `pattern`. The value passed the `pattern` argument should be regular expression. The `str_detect()` function returns `TRUE` if it finds the pattern in the string and `FALSE` if it does not find the pattern in the string.
- Instead of having a single `symptoms` column that can take different combinations of the values `pain`, `headache`, and `nausea`, we create a new column for each value – the so-called dummy variables.
- Each dummy variable can take the value `TRUE`, `FALSE`, or `NA`. The value for each dummy variable is `TRUE` in rows were that symptom was reported and `FALSE` in rows where the symptom was not reported. For example, the value in the first row of the `pain` column is `TRUE` because the value in the first row of `symptoms` column ("Pain", "Headache", "Nausea") includes "Pain". However, the value in the fourth row of the `pain` column is `FALSE` because the value in the fourth row of `symptoms` column ("Nausea", "Headache") does not include "Pain".

Now, we can much more easily figure out how many people had each symptom.

```
table(ehr$headache)
```

```
FALSE  TRUE
    4     8
```

we should acknowledge that dummy variables typically take the values 0 and 1 instead of FALSE and TRUE. We can easily coerce our dummy variable values to 0/1 using the `as.numeric()` function. For example:

```r
ehr %>%
  select(pain) %>%
  mutate(pain_01 = as.numeric(pain))
```

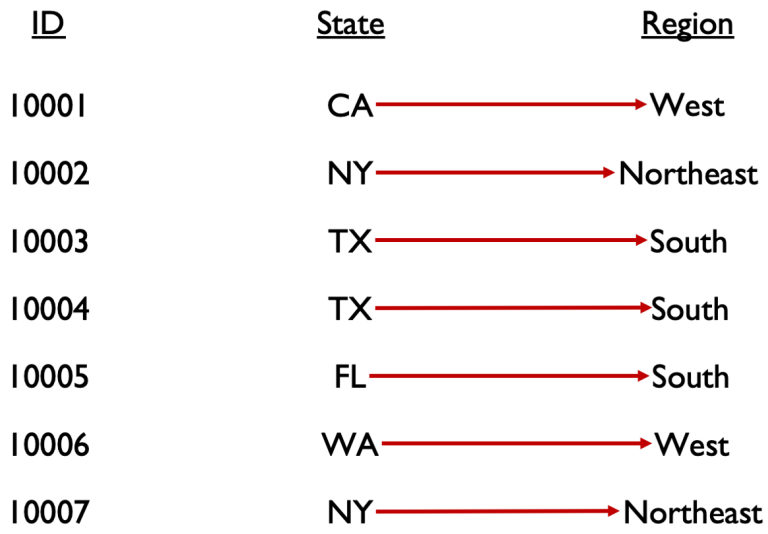
```
# A tibble: 15 x 2
   pain  pain_01
   <lgl>   <dbl>
 1 TRUE        1
 2 TRUE        1
 3 TRUE        1
 4 FALSE       0
 5 TRUE        1
 6 FALSE       0
 7 TRUE        1
 8 NA         NA
 9 TRUE        1
10 NA         NA
11 FALSE       0
12 TRUE        1
13 FALSE       0
14 TRUE        1
15 NA         NA
```

However, this step is sort of unnecessary in most cases because R treats `TRUE` and `FALSE` as `1` and `0` respectively when logical (i.e., TRUE/FALSE) vectors are passed to functions or operators that perform a mathematical operation.

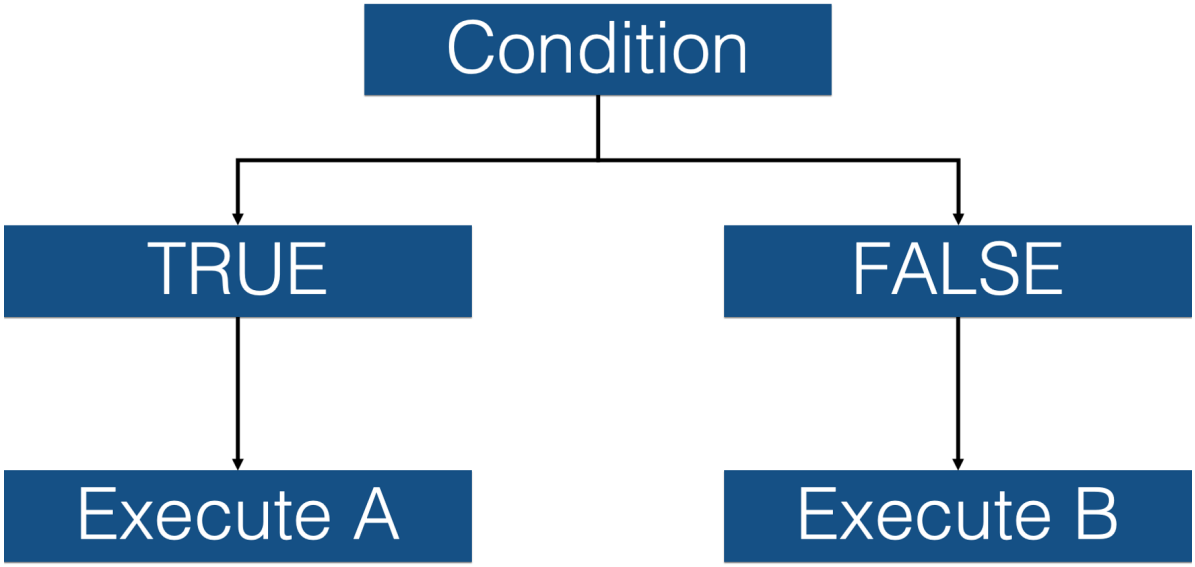
That concludes the chapter on working with character strings. Don't beat yourself up if you're feeling confused about regular expressions. They are really tough to wrap your head around at first! But, at least now you know they exist and can be useful for manipulating character strings. If you come across more complicated situations in the future, we suggest starting by checking out the stringr cheat sheet and practicing with the regular expressions 101 online regex tester and debugger before writing any R code.

30 Conditional Operations

There will often be times that we want to modify the values in one column of our data based on the values in one or more other columns in our data. For example, maybe we want to create a column that contains the region of the country someone is from, based on another column that contains the state they are from.



we don't really have a way to do this with the tools we currently have in our toolbox. We can manually type out all the region values, but that isn't very scalable. Wouldn't it be nice if we could just give R some rules, or conditions (e.g., TX is in the South, CA is in the West), and have R fill in the region values for us? Well, that's exactly what we are going to learn how to do in this chapter.



These kinds of operations are called conditional operations because we type in a set of conditions, R evaluates those conditions, and then executes a different process or procedure based on whether or not the condition is met.

As a silly example, let's say that we want our daughters to wear a raincoat if it's raining outside, but we don't want them to wear a raincoat if it is not raining outside. So, we give them a conditional request: "If it's raining outside, then make sure to wear your raincoat, please. Otherwise, please don't wear your raincoat."

In this hypothetical scenario, they say, "yes, dad," and then go to the window to see if it's raining. Then, they choose their next action (i.e., raincoat wearing) depending on whether the condition (raining) is met or not.

Just like we have to ask our daughters to put on a raincoat using conditional logic, we sometimes have to ask R to execute commands using conditional logic. Additionally, we have to do so in a way that R understands. For example, we can use `dplyr`'s `if_else()` function to ask R to execute commands conditionally. Let's go ahead and take a look at an example now:

```
library(dplyr)
```

```
rainy_days <- tibble(
  day     = 1:5,
  weather = c("rain", "rain", "no rain", "rain", "no rain")
) %>%
  print()
```

```
# A tibble: 5 x 2
    day weather
  <int> <chr>
1     1 rain
2     2 rain
3     3 no rain
```

```
4     4 rain
5     5 no rain
```

Here's what we did above:

- we simulated some data that contains information about whether or not it rained on each of 5 days.

Now, let's say that we want to create a new column in our data frame called `raincoat`. We want the value of `raincoat` to be `wear` on rainy days and `no wear` on days when it isn't raining. Here's how we can do that with the if_else() function:

```
rainy_days %>%
  mutate(
    raincoat = if_else(
      condition = weather == "rain",
      true      = "wear",
      false     = "no wear"
    )
  )
```

```
# A tibble: 5 x 3
    day weather raincoat
  <int> <chr>   <chr>
1     1 rain    wear
2     2 rain    wear
3     3 no rain no wear
4     4 rain    wear
5     5 no rain no wear
```

Here's what we did above:

- we used `dplyr`'s `if_else()` function to assign the values `wear` and `no wear` to the column `raincoat` conditional on the values in each row of the `weather` column.
- You can type `?if_else` into our R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `if_else()` function is the `condition` argument. The condition should typically be composed of a series of operands and operators (we'll talk more about these soon) that tell R the condition(s) that we want it to test. For example, is the value of `weather` equal to `rain`?

- The second argument to the `if_else()` function is the `true` argument. The value passed to the `true` argument tells R what value the `if_else()` function should return when the condition is `TRUE`. In this case, we told `if_else()` to return the character value `wear`.
- The third argument to the `if_else()` function is the `false` argument. The value passed to the `false` argument tells R what value the `if_else()` function should return when the condition is `FALSE`. In this case, we told `if_else()` to return the character value `no wear`.
- Finally, we assigned all the values returned by the `if_else()` function to a new column that we named `raincoat`.

> **Note**
>
> For the rest of the book, we will pass values to the `if_else()` function by position instead of name. In other words, we won't write `condition =`, `true =`, or `false =` anymore. However, the first value passed to the `if_else()` function will always be passed to the `condition` argument, the second value will always be passed to the `true` argument, and the third value will always be passed to the `false` argument.

Before moving on, let's dive into this a little further. R must always be able to reduce whatever value we pass to the `condition` argument of `if_else()` to TRUE or FALSE. That's how R views any expression we pass to the `condition` argument. We can literally even pass the value `TRUE` or the value `FALSE` (not that doing so has much practical application):

```
if_else(TRUE, "wear", "no wear")
```

```
[1] "wear"
```

Because the value passed to the `condition` argument is `TRUE` (in this case, literally), the `if_else()` function returns the value `wear`. What happens if we use this code to assign values to the `raincoat` column?

```
rainy_days %>%
  mutate(
    raincoat = if_else(TRUE, "wear", "no wear")
  )
```

```
# A tibble: 5 x 3
    day weather raincoat
  <int> <chr>   <chr>
1     1 rain    wear
```

```
2     2 rain    wear
3     3 no rain wear
4     4 rain    wear
5     5 no rain wear
```

Again, the `if_else()` function returns the value `wear` because the value passed to the `condition` argument is `TRUE`. Then, R uses its recycling rules to copy the value `wear` to every row of the `raincoat` column. What would do you think will happen if we pass the value `FALSE` to the `condition` argument instead?

```
rainy_days %>%
  mutate(
    raincoat = if_else(FALSE, "wear", "no wear")
  )
```

```
# A tibble: 5 x 3
    day weather raincoat
  <int> <chr>   <chr>
1     1 rain    no wear
2     2 rain    no wear
3     3 no rain no wear
4     4 rain    no wear
5     5 no rain no wear
```

Hopefully, that was the result you expected. The `if_else()` function returns the value `no wear` because the value passed to the `condition` argument is `FALSE`. Then, R uses its recycling rules to copy the value `no wear` to every row of the `raincoat` column.

we can take this a step further and actually pass a vector of logical (`TRUE`/`FALSE`) values to the `condition` argument. For example:

```
rainy_days %>%
  mutate(
    raincoat = if_else(c(TRUE, TRUE, FALSE, TRUE, FALSE), "wear", "no wear")
  )
```

```
# A tibble: 5 x 3
    day weather raincoat
  <int> <chr>   <chr>
1     1 rain    wear
2     2 rain    wear
```

```
3     3 no rain no wear
4     4 rain    wear
5     5 no rain no wear
```

In reality, that's sort of what we did in the very first `if_else()` example above. But, instead of typing the values manually, we used an expression that returned a vector of logical values. Specifically, we used the equality operator (`==`) to check whether or not each value in the `weather` column was equal to the value "rain" or not.

```
rainy_days$weather == "rain"
```

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

That pretty much covers the basics of how the `if_else()` function works. Next, let's take a look at some of the different combinations of operands and operators that we can combine and pass to the `condition` argument of the `if_else()` function.

30.1 Operands and operators

Let's start by taking a look at some commonly used operands:

Operands	
Type	Example
Variables	name ht_in dob
Character Constants	"Alice" "Married" "May"
Numeric Constants	500 0 -3
Date Constants	2000-01-01 2020-02-26 21:33:09

As we can see in the table above, operands are the *values* we want to check, or test. Operands can be variables or they can be individual values (also called constants). The example above (`weather == "rain"`) contained two operands; the variable `weather` and the character constant `"rain"`. The operator we used in this case was the equality operator (`==`). Next, let's take a look at some other commonly used operators.

Comparison Operators and Numeric Variables			
Symbol(s)	**Definition**	**Example**	**Result**
==	Equal to	1 == 1 1 == 2	TRUE FALSE
!=	Not equal to	1 != 1 1 != 2	FALSE TRUE
>	Greater than	1 > 2 2 > 1	FALSE TRUE
<	Less than	1 < 2 2 < 1	TRUE FALSE
>=	Greater than or equal to	1 >= 1 1 >= 2	TRUE FALSE
<=	Less than or equal to	1 <= 1 1 <= 2	TRUE TRUE
%in%	Equal to one of a list	1 %in% c(1, 2) 1 %in% c(2, 3)	TRUE FALSE
is.na	Is missing	is.na(NA) !is.na(NA)	TRUE FALSE

Comparison Operators and Character Variables			
Symbol(s)	**Definition**	**Example**	**Result**
==	Equal to	"A" == "A" "A" == "a"	TRUE FALSE
!=	Not equal to	"A" != "A" "A" != "a"	FALSE TRUE
>	Greater than	"A" > "a" "A" > "B"	TRUE FALSE
<	Less than	"A" < "a" "A" < "B"	FALSE TRUE
>=	Greater than or equal to	"A" >= "a" "A" >= "B"	TRUE FALSE
<=	Less than or equal to	"A" <= "a" "A" <= "B"	FALSE TRUE
%in%	Equal to one of a list	"A" %in% c("A", "B") "A" %in% c("a", "b")	TRUE FALSE
is.na	Is missing	is.na(NA) !is.na(NA)	TRUE FALSE

Arithmetic Operators and Numeric Variables			
Symbol(s)	**Definition**	**Example**	**Result**
+	Addition	1 + 2 2 + 1	3 3
-	Subtraction	1 - 2 2 -1	-1 1
*	Multiplication	1 * 2 2 * 1	2 2
/	Division	1 / 2 2 / 1 1 / 0	0.5 2 0 Inf
^ *or* **	Exponentiation	2 ** 2 2 ^ 2	4 4
%%	Modulus	1 %% 2 2 %% 2 3 %% 2 4 %% 2	1 0 1 0

Logical Operators			
Symbol(s)	**Definition**	**Example**	**Result**
&	AND	TRUE & TRUE TRUE & FALSE X <- 1 X == 1 & X == 2	TRUE FALSE FALSE
|	OR	TRUE | TRUE TRUE | FALSE X <- 1 X == 1 | X == 2	TRUE TRUE TRUE
!	NOT	!TRUE !FALSE X <- 1 !X == 1 !X == 2	FALSE TRUE FALSE TRUE

we think that most of the operators above will be familiar, or a least intuitive, for most of you. However, we do want to provide a little bit of commentary for a few of them.

- we haven't seen the `%in%` operator before, but we will wait to discuss it below.
- Some of you may have been a little surprised by the results we get from using less than (`<`) and greater than (`>`) with characters. It's basically just testing alphabetical order. A comes before B in the alphabet, so A is less than B. Additionally, when two letters are the same, the upper-case letter is considered greater than the lowercase letter. However, alphabetical order takes precedence over case. So, b is still greater than A even though b is lowercase and A is upper case.
- Many of you may not have seen the modulus operator (`%%`) before. The modulus operator returns the remainder that is left after dividing two numbers. For example, 4 divided by 2 is 2 with a remainder of 0 because 2 goes into 4 *exactly* two times. Said another way, 2 * 2 = 4 and 4 - 4 = 0. So, `4 %% 2 = 0`. However, 3 divided by 2 is 1 with a remainder of 1 because 2 goes into 3 one time with 1 left over. Said another way, 2 * 1 = 2 and 3 - 2 = 1. So, `3 %% 2 = 1`. How is this useful? Well, the only times we can remember using the modulus operator have been when we needed to separate even and odd rows of a data frame. For example, let's say that we have a data frame where each person has two rows. The first row always corresponds to treatment A and the second row always corresponds to treatment B. However, for some reason (maybe blinding?), there was no `treatment` column in the data when we received it. We could use the modulus operator to add a `treatment` column like this:

```r
df <- tibble(
  id        = c(1, 1, 2, 2),
  outcome   = c(0, 1, 1, 1)
) %>%
  print()
```

```
# A tibble: 4 x 2
     id outcome
  <dbl>   <dbl>
1     1       0
2     1       1
3     2       1
4     2       1
```

```r
df %>%
  mutate(
    # Odd rows are treatment A
    # Even rows are treatment B
    treatment = if_else(row_number() %% 2 == 1, "A", "B")
  )
```

```
# A tibble: 4 x 3
     id outcome treatment
  <dbl>   <dbl> <chr>
1     1       0 A
2     1       1 B
3     2       1 A
4     2       1 B
```

- we also want to remind you that we should always use the `is.na()` function to check for missing values. Not the equality operator (`==`). Using the equality operator when there are missing values can give results that may be unexpected. For example:

```r
df <- tibble(
  name1 = c("Jon", "John", NA),
  name2 = c("Jon", "Jon", "Jon")
)
```

```r
df %>%
  mutate(
    name_match = name1 == name2
  )
```

```
# A tibble: 3 x 3
  name1 name2 name_match
  <chr> <chr> <lgl>
1 Jon   Jon   TRUE
2 John  Jon   FALSE
3 <NA>  Jon   NA
```

Many of us would expect the third value of the `name_match` column to be `FALSE` instead of `NA`. There are a couple of different ways we can get `FALSE` in the third row instead of `NA`. One way, although not necessarily the best way, is to use the `if_else()` function:

```
df %>%
  mutate(
    name_match = name1 == name2,
    name_match = if_else(is.na(name_match), FALSE, name_match)
  )
```

```
# A tibble: 3 x 3
  name1 name2 name_match
  <chr> <chr> <lgl>
1 Jon   Jon   TRUE
2 John  Jon   FALSE
3 <NA>  Jon   FALSE
```

Here's what we did above:

- we used `dplyr`'s `if_else()` function to assign the value `FALSE` to the column `name_match` where the original value of `name_match` was `NA`.
- The value we passed to the `condition` argument was `is.na(name_match)`. In doing so, we asked R to check each value of the `name_match` column and see if it was `NA`.
- If it was `NA`, then we wanted to return the value that we passed to the `true` argument. Somewhat confusingly, the value we passed to the `true` argument was `FALSE`. All that means is that we wanted `if_else()` to return the literal value `FALSE` when the value for `name_match` was `NA`.
- If the value in `name_match` was NOT `NA`, then we wanted to return the value that we passed to the `false` argument. In this case, we asked R to return the value that already exists in the `name_match` column.
- In more informal language, we asked R to replace missing values in the `name_match` column with `FALSE` and leave the rest of the values unchanged.

30.2 Testing multiple conditions simultaneously

So far, we have only ever passed one condition to the `condition` argument of the `if_else()` function. However, we can pass as many conditions as we want. Having said that, more than 2, or maybe 3, gets very convoluted. Let's go ahead and take a look at a couple of examples now. We'll start by simulating some blood pressure data:

```
blood_pressure <- tibble(
  id     = 1:10,
  sysbp  = c(152, 120, 119, 123, 135, 83, 191, 147, 209, 166),
  diasbp = c(78, 60, 88, 76, 85, 54, 116, 95, 100, 106)
) %>%
  print()
```

```
# A tibble: 10 x 3
      id sysbp diasbp
   <int> <dbl>  <dbl>
 1     1   152     78
 2     2   120     60
 3     3   119     88
 4     4   123     76
 5     5   135     85
 6     6    83     54
 7     7   191    116
 8     8   147     95
 9     9   209    100
10    10   166    106
```

A person may be categorized as having normal blood pressure when their systolic blood pressure is less than 120 mmHG *AND* their diastolic blood pressure is less than 80 mmHG. We can use this information and the `if_else()` function to create a new column in our data frame that contains information about whether each person in our simulated data frame has normal blood pressure or not:

```
blood_pressure %>%
  mutate(bp = if_else(sysbp < 120 & diasbp < 80, "Normal", "Not Normal"))
```

```
# A tibble: 10 x 4
      id sysbp diasbp bp
   <int> <dbl>  <dbl> <chr>
 1     1   152     78 Not Normal
```

```
 2     2   120     60 Not Normal
 3     3   119     88 Not Normal
 4     4   123     76 Not Normal
 5     5   135     85 Not Normal
 6     6    83     54 Normal
 7     7   191    116 Not Normal
 8     8   147     95 Not Normal
 9     9   209    100 Not Normal
10    10   166    106 Not Normal
```

Here's what we did above:

- we used `dplyr`'s `if_else()` function to create a new column in our data frame (`bp`) that contains information about whether each person has normal blood pressure or not.
- we actually passed two conditions to the `condition` argument. The first condition was that the value of `sysbp` had to be less than `120`. The second condition was that the value of `diasbp` had to be less than `80`.
- Because we separated these conditions with the AND operator (`&`), both conditions had to be true in order for the `if_else()` function to return the value we passed to the `true` argument – `Normal`. Otherwise, the `if_else()` function returned the value we passed to the `false` argument – `Not Normal`.
- Participant 2 had a systolic blood pressure of 120 and a diastolic blood pressure of 60. Although 60 is less than 80 (condition number 2), 120 is not less than 120 (condition number 1). So, the value returned by the `if_else()` function was `Not Normal`.
- Participant 3 had a systolic blood pressure of 119 and a diastolic blood pressure of 88 Although 119 is less than 120 (condition number 1), 88 is not less than 80 (condition number 2). So, the value returned by the `if_else()` function was `Not Normal`.
- Participant 6 had a systolic blood pressure of 83 and a diastolic blood pressure of 54. In this case, conditions 1 *and* 2 were met. So, the value returned by the `if_else()` function was `Normal`.

This is useful! However, in some cases, we need to be able to test conditions sequentially, rather than simultaneously, and return a different value for each condition.

30.3 Testing a sequence of conditions

Let's say that we wanted to create a new column in our `blood_pressure` data frame that contains each person's blood pressure category according to the following scale:

Category	Systolic mmHg		Diastolic mmHG
Normal	Less than 120	AND	Less than 80
Elevated	120 – 129	AND	Less than 80
Hypertension Stage 1	130 – 139	OR	80-89
Hypertension Stage 2	140 or higher	OR	90 or higher

This is the perfect opportunity to use `dplyr`'s `case_when()` function. Take a look:

```
blood_pressure %>%
  mutate(
    bp = case_when(
      sysbp < 120 & diasbp < 80                               ~ "Normal",
      sysbp >= 120 & sysbp < 130 & diasbp < 80                ~ "Elevated",
      sysbp >= 130 & sysbp < 140 | diasbp >= 80 & diasbp < 90 ~ "Hypertension Stage 1",
      sysbp >= 140 | diasbp >= 90                             ~ "Hypertension Stage 2"
    )
  )
```

```
# A tibble: 10 x 4
      id sysbp diasbp bp
   <int> <dbl>  <dbl> <chr>
 1     1   152     78 Hypertension Stage 2
 2     2   120     60 Elevated
 3     3   119     88 Hypertension Stage 1
 4     4   123     76 Elevated
 5     5   135     85 Hypertension Stage 1
 6     6    83     54 Normal
 7     7   191    116 Hypertension Stage 2
 8     8   147     95 Hypertension Stage 2
```

```
 9     9   209    100 Hypertension Stage 2
10    10   166    106 Hypertension Stage 2
```

Here's what we did above:

- we used `dplyr`'s `case_when()` function to create a new column in our data frame (`bp`) that contains information about each person's blood pressure category.
- You can type `?case_when` into our R console to view the help documentation for this function and follow along with the explanation below.
- The `case_when()` function only has a single argument – the `...` argument. You should pass one or more two-sided formulas separated by commas to this argument. What in the heck does that mean?
  - When the help documentation refers to a two-sided formula, it means this: `LHS ~ RHS`. Here, `LHS` means left-hand side and `RHS` means right-hand side.
  - The `LHS` should be the condition or conditions that we want to test. You can think of this as being equivalent to the `condition` argument of the `if_else()` function.
  - The `RHS` should be the value we want the `case_when()` function to return when the condition on the left-hand side is met. You can think of this as being equivalent to the `true` argument of the `if_else()` function.
  - The tilde symbol (`~`) is used to separate the conditions on the left-hand side and the return values on the right-hand side.
- The `case_when()` function doesn't have a direct equivalent to the `if_else()` function's `false` argument. Instead, it evaluates each two-sided formula sequentially until if finds a condition that is met. If it never finds a condition that is met, then it returns an `NA`. We will expand on this more below.
- Finally, we assigned all the values returned by the `case_when()` function to a new column that we named `bp`.

> **Note**
>
> Traditionally, the tilde symbol (`~`) is used to represent relationships in a statistical model. Here, it doesn't have that meaning. We assume this symbol was picked somewhat out of necessity. Remember, any of the comparison operators, arithmetic operators, and logical operators may be used to define a condition in the left-hand side, and commas are used to separated multiple two-sided formulas. Therefore, there aren't very many symbols left to choose from. Therefore, tilde it is. That's our guess anyway.

The `case_when()` function was really useful for creating the `bp` column above, but there was also a lot going on there. Next, we'll take a look at a slightly less complex example and clarify a few things along the way.

30.4 Recoding variables

In epidemiology, recoding variables is really common. For example, we may collect information about people's ages as a continuous variable, but decide that it makes more sense to **collapse** age into age categories for our analysis. Let's say that our analysis plan calls for assigning each of our participants to one of the following age categories:

1 = `child` when the participant is less than 12 years old
2 = `adolescent` when the participant is between the ages of 12 and less than 18
3 = `adult` when the participant is 18 years old or older

> **Note**
>
> You may not have ever heard of **collapsing** variables before. It simply means combing two or more values of our variable. We can collapse continuous variables into categories, as we discussed in the example above, or we can collapse categories into broader categories (as we will see with the race category example below). After we collapse a variable, it always contains fewer (and broader) possible values than it contained before we collapsed it.

we're going to show you how to do this below using the `case_when()` function. However, we're going to do it piecemeal so that we can highlight a few important concepts. First, let's simulate some data that includes 10 participant's ages.

```
# Simulate some age data
set.seed(123)
ages <- tibble(
  id  = 1:10,
  age = c(sample(1:30, 9, TRUE), NA)
) %>%
  print()
```

```
# A tibble: 10 x 2
      id   age
   <int> <int>
 1     1    15
 2     2    19
 3     3    14
```

```
 4     4     3
 5     5    10
 6     6    18
 7     7    22
 8     8    11
 9     9     5
10    10    NA
```

Then, let's start the process of collapsing the `age` column into a new column called `age_3cat` that contains the 3 age categories we discussed above:

```
ages %>%
  mutate(
    age_3cat = case_when(
      age < 12 ~ 1
    )
  )
```

```
# A tibble: 10 x 3
      id   age age_3cat
   <int> <int>    <dbl>
 1     1    15       NA
 2     2    19       NA
 3     3    14       NA
 4     4     3        1
 5     5    10        1
 6     6    18       NA
 7     7    22       NA
 8     8    11        1
 9     9     5        1
10    10    NA       NA
```

Here's what we did above:

- we used `dplyr`'s `case_when()` function to create a new column in our data frame (`age_3cat`) that will eventually categorize each participant into one of 3 categories depending on their continuous age value.
- Notice that we only passed one two-sided formula to the `case_when()` function – `age < 12 ~ 1`.
  - The `RHS` of the two-sided formula is `age < 12`. This tells the `case_when()` function to check whether or not every value in the `age` column is less than `12` or not.

  - The `LHS` of the two-sided formula is `1`. This tells the `case_when()` function what value to return each time it finds a value less than `12` in the `age` column.
  - The tilde symbol is used to separate the `RHS` and the `LHS` of the two-sided formula.

- Here is how the `case_when()` function basically works. It will test the condition on the left-hand side for each value of the variable, or variables, passed to the left-hand side (i.e., `age`). If the condition is met (i.e., `< 12`), then it will return the value on the right-hand side of the tilde (i.e., `1`). If the condition is not met, it will test the condition in the next two-sided formula. When there are no more two-sided formulas, then it will return an `NA`.
  - Above, the first value in `age` is `15`. `15` is *NOT* less than `12`. So, `case_when()` tries to move on to the next two-sided formula. However, there is no next two-sided formula. So, the first value returned by the `case_when()` function is `NA`. The same is true for the next two values of age.
  - The fourth value in `age` is `3`. `3` is less than `12`. So, the fourth value returned by the `case_when()` function is `1`. And so on…
  - Finally, after the `case_when()` function has tested all conditions, the returned values are assigned to a new column that we named `age_3cat`.

- Notice that we named the new variable `age_3cat`. We're not sure where we picked up this naming convention, but we use it a lot when we collapse variables. The basic format is the name of variable we're collapsing, an underscore, and the number of categories in the collapsed variable. We like using this convention for two reasons. First, the resulting column names are meaningful and informative. Second, we don't have to spend any time trying to think of a different meaningful or informative name for my new variable. It's totally fine if you don't adopt this naming convention, but we would recommend that you try to use names that are more informative than `age2` or something like that.
- Notice that we used a number (`1`) on the right-hand side of the two-sided formula above. We could have used a character value instead (i.e., `child`); however, for reasons we discussed in the section on factor variables, we prefer to recode my variables using numeric categories and then later creating a factor version of the variable using the `_f` naming convention.

Now, let's add a second two-sided formula to our `case_when()` function.

```
ages %>%
  mutate(
    age_3cat = case_when(
      age < 12             ~ 1,
      age >= 12 & age < 18 ~ 2
```

```
    )
  )
```

```
# A tibble: 10 x 3
      id   age age_3cat
   <int> <int>    <dbl>
 1     1    15        2
 2     2    19       NA
 3     3    14        2
 4     4     3        1
 5     5    10        1
 6     6    18       NA
 7     7    22       NA
 8     8    11        1
 9     9     5        1
10    10    NA       NA
```

Here's what we did above:

- we used `dplyr`'s `case_when()` function to create a new column in our data frame (`age_3cat`) that will eventually categorize each participant into one of 3 categories depending on their continuous age value.
- Notice that this time we passed two two-sided formulas to the `case_when()` function – `age < 12 ~ 1` and `age >= 12 & age < 18 ~ 2`.
  - Notice that we separated the two two-sided formulas with a comma (i.e., immediately after the `1` in `age < 12 ~ 1`.
  - Notice that the second two-sided formula is actually testing two conditions. First, it tests whether or not the value of age is greater than or equal to 12. Then, it tests whether or not the value of age is less than 18.
  - Because we separated the two conditions with the and operator (`&`), both must be TRUE for `case_when()` to return the value `2`. Otherwise, it will move on to the next two-sided formula.
  - Above, the first value in `age` is `15`. `15` is *NOT* less than `12`. So, `case_when()` moves on to evaluate the next two-sided formula. `15` is greater than or equal to `12` *AND* `15` is less than `18`. Because both conditions of the second two-sided formula were met, `case-when()` returns the value on the right-hand side of the second two-sided formula – `2`. So, the first value returned by the `case_when()` function is `2`.

  - The second value in `age` is `19`. `19` is *NOT* less than `12`. So, `case_when()` moves on to evaluate the next two-sided formula. `19` is greater than or equal to `12`, but `19` is *NOT* less than `18`. So, `case_when()` tries to move on to the next two-sided formula. However, there is no next two-sided formula. So, the second value returned by the `case_when()` function is `NA`.
  - The fourth value in `age` is `3`. `3` is less than `12`. So, the fourth value returned by the `case_when()` function is `1`. **At this point, because a condition was met, `case_when()` does not continue checking the current value of `age` against the remaining two-sided formulas. It returns a `1` and moves on to the next value of `age`.**
  - Finally, after the `case_when()` function has tested all conditions, the returned values are assigned to a new column that we named `age_3cat`.

- In everyday speech, we may express the second two-sided condition above as "categorize all people between the ages of 12 and 18 as an adolescent." we want to make two points about that before moving on.
  - First, while that statement may be totally reasonable in everyday speech, it isn't quite specific enough for what we are trying to do here. "Between 12 and 18" is a little bit ambiguous. What category is a person put in if they are exactly 12? What category are they put in if they are exactly 18? So, clearly we need to be more precise. We're not aware of any hard and fast rules for making these kinds of decisions about categorization, but we tend to *include* the lower end of the range in the current category and *exclude* the value on the upper end of the range in the current category. So, in the example above, we would say, "categorize all people between the ages of 12 and less than 18 as an adolescent."
  - Second, when we are testing for a "between" condition like this one, we often see students write code like this: `age >= 12 & < 18`. R won't understand that. We have to use the column name in each condition to be tested (i.e., `age >= 12 & age < 18`), even though it doesn't change. Otherwise, we get an error that looks something like this:

```
ages %>%
  mutate(
    age_3cat = case_when(
      age < 12             ~ 1,
      age >= 12 & < 18     ~ 2
    )
  )
```

```
Error in parse(text = input): <text>:5:19: unexpected '<'
4:       age < 12             ~ 1,
5:       age >= 12 & <
                     ^
```

Ok, let's go ahead and wrap up this age category variable:

```
ages %>%
  mutate(
    age_3cat = case_when(
      age < 12             ~ 1,
      age >= 12 & age < 18 ~ 2,
      age >= 18            ~ 3
    )
  )
```

```
# A tibble: 10 x 3
      id   age age_3cat
   <int> <int>    <dbl>
 1     1    15        2
 2     2    19        3
 3     3    14        2
 4     4     3        1
 5     5    10        1
 6     6    18        3
 7     7    22        3
 8     8    11        1
 9     9     5        1
10    10    NA       NA
```

Here's what we did above:

- we used `dplyr`'s `case_when()` function to create a new column in our data frame (`age_3cat`) that categorized each participant into one of 3 categories depending on their continuous age value.

30.5 case_when() is lazy

What do we mean when we say that `case_when()` is lazy? Well, it may not have registered when we mentioned it above, but `case_when()` stops evaluating two-sided functions for a value as soon as it finds one that is `TRUE`. For example:

```r
df <- tibble(
  number = c(1, 2, 3)
) %>%
  print()
```

```
# A tibble: 3 x 1
  number
   <dbl>
1      1
2      2
3      3
```

```r
df %>%
  mutate(
    size = case_when(
      number < 2 ~ "Small",
      number < 3 ~ "Medium",
      number < 4 ~ "Large"
    )
  )
```

```
# A tibble: 3 x 2
  number size
   <dbl> <chr>
1      1 Small
2      2 Medium
3      3 Large
```

Why wasn't the value for the `size` column `Large` in every row of the data frame? After all, `1`, `2`, and `3` are all less than `4`, and `number < 4` was the final possible two-sided formula that could have been evaluated for each value of `number`. The answer is that `case_when()` is lazy. The first value in `number` is `1`. `1` is less than `2`. So, the condition in the first two-sided formula evaluates to `TRUE`. So, `case_when()` immediately returns the value on the right-hand side (`Small`) and **does not continue checking two-sided formulas. It moves on to the next value of `number`.**

The fact that `case_when()` is lazy isn't a bad thing. It's just something to be aware of. In fact, we can often use it to our advantage. For example, we can use `case_when()`'s laziness to rewrite the `age_3cat` code from above a little more succinctly:

```r
ages %>%
  mutate(
    age_3cat = case_when(
      age < 12  ~ 1,
      age < 18  ~ 2,
      age >= 18 ~ 3
    )
  )
```

```
# A tibble: 10 x 3
      id   age age_3cat
   <int> <int>    <dbl>
 1     1    15        2
 2     2    19        3
 3     3    14        2
 4     4     3        1
 5     5    10        1
 6     6    18        3
 7     7    22        3
 8     8    11        1
 9     9     5        1
10    10    NA       NA
```

Here's what we did above:

- Because `case_when()` is lazy, we were able to omit the `age >= 12` condition from the second two-sided formula. It's unnecessary because the value `1` is immediately returned for every person with an `age` value less than `12`. By definition, any value being evaluated in the second two-sided function (`age < 18`) has an age value greater than or equal to 12.

30.6 Recode missing

We've already talked about how R uses the special `NA` value to represent missing data. We've also learned how to convert other representations of missing data (e.g., ".") to `NA` when we are importing data. However, It is extremely common for data sets that we use in epidemiology to include "don't know" and "refused" answer options in addition to true "missing". By convention, those options are often coded as `7` and `9`. For questions that include `7` or more response options (e.g., month), then `77` and `99` are commonly used to represent "don't know" and "refused". For questions that include `77` or more response options (e.g., age), then `777` and `999` are commonly used to represent "don't know" and "refused".

Differentiating between true missing (i.e., the respondent was never asked the question or just left the response blank on a written questionnaire), don't know (i.e., the respondent doesn't know the answer), and refused (i.e., the respondent knows the answer, but doesn't want to reveille it – possibly out of shame, fear, or embarrassment) can be of some interest for survey design purposes. However, all three of the values described above typically just amount to missing data by the time we get around to the substantive analyses. In other words, knowing that a person refused to give their age doesn't help me figure out how old they are any more than if they had never been asked at all. Therefore, we commonly use conditional operations in epidemiology to recode these kinds of values to explicitly missing values (`NA`).

We'll walk through an example below, but first we will simulate some additional data. Specifically, we'll add a `race` column and a `hispanic` column to our `ages` data frame, and name the new data frame `demographics`.

Let's assume that we have a survey that asks people what race they most identify with. For the moment, let's assume that they can only select one race. Further, let's say that the options they are given to select from are:

1 = White
2 = Black or African American
3 = American Indian or Alaskan Native
4 = Asian
5 = Pacific Islander
7 = Don't know
9 = Refused

Let's say that we also ask if they self-identify their ethnicity as Hispanic or not. The options they are given to select from are:

0 = No, not Hispanic
1 = Yes, Hispanic
7 = Don't know
9 = Refused

```
demographics <- ages %>%
  mutate(
    race     = c(1, 2, 1, 4, 7, 1, 2, 9, 1, 3),
    hispanic = c(7, 0, 1, 0, 1, 0, 1, 9, 0, 1)
  ) %>%
  print()
```

```
# A tibble: 10 x 4
      id   age  race hispanic
   <int> <int> <dbl>    <dbl>
```

```
 1     1    15     1        7
 2     2    19     2        0
 3     3    14     1        1
 4     4     3     4        0
 5     5    10     7        1
 6     6    18     1        0
 7     7    22     2        1
 8     8    11     9        9
 9     9     5     1        0
10    10    NA     3        1
```

A very common way that we may want to transform data like this is to collapse race and ethnicity into as single combined race and ethnicity column. Further, notice that American Indian or Alaskan Native race and Asian race are only observed once each, and Pacific Islander race is not observed at all. When values are observed very few times in the data like this, it is common to collapse them into an "other" category. Therefore, our new combined race and ethnicity column will have the following possible values:

1 = White, non-Hispanic
2 = Black, non-Hispanic
3 = Hispanic, any race
4 = Other race, non-Hispanic

There are multiple ways that we can create this new column. We could start by using `if_else()` to recode `7` and `9` to missing:

```
demographics %>%
  mutate(
    # Recode 7 and 9 to missing
    race_recode = if_else(race == 7 | race == 9, NA, race),
    hispanic_recode = if_else(hispanic == 7 | hispanic == 9, NA, hispanic)
  )
```

```
# A tibble: 10 x 6
      id   age  race hispanic race_recode hispanic_recode
   <int> <int> <dbl>    <dbl>       <dbl>           <dbl>
 1     1    15     1        7           1              NA
 2     2    19     2        0           2               0
 3     3    14     1        1           1               1
 4     4     3     4        0           4               0
 5     5    10     7        1          NA               1
 6     6    18     1        0           1               0
 7     7    22     2        1           2               1
```

```
 8     8    11     9        9          NA              NA
 9     9     5     1        0           1               0
10    10    NA     3        1           3               1
```

We intentionally made this error because it's a really easy one to make, and you will probably make it too. If we look back to the Let's get programming chapter, we will see that we briefly discussed the `NA` value being type logical by default. We also talked about "type coercion" and how most of the time we don't have to worry about it. We said that R generally coerces `NA` to `NA_character`, `NA_double`, or whatever specific version of `NA` is most appropriate for the data type automatically. We also said that there are some exceptions. Notably, when using the `if_else()` and `case_when()` functions, R will throw an error instead of automatically type coercing. Finally, we said we would discuss it later. It's later now. Long story short, the developers of the `if_else()` function do this on purpose to make the function's returned result more predictable and slightly faster.

For us, this just means that we have to remember to use `NA_character`, `NA_integer`, or `NA_real` as appropriate. For example, the error message above says, "`false` must be a logical vector, not a double vector." This means that the value we passed to the `false` argument was type double, but `if_else()` was expecting it to be type logical. Why? Well, `if_else()` was expecting it to be type logical because the value we passed to the `true` argument (`NA`) is type logical, and vectors can only ever have one type. To fix this error, we simply need to change the value we are passing to the `true` argument from logical (`NA`) to double (`NA_real`) so that it matches the values we are passing to the `false` argument.

Let's try again using `NA_real` instead of `NA`.

```
demographics %>%
  mutate(
    # Recode 7 and 9 to missing
    race_recode = if_else(race == 7 | race == 9, NA_real_, race),
    hispanic_recode = if_else(hispanic == 7 | hispanic == 9, NA_real_, hispanic)
  )
```

```
# A tibble: 10 x 6
      id   age  race hispanic race_recode hispanic_recode
   <int> <int> <dbl>    <dbl>       <dbl>           <dbl>
 1     1    15     1        7           1              NA
 2     2    19     2        0           2               0
 3     3    14     1        1           1               1
 4     4     3     4        0           4               0
 5     5    10     7        1          NA               1
 6     6    18     1        0           1               0
 7     7    22     2        1           2               1
```

```
 8     8    11     9        9          NA              NA
 9     9     5     1        0           1               0
10    10    NA     3        1           3               1
```

Great! We can move on with creating our new race and ethnicity column now that we've explicitly transformed 7's and 9's to NA. There's nothing "new" in the code below, so we're not going to explain it line-by-line. However, it's a little bit dense, so we recommend that you take a few minutes to review it thoroughly and make sure you understand what each line is doing.

```
demographics %>%
  mutate(
    # Recode 7 and 9 to missing
    race_recode     = if_else(race == 7 | race == 9, NA_real_, race),
    hispanic_recode = if_else(hispanic == 7 | hispanic == 9, NA_real_, hispanic),
    race_eth_4cat   = case_when(
      # White, non-Hispanic
      race_recode == 1 & hispanic_recode == 0 ~ 1,
      # Black, non-Hispanic
      race_recode == 2 & hispanic_recode == 0 ~ 2,
      # American Indian or Alaskan Native to Other race, non-Hispanic
      race_recode == 3 & hispanic_recode == 0 ~ 4,
      # Asian to Other race, non-Hispanic
      race_recode == 4 & hispanic_recode == 0 ~ 4,
      # Pacific Islander to Other race, non-Hispanic
      race_recode == 4 & hispanic_recode == 0 ~ 4,
      # Hispanic, any race
      hispanic_recode == 1                    ~ 3
    )
  )
```

```
# A tibble: 10 x 7
      id   age  race hispanic race_recode hispanic_recode race_eth_4cat
   <int> <int> <dbl>    <dbl>       <dbl>           <dbl>         <dbl>
 1     1    15     1        7           1              NA            NA
 2     2    19     2        0           2               0             2
 3     3    14     1        1           1               1             3
 4     4     3     4        0           4               0             4
 5     5    10     7        1          NA               1             3
 6     6    18     1        0           1               0             1
 7     7    22     2        1           2               1             3
 8     8    11     9        9          NA              NA            NA
```

```
 9     9     5     1        0           1               0             1
10    10    NA     3        1           3               1             3
```

The code above works, and it is very explicit. However, we can definitely make it more succinct and easier to read. For example:

```
demographics %>%
  mutate(
    race_eth_4cat = case_when(
      is.na(hispanic) | hispanic %in% c(7, 9) ~ NA_real_, # Unknown ethnicity
      hispanic == 1                           ~ 3,        # Hispanic, any race
      is.na(race) | race %in% c(7, 9)         ~ NA_real_, # non-Hispanic, unknown race
      race == 1                               ~ 1,        # White, non-Hispanic
      race == 2                               ~ 2,        # Black, non-Hispanic
      TRUE                                    ~ 4         # Other race, non-Hispanic
    )
  )
```

```
# A tibble: 10 x 5
      id   age  race hispanic race_eth_4cat
   <int> <int> <dbl>    <dbl>         <dbl>
 1     1    15     1        7            NA
 2     2    19     2        0             2
 3     3    14     1        1             3
 4     4     3     4        0             4
 5     5    10     7        1             3
 6     6    18     1        0             1
 7     7    22     2        1             3
 8     8    11     9        9            NA
 9     9     5     1        0             1
10    10    NA     3        1             3
```

Here's what we did above:

- We used `dplyr`'s `case_when()` function to create a new column in our data frame (`race_eth_4cat`) that categorized each participant into one of 4 race and ethnicity categories depending on their values in the `race` column and the `hispanic` column.
- Compared to the first method we used, the second method doesn't explicitly create new `race` and `hispanic` columns with the 7's and 9's recoded to `NA`. In the second method, those columns aren't needed.

- The very first two-sided formula tells `case_when()` to set the value of `race_eth_4cat` to `NA_real_` when the value of `hispanic` is missing.
  - We put this two-sided formula first because if we don't know a person's Hispanic ethnicity, then we can't put them into any category of `race_eth_4cat`. All categories of `race_eth_4cat` are dependent on a known value for `hispanic`. For example, look at participant number 1. They reported being white, but they don't give their ethnicity. Which category do we put them in? We can't put them in `White, non-Hispanic` because they very well could be Hispanic. We can't put them in `Hispanic, any race` because they very well could be non-Hispanic. We don't know. We never will. We code them as missing and don't evaluate any further. And because `case_when()` is lazy, any other participants with a missing value for `hispanic` would also only have this first two-sided formula evaluated.
  - There were no actual `NA` values in the `hispanic` column, but we put it in the code for completeness. There will be some true missing (`NA`) values in most real-world data sets.
  - Notice that we finally used the `%in%` operator above (`hispanic %in% c(7, 9)`). This is equivalent to typing `hispanic == 7 | hispanic == 9`. Notice that's an *OR*. In this case, it doesn't save us a ton of typing and visual clutter, but in many cases it can.
- The second two-sided formula tells `case_when()` to set the value of `race_eth_4cat` to `3` (i.e., Hispanic any-race) when the value of `hispanic` is `1`. Why did we put this second? If we know that someone is Hispanic, does it matter what race they reported? Nope! No matter what race they reported (even missing race) they get coded as `Hispanic, any race`. And because `case_when()` is lazy, putting this two-sided formula second has two advantages:
  - Any other participants with a value of `1` for `hispanic` would only have the first two two-sided formulas evaluated. In other words, for each Hispanic participant, R would only evaluate 2 two-sided formulas instead of the 6 we used in the first method. With only 10 participants in the data, we won't notice any performance improvement. But, this performance improvement can add up when we have thousands or millions of rows.
  - It allows us to remove the `hispanic == 0` from the remaining two-sided formulas. Think about it. All participants with a missing value for `hispanic` were accounted for in the first two-sided formula. All participants with a `1` for `hispanic` were accounted for in the second two-sided formula. By definition, any participant left in the data must have a value of `0` for `hispanic`. There's no need to write that code and there's no need for R to evaluate that condition. Less typing for us and further performance improvements to boot.

- The third two-sided formula tells `case_when()` to set the value of `race_eth_4cat` to `NA_real_` when the value of `race` is missing. At this point in the code, there are no participants left with a value of `1` for `hispanic`. Therefore, if they are missing a value for `race` we won't be able to assign them a value for `race_eth_4cat`. We code them as missing and don't evaluate any further.
- The fourth and fifth two-sided formulas tell `case_when()` to set the value of `race_eth_4cat` to `1` and `2` respectively when the value of `race` is `1` and `2`.
- The final two-sided formula is simply `TRUE ~ 4`. This tells `case_when()` to set the value of `race_eth_4cat` to `4` when none of the other two-sided formulas above evaluated to `TRUE`. Why did we do this? Well, every participant with missing data has been accounted for, every `Hispanic` participant has been accounted for, every `White, non-Hispanic` participant has been accounted for, and every `Black, non-Hispanic` participant has been accounted for. Because case_when() is lazy, we know that any participant that makes it to this part of the code must fall into the `Other race, non-Hispanic` category.
    - Notice that there is nothing at all about `race` or `hispanic` in this two-sided formula. It just says `TRUE`. What does `case_when()` do when a condition on the left-hand side evaluates to `TRUE`? It returns the value on the right-hand side. In this case `4`.

> **Warning**
>
> Sometimes, adding an a final `TRUE` condition like the one above can be really useful. However, we have to be really careful. We can easily get unintended results if we aren't absolutely sure that we've already accounted for every possible combination of relevant conditions in the two-sided formulas that come before.

Let's go ahead and wrap up this chapter with one consolidated code chunk that cleans our demographics data:

```
demographics %>%
  # Recode variables
  mutate(
    # Collapse continuous age into 3 categories
    age_3cat = case_when(
      age < 12  ~ 1, # child
      age < 18  ~ 2, # adolescent
      age >= 18 ~ 3  # adult
    ),
    age_3cat_f = factor(
      age_3cat,
      labels = c("child", "adolescent", "adult")
    ),
```

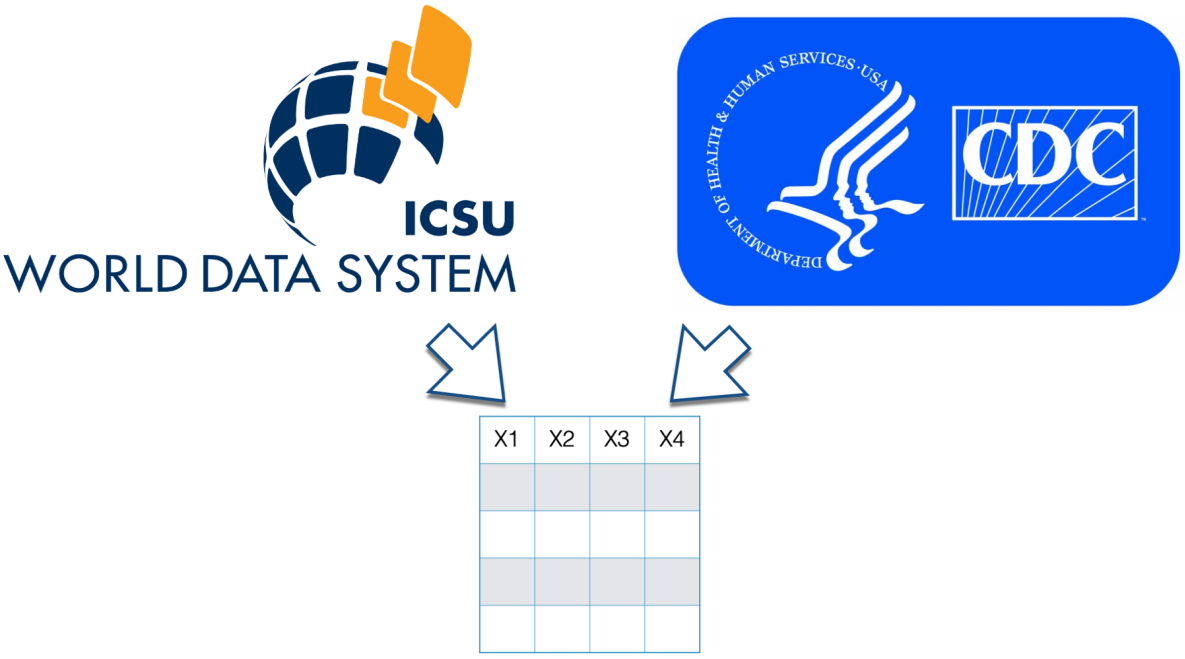
```
    # Combine race and ethnicity
    race_eth_4cat = case_when(
      is.na(hispanic) | hispanic %in% c(7, 9) ~ NA_real_, # Unknown ethnicity
      hispanic == 1                           ~ 3,        # Hispanic, any race
      is.na(race) | race %in% c(7, 9)         ~ NA_real_, # non-Hispanic, unknown race
      race == 1                               ~ 1,        # White, non-Hispanic
      race == 2                               ~ 2,        # Black, non-Hispanic
      TRUE                                    ~ 4         # Other race, non-Hispanic
    ),
    race_eth_4cat_f = factor(
      race_eth_4cat,
      labels = c(
        "White, non-Hispanic", "Black, non-Hispanic", "Hispanic, any race",
        "Other race, non-Hispanic"
      )
    )
  )
```

```
# A tibble: 10 x 8
      id   age  race hispanic age_3cat age_3cat_f race_eth_4cat race_eth_4cat_f
   <int> <int> <dbl>    <dbl>    <dbl> <fct>              <dbl> <fct>
 1     1    15     1        7        2 adolescent            NA <NA>
 2     2    19     2        0        3 adult                  2 Black, non-Hisp~
 3     3    14     1        1        2 adolescent             3 Hispanic, any r~
 4     4     3     4        0        1 child                  4 Other race, non~
 5     5    10     7        1        1 child                  3 Hispanic, any r~
 6     6    18     1        0        3 adult                  1 White, non-Hisp~
 7     7    22     2        1        3 adult                  3 Hispanic, any r~
 8     8    11     9        9        1 child                 NA <NA>
 9     9     5     1        0        1 child                  1 White, non-Hisp~
10    10    NA     3        1       NA <NA>                   3 Hispanic, any r~
```

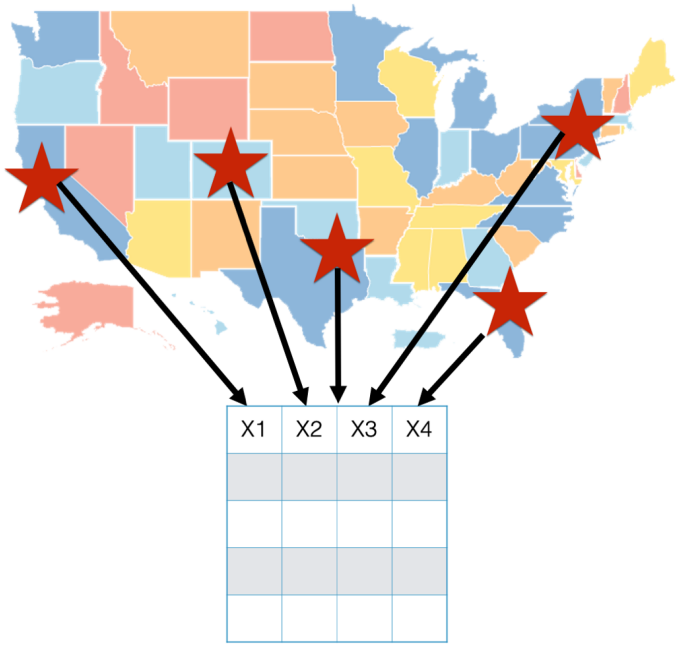
Now that we've mastered conditional operations, we can use them to help us navigate another common data collection technique in epidemiology – skip patterns.

31 Working with Multiple Data Frames

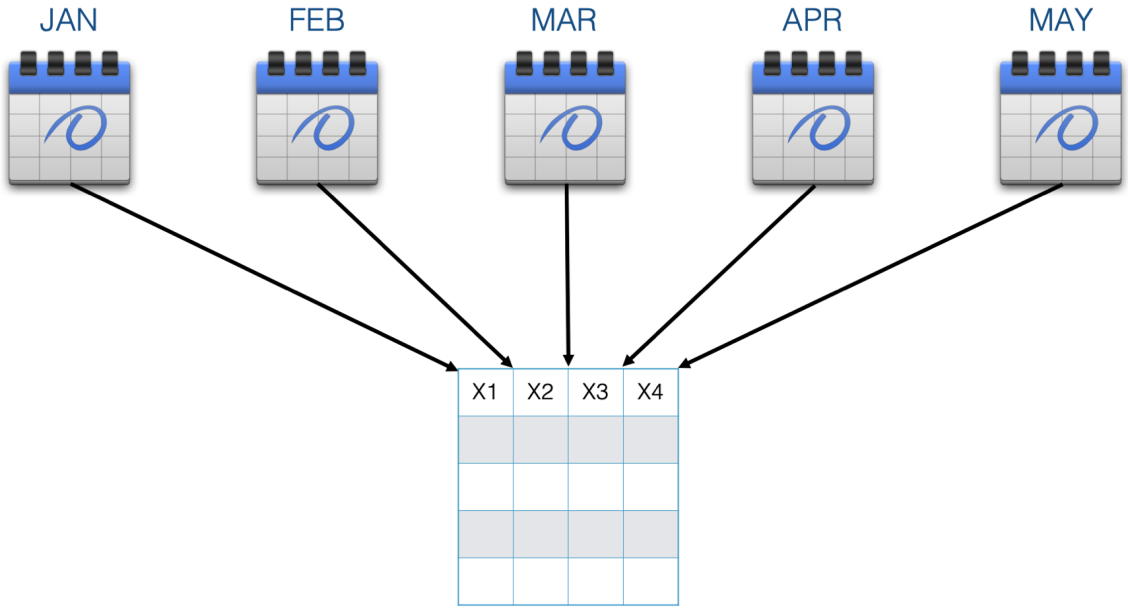
Up to this point, the data we've needed has always been stored in a single data frame. However, that won't always be the case. At times we may need to combine data from multiple agencies in order to complete your analysis.



Additionally, large studies often gather data at multiple sites.

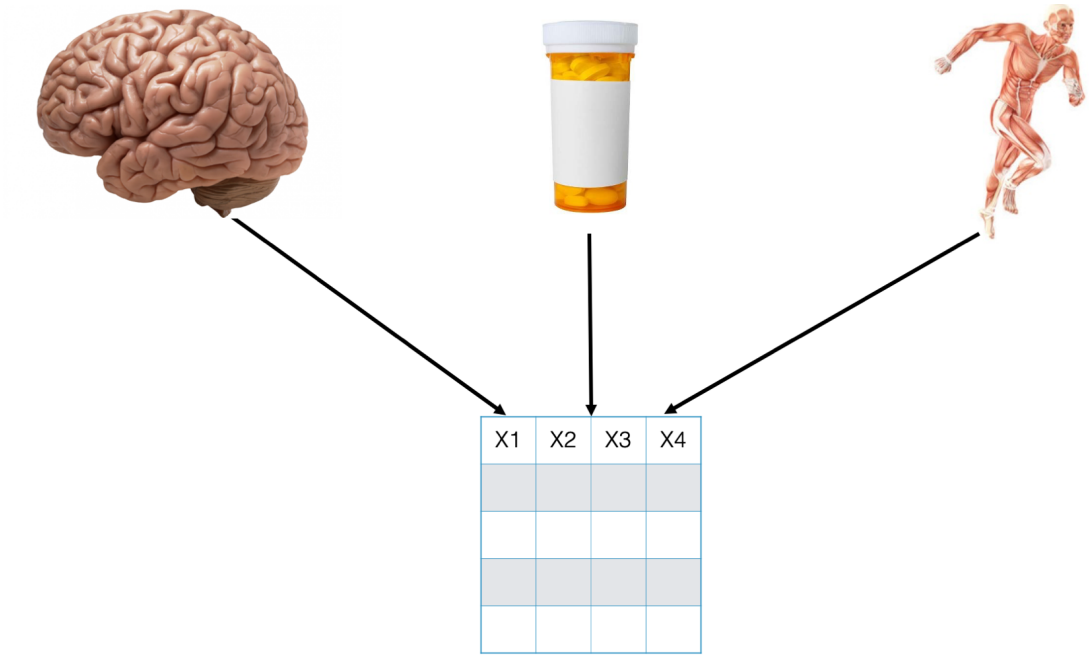


Or, data is sometimes gathered over long periods of time. When this happens, it is not uncommon for observations across the study sites or times to be stored as separate data sets.

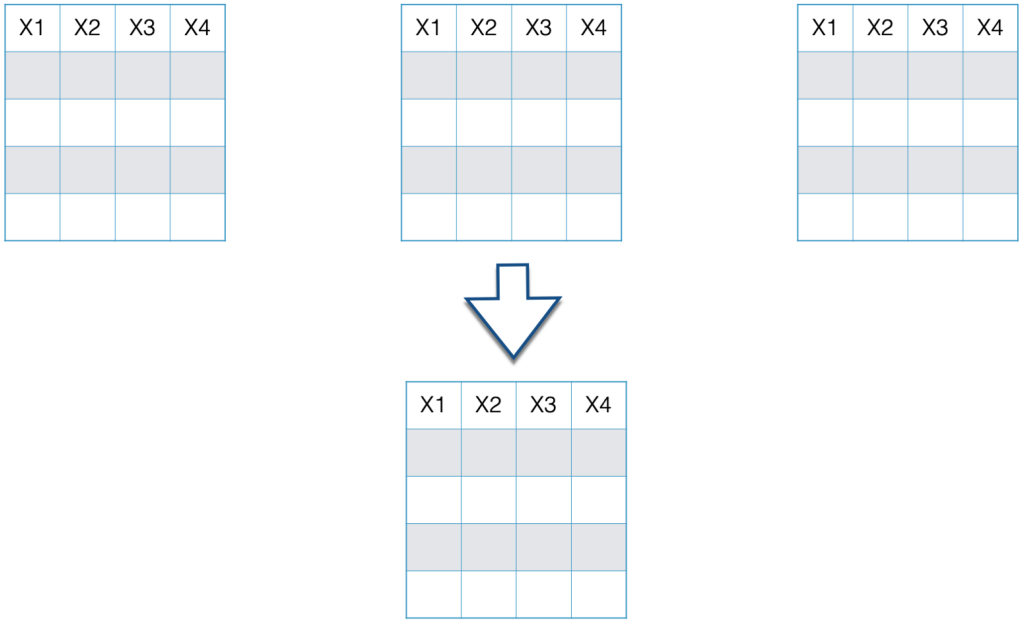


Another common scenario in which you end up with multiple data sets for the same study is

when researchers use different data sets to record the results of different survey instruments or groups of similar instruments.



In any of these cases, you may need to combine data from across data sets in order to complete your analysis.



This combining of data comes in two basic forms: combining vertically and combining horizontally. First we'll learn about combining vertically, or adding rows. Later, we'll learn about combining horizontally, or adding columns.

Below we have two separate data frames - data frame one and data frame two. In this case both data frames contain the exact same variables: Var1, Var2, and Var3. However, they aren't identical because they contain different observations.

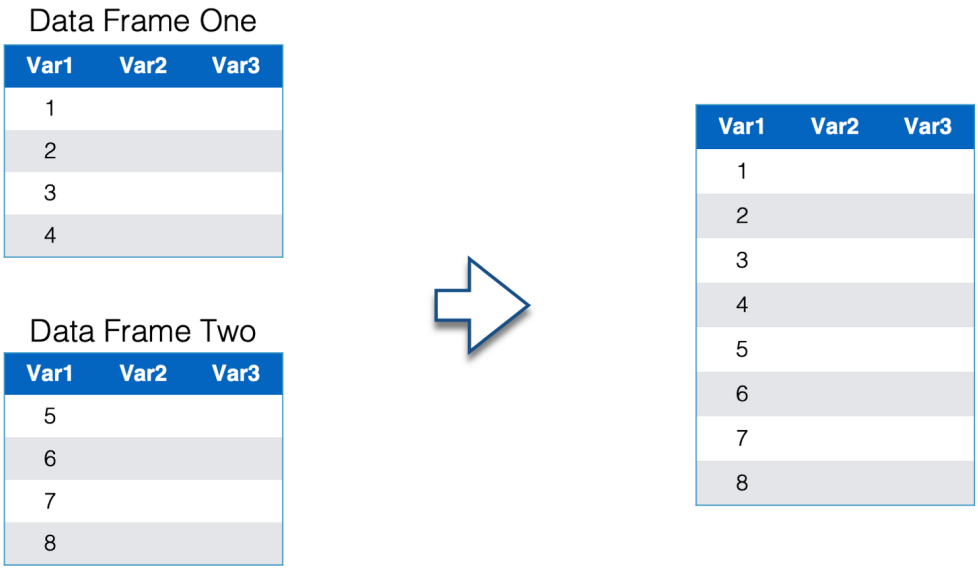
Data Frame One

Var1	Var2	Var3
1		
2		
3		
4		

Data Frame Two

Var1	Var2	Var3
5		
6		
7		
8		

Now, you want to combine these two data frames and end up with one data frame that includes the observations from data frame two listed directly below the observations from data frame one. This is a situation where we want to combine data frames vertically.



When combining data frames vertically, one of the most important questions to ask is, "do the data frames have variables in common?" Just by examining data frame one and data frame two, you can see that the variables have the same names. How can you check to make sure that the variables also contain the same type of data? Well, you can use the `str()` or `glimpse()` functions to compare the details of the columns in the two data frames.

Sometimes, you might find that columns that have different names across data frames contain the same data. For example, suppose that data frame one has a variable named ID and data frame two has a variable named subject ID. In this situation you might want R to combine these two variables when you combine data frames.

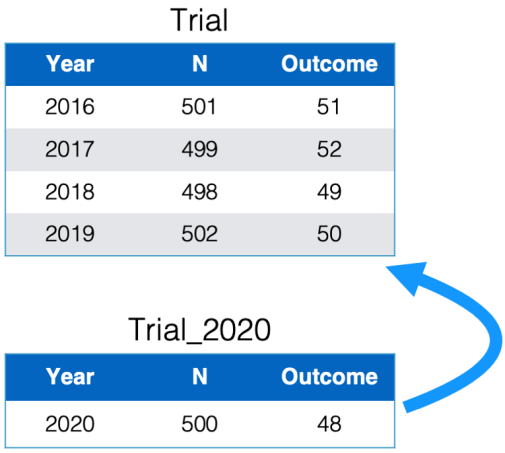
On the other hand, you may find that variables that have the same name across data frames, actually contain different data. For example, both data frames may contain the variable `date`. But, one `date` variable might store birth date and the other might store date of admission. You would not want to combine these two variables.

As you may have guessed, when combining data frames vertically, it's easiest to combine data frames that have identical variables. However, you will also learn how to combine data frames that have different variables.

31.1 Combining data frames vertically: Adding rows

Suppose you are working on a multisite clinical trial recruiting participants over multiple years. You have a data frame named Trial, that stores the number of participants recruited each year,

as well as the number of participants who experienced the outcome of interest. Another data frame named Trial_2020 was just sent to you with the recruitment numbers for the year 2020.



You want to add the observations about the participants recruited in 2020 to the master data frame so that it contains the information about all years. To do this, you bind the rows in the `trial_2020` data frame to the `trial` data frame.

Let's go ahead and load `dplyr`:

```
library(dplyr)
```

And simulate our data frames:

```
trial <- tibble(
  year    = c(2016, 2017, 2018, 2019),
  n       = c(501, 499, 498, 502),
  outcome = c(51, 52, 49, 50)
) %>%
  print()
```

```
# A tibble: 4 x 3
   year     n outcome
```

```
  <dbl> <dbl>   <dbl>
1  2016   501      51
2  2017   499      52
3  2018   498      49
4  2019   502      50
```

```
trial_2020 <- tibble(
  year    = 2020,
  n       = 500,
  outcome = 48
) %>%
  print()
```

```
# A tibble: 1 x 3
   year     n outcome
  <dbl> <dbl>   <dbl>
1  2020   500      48
```

we can see above that column names and types in both data frames are identical. In this case, we can easily bind them together vertically with `dplyr`'s `bind_rows()` function:

```
trial %>%
  bind_rows(trial_2020)
```

```
# A tibble: 5 x 3
   year     n outcome
  <dbl> <dbl>   <dbl>
1  2016   501      51
2  2017   499      52
3  2018   498      49
4  2019   502      50
5  2020   500      48
```

Here's what we did above:

- we used `dplyr`'s `bind_rows()` function to vertically stack, or bind, the rows in `trial_2020` to the rows in `trials`.
- You can type `?bind_rows` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `bind_rows()` function is the `...` argument. Typically, we will pass one or more data frames that we want to combine to the `...` argument.

31.1.1 Combining more than 2 data frames

What if we want to vertically combine more than two data frames? This isn't a problem. Thankfully, `bind_rows()` lets us pass as many data frames as we want to the `...` argument. For example:

```
trial_2021 <- tibble(
  year      = 2021,
  n         = 598,
  outcome   = 57
) %>%
  print()
```

```
# A tibble: 1 x 3
   year     n outcome
  <dbl> <dbl>   <dbl>
1  2021   598      57
```

```
trial %>%
  bind_rows(trial_2020, trial_2021)
```

```
# A tibble: 6 x 3
   year     n outcome
  <dbl> <dbl>   <dbl>
1  2016   501      51
2  2017   499      52
3  2018   498      49
4  2019   502      50
5  2020   500      48
6  2021   598      57
```

31.1.2 Adding rows with differing columns

What happens when the data frames we want to combine don't have identical sets of columns? For example, let's say that we started collecting data on adverse events for the first time in 2020. In this case, `trials_2020` would contain a column that `trials` doesn't contain. Can we still row bind our two data frames? Let's see:

```
trial_2020 <- tibble(
  year      = 2020,
  n         = 500,
  outcome   = 48,
  adv_event = 3 # Here is the new column
) %>%
  print()
```

```
# A tibble: 1 x 4
   year     n outcome adv_event
  <dbl> <dbl>   <dbl>     <dbl>
1  2020   500      48         3
```

```
trial %>%
  bind_rows(trial_2020)
```

```
# A tibble: 5 x 4
   year     n outcome adv_event
  <dbl> <dbl>   <dbl>     <dbl>
1  2016   501      51        NA
2  2017   499      52        NA
3  2018   498      49        NA
4  2019   502      50        NA
5  2020   500      48         3
```

we sure can! R just sets the value of `adv_event` to `NA` in the rows that came from the `trial` data frame.

31.1.3 Differing column positions

Next, let's say that the person doing data entry accidently put the columns in a different order in 2020. Is `bind_rows()` able to figure out which columns go together?

```
trial_2020 <- tibble(
  year      = 2020,
  n         = 500,
  adv_event = 3, # This was previously the fourth column
  outcome   = 48 # This is the thrid column in trial
) %>%
  print()
```

```
# A tibble: 1 x 4
   year     n adv_event outcome
  <dbl> <dbl>     <dbl>   <dbl>
1  2020   500         3      48
```

```
trial %>%
  bind_rows(trial_2020)
```

```
# A tibble: 5 x 4
   year     n outcome adv_event
  <dbl> <dbl>   <dbl>     <dbl>
1  2016   501      51        NA
2  2017   499      52        NA
3  2018   498      49        NA
4  2019   502      50        NA
5  2020   500      48         3
```

Yes! The `bind_rows()` function binds the data frames together based on column names. So, having our columns in a different order in the two data frames isn't a problem. But, what happens when we have different column names?

31.1.4 Differing column names

As a final wrinkle, let's say that the person doing data entry started using different column names in 2020 as well. For example, below, the `n` column is now named `count` and the `outcome` column is now named `outcomes`. Will `bind_rows()` still be able to vertically combine these data frames?

```
trial_2020 <- tibble(
  year      = 2020,
  count     = 500,
  adv_event = 3,
  outcomes  = 48
) %>%
  print()
```

```
# A tibble: 1 x 4
   year count adv_event outcomes
  <dbl> <dbl>     <dbl>    <dbl>
1  2020   500         3       48
```

```
trial %>%
  bind_rows(trial_2020)
```

```
# A tibble: 5 x 6
   year     n outcome count adv_event outcomes
  <dbl> <dbl>   <dbl> <dbl>     <dbl>    <dbl>
1  2016   501      51    NA        NA       NA
2  2017   499      52    NA        NA       NA
3  2018   498      49    NA        NA       NA
4  2019   502      50    NA        NA       NA
5  2020    NA      NA   500         3       48
```

In this case, `bind_rows()` plays it safe and doesn't make any assumptions about whether columns with different names belong together or not. However, we only need to rename the columns in one data frame or the other to fix this problem. We could do this in separate steps like this:

```
trial_2020_rename <- trial_2020 %>%
  rename(
    n = count,
    outcome = outcomes
  )

trial %>%
  bind_rows(trial_2020_rename)
```

```
# A tibble: 5 x 4
   year     n outcome adv_event
  <dbl> <dbl>   <dbl>     <dbl>
1  2016   501      51        NA
2  2017   499      52        NA
3  2018   498      49        NA
4  2019   502      50        NA
5  2020   500      48         3
```

Or, we could rename and bind in a single step by nesting functions like this:

```
trial %>%
  bind_rows(
    trial_2020 %>%
```

```
    rename(
      n = count,
      outcome = outcomes
    )
)
```

```
# A tibble: 5 x 4
   year     n outcome adv_event
  <dbl> <dbl>   <dbl>     <dbl>
1  2016   501      51        NA
2  2017   499      52        NA
3  2018   498      49        NA
4  2019   502      50        NA
5  2020   500      48         3
```

Here's what we did above:

- we *nested* the code that we previously used to create the `trial_2020_rename` data frame inside of the `bind_rows()` function instead creating the actual `trial_2020_rename` data frame and passing it to `bind_rows()`.
- I don't think you can really say that one method is "better" or "worse". The first method requires two steps and creates a data frame in our global environment that we may or may not ever need again (i.e., potentially just clutter). However, one could make an argument that the first method is also easier to glance at and read. I would typically use the second method, but this is really just a personal preference in this case.

And that's pretty much it. The `bind_rows()` function makes it really easy to combine R data frames vertically. Next, let's learn how to combine data frames horizontally.

31.2 Combining data frames horizontally: Adding columns

In this section we will once again begin with two separate data frames - data frame one and data frame two. But, unlike before, these data frames share only one variable in common. And, the data contained in both data frames pertains to the same observations.

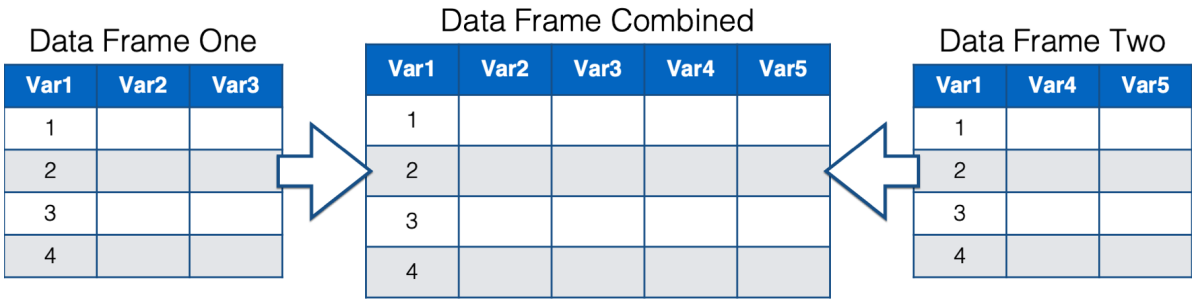
Data Frame One

Var1	Var2	Var3
1		
2		
3		
4		

Data Frame Two

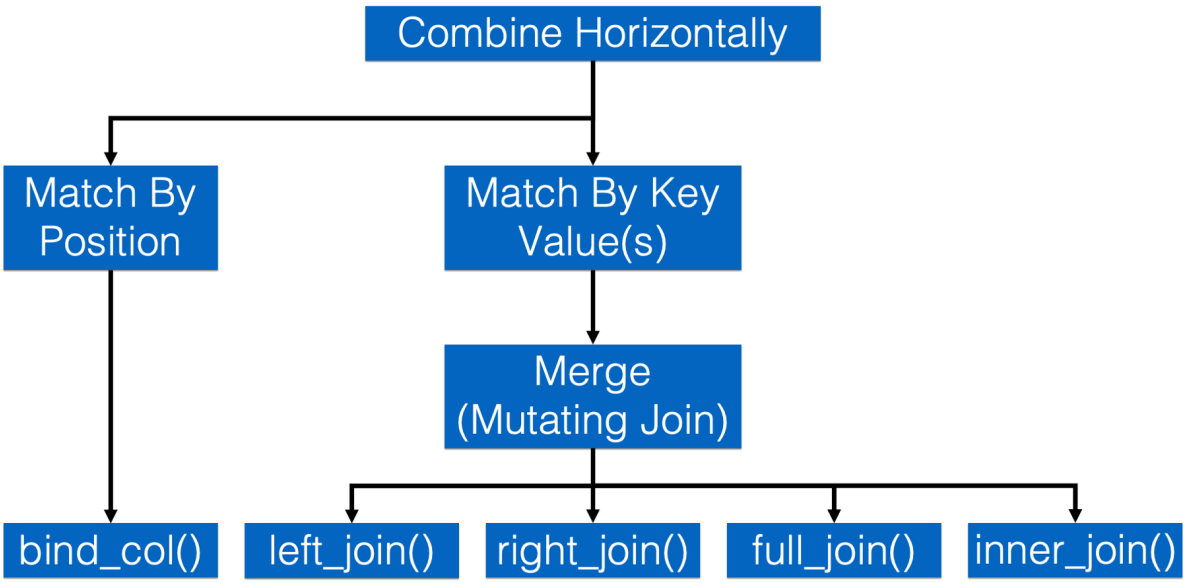
Var1	Var4	Var5
1		
2		
3		
4		

Our goal is once again to combine these data frames. But, this time we want to combine them horizontally. In other words, we want a combined data frame that combines all the *columns* from data frame one and data frame two.



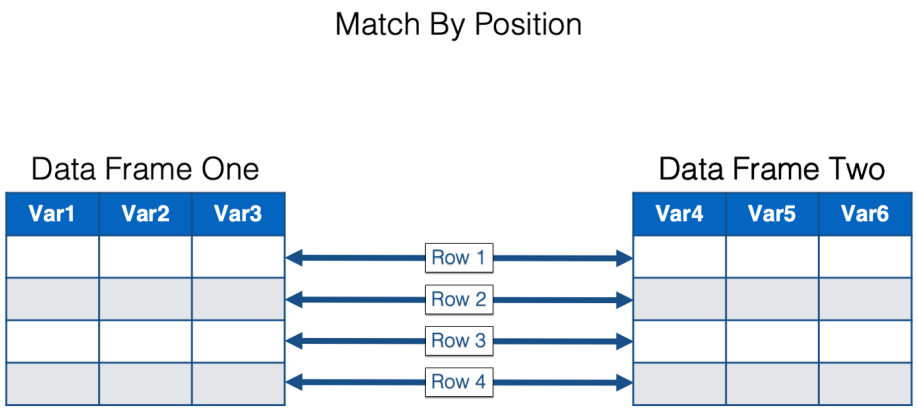
Combining data frames horizontally can be slightly more complicated than combining them

vertically. As shown in the following flow chart, we can either match the rows of our two data frames up by position or by key values.



31.2.1 Combining data frames horizontally by position

In the simplest case, we match the rows in our data frames up by position. In other words, row 1 in data frame one is matched up with row 1 in data frame two, row 2 in data frame one is matched up with row 2 in data frame two, and so on. Row n (meaning, any number) in data frame one always gets matched to row n in data frame two, regardless of the values in any column of those rows.



Combining data frames horizontally by position is very easy in R. We just use `dplyr`'s `bind_cols()` function similarly to the way used `bind_rows()` above. Just remember that when we horizontally combine data frames by position both data frames must have the same number of rows. For example:

```
df1 <- tibble(
  color = c("red", "green", "blue"),
  size  = c("small", "medium", "large")
) %>%
  print()
```

```
# A tibble: 3 x 2
  color size
  <chr> <chr>
1 red   small
2 green medium
3 blue  large
```

```
df2 <- tibble(
  amount = c(1, 4, 3),
  dose   = c(10, 20, 30)
) %>%
  print()
```

```
# A tibble: 3 x 2
  amount  dose
   <dbl> <dbl>
1      1    10
2      4    20
3      3    30
```

```
df1 %>%
  bind_cols(df2)
```

```
# A tibble: 3 x 4
  color size   amount  dose
  <chr> <chr>   <dbl> <dbl>
1 red   small       1    10
2 green medium      4    20
3 blue  large       3    30
```

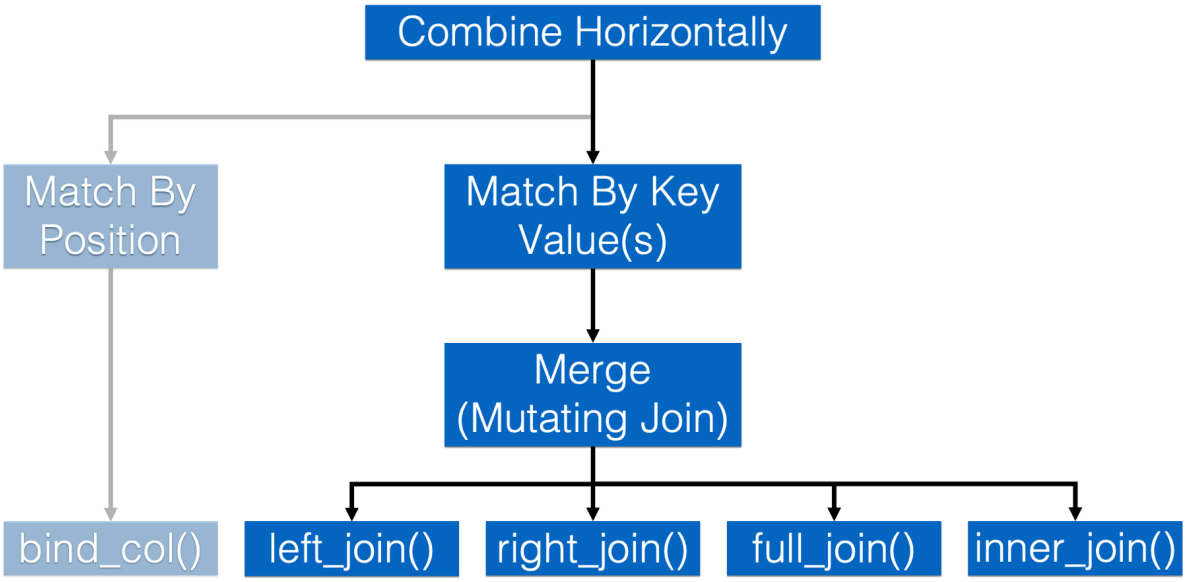
Here's what we did above:

- we used `dplyr`'s `bind_cols()` function to horizontally bind the columns in `df1` to the columns in `df2`.
- You can type `?bind_cols` into your R console to view the help documentation for this function and follow along with the explanation below.
- The only argument to the `bind_cols()` function is the `...` argument. Typically, we will pass one or more data frames that we want to combine to the `...` argument.

In general, it's a bad idea to combine data frames that contain different kinds of information (i.e., variables) about the same set of people (or places or things) in this way. It's difficult to ensure that the information in row n in both data frames is really about the same person (or place or thing). However, we do sometimes find `bind_cols()` to be useful when we're writing our own functions in R. We haven't quite learned how to do that yet, but we will soon.

31.2.2 Combining data frames horizontally by key values

In all the examples from here on out we will match the rows of our data frames by one or more key values.

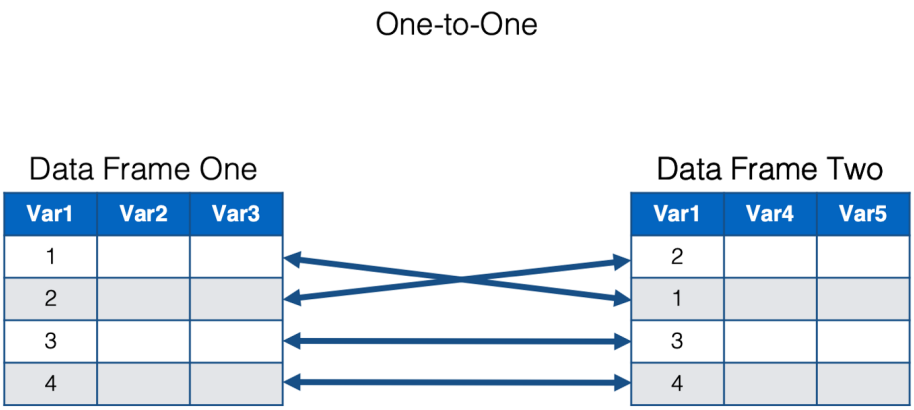


In epidemiology, the term I most often hear used for combining data frames in this way is **merging**. So, I will mostly use that term below. However, in other disciplines it is common to use the term **joining**, or performing a data **join**, to mean the same thing. The `dplyr` package, in specific, refers to these as "mutating joins."

31.2.2.1 Relationship types

When we merge data frames it's important to ask ourselves, "what is the relationship between the observations in the original data frames?" The observations can be related in several different ways.

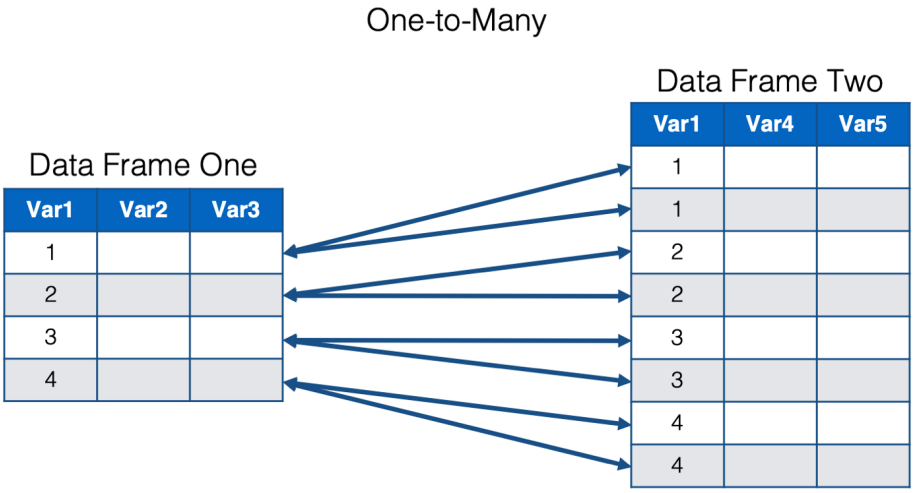
In a one-to-one relationship, a single observation in one data frame is related to no more than one observation in the other data frame. We know how to align, or connect, the rows in the two data frames based on the values of one or more common variables.



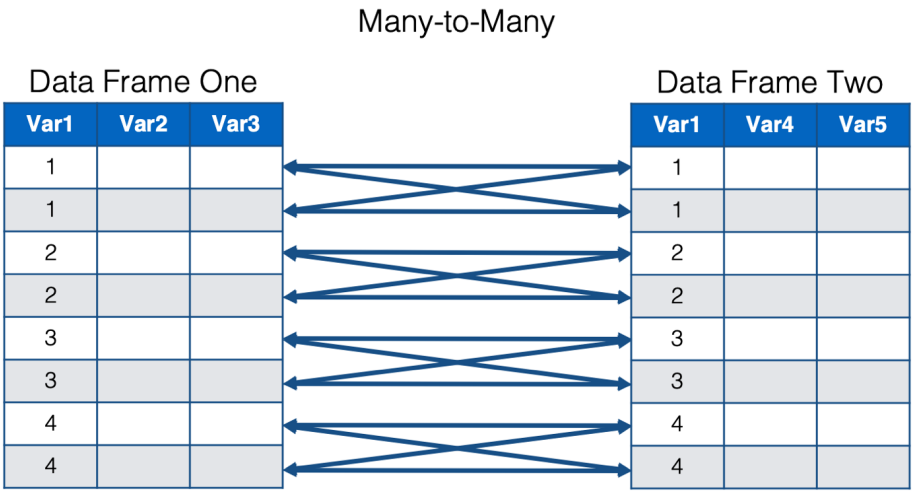
This common variable, or set of common variables, is also called a **key**. When we use the values in the key to match rows in our data frames, we can say that we are *matching on key values*.

In the example above, There is one key column – `Var1`. Both data frames contain the column named `Var1`, and the values of that column tell R how to align the rows in both data frames so that all the values in that row contain data are about the same person, place, or thing. In the example above, we know that the first row of data frame one goes with the *second* row of data frame two because both rows have the same key value – `1`.

In a one-to-many relationship, a single observation in one data frame is related to multiple observations in the other data frame.



And finally, in a many-to-many relationship, multiple observations in one data frame are related to multiple observations in the other data frame.



Many-to-many relationships are messy and are generally best avoided, if possible. In practice, we're not sure that we've ever merged two data frames that had a *true* many-to-many

relationship. We emphasize *true* because we have definitely merged data frames that had a many-to-many relationship when matching on a single key column. However, after matching on multiple key columns (e.g., study id and date instead of just study id), the relationship became one-to-one or one-to-many. We'll see an example of matching on multiple key columns later.

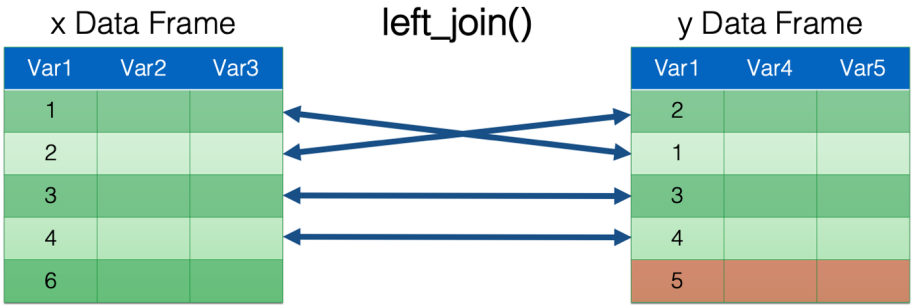
31.2.2.2 dplyr join types

In this chapter, we will merge data frames using one of `dplyr`'s four mutating join functions.

The first three arguments to all four of `dplyr`'s mutating join functions are: `x`, `y`, and `by`. You should pass the names of the data frames you want to merge to the `x` and `y` arguments respectively. You should pass the name(s) of the key column(s) to the `by` argument. In many cases, you will get a different merge result depending on which data frame you pass to the `x` and `y` arguments, and which mutating join function you use. Below, we will give you a brief overview of each of the mutating join functions, and then we will jump into some examples.

The four mutating join functions are:

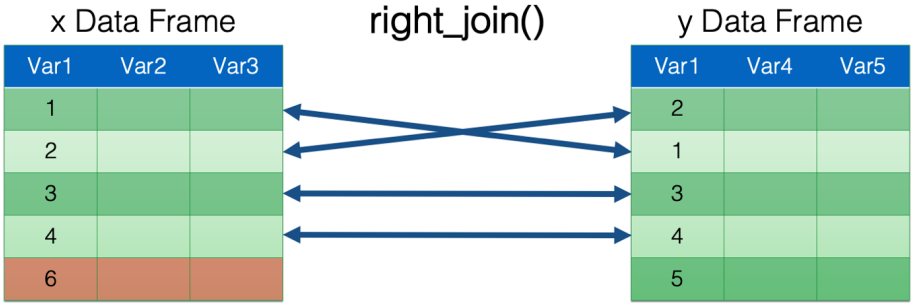
1. `left_join()`. This is probably the join function that you will use the most. It's important to remember that `left_join()` keeps all the rows from the `x` data frame in the resulting combined data frame. However, it only keeps the rows from the `y` data frame that have a key value match in the `x` data frame. The values for columns with no key value match in the opposite data frame are set to `NA`.



Result

Var1	Var2	Var3	Var4	Var5
1				
2				
3				
4				
6			NA	NA

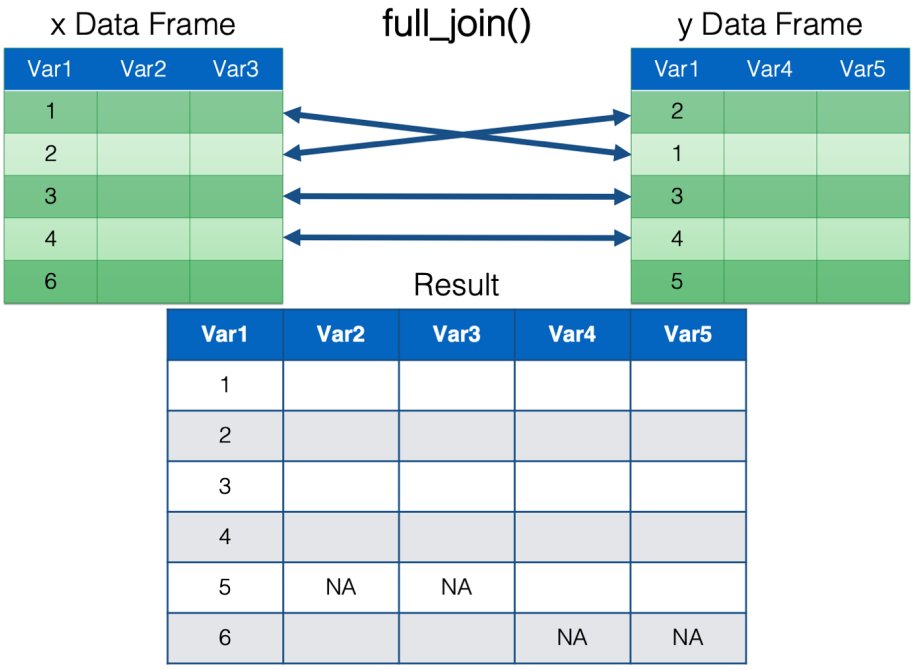
2. `right_join()`. This is just the mirror opposite of `left_join()`. Accordingly, `right_join()` keeps all the rows from the `y` data frame in the resulting combined data frame, and only keep the rows from the `x` data frame that have a key value match in the `y` data frame. The values for columns with no key value match in the opposite data frame are set to `NA`.



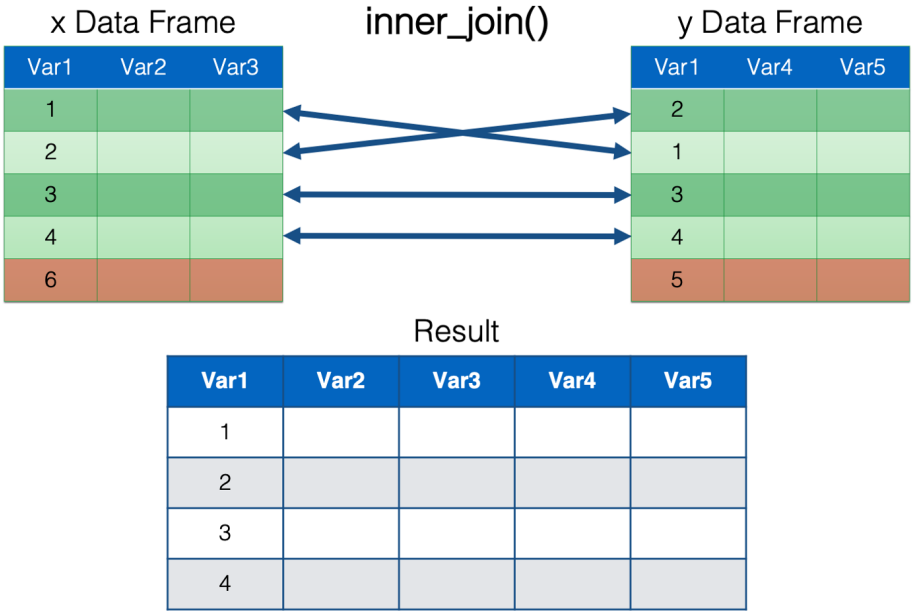
Result

Var1	Var2	Var3	Var4	Var5
1				
2				
3				
4				
5	NA	NA		

3. `full_join()`. Full join keeps all the rows from both data frames in the resulting combined data frame. The values for columns with no key value match in the opposite data frame are set to `NA`.



4. `inner_join()`. Inner join keeps only the rows from both data frames that have a key value match in the opposite data frame in the resulting combined data frame.



Now that we have a common vocabulary, let's take a look at some more concrete examples.

Suppose we are analyzing data from a study of aging and functional ability. At baseline, we assigned a study id to each of our participants. We then ask them their date of birth and their race and ethnicity. We saved that information in a data frame called `demographics`.

```
demographics <- tibble(
  id       = c("1001", "1002", "1003", "1004"),
  dob      = as.Date(c("1968-12-14", "1952-08-03", "1949-05-27", "1955-03-12")),
  race_eth = c(1, 2, 2, 4)
) %>%
  print()
```

```
# A tibble: 4 x 3
  id    dob        race_eth
  <chr> <date>        <dbl>
1 1001  1968-12-14        1
2 1002  1952-08-03        2
3 1003  1949-05-27        2
4 1004  1955-03-12        4
```

Then, we asked our participants to do a series of functional tests. The functional tests included measuring grip strength in their right hand (`grip_r`) and grip strength in their left hand (`grip_l`). We saved each measure, along with their study id, in a separate data frame called `grip_strength`.

```
grip_strength <- tibble(
  id     = c("1002", "1001", "1003", "1004"),
  grip_r = c(32, 28, 32, 22),
  grip_l = c(30, 30, 28, 22)
) %>%
  print()
```

```
# A tibble: 4 x 3
  id    grip_r grip_l
  <chr>  <dbl>  <dbl>
1 1002      32     30
2 1001      28     30
3 1003      32     28
4 1004      22     22
```

Now, we want to merge these two data frames together so that we can include age, race/ethnicity, and grip strength in our analysis.

Let's first ask ourselves, "what is the relationship between the observations in `demographics` and the observations in `grip_strength`?"

31.2.2.3 One-to-one relationship merge

It's a one-to-one relationship because each participant in `demographics` has no more than one corresponding row in `grip_strength`. Since both data frames have exactly four rows, we can go ahead hand combine them horizontally using `bind_cols()` like this:

```
demographics %>%
  bind_cols(grip_strength)
```

```
New names:
* `id` -> `id...1`
* `id` -> `id...4`
```

```
# A tibble: 4 x 6
  id...1 dob        race_eth id...4 grip_r grip_l
  <chr>  <date>        <dbl> <chr>   <dbl>  <dbl>
1 1001   1968-12-14        1 1002       32     30
2 1002   1952-08-03        2 1001       28     30
3 1003   1949-05-27        2 1003       32     28
4 1004   1955-03-12        4 1004       22     22
```

Here's what we did above:

- we used `dplyr`'s `bind_cols()` function to horizontally bind the columns in `demographics` to the columns in `grip_strength`. **This was a bad idea!**
- Notice the message that `bind_cols()` gave us this time: `New names: * id -> id...1 * id -> id...2`. This is telling us that both data frames had a column named `id`. If `bind_cols()` had left the column names as-is, then the resulting combined data frame would have had two columns named `id`, which isn't allowed.
- **More importantly**, notice the demographic data for participant 1001 is now aligned with the grip strength data for participant 1002, and vice versa. The grip strength data was recorded in the order that participants came in to have their grip strength measured. In this case, participant 1002 came in before 1001. Remember that `bind_cols()` matches rows by position, which results in mismatched data in this case.

Now, let's learn a better way to merge these two data frames – `dplyr`'s `left_join()` function:

```
demographics %>%
  left_join(grip_strength, by = "id")
```

```
# A tibble: 4 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
```

Here's what we did above:

- we used `dplyr`'s `left_join()` function to perform a one-to-one merge of the `demographics` data frame with the `grip_strength` data frame.
- You can type `?left_join` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `left_join()` function is the `x` argument. You should pass a data frame to the `x` argument.
- The second argument to the `left_join()` function is the `y` argument. You should pass a data frame to the `y` argument.
- The third argument to the `left_join()` function is the `by` argument. You should pass the name of the column, or columns, that contain the key values. The column name should be wrapped in quotes.
- Notice that the demographics and grip strength data are now correctly aligned for participants 1001 and 1002 even though they were still misaligned in the original data frames. That's because row position is irrelevant when we match by key values.
- Notice that the result above only includes a single `id` column. This is because we aren't simply smooshing two data frames together, side-by-side. We are integrating information from across the two data frames based on the value of the key column – `id`.

The merge we did above is about as simple as it gets. It was a one-to-one merge where every key value in the `x` data frame had one, and only one, matching key value in the `y` data frame. Therefore, in this simple case, all four join types give us the same result:

```r
# Right join
demographics %>%
  right_join(grip_strength, by = "id")
```

```
# A tibble: 4 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
```

```r
# Full join
demographics %>%
  full_join(grip_strength, by = "id")
```

```
# A tibble: 4 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
```

```r
# Inner join
demographics %>%
  inner_join(grip_strength, by = "id")
```

```
# A tibble: 4 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
```

Additionally, aside from the order of the rows and columns in the resulting combined data frame, it makes no difference which data frame you pass to the `x` and `y` arguments in this case:

```
# Switching order
grip_strength %>%
  left_join(demographics, by = "id")
```

```
# A tibble: 4 x 5
  id    grip_r grip_l dob        race_eth
  <chr>  <dbl>  <dbl> <date>        <dbl>
1 1002      32     30 1952-08-03        2
2 1001      28     30 1968-12-14        1
3 1003      32     28 1949-05-27        2
4 1004      22     22 1955-03-12        4
```

As our merges get more complex, we will get different results depending on which join function we choose and the ordering in which we pass our data frames to the `x` and `y` arguments. We're not going to attempt to cover every possible combination. But, we are going to try to give you a flavor for some of the scenarios we believe you are most likely to encounter in practice.

31.2.2.4 Differing rows

In the real world, participants don't always attend scheduled visits. Let's suppose that there was actually a fifth participant that we collected baseline data from:

```
demographics <- tibble(
  id       = c("1001", "1002", "1003", "1004", "1005"),
  dob      = as.Date(c(
    "1968-12-14", "1952-08-03", "1949-05-27", "1955-03-12", "1942-06-07"
  )),
  race_eth = c(1, 2, 2, 4, 3)
) %>%
  print()
```

```
# A tibble: 5 x 3
  id    dob        race_eth
  <chr> <date>        <dbl>
1 1001  1968-12-14        1
2 1002  1952-08-03        2
3 1003  1949-05-27        2
4 1004  1955-03-12        4
5 1005  1942-06-07        3
```

However, participant 1005 never made it back in for a grip strength test. Now, what do you think will happen when we merge `demographics` and `grip_strength` using `left_join()`?

```
demographics %>%
  left_join(grip_strength, by = "id")
```

```
# A tibble: 5 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
5 1005  1942-06-07        3     NA     NA
```

The resulting data frame includes *all* rows from the `demographics` data frame *and all* the rows from the `grip_strength` data frame. Because participant 1005 never had their grip strength measured, and therefore, had no rows in the `grip_strength` data frame, their values for `grip_r` and `grip_l` are set to missing.

This scenario is a little a different than the one above. It's still a one-to-one relationship because each participant in `demographics` has no more than one corresponding row in `grip_strength`. However, every key value in the `x` data frame no longer has one, and only one, matching key value in the `y` data frame. Therefore, we will now get different results depending on which join function we choose, and the order in which we pass our data frames to the `x` and `y` arguments. Before reading further, think about what you expect the results from each join function to look like. Think about what you expect the results of switching the data frame order to look like.

```
# Right join
demographics %>%
  right_join(grip_strength, by = "id")
```

```
# A tibble: 4 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
```

```r
# Full join
demographics %>%
  full_join(grip_strength, by = "id")
```

```
# A tibble: 5 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
5 1005  1942-06-07        3     NA     NA
```

```r
# Inner join
demographics %>%
  inner_join(grip_strength, by = "id")
```

```
# A tibble: 4 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
```

```r
# Switching order
grip_strength %>%
  left_join(demographics, by = "id")
```

```
# A tibble: 4 x 5
  id    grip_r grip_l dob        race_eth
  <chr>  <dbl>  <dbl> <date>        <dbl>
1 1002      32     30 1952-08-03        2
2 1001      28     30 1968-12-14        1
3 1003      32     28 1949-05-27        2
4 1004      22     22 1955-03-12        4
```

Well, were those the results you expected? In practice, the "correct" result depends on what we are trying to do. In the scenario above, we would probably tend to want the result from `left_join()` or `full_join()` in most cases. The reason is that it's much harder to add data into our analysis that never made it into our combined data frame than it is to drop rows from our results data frame that we don't need for our analysis.

31.2.2.5 Differing key column names

Sometimes the key columns will have different names across data frames. For example, let's imagine that the team collecting the grip strength data named the participant id column `pid` instead of `id`:

```
grip_strength <- tibble(
  pid    = c("1002", "1001", "1003", "1004"),
  grip_r = c(32, 28, 32, 22),
  grip_l = c(30, 30, 28, 22)
) %>%
  print()
```

```
# A tibble: 4 x 3
  pid   grip_r grip_l
  <chr>  <dbl>  <dbl>
1 1002      32     30
2 1001      28     30
3 1003      32     28
4 1004      22     22
```

If we try to merge `demographics` and `grip_strength` as we did before, we will get an error.

```
demographics %>%
  left_join(grip_strength, by = "id")
```

```
Error in `left_join()`:
! Join columns in `y` must be present in the data.
x Problem with `id`.
```

This error is `left_join()` telling us that it couldn't find a column named `id` in both data frames. To get around this error, we can simply tell `left_join()` which column is the matching key column in the opposite data frame using a named vector like this:

```
demographics %>%
  left_join(grip_strength, by = c("id" = "pid"))
```

```
# A tibble: 5 x 5
  id    dob        race_eth grip_r grip_l
  <chr> <date>        <dbl>  <dbl>  <dbl>
```

```
1 1001  1968-12-14        1     28     30
2 1002  1952-08-03        2     32     30
3 1003  1949-05-27        2     32     28
4 1004  1955-03-12        4     22     22
5 1005  1942-06-07        3     NA     NA
```

Just make sure that the first column name you pass to the named vector (i.e., `"id"`) is the name of the key column in the `x` data frame and that the second column name you pass to the named vector (i.e., `"pid"`) is the name of the key column in the `y` data frame.

31.2.2.6 One-to-many relationship merge

Now suppose that our grip strength study has a longitudinal design. The demographics data was only collected at enrollment into the study. After all, race and dob don't change. There's no need to ask our participants about them at every follow-up interview.

```
demographics
```

```
# A tibble: 5 x 3
  id    dob        race_eth
  <chr> <date>        <dbl>
1 1001  1968-12-14        1
2 1002  1952-08-03        2
3 1003  1949-05-27        2
4 1004  1955-03-12        4
5 1005  1942-06-07        3
```

Grip strength, however, was measured pre and post some intervention.

```
grip_strength <- tibble(
  id     = rep(c("1001", "1002", "1003", "1004"), each = 2),
  visit  = rep(c("pre", "post"), 4),
  grip_r = c(32, 33, 28, 27, 32, 34, 22, 27),
  grip_l = c(30, 32, 30, 30, 28, 30, 22, 26)
) %>%
  print()
```

```
# A tibble: 8 x 4
  id    visit grip_r grip_l
  <chr> <chr>  <dbl>  <dbl>
```

```
1 1001  pre       32     30
2 1001  post      33     32
3 1002  pre       28     30
4 1002  post      27     30
5 1003  pre       32     28
6 1003  post      34     30
7 1004  pre       22     22
8 1004  post      27     26
```

Now what is the relationship of these two data frames?

These data frames have a one-to-many relationship because at least one observation in one data frame is related to multiple observations in the other data frame. The `demographics` data frame has one observation for each value of `id`. The `grip_strength` data frame has two observations for each value of the `id`'s `1001` through `1004`.

Now, to conduct our analysis, we need to combine the data in `demographics` with the data in the longitudinal `grip_strength` data frame. And how will we ask R to merge these two data frames? Well, here is some good news. To perform a one-to-many or many-to-many merge, we use the exact same syntax that we used to perform a one-to-one merge. R will figure out the relationship between the data frames automatically. Take a look:

```
demographics %>%
  left_join(grip_strength, by = "id")
```

```
# A tibble: 9 x 6
  id    dob        race_eth visit grip_r grip_l
  <chr> <date>        <dbl> <chr>  <dbl>  <dbl>
1 1001  1968-12-14        1 pre       32     30
2 1001  1968-12-14        1 post      33     32
3 1002  1952-08-03        2 pre       28     30
4 1002  1952-08-03        2 post      27     30
5 1003  1949-05-27        2 pre       32     28
6 1003  1949-05-27        2 post      34     30
7 1004  1955-03-12        4 pre       22     22
8 1004  1955-03-12        4 post      27     26
9 1005  1942-06-07        3 <NA>      NA     NA
```

31.2.2.7 Multiple key columns

Let's throw one more little wrinkle into our analysis. Let's say that each participant had a medical exam prior to being sent into the gym to do their functional assessments. The results

of that medical exam, along with the participant's study id, were recorded in the university hospital system's electronic medical records. As part of that medical exam, each participant's weight was recorded. Luckily, we were given access to the electronic medical records, which look like this:

```
emr <- tibble(
  id     = rep(c("1001", "1002", "1003", "1004"), each = 2),
  visit  = rep(c("pre", "post"), 4),
  weight = c(105, 99, 200, 201, 136, 133, 170, 175)
) %>%
  print()
```

```
# A tibble: 8 x 3
  id    visit weight
  <chr> <chr>  <dbl>
1 1001  pre      105
2 1001  post      99
3 1002  pre      200
4 1002  post     201
5 1003  pre      136
6 1003  post     133
7 1004  pre      170
8 1004  post     175
```

Now, we would like to add participant weight to our analysis. Our first attempt might look something like this:

```
demographics %>%
  left_join(grip_strength, emr, by = "id")
```

```
# A tibble: 9 x 6
  id    dob        race_eth visit grip_r grip_l
  <chr> <date>        <dbl> <chr>  <dbl>  <dbl>
1 1001  1968-12-14        1 pre       32     30
2 1001  1968-12-14        1 post      33     32
3 1002  1952-08-03        2 pre       28     30
4 1002  1952-08-03        2 post      27     30
5 1003  1949-05-27        2 pre       32     28
6 1003  1949-05-27        2 post      34     30
7 1004  1955-03-12        4 pre       22     22
8 1004  1955-03-12        4 post      27     26
9 1005  1942-06-07        3 <NA>      NA     NA
```

Of course, that doesn't work because `left_join()` can only merge two data frames at a time – `x` and `y`. The `emr` data frame was ignored. Then we think, "hmmm, maybe we should try merging them sequentially." In other words, merge `demographics` and `grip_strength` first. Then merge the combined `demographics/grip_strength` data frame with `emr`. So, our next attempt might look like this:

```
demographics %>%
  left_join(grip_strength, by = "id") %>%
  left_join(emr, by = "id")
```

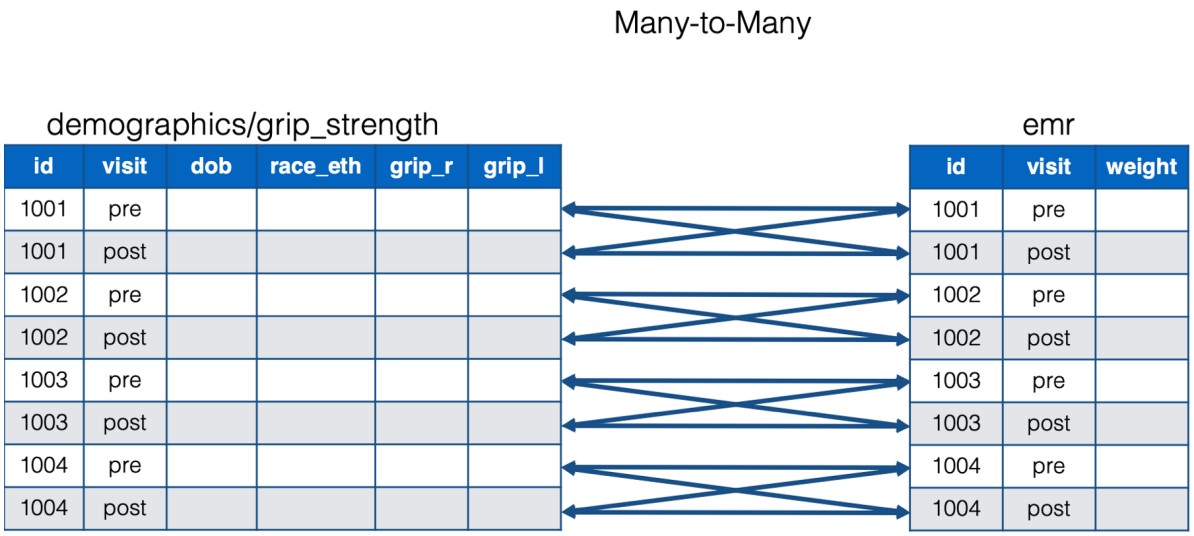
```
Warning in left_join(., emr, by = "id"): Detected an unexpected many-to-many relationship be
i Row 1 of `x` matches multiple rows in `y`.
i Row 1 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
  "many-to-many"` to silence this warning.
```

```
# A tibble: 17 x 8
   id    dob        race_eth visit.x grip_r grip_l visit.y weight
   <chr> <date>        <dbl> <chr>    <dbl>  <dbl> <chr>    <dbl>
 1 1001  1968-12-14       1 pre         32     30 pre        105
 2 1001  1968-12-14       1 pre         32     30 post        99
 3 1001  1968-12-14       1 post        33     32 pre        105
 4 1001  1968-12-14       1 post        33     32 post        99
 5 1002  1952-08-03       2 pre         28     30 pre        200
 6 1002  1952-08-03       2 pre         28     30 post       201
 7 1002  1952-08-03       2 post        27     30 pre        200
 8 1002  1952-08-03       2 post        27     30 post       201
 9 1003  1949-05-27       2 pre         32     28 pre        136
10 1003  1949-05-27       2 pre         32     28 post       133
11 1003  1949-05-27       2 post        34     30 pre        136
12 1003  1949-05-27       2 post        34     30 post       133
13 1004  1955-03-12       4 pre         22     22 pre        170
14 1004  1955-03-12       4 pre         22     22 post       175
15 1004  1955-03-12       4 post        27     26 pre        170
16 1004  1955-03-12       4 post        27     26 post       175
17 1005  1942-06-07       3 <NA>        NA     NA <NA>        NA
```

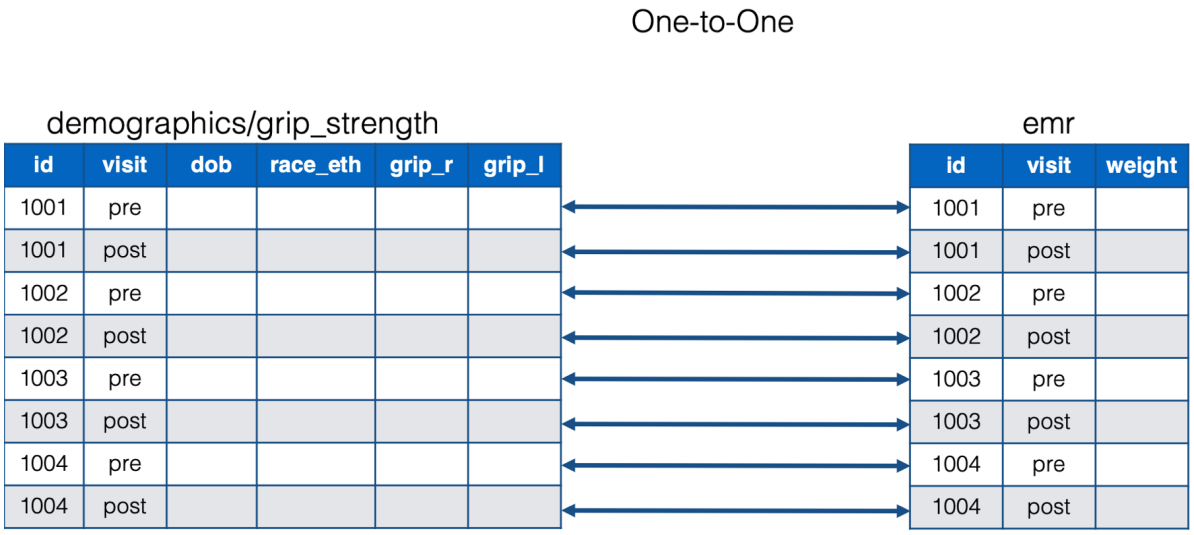
But, if you look closely, that isn't what we want either. Each participant didn't have four visits. They only had two. Here's the problem. Each participant in the combined `demographics/grip_strength` data frame has two rows (i.e., one for pre and one for post). Each participant in the `emr` data frame also has two rows (i.e., one for pre and one for post).

Above, we told `left_join()` to join by `id`. So, `left_join()` aligns all rows with matching key values – `id`'s.

For example, row one in the combined `demographics/grip_strength` data frame has the key value `1001`. So, `left_join()` aligns row one in the combined `demographics/grip_strength` data frame with rows one *and* two in the `emr` data frame. Next, row *two* in the combined `demographics/grip_strength` data frame has the key value `1001`. So, `left_join()` aligns row *two* in the combined `demographics/grip_strength` data frame with rows one *and* two in the `emr` data frame. This results in 2 * 2 = 4 rows for each id - a **many-to-many** merge.



But in reality, study id alone no longer uniquely identifies observations in our data. Now, observations are uniquely identified by study id and visit. For example, `1001` *and* `pre` are a unique observation, `1001` *and* `post` are a unique observation, `1002` *and* `pre` are a unique observation, and so on. We now have two key columns that identify unique observations. And once we give that information to `left_join`, the relationship between the data frames becomes a one-to-one relationship. In other words, each observation (defined by `id` *and* `visit`) in one data frame is related to no more than one observation (defined by `id` *and* `visit`) in the other data frame.



Here is how we tell `left_join()` to merge our data frames by `id` *and* `visit`:

```
demographics %>%
  left_join(grip_strength, by = "id") %>%
  left_join(emr, by = c("id", "visit"))
```

```
# A tibble: 9 x 7
  id    dob        race_eth visit grip_r grip_l weight
  <chr> <date>        <dbl> <chr>  <dbl>  <dbl>  <dbl>
1 1001  1968-12-14        1 pre       32     30    105
2 1001  1968-12-14        1 post      33     32     99
3 1002  1952-08-03        2 pre       28     30    200
4 1002  1952-08-03        2 post      27     30    201
5 1003  1949-05-27        2 pre       32     28    136
6 1003  1949-05-27        2 post      34     30    133
7 1004  1955-03-12        4 pre       22     22    170
8 1004  1955-03-12        4 post      27     26    175
9 1005  1942-06-07        3 <NA>      NA     NA     NA
```

Here's what we did above:

- We used `dplyr`'s `left_join()` function to perform a one-to-many merge of the `demographics` data frame with the `grip_strength` data frame. Then,

we used `left_join()` again to perform a one-to-one merge of the combined `demographics/grip_strength` data frame with the `emr` data frame.

- We told `left_join()` that it needed to match the values in the `id` key column *and* the values in the `visit` key column in order to align the rows in the combined `demographics/grip_strength` data frame with the `emr` data frame.

We now have a robust set of tools we can use to work with data that is stored in more than one data frame – a common occurrence in epidemiology!

32 Restructuring Data frames

we've already seen data frames with a couple of different structures, but we haven't explicitly discussed those structures yet. When we say structure, we basically mean the way the data is organized into columns and rows. Traditionally, data are described as being organized in one of two ways:

1. With a **person-level**, or **wide**, structure. In person-level data, each person (observational unit) has one observation (row) and a separate column contains data for each measurement. For example:

id	sex	weight_3	weight_6	weight_9	weight_12
1001	F	9	13	16	17
1002	F	11	16	17	20
1003	M	17	20	23	24
1004	F	16	18	21	22
1005	M	11	15	16	18
1006	M	17	21	25	26
1007	M	16	17	19	21
1008	F	15	16	18	19

Figure 32.1: Baby weights at 3, 6, 9, and 12 months.

2. With a **person-period**, or **long**, structure. In the person-period data structure each person (observational unit) has multiple observations – one for each measurement occasion.

id	months	sex	weight
1001	3	F	9
1001	6	F	13
1001	9	F	16
1001	12	F	17
1002	3	F	11
1002	6	F	16
1002	9	F	17
1002	12	F	20

Figure 32.2: Baby weights at 3, 6, 9, and 12 months. Babies 1001 and 1002 only.

> **Note**
>
> Often, people are our observational unit in epidemiology. However, our observational units could also be schools, states, or air quality monitors. It's the entity from which we are gathering data.

In some cases, only the person-level data structure will practically make sense. For example, the table below contains the sex, weight, length, head circumference, and abdominal circumference for eight newborn babies measured cross-sectionally (i.e., at one point in time) at birth.

id	sex	weight	length	head	abdomen
1001	F	7	17	31	16
1002	F	9	19	35	13
1003	M	15	23	38	14
1004	F	14	20	33	14
1005	M	9	18	31	14
1006	M	15	22	33	16
1007	M	14	21	35	13
1008	F	13	18	37	16

Figure 32.3: Various measurements take at birth for 8 newborn babies.

In this table, each baby has one observation (row) and a separate column contains data for each measurement. Further, each measurement is only taken on *one* occasion. There really is no other structure that makes sense for this data.

For contrast, the next table below is also person-level data. It contains the weight in pounds for eight babies at ages 3 months, 6 months, 9 months, and 12 months.

id	sex	weight_3	weight_6	weight_9	weight_12
1001	F	9	13	16	17
1002	F	11	16	17	20
1003	M	17	20	23	24
1004	F	16	18	21	22
1005	M	11	15	16	18
1006	M	17	21	25	26
1007	M	16	17	19	21
1008	F	15	16	18	19

Figure 32.4: Baby weights at 3, 6, 9, and 12 months

Notice that each baby still has one, and only one, row. This time, however, there are only 2 measurements – sex and weight. Sex is measured on one occasion, but weight is measured on four occasions, and a *new column* is created in the data frame for each subsequent measure of weight. So, although each baby has a single *row* in the data, they really have four *observations* (i.e., measurement occasions). **Notice that this is the first time that we've explicitly drawn a distinction between a row and an observation.** Further, unlike the first table we saw, this table could actually be structured in a different way.

An alternative, and often preferable, data structure for data with repeated measures is the person-period, or long, data structure. Below, we look at the baby weights again. In the interest of saving space, we're only looking at the first two babies from the previous table of data.

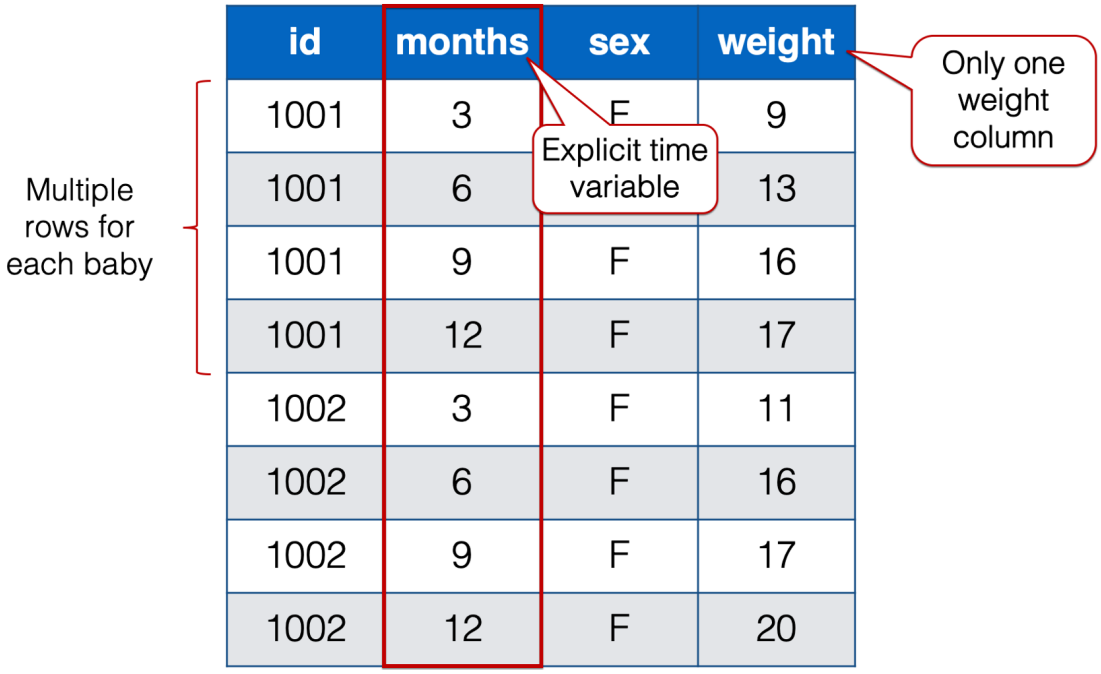


Figure 32.5: Baby weights at 3, 6, 9, and 12 months. Babies 1001 and 1002 only.

Notice that each baby in the person-period table has four rows – one for each weight measurement. Also notice that there is a new variable in the person-period data that explicitly records time (i.e., `months`).

> **Note**
>
> Let's quickly learn a couple of new terms: *time-varying* and *time-invariant* variables. In the data above, `sex` is time invariant. It remains constant over all 4 measurement occasions for each baby. Not only that, but for all intents and purposes it isn't really *allowed* to change. The `weight` variable, on the other hand, is time varying. The weight values change over time. And not only do they change, but the amount, rate, and/or shape of their change may be precisely what this researcher is interested in.

Below, we can compare the person-level version of the baby weight data to the person-period version of the baby weight data. we are only including babies 1001 and 1002 in the interest of saving space. As you can see, given the same data, the person-level structure is wider (i.e., more *columns*) than the person-period data and the person-period structure is longer (i.e., more *rows*) than the person-level data. That's why the two structures are sometimes referred to as wide and long respectively.

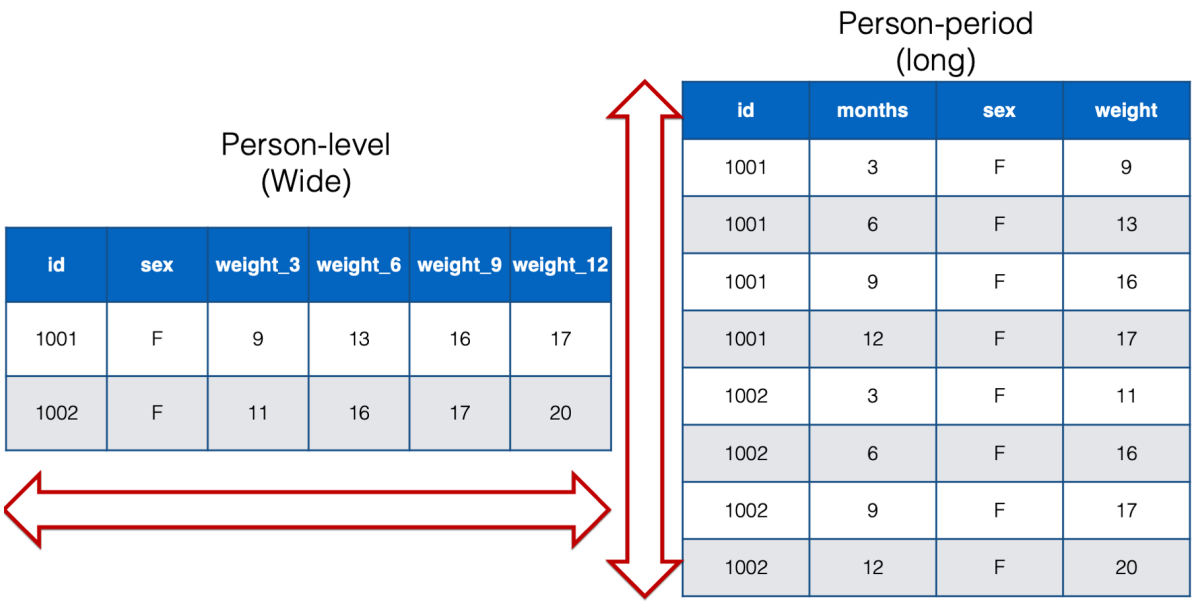


Figure 32.6: Comparing wide and long data for the babies 1001 and 1002.

Ok, so this data *can* be structured in either a person-level *or* a person-period format, but which structure *should* we use?

Well, in general, we are going to suggest that you use the person-period structure for the kind of longitudinal data we have above for the following reasons:

1. It contains an explicit time variable. The time information may be descriptively interesting on its own, or we may need to include it in our statistical models. In fact, many longitudinal analyses will require that our data have a person-period structure. For example, mixed models, gereralized estimating equations, and survival analysis.
2. The person-period structure can be more efficient when we the intervals between repeated measures vary across observational units. For example, in the data above the baby weight columns were named `weight_3`, `weight_6`, `weight_9`, and `weight_12`, which indicated each baby's weight at a 3-month, 6-month, 9-month, and 12-month checkup. However, what if the study needed a more precise measure of each baby's age. Let's say that we needed to record each baby's weight at their precise age in days at each checkup. That might look something like the following if structured in a person-level format:

id	sex	weight_36	weight_84	weight_150	weight_173	weight_214	weight_222	weight_317	weight_332
1001	F	9	NA	NA	13	NA	16	17	NA
1002	F	NA	11	16	NA	17	NA	NA	20

Figure 32.7: Baby weights at age in days. Babies 1001 and 1002 only.

Notice all the missing data in this format – even with only two babies. For example, baby 1001 had her first check-up at 36 days old. She was 9 lbs. Baby 1002, however, didn't have her first checkup until she was 84 days old. So, baby 1002 has a missing value for `weight_36`. That pattern continues throughout the data. Now, just try to imagine what this would look like for tens, hundreds, or thousands of babies. It would be a mess! By contrast, the person-period version of this data is much more efficient. In fact, it looks almost identical to the first person-period version of this data:

id	days	sex	weight
1001	36	F	9
1001	173	F	13
1001	222	F	16
1001	317	F	17
1002	84	F	11
1002	150	F	16
1002	214	F	17
1002	332	F	20

Figure 32.8: Baby weights at age in days. Babies 1001 and 1002 only.

3. For essentially the same reasons already discussed above, the person-period format is better suited for handling time-varying predictors. In the baby weight data, the only predictor variable (other than time) was sex, which is time invariant. Regardless of which structure we use, sex only requires one column in the data frame because it never changes. However, imagine a scenario where we also collect height and information about diet at each visit. Using a person-level structure to store these variables would have the same limitations that we already discussed above (i.e., no explicit measure of time, incompatibility with many analysis techniques, and potentially inefficient storage).
4. Many of the "tidyverse" packages we use in this book (e.g., `dplyr` and `ggplot2`) assume, or at least work best, with data organized in a person-period, or long, format.

So, does this mean that we should *never* organize our data frames in a person-level format? Of course not! There are going to be some occasions when there are advantages to organizing our data frames in a person-level format. For example:

1. Many people prefer the person-level format during the data entry process because it can require less typing. Thinking about our baby weight data above, we would only need to type one new value at each checkup (i.e., weight) if the data is organized in a person-level format. However, if the data is organized in a person-period format, we have to type three new values (i.e., id, sex, and weight). This limitation grows with the number of time-invariant variables in the data.

2. There are some analyses that will require that our data have a person-level structure. For example, the traditional ANOVA and MANOVA techniques assume the wide format.
3. There are times when our data is easier to manipulate when it is organized in a person-level format.
4. There are times when it's advantageous to restructure statistical results from a longer format to a wider format to present them in the most effective way possible.

Luckily, we rarely have to choose one structure or the other in an absolute sense. The `tidyr` package generally makes it very easy for us to restructure ("reshape" is another commonly used term) our data frames from wide to long and back again. This allows us to organize our data in the manner that is best suited for the particular task at hand. Let's go ahead and take a look at some examples.

32.1 The tidyr package

The tools we will use for restructuring our data will primarily come from a package we haven't used before in this book – `tidyr`. If you haven't already done so, and you'd like to follow along, please install and load `tidyr`, `dplyr`, and `ggplot2` now.

```
library(tidyr)
library(dplyr)
library(ggplot2)
```

32.2 Pivoting longer

In epidemiology, it's common for data that we analyze to be measured on multiple occasions. It's also common for repeated measures data to be entered into a spreadsheet or database in such a way that each new measure is a new column. We saw an example of this above:

id	sex	weight_3	weight_6	weight_9	weight_12
1001	F	9	13	16	17
1002	F	11	16	17	20
1003	M	17	20	23	24
1004	F	16	18	21	22
1005	M	11	15	16	18
1006	M	17	21	25	26
1007	M	16	17	19	21
1008	F	15	16	18	19

Figure 32.9: Baby weights at 3, 6, 9, and 12 months

we already concluded that this data has a person-level (wide) structure. As discussed above, many techniques that we may want to use to analyze this data will require us to restructure it to a person-period format. Let's go ahead and walk through a demonstration of how do that. We will start by simulating this data in R:

```
babies <- tibble(
  id        = 1001:1008,
  sex       = c("F", "F", "M", "F", "M", "M", "M", "F"),
  weight_3  = c(9, 11, 17, 16, 11, 17, 16, 15),
  weight_6  = c(13, 16, 20, 18, 15, 21, 17, 16),
  weight_9  = c(16, 17, 23, 21, 16, 25, 19, 18),
  weight_12 = c(17, 20, 24, 22, 18, 26, 21, 19)
) %>%
  print()
```

```
# A tibble: 8 x 6
     id sex   weight_3 weight_6 weight_9 weight_12
  <int> <chr>    <dbl>    <dbl>    <dbl>     <dbl>
1  1001 F            9       13       16        17
2  1002 F           11       16       17        20
```

```
3  1003 M           17       20       23       24
4  1004 F           16       18       21       22
5  1005 M           11       15       16       18
6  1006 M           17       21       25       26
7  1007 M           16       17       19       21
8  1008 F           15       16       18       19
```

Now, let's use the `pivot_longer()` function to restructure the `babies` data frame to a person-period format:

```
babies_long <- babies %>%
  pivot_longer(
    cols         = starts_with("weight"),
    names_to     = "months",
    names_prefix = "weight_",
    values_to    = "weight"
  ) %>%
  print()
```

```
# A tibble: 32 x 4
      id sex   months weight
   <int> <chr> <chr>   <dbl>
 1  1001 F     3           9
 2  1001 F     6          13
 3  1001 F     9          16
 4  1001 F     12         17
 5  1002 F     3          11
 6  1002 F     6          16
 7  1002 F     9          17
 8  1002 F     12         20
 9  1003 M     3          17
10  1003 M     6          20
# i 22 more rows
```

Here's what we did above:

- we used `tidyr`'s `pivot_longer()` function to restructure the `babies` data frame from person-level (wide) to person-period (long).
- You can type `?pivot_longer` into your R console to view the help documentation for this function and follow along with the explanation below.

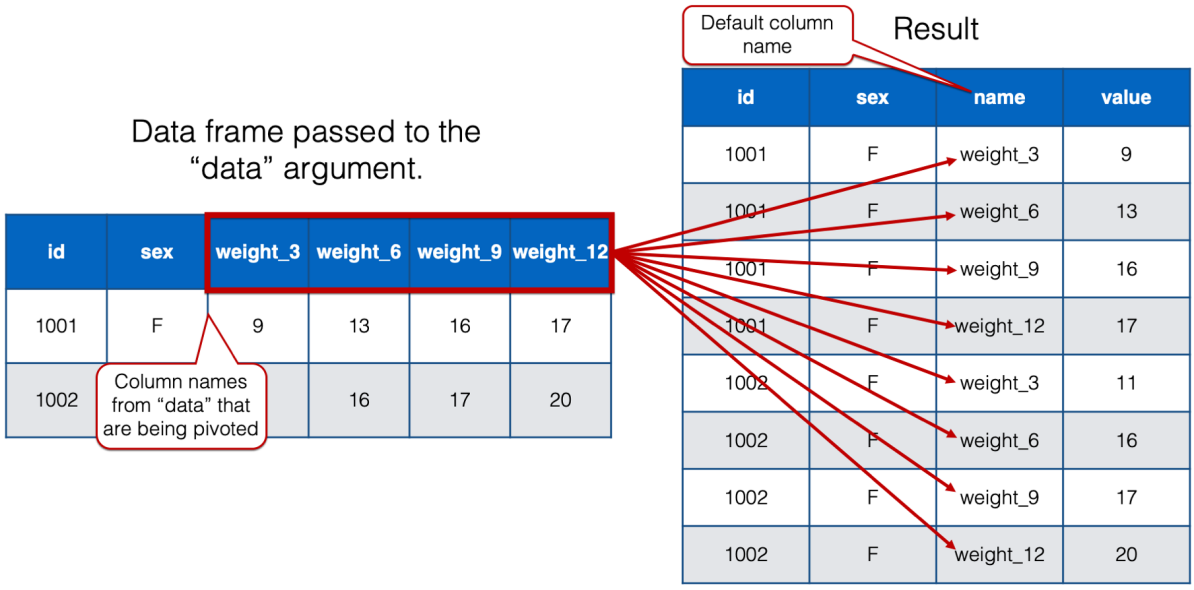
- The first argument to the `pivot_longer()` function is the `data` argument. You should pass the name of the data frame you want to restructure to the `data` argument. Above, we passed the `babies` data frame to the `data` argument using a pipe operator.
- The second argument to the `pivot_longer()` function is the `cols` argument. You should pass the name of the columns you want to make longer to the `cols` argument. Above, we passed the names of the four weight columns to the `cols` argument. The `cols` argument actually accepts tidy-select argument modifiers. We first discussed tidy-select argument modifiers in the chapter on subsetting data frames. In the example above, we used the `starts_with()` tidy-select modifier to simplify our code. Instead of passing each column name directly to the `cols` argument, we asked `starts_with()` to pass the name of any column that has a column name that starts with the word "weight" to the `cols` argument.
- The third argument to the `pivot_longer()` function is the `names_to` argument. You should pass the `names_to` argument a character string or character vector that tells `pivot_longer()` what you want to name the column that will contain the previous column names that were pivoted. By default, the value passed to the `names_to` argument is `"name"`. We passed the value `"months"` to the `names_to` argument. This tells `pivot_longer()` what to name the column that contains the names of the previous column names. If that seems really confusing, I'm with you. Unfortunately, we don't currently know a better way to write it, but we will *show* you what the `names_to` argument does below.
- The fourth argument to the `pivot_longer()` function is the `names_prefix` argument. You should pass the `names_prefix` argument a regular expression that tells `pivot_longer()` what to remove from the start of each of the previous column names that we pivoted. By default, the value passed to the `names_prefix` argument is `NULL` (i.e., it doesn't remove anything). We passed the value `"weight_"` to the `names_prefix` argument. This tells `pivot_longer()` that we want to remove the character string "weight_" from the start of each of the previous column names that we pivoted. For example, removing "weight_" from "weight_3" results in the value "3", removing "weight_" from "weight_6" results in the value "6", and so on. Again, we will show you what the `names_prefix` argument does below.
- The eighth argument (we left the 5th, 6th, and 7th arguments at their default values) to the `pivot_longer()` function is the `values_to` argument. You should pass the `values_to` argument a character string or character vector that tells `pivot_longer()` what you want to name the column that will contain the values from the columns that were pivoted. By default, the value passed to the `values_to` argument is `"value"`. We passed the value `"weight"` to the `values_to` argument. This tells `pivot_longer()` what to name the column that contains values from the columns that were pivoted. we will demonstrate what the `values_to` argument does below as well.

32.2.1 The names_to argument

The official help documentation for `pivot_longer()` says that the value passed to the `names_to` argument should be "a string specifying the name of the column to create from the data stored in the column names of data." we don't blame you if you feel like that's a little bit difficult to wrap your head around. Let's take a look at the result we get when we don't adjust the value passed to the `names_to` argument:

```
babies %>%
  pivot_longer(
    cols = starts_with("weight")
  )
```

```
# A tibble: 32 x 4
      id sex   name      value
   <int> <chr> <chr>     <dbl>
 1  1001 F     weight_3      9
 2  1001 F     weight_6     13
 3  1001 F     weight_9     16
 4  1001 F     weight_12    17
 5  1002 F     weight_3     11
 6  1002 F     weight_6     16
 7  1002 F     weight_9     17
 8  1002 F     weight_12    20
 9  1003 M     weight_3     17
10  1003 M     weight_6     20
# i 22 more rows
```



As you can see, when we only pass a value to the `cols` argument, `pivot_longer()` creates a new column that contains the column names from the data frame passed to the `data` argument, that are being pivoted into long format. By default, `pivot_longer()` names that column `name`. However, that name isn't very informative. We will go ahead and change the column name to "months" because we know that this column will eventually contain month values. We do so by passing the value `"months"` to the `names_to` argument like this:

```
babies %>%
  pivot_longer(
    cols      = starts_with("weight"),
    names_to = "months"
  )
```

```
# A tibble: 32 x 4
      id sex   months    value
   <int> <chr> <chr>     <dbl>
 1  1001 F     weight_3      9
 2  1001 F     weight_6     13
 3  1001 F     weight_9     16
 4  1001 F     weight_12    17
 5  1002 F     weight_3     11
 6  1002 F     weight_6     16
 7  1002 F     weight_9     17
```

```
 8  1002 F     weight_12    20
 9  1003 M     weight_3     17
10  1003 M     weight_6     20
# i 22 more rows
```

32.2.2 The names_prefix argument

The official help documentation for `pivot_longer()` says that the value passed to the `names_prefix` argument should be "a regular expression used to remove matching text from the start of each variable name." Passing a value to this argument can be really useful when column names actually contain data values, which was the case above. Take the column name "weight_3" for example. The "weight" part is truly a column name – it tells us what the values in that column are. They are weights. The "3" part is actually a separate data value meaning "3 months." If we can remove the "weight_" part of the column name, then what remains is a useful column of information – time measured in months. Passing the value "weight_" to the `names_prefix` argument does exactly that.

```
babies %>%
  pivot_longer(
    cols         = starts_with("weight"),
    names_to     = "months",
    names_prefix = "weight_"
  )
```

```
# A tibble: 32 x 4
      id sex   months value
   <int> <chr> <chr>  <dbl>
 1  1001 F     3          9
 2  1001 F     6         13
 3  1001 F     9         16
 4  1001 F     12        17
 5  1002 F     3         11
 6  1002 F     6         16
 7  1002 F     9         17
 8  1002 F     12        20
 9  1003 M     3         17
10  1003 M     6         20
# i 22 more rows
```

Now, the value passed to the `names_prefix` argument can be any regular expression. So, we could have written a more complicated, and flexible, regular expression like this:

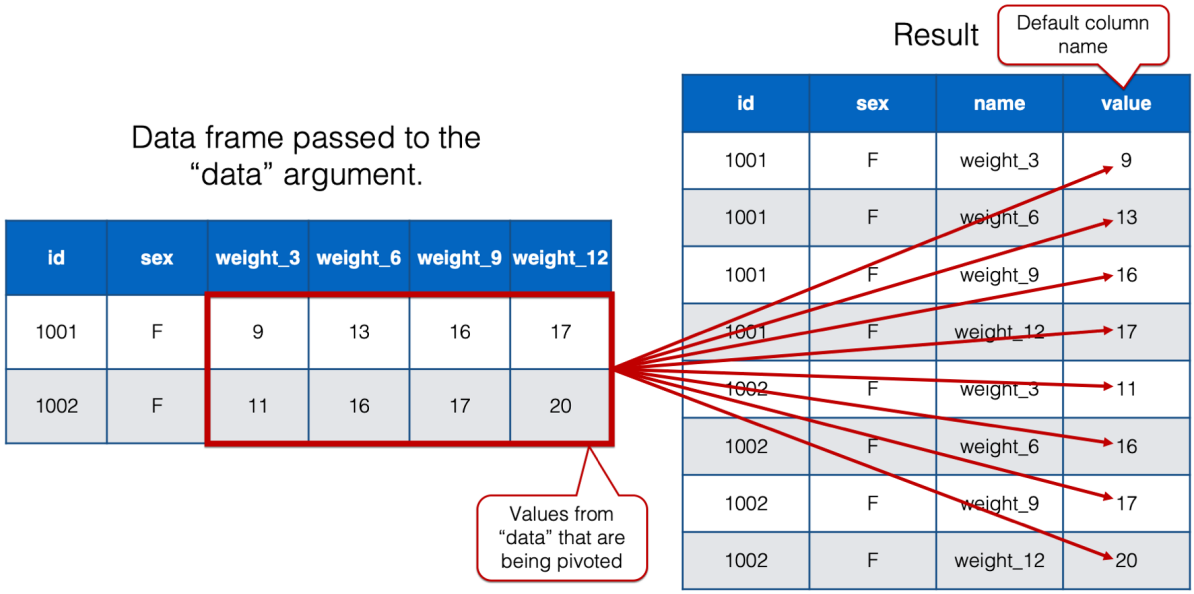
```
babies %>%
  pivot_longer(
    cols         = starts_with("weight"),
    names_to     = "months",
    names_prefix = "\\w+_"
  )
```

```
# A tibble: 32 x 4
      id sex   months value
   <int> <chr> <chr>  <dbl>
 1  1001 F     3          9
 2  1001 F     6         13
 3  1001 F     9         16
 4  1001 F     12        17
 5  1002 F     3         11
 6  1002 F     6         16
 7  1002 F     9         17
 8  1002 F     12        20
 9  1003 M     3         17
10  1003 M     6         20
# i 22 more rows
```

The regular expression above would have removed *any* word characters followed by an underscore. However, in this case, the value `"weight_"` is straightforward and gets the job done.

32.2.3 The values_to argument

The official help documentation for `pivot_longer()` says that the value passed to the `values_to` argument should be "a string specifying the name of the column to create from the data stored in cell values." All that means is that we use this argument to name the column that contains the values that were pivoted.



By default, `pivot_longer()` names that column "value." However, we will once again want a more informative column name in our new data frame. So, we'll go ahead and change the column name to "weight" because that's what the values in that column are – weights. We do so by passing the value `"weight"` to the `values_to` argument like this:

```
babies %>%
  pivot_longer(
    cols          = starts_with("weight"),
    names_to      = "months",
    names_prefix = "weight_",
    values_to     = "weight"
  )
```

```
# A tibble: 32 x 4
      id sex   months weight
   <int> <chr> <chr>   <dbl>
 1  1001 F     3           9
 2  1001 F     6          13
 3  1001 F     9          16
 4  1001 F     12         17
 5  1002 F     3          11
 6  1002 F     6          16
 7  1002 F     9          17
```

```
 8  1002 F      12          20
 9  1003 M      3           17
10  1003 M      6           20
# i 22 more rows
```

32.2.4 The names_transform argument

As one little final touch on the data restructuring at hand, it would be nice to coerce the `months` column from type character to type integer. We already know how to do this with `mutate()`:

```
babies %>%
  pivot_longer(
    cols          = starts_with("weight"),
    names_to      = "months",
    names_prefix = "weight_",
    values_to     = "weight"
  ) %>%
  mutate(months = as.integer(months))
```

```
# A tibble: 32 x 4
      id sex   months weight
   <int> <chr>  <int>  <dbl>
 1  1001 F          3      9
 2  1001 F          6     13
 3  1001 F          9     16
 4  1001 F         12     17
 5  1002 F          3     11
 6  1002 F          6     16
 7  1002 F          9     17
 8  1002 F         12     20
 9  1003 M          3     17
10  1003 M          6     20
# i 22 more rows
```

However, we can also do this directly inside the `pivot_longer()` function by passing a list of column names paired with type coercion functions. For example:

```
babies %>%
  pivot_longer(
    cols             = starts_with("weight"),
```

```
    names_to        = "months",
    names_prefix    = "weight_",
    names_transform = list(months = as.integer),
    values_to       = "weight"
  )
```

```
# A tibble: 32 x 4
      id sex   months weight
   <int> <chr>  <int>  <dbl>
 1  1001 F          3      9
 2  1001 F          6     13
 3  1001 F          9     16
 4  1001 F         12     17
 5  1002 F          3     11
 6  1002 F          6     16
 7  1002 F          9     17
 8  1002 F         12     20
 9  1003 M          3     17
10  1003 M          6     20
# i 22 more rows
```

Here's what we did above:

- we coerced the `months` column from type character to type integer by passing the value `list(months = as.integer)` to the `names_transform` argument. The list passed to `names_transform` should contain one or more column names paired with a type coercion function. The column name and type coercion function should be paired using an equal sign. Multiple pairs should be separated by commas.

32.2.5 Pivoting multiple sets of columns

Let's add a little layer of complexity to our situation. Let's say that our `babies` data frame also includes each baby's length in inches measured at each visit:

```
set.seed(123)
babies <- tibble(
  id       = 1001:1008,
  sex      = c("F", "F", "M", "F", "M", "M", "M", "F"),
  weight_3  = c(9, 11, 17, 16, 11, 17, 16, 15),
  weight_6  = c(13, 16, 20, 18, 15, 21, 17, 16),
  weight_9  = c(16, 17, 23, 21, 16, 25, 19, 18),
```

```
  weight_12 = c(17, 20, 24, 22, 18, 26, 21, 19),
  length_3  = c(17, 19, 23, 20, 18, 22, 21, 18),
  length_6  = round(length_3 + rnorm(8, 2, 1)),
  length_9  = round(length_6 + rnorm(8, 2, 1)),
  length_12 = round(length_9 + rnorm(8, 2, 1)),
) %>%
  print()
```

```
# A tibble: 8 x 10
     id sex   weight_3 weight_6 weight_9 weight_12 length_3 length_6 length_9
  <int> <chr>    <dbl>    <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1  1001 F            9       13       16        17       17       18       19
2  1002 F           11       16       17        20       19       21       23
3  1003 M           17       20       23        24       23       27       30
4  1004 F           16       18       21        22       20       22       24
5  1005 M           11       15       16        18       18       20       22
6  1006 M           17       21       25        26       22       26       28
7  1007 M           16       17       19        21       21       23       24
8  1008 F           15       16       18        19       18       19       23
# i 1 more variable: length_12 <dbl>
```

Here is what we want our final data frame to look like:

```
babies %>%
  pivot_longer(
    cols      = c(-id, -sex),
    names_to  = c(".value", "months"),
    names_sep = "_"
  )
```

```
# A tibble: 32 x 5
      id sex   months weight length
   <int> <chr> <chr>   <dbl>  <dbl>
 1  1001 F     3           9     17
 2  1001 F     6          13     18
 3  1001 F     9          16     19
 4  1001 F     12         17     21
 5  1002 F     3          11     19
 6  1002 F     6          16     21
 7  1002 F     9          17     23
 8  1002 F     12         20     23
```

```
 9  1003 M     3          17     23
10  1003 M     6          20     27
# i 22 more rows
```

Next, we'll walk through getting to this result step-by-step.

we are once again starting with a person-level data frame, and we once again want to restructure it to a person-period data frame. This is the result we get if we use the same code we previously used to restructure the data frame that didn't include each baby's length:

```
babies_long <- babies %>%
  pivot_longer(
    cols         = starts_with("weight"),
    names_to     = "months",
    names_prefix = "weight_",
    values_to    = "weight"
  ) %>%
  print()
```

```
# A tibble: 32 x 8
      id sex   length_3 length_6 length_9 length_12 months weight
   <int> <chr>    <dbl>    <dbl>    <dbl>     <dbl> <chr>   <dbl>
 1  1001 F           17       18       19        21 3           9
 2  1001 F           17       18       19        21 6          13
 3  1001 F           17       18       19        21 9          16
 4  1001 F           17       18       19        21 12         17
 5  1002 F           19       21       23        23 3          11
 6  1002 F           19       21       23        23 6          16
 7  1002 F           19       21       23        23 9          17
 8  1002 F           19       21       23        23 12         20
 9  1003 M           23       27       30        33 3          17
10  1003 M           23       27       30        33 6          20
# i 22 more rows
```

Because we aren't passing any of the `length_` columns to the `cols` argument, `pivot_longer()` is treating them like the other time-invariant variables (i.e., `id` and `sex`). Their values are just being recycled across every row within each id. So, let's add the `length_` columns to the `cols` argument and see what happens:

```
babies_long <- babies %>%
  pivot_longer(
    cols         = c(-id, -sex),
```

```
    names_to     = "months",
    names_prefix = "weight_",
    values_to    = "weight"
  ) %>%
  print()
```

```
# A tibble: 64 x 4
      id sex   months    weight
   <int> <chr> <chr>      <dbl>
 1  1001 F     3              9
 2  1001 F     6             13
 3  1001 F     9             16
 4  1001 F     12            17
 5  1001 F     length_3      17
 6  1001 F     length_6      18
 7  1001 F     length_9      19
 8  1001 F     length_12     21
 9  1002 F     3             11
10  1002 F     6             16
# i 54 more rows
```

Here's what we did above:

- we passed the `weight_` and `length_` columns to the `cols` argument *indirectly* by passing the value `c(-id, -sex)`. Basically, this tells `pivot_longer()` that we would like to pivot every column *except* `id` and `sex`.

Now, we are pivoting both the `weight_` columns and the `length_` columns. That's an improvement. However, we obviously still don't have the result we want.

Remember that the value passed to the `names_prefix` argument is used to remove matching text from the start of each variable name. Passing the value `"weight_"` to the `names_prefix` argument made sense when all of our pivoted columns began with the character sting "weight_". Now, however, some of our pivoted columns begin with the character string "length_". That's why we are still seeing values in the `months` column like `length_3`, `length_6`, and so on.

Now, your first instinct might be to just add `"length_"` to the `names_prefix` argument. Unfortunately, that doesn't work:

```
babies_long <- babies %>%
  pivot_longer(
    cols         = c(-id, -sex),
    names_to     = "months",
```

```
    names_prefix = c("weight_", "length_"),
    values_to    = "weight"
  ) %>%
  print()
```

```
Warning in gsub(vec_paste0("^", names_prefix), "", cols): argument 'pattern'
has length > 1 and only the first element will be used
```

```
# A tibble: 64 x 4
      id sex   months    weight
   <int> <chr> <chr>      <dbl>
 1  1001 F     3              9
 2  1001 F     6             13
 3  1001 F     9             16
 4  1001 F     12            17
 5  1001 F     length_3      17
 6  1001 F     length_6      18
 7  1001 F     length_9      19
 8  1001 F     length_12     21
 9  1002 F     3             11
10  1002 F     6             16
# i 54 more rows
```

Instead, we need to drop the `names_prefix` argument altogether before we can move forward to the correct solution:

```
babies_long <- babies %>%
  pivot_longer(
    cols      = c(-id, -sex),
    names_to  = "months",
    values_to = "weight"
  ) %>%
  print()
```

```
# A tibble: 64 x 4
      id sex   months    weight
   <int> <chr> <chr>      <dbl>
 1  1001 F     weight_3       9
 2  1001 F     weight_6      13
 3  1001 F     weight_9      16
```

```
 4  1001 F     weight_12     17
 5  1001 F     length_3      17
 6  1001 F     length_6      18
 7  1001 F     length_9      19
 8  1001 F     length_12     21
 9  1002 F     weight_3      11
10  1002 F     weight_6      16
# i 54 more rows
```

Additionally, not all the values in the third column (i.e., `weight`) are weights. Half of those values are lengths. So, we also need to drop the `values_to` argument:

```
babies_long <- babies %>%
  pivot_longer(
    cols     = c(-id, -sex),
    names_to = "months"
  ) %>%
  print()
```

```
# A tibble: 64 x 4
      id sex   months    value
   <int> <chr> <chr>     <dbl>
 1  1001 F     weight_3      9
 2  1001 F     weight_6     13
 3  1001 F     weight_9     16
 4  1001 F     weight_12    17
 5  1001 F     length_3     17
 6  1001 F     length_6     18
 7  1001 F     length_9     19
 8  1001 F     length_12    21
 9  1002 F     weight_3     11
10  1002 F     weight_6     16
# i 54 more rows
```

Believe it or not, we are actually pretty close to accomplishing our goal. Next, we need to somehow tell `pivot_longer()` that the column names we are pivoting contain a description of the values (i.e., heights and weights) *and* time values (i.e., 3, 6, 9, and 12 months). Notice that in all cases, the description and the time value are separated by an underscore. It turns out that we can use the `names_sep` argument to give `pivot_longer()` this information.

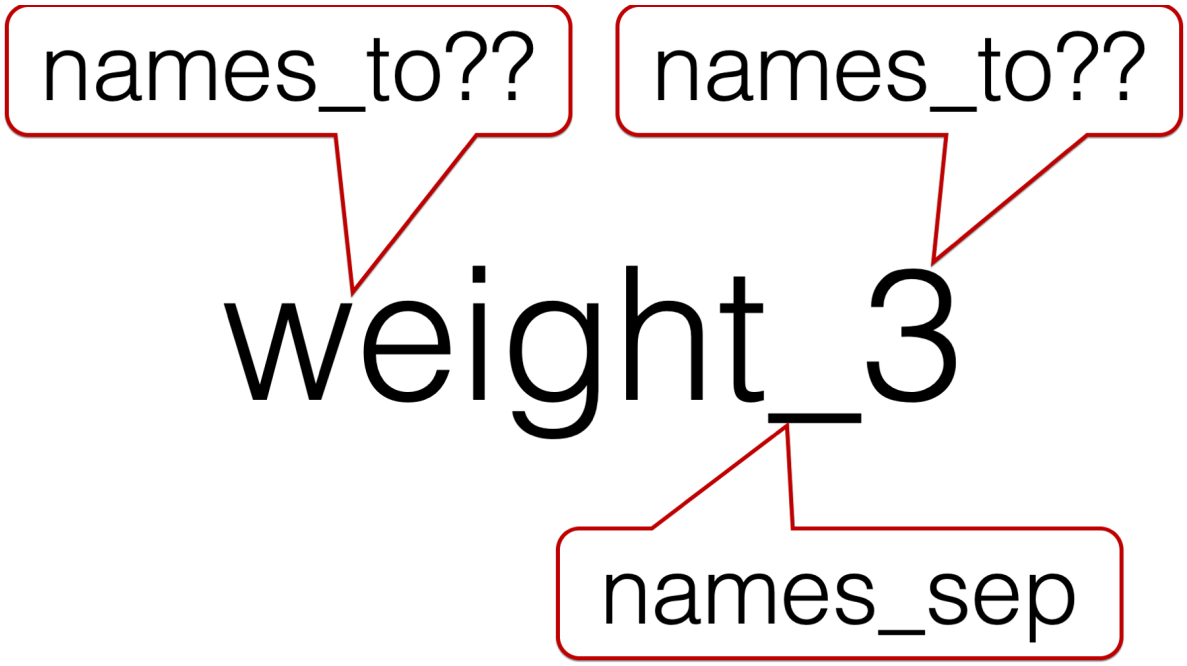
32.2.6 The names_sep argument

Let's start by simply passing the adding the `names_sep` argument to the `pivot_longer()` function and pass it the value that separates our description and our time value:

```
babies_long <- babies %>%
  pivot_longer(
    cols      = c(-id, -sex),
    names_to  = "months",
    names_sep = "_"
  ) %>%
  print()
```

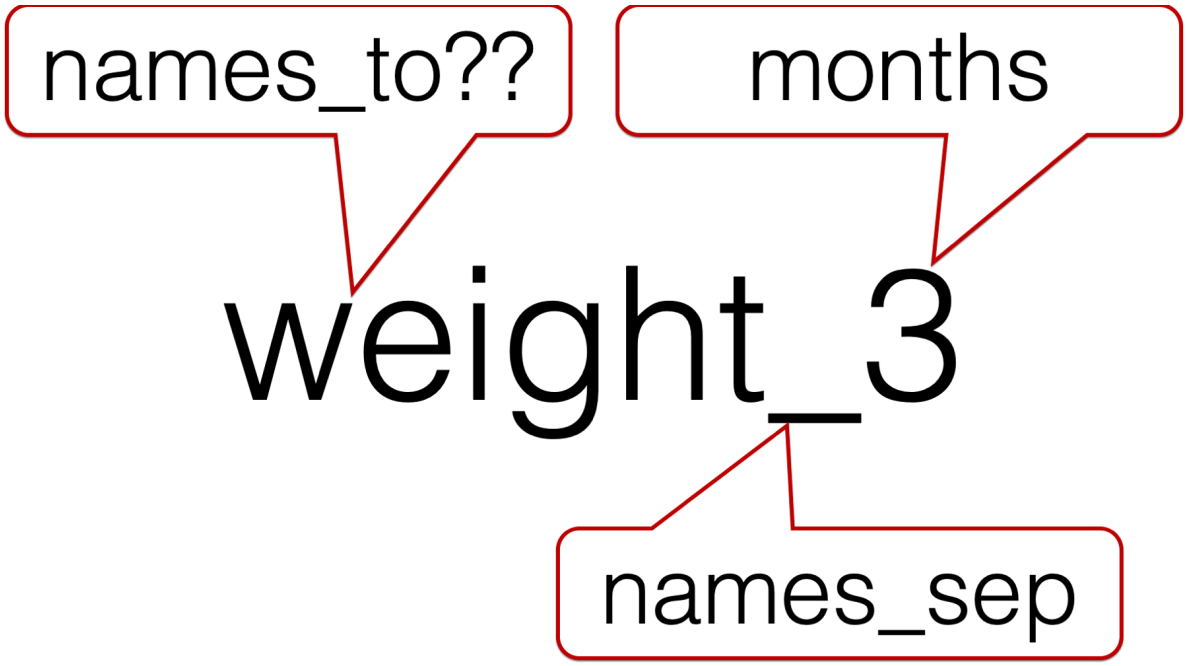
```
Error in `pivot_longer()`:
! `names_sep` can't be used with a length 1 `names_to`.
```

And we get an error. The reason we get an error can be seen in the following figure:



we are asking `pivot_longer()` to break up each column name (e.g., `weight_3`) at the underscore. That results in creating two separate character strings. In this case, the character string "weight" and the character string "3". However, we only passed one value to the `names_to` argument – `"months"`. So, which character string should `pivot_longer()` put in the `months` column? Of course, we know that the answer is "3", but `pivot_longer()` doesn't know that.

So, we have to pass two values to the names_to argument. But, what values should we pass?



we obviously want to character string that comes after the underscore to be called "months". However, we can't call the character string in front of the underscore "weight" because this column isn't just identifying rows that contain weights. Similarly, we can't call the character string in front of the underscore "length" because this column isn't just identifying rows that contain lengths. For lack of a better idea, let's just call it "measure".

```
babies_long <- babies %>%
  pivot_longer(
    cols      = c(-id, -sex),
    names_to  = c("measure", "months"),
    names_sep = "_"
  ) %>%
  print()
```

```
# A tibble: 64 x 5
      id sex   measure months value
   <int> <chr> <chr>   <chr>  <dbl>
 1  1001 F     weight  3          9
 2  1001 F     weight  6         13
 3  1001 F     weight  9         16
 4  1001 F     weight  12        17
```

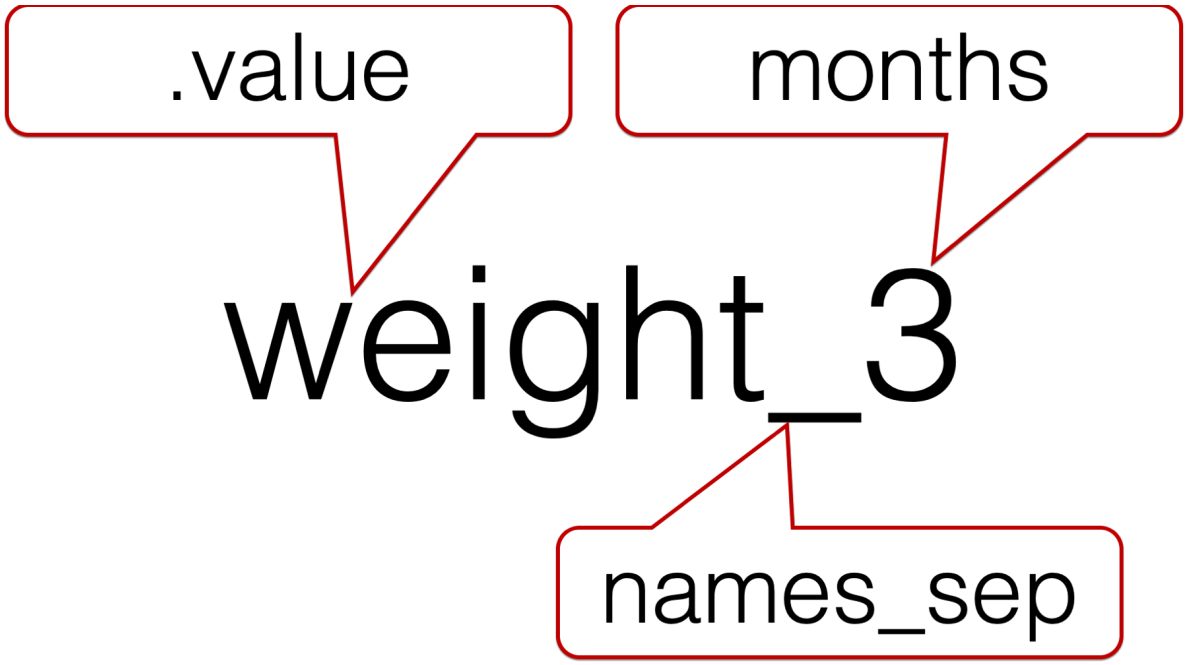
```
 5  1001 F     length  3         17
 6  1001 F     length  6         18
 7  1001 F     length  9         19
 8  1001 F     length  12        21
 9  1002 F     weight  3         11
10  1002 F     weight  6         16
# i 54 more rows
```

That sort of works. Except, what we really want is one row for each combination of id and months, each containing a value for weight and length. Instead, we have two rows for each combination of id and months. One set of rows contains weights and the other set of rows contains lengths.

What we really need is for `pivot_longer()` to make `weight` one column and `length` a separate column, and then put the appropriate values from `value` under each. We can do this with the `.value` special value.

32.2.7 The .value special value

The official help documentation for `pivot_longer()` says that the `.value` special value "indicates that [the] component of the name defines the name of the column containing the cell values, overriding values_to." Said another way, `.value` tells `pivot_longer()` the character string in front of the underscore is the value description. Further, `.value` tells `pivot_longer()` to create a new column for each unique character string that is in front of the underscore.



Now, let's add the `.value` special value to our code:

```
babies_long <- babies %>%
  pivot_longer(
    cols      = c(-id, -sex),
    names_to  = c(".value", "months"),
    names_sep = "_",
    names_transform = list(months = as.integer)
  ) %>%
  print()
```

```
# A tibble: 32 x 5
      id sex   months weight length
   <int> <chr>  <int>  <dbl>  <dbl>
 1  1001 F          3      9     17
 2  1001 F          6     13     18
 3  1001 F          9     16     19
 4  1001 F         12     17     21
 5  1002 F          3     11     19
 6  1002 F          6     16     21
 7  1002 F          9     17     23
 8  1002 F         12     20     23
 9  1003 M          3     17     23
```

```
10  1003 M          6     20     27
# i 22 more rows
```

And that is exactly the result we wanted. However, there was one little detail we didn't cover. How does `.value` know to create a new column for each unique character string that is in *front* of the underscore. Why didn't it create a new column for each unique character string that is *behind* the underscore?

The answer is simple. It knows because of the ordering we used in the value we passed to the `names_to` argument. If we changed the order to `c("months", ".value")`, `pivot_longer()` would have created a new column for each unique character string that is *behind* the underscore. Take a look:

```
babies %>%
  pivot_longer(
    cols      = c(-id, -sex),
    names_to  = c("months", ".value"),
    names_sep = "_"
  )
```

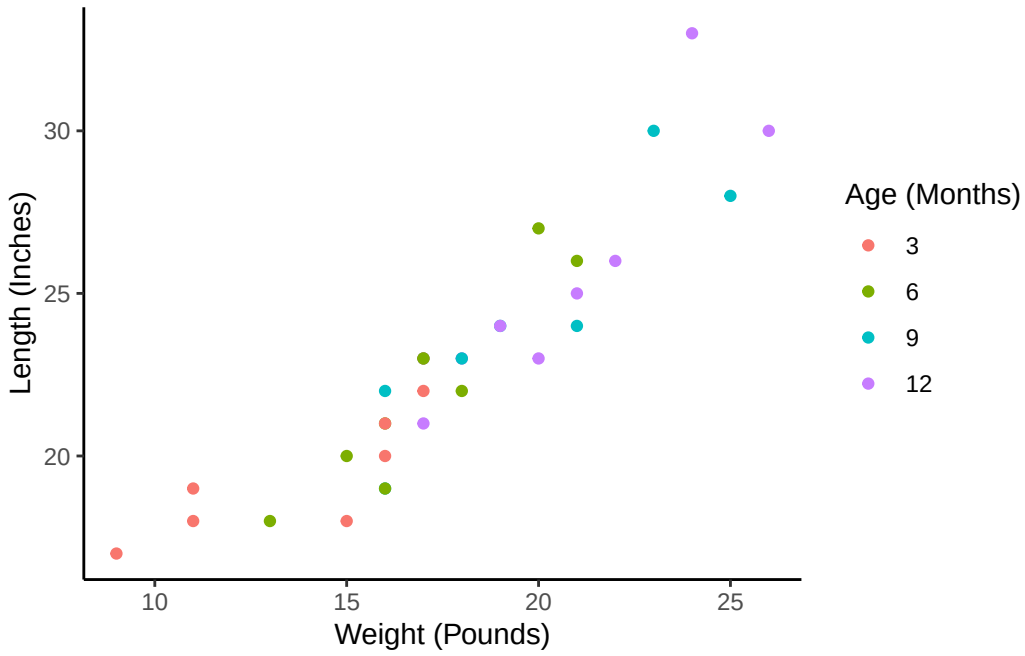
```
# A tibble: 16 x 7
      id sex   months   `3`   `6`   `9`  `12`
   <int> <chr> <chr>  <dbl> <dbl> <dbl> <dbl>
 1  1001 F     weight     9    13    16    17
 2  1001 F     length    17    18    19    21
 3  1002 F     weight    11    16    17    20
 4  1002 F     length    19    21    23    23
 5  1003 M     weight    17    20    23    24
 6  1003 M     length    23    27    30    33
 7  1004 F     weight    16    18    21    22
 8  1004 F     length    20    22    24    26
 9  1005 M     weight    11    15    16    18
10  1005 M     length    18    20    22    23
11  1006 M     weight    17    21    25    26
12  1006 M     length    22    26    28    30
13  1007 M     weight    16    17    19    21
14  1007 M     length    21    23    24    25
15  1008 F     weight    15    16    18    19
16  1008 F     length    18    19    23    24
```

So, be careful about the ordering of the values you pass to the `names_to` argument.

32.2.8 Why person-period?

Why might we want the `babies` data in this person-period format? Well, as we discussed above, there are many analytic techniques that require our data to be in this format. Unfortunately, those techniques are beyond the scope of this chapter. However, this person-period format is still necessary for something as simple as plotting baby weight against baby height as we've done in the scatter plot below:

```
babies_long %>%
  mutate(months = factor(months, c(3, 6, 9, 12))) %>%
  ggplot() +
    geom_point(aes(weight, length, color = months)) +
    labs(
      x = "Weight (Pounds)",
      y = "Length (Inches)",
      color = "Age (Months)"
    ) +
    theme_classic()
```



32.3 Pivoting wider

As previously discussed, the person-period, or long, data structure is *usually* preferable for longitudinal data analysis. However, there are times when the person-level data structure is preferable, or even necessary. Further, there are times when we have tables of analysis results, as opposed than actual data values, that we need to restructure for ease of interpretation. We will demonstrate how to do both below.

We'll start by learning how to restructure, or reshape, our person-period `babies_long` data frame back to a person-level format. As a reminder, here is what our `babies_long` data frame currently looks like:

```
babies_long
```

```
# A tibble: 32 x 5
      id sex   months weight length
   <int> <chr>  <int>  <dbl>  <dbl>
 1  1001 F          3      9     17
 2  1001 F          6     13     18
 3  1001 F          9     16     19
 4  1001 F         12     17     21
 5  1002 F          3     11     19
 6  1002 F          6     16     21
 7  1002 F          9     17     23
 8  1002 F         12     20     23
 9  1003 M          3     17     23
10  1003 M          6     20     27
# i 22 more rows
```

As you probably guessed, we will use `tidyr`'s `pivot_wider()` function to restructure the data:

```
babies <- babies_long %>%
  pivot_wider(
    names_from  = "months",
    values_from = c("weight", "length")
  ) %>%
  print()
```

```
# A tibble: 8 x 10
     id sex   weight_3 weight_6 weight_9 weight_12 length_3 length_6 length_9
```

```
  <int> <chr>    <dbl>    <dbl>    <dbl>      <dbl>    <dbl>    <dbl>    <dbl>
1  1001 F            9       13       16         17       17       18       19
2  1002 F           11       16       17         20       19       21       23
3  1003 M           17       20       23         24       23       27       30
4  1004 F           16       18       21         22       20       22       24
5  1005 M           11       15       16         18       18       20       22
6  1006 M           17       21       25         26       22       26       28
7  1007 M           16       17       19         21       21       23       24
8  1008 F           15       16       18         19       18       19       23
# i 1 more variable: length_12 <dbl>
```

Here's what we did above:

- We used `tidyr`'s `pivot_wider()` function to restructure the `babies_long` data frame from person-period (long) to person-level (wide).
- You can type `?pivot_wider` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `pivot_wider()` function is the `data` argument. You should pass the name of the data frame you want to restructure to the `data` argument. Above, we passed the `babies_long` data frame to the `data` argument using a pipe operator.
- The third argument (we left the second argument at its default value) to the `pivot_wider()` function is the `names_from` argument. You should pass this argument the name of a column, or columns, that exists in the data frame you passed to the `data` argument. The column(s) you choose should contain values that you want to become column names in the wide data frame. That's a little be confusing, and our example above is sort of subtle, so here is a more obvious example:

```
df <- tribble(
  ~id, ~measure, ~lbs_inches,
  1, "weight", 9,
  1, "length", 17,
  2, "weight", 11,
  2, "length", 19
) %>%
  print()
```

```
# A tibble: 4 x 3
     id measure lbs_inches
  <dbl> <chr>        <dbl>
1     1 weight           9
```

```
2     1 length          17
3     2 weight          11
4     2 length          19
```

- In the data frame above, the values in the column named `measure` are what we want to use as column names in our wide data frame. Therefore, we would pass `"measure"` to the `names_to` argument of `pivot_wider()`:

```
df %>% pivot_wider(
  names_from  = "measure",
  values_from = "lbs_inches"
)
```

```
# A tibble: 2 x 3
     id weight length
  <dbl>  <dbl>  <dbl>
1     1      9     17
2     2     11     19
```

- Our `babies` example was more subtle in the sense that the long version of our data frame already had columns named `weight` and `height`. However, we essentially wanted to *change* those column names by *adding* the values from the column named `months` to the current column names. So, `weight` to `weight_3`, with the "3" coming from the column `months`.
- The ninth argument (we left the fourth through eighth arguments at their default value) to the `pivot_wider()` function is the `values_from` argument. You should pass this argument the name of a column, or columns, that exists in the data frame you passed to the `data` argument. The column(s) you choose should contain values for the new columns you want to create in the new wide data frame. In our `babies` data frame, we wanted to pull the values from the `weight` and `length` columns respectively.
- The combination of arguments (i.e., `names_from  = "months"` and `values_from = c("weight", "length")`) that we passed to `pivot_wider()` above essentially said, "make new columns from each combination of the values in the column named `months` and the column names `weight` and `length`. So, `weight_3`, `weight_6`, etc. Then, the values you put in each column should come from the intersection of `month` and `weight` (for the `weight_#`) columns, or `month` and `length` (for the `length_#`) columns.

32.3.1 Why person-level?

Why might we want the `babies` data in this person-level format? Well, as we discussed above, there are a handful analytic techniques that require our data to be in this format.

Unfortunately, those techniques are beyond the scope of this chapter. However, this person-level format is still useful for something as simple as calculating descriptive statistics about time-invariant variables. For example, the number of female and male babies in our data frame:

```
babies %>%
  count(sex)
```

```
# A tibble: 2 x 2
  sex       n
  <chr> <int>
1 F         4
2 M         4
```

32.4 Pivoting summary statistics

What do I mean by pivoting "summary statistics?" Well, in all the examples above we were manipulating the actual data values that were gathered about our observational units – babies. However, the ultimate goal of doing this kind of data management is typically to analyze it. In other words, we can often learn more from collapsing our data into a relatively small number of summary statistics than we can by viewing the actual data values themselves. Having said that, not all ways of organizing our summary statistics are equally informative. Or, perhaps it's more accurate to say that not all ways of organizing our summary statistics convey the information with equal efficiency.

There are probably a near-infinite number of possible examples of manipulating summary statistics that we could discuss. Obviously, I can't cover them all. However, I will walk through two examples below that are intended to give you a feel for what we are talking about.

32.4.1 Pivoting summary statistics wide to long

Our first example is a pretty simple one. Let's say that we are working with our person-level `babies` data frame. In this scenario, we want to calculate the mean and standard deviation of weight at the 3, 6, 9, and 12-month follow-up visits. We might do the calculations like this:

```
mean_weights <- babies %>%
  summarise(
    mean(weight_3),
    sd(weight_3),
```

```
    mean(weight_6),
    sd(weight_6),
    mean(weight_9),
    sd(weight_9),
    mean(weight_12),
    sd(weight_12),
  ) %>%
  print()
```

```
# A tibble: 1 x 8
  `mean(weight_3)` `sd(weight_3)` `mean(weight_6)` `sd(weight_6)`
             <dbl>          <dbl>            <dbl>          <dbl>
1               14           3.16               17           2.62
# i 4 more variables: `mean(weight_9)` <dbl>, `sd(weight_9)` <dbl>,
#   `mean(weight_12)` <dbl>, `sd(weight_12)` <dbl>
```

Side Note: This is not the most efficient way to do this analysis. We are only doing the analysis in this way to give us an excuse to use `pivot_longer()` to restructure some summary statistics.

By default, the mean and standard deviation are organized in a single row, side-by-side. One issue with organizing our results this way is that is that they don't all fit on the screen at the same time. However, even if they did, it's much more difficult for our brains to quickly scan the numbers and make comparisons across months when the summary statistics are organized this way than when they are stacked on top of each other. Take a look for yourself below:

```
mean_weights %>%
  pivot_longer(
    cols = everything(),
    names_to = c(".value", "measure", "months"),
    names_pattern = "(\\w+)\\((\\w+)_(\\d+)"
  )
```

```
# A tibble: 4 x 4
  measure months  mean    sd
  <chr>   <chr>  <dbl> <dbl>
1 weight  3       14    3.16
2 weight  6       17    2.62
3 weight  9       19.4  3.34
4 weight  12      20.9  3.04
```

Here's what we did above:

- We used `tidyr`'s `pivot_longer()` function to restructure our data frame of summary statistics from wide to long.
- The only new argument above is the `names_pattern` argument. You should pass a regular expression to the `names_pattern` argument. This regular expression will tell `pivot_longer()` how to break up the original column names and repurpose them for the new column names. The regular expression we used above is not intended to be the main lesson here. But, I'm sure that some of you will be curious about how it works, so I will try to briefly explain it below. In a way, this is how R interprets the regular expression above (feel free to skip if you aren't interested):

```
stringr::str_match("mean(weight_3)", "(\\w+)\\((\\w+)_(\\d+)")
```

```
     [,1]            [,2]   [,3]     [,4]
[1,] "mean(weight_3" "mean" "weight" "3"
```

- We haven't used parentheses yet in our regular expressions, but they create something called "capturing groups." Instead of saying, "look for this *one thing* in the character string," we say "look for these *groups of things* in this character string."
- The first capture group in the regular expression is `(\\w+)`. This tells R to look for one or more word characters. The value that R grabs as part of this first capture group is given under the second result (i.e., `[,2]`) above – `"mean"`.
- Then, the regular expression tells R to look for a literal open parenthesis `\\(`. However, this parenthesis is not included in a capture group. In this case, it's really just used as landmark to tell R where the first capture group stops, and the second capture group starts.
- The second capture group in the regular expression is another `(\\w+)`. This again tells R to look for one or more word characters, but this time, R starts look for the word characters *after* the open parenthesis. The value that R grabs as part of the second capture group is given under the third result (i.e., `[,3]`) above – `"weight"`.
- Next, the regular expression tells R to look for a literal underscore `_`. However, this underscore is not included in a capture group. In this case, it's really just used as landmark to tell R where the second capture group stops, and the third capture group starts.
- The third and final capture group in the regular expression is `(\\d+)`. This tells R to look for one or more digits after the underscore. The value that R grabs as part of the third capture group is given under the third result (i.e., `[,4]`) above – `"3"`.

- Finally, R matches the values it grabs in each of the three capture groups with the three values passed to the `names_to` argument, which are `".value"`, `"measure"`, and `"months"`. We already discussed the .value special value above. Similar to before, .value will create a new column for each unique value captured in the first capture group. In this case, `mean` and `sd`. Next, the values captured in the second capture group are assigned to a column named `measure`. Finally, the values captured in the third capture group are assigned to a column named `months`.

32.4.2 Pivoting summary statistics long to wide

This next example comes from an actual project I was involved with. As a part of this project, researchers asked the parents of elementary-aged children about series of sun protection behaviors. Below, I'm not simulating the data that was collected. Rather, I am simulating a small part of the results of one of the early descriptive analyses we conducted:

```
summary_stats <- tribble(
  ~period, ~behavior, ~value, ~n, ~n_total, ~percent,
  "School Year Weekends", "Long sleeve shirt", "Never", 6, 78,  8,
  "School Year Weekends", "Long sleeve shirt", "Seldom", 16, 78,    21,
  "School Year Weekends", "Long sleeve shirt", "Sometimes", 33, 78, 42,
  "School Year Weekends", "Long sleeve shirt", "Often", 17, 78, 22,
  "School Year Weekends", "Long sleeve shirt", "Always", 6, 78, 8,
  "School Year Weekends", "Long Pants", "Never", 5, 79, 6,
  "School Year Weekends", "Long Pants", "Seldom",   15, 79, 19,
  "School Year Weekends", "Long Pants", "Sometimes", 32, 79, 41,
  "School Year Weekends", "Long Pants", "Often", 19, 79, 24,
  "School Year Weekends", "Long Pants", "Always",   8, 79, 10,
  "Summer", "Long sleeve shirt", "Never",   9, 80, 11,
  "Summer", "Long sleeve shirt", "Seldom", 18, 80, 22,
  "Summer", "Long sleeve shirt", "Sometimes", 31,   80, 39,
  "Summer", "Long sleeve shirt", "Often",   14, 80, 18,
  "Summer", "Long sleeve shirt", "Always", 8,   80, 10,
  "Summer", "Long Pants", "Never", 7,   76, 9,
  "Summer", "Long Pants", "Seldom", 16, 76, 21,
  "Summer", "Long Pants", "Sometimes", 27,  76, 36,
  "Summer", "Long Pants", "Often", 18, 76,  24,
  "Summer", "Long Pants", "Always", 8, 76,  11
) %>%
  print()
```

```
# A tibble: 20 x 6
```

	period	behavior	value	n	n_total	percent
	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>
1	School Year Weekends	Long sleeve shirt	Never	6	78	8
2	School Year Weekends	Long sleeve shirt	Seldom	16	78	21
3	School Year Weekends	Long sleeve shirt	Sometimes	33	78	42
4	School Year Weekends	Long sleeve shirt	Often	17	78	22
5	School Year Weekends	Long sleeve shirt	Always	6	78	8
6	School Year Weekends	Long Pants	Never	5	79	6
7	School Year Weekends	Long Pants	Seldom	15	79	19
8	School Year Weekends	Long Pants	Sometimes	32	79	41
9	School Year Weekends	Long Pants	Often	19	79	24
10	School Year Weekends	Long Pants	Always	8	79	10
11	Summer	Long sleeve shirt	Never	9	80	11
12	Summer	Long sleeve shirt	Seldom	18	80	22
13	Summer	Long sleeve shirt	Sometimes	31	80	39
14	Summer	Long sleeve shirt	Often	14	80	18
15	Summer	Long sleeve shirt	Always	8	80	10
16	Summer	Long Pants	Never	7	76	9
17	Summer	Long Pants	Seldom	16	76	21
18	Summer	Long Pants	Sometimes	27	76	36
19	Summer	Long Pants	Often	18	76	24
20	Summer	Long Pants	Always	8	76	11

- The `period` column contains the time frame the researchers were asking the parents about. It can take the values `School Year Weekends` or `Summer`.
- The `behavior` column contains each of the specific behaviors that the researchers were interested in. Above, `behavior` takes only the values `Long sleeve shirt` and `Long Pants`.
- The `value` column contains the possible answer choices that parents could select from.
- The `n` column contains the number of parents who selected the response in `value` for the behavior in `behavior` and the time frame in `period`. For example, `n` = 6 in the first row indicates that six parents said that their child never wears long sleeve shirts on weekends during the school year.
- The `n_total` column is the sum of `n` for each period/behavior combination.
- The `percent` column contains the percentage of parents who selected the response in `value` for the behavior in `behavior` and the time frame in `period`. For example, `percent` = 8 in the first row indicates that 8 percent of parents said that their child never wears long sleeve shirts on weekends during the school year.

These results are relatively difficult to scan and get a feel for. In particular, these researchers were interested in whether or not engagement in these protective behaviors differed by period.

In other words, were kids more likely to wear long sleeve shirts on weekends during the school year than they were during the summer? It's difficult to answer that quickly with the way the summary statistics above are organized. We can improve the interpretability of our results by combining `n` and `percent` into a single character string, and pivoting them wider so that the two periods are presented side-by-side:

```
summary_stats %>%
  # Combine n and percent into a single character string
  mutate(n_percent = paste0(n, " (", percent, ")")) %>%
  # We no longer need n, n_total, percent
  select(-n:-percent) %>%
  pivot_wider(
    names_from = "period",
    values_from = "n_percent"
  )
```

```
# A tibble: 10 x 4
   behavior          value     `School Year Weekends` Summer
   <chr>             <chr>     <chr>                  <chr>
 1 Long sleeve shirt Never     6 (8)                  9 (11)
 2 Long sleeve shirt Seldom    16 (21)                18 (22)
 3 Long sleeve shirt Sometimes 33 (42)                31 (39)
 4 Long sleeve shirt Often     17 (22)                14 (18)
 5 Long sleeve shirt Always    6 (8)                  8 (10)
 6 Long Pants        Never     5 (6)                  7 (9)
 7 Long Pants        Seldom    15 (19)                16 (21)
 8 Long Pants        Sometimes 32 (41)                27 (36)
 9 Long Pants        Often     19 (24)                18 (24)
10 Long Pants        Always    8 (10)                 8 (11)
```

The layout of our summary statistics above is now much more compact. Further, it's much easier to compare behaviors between the two time periods. For example, we can see that a slightly higher percentage of people (11%) reported that their child never wears a long sleeve shirt during the summer as compared to weekends during the school year (8%).

32.5 Tidy data

As I said above, the person-level (wide) and person-period (long) data structures are the *traditional* way of classifying how longitudinal (or repeated measures) data are organized. In reality, however, structuring data in a way that is most conducive to analysis is often more

complicated than the examples above would lead you to believe. Simply thinking about data structure in terms of wide and long sometimes leaves us with an incomplete model for how to take many real-world data sets and prepare them for conducting analysis in an efficient way. In his seminal paper on the topic, Hadley Wickham, provides us with a set of guidelines for systematically (re)structuring our data in a way that is consistent, and generally optimized for analysis. He refers to this process as "tidying" our data, and to the resulting data frame as "tidy data".[8]

> **Note**
>
> If you are interested, you can download the entire article for free from the Journal of statistical Software here.

The three basic guidelines for tidy data are:

1. Each variable (i.e., measurement or characteristic about the observational unit) must have its own column.
2. Each observation (i.e. the people, places, or things we are interested in characterizing or comparing *at a particular occasion*) must have its own row.
3. Each value must have its own cell.

According to the tidy data philosophy, any data frame that does not conform to the guidelines above is considered "messy" data. In my opinion, it's kind of hard to read the guidelines above and wrap your head around what tidy data is. I think it's actually easier to get a *feel* for tidy data by looking at examples of data that are not tidy. Let's go ahead and take a look at a few examples:

32.5.1 Each variable must have its own column

What does it mean for every variable to have its own column? Well, let's say we interested the rate of neural tube defects by state. So, we pull some data from a government website that looks like this:

```
births_ntd <- tibble(
  state   = rep(c("CA", "FL", "TX"), each = 2),
  outcome = rep(c("births", "neural tube defects"), 3),
  count   = c(454920, 318, 221542, 155, 378624, 265)
) %>%
  print()
```

```
# A tibble: 6 x 3
  state outcome              count
  <chr> <chr>                <dbl>
1 CA    births              454920
2 CA    neural tube defects    318
3 FL    births              221542
4 FL    neural tube defects    155
5 TX    births              378624
6 TX    neural tube defects    265
```

In this case, there is only one `count` column, but that column really contains two variables: the count of live births and the count of neural tube defects. Further, the `outcome` column doesn't really contain "data." In this case, the values stored in the `outcome` column are really data *labels*. We can tidy this data using the `pivot_wider()` function:

```
births_ntd %>%
  pivot_wider(
    names_from  = "outcome",
    values_from = "count"
  )
```

```
# A tibble: 3 x 3
  state births `neural tube defects`
  <chr>  <dbl>                 <dbl>
1 CA    454920                   318
2 FL    221542                   155
3 TX    378624                   265
```

Now, `births` and `neural tube defects` each have their own column. It might also be a good idea to remove the spaces from `neural tube defects` and make it clear that the values in each column are counts. But, I'm going to leave that to you.

Another common violation of the "each variable must have its own column" guideline is when column names contain data values. We already saw an example of this above. Our `weight_` and `length_` column names actually had time data embedded in them.

In the example below, each column name contains two data values (i.e., sex and year); however, neither variable currently has a column in the data:

```
births_sex <- tibble(
  state  = c("CA", "FL", "TX"),
  f_2018 = c(222911, 108556, 185526),
```

```
  m_2018 = c(232009, 112986, 193098)
) %>%
  print()
```

```
# A tibble: 3 x 3
  state f_2018 m_2018
  <chr>  <dbl>  <dbl>
1 CA    222911 232009
2 FL    108556 112986
3 TX    185526 193098
```

In this case, we can tidy the data by giving `sex` and `year` a column, and giving the other data values (i.e., count of live births) a more informative column name. We can do so with the `pivot_longer()` function:

```
births_sex %>%
  pivot_longer(
    cols      = -state,
    names_to  = c("sex", "year"),
    names_sep = "_",
    values_to = "births"
  )
```

```
# A tibble: 6 x 4
  state sex   year  births
  <chr> <chr> <chr>  <dbl>
1 CA    f     2018  222911
2 CA    m     2018  232009
3 FL    f     2018  108556
4 FL    m     2018  112986
5 TX    f     2018  185526
6 TX    m     2018  193098
```

32.5.2 Each observation must have its own row

Our person-level `babies` data frame above also violated this guideline.

```
babies
```

```
# A tibble: 8 x 10
     id sex   weight_3 weight_6 weight_9 weight_12 length_3 length_6 length_9
  <int> <chr>    <dbl>    <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1  1001 F            9       13       16        17       17       18       19
2  1002 F           11       16       17        20       19       21       23
3  1003 M           17       20       23        24       23       27       30
4  1004 F           16       18       21        22       20       22       24
5  1005 M           11       15       16        18       18       20       22
6  1006 M           17       21       25        26       22       26       28
7  1007 M           16       17       19        21       21       23       24
8  1008 F           15       16       18        19       18       19       23
# i 1 more variable: length_12 <dbl>
```

Notice that each baby in this data has one `row`, but that each row actually contains four unique observations – at 3, 6, 9, and 12 months. As another example, let's say that we've once again downloaded birth count data from a government website. This time, we are interested in investigating the absolute change in live births over the decade between 2010 and 2020. That data may look like this:

```
births_decade <- tibble(
  state  = c("CA", "FL", "TX"),
  `2010` = c(409428, 199388, 340762),
  `2020` = c(454920, 221542, 378624)
) %>%
  print()
```

```
# A tibble: 3 x 3
  state `2010` `2020`
  <chr>  <dbl>  <dbl>
1 CA    409428 454920
2 FL    199388 221542
3 TX    340762 378624
```

In this example, each state has a single row, but multiple observations. We can once again tidy this data using the `pivot_longer()` function:

```
births_decade %>%
  pivot_longer(
    cols      = -state,
    names_to  = "year",
    values_to = "births"
  )
```

```
# A tibble: 6 x 3
  state year  births
  <chr> <chr>  <dbl>
1 CA    2010  409428
2 CA    2020  454920
3 FL    2010  199388
4 FL    2020  221542
5 TX    2010  340762
6 TX    2020  378624
```

32.5.3 Each value must have its own cell

In my personal experience, violations of this guideline are rarer than violations of the first two guidelines. However, let's imagine a study where we are monitoring the sleeping habits of newborn babies. Specifically, we are interested in the range of lengths of time they sleep. That data could be recorded the following way:

```
baby_sleep <- tibble(
  id          = c(1001, 1002, 1003),
  sleep_range = c(".5-2", ".75-2.4", "1.1-3.8")
) %>%
  print()
```

```
# A tibble: 3 x 2
     id sleep_range
  <dbl> <chr>
1  1001 .5-2
2  1002 .75-2.4
3  1003 1.1-3.8
```

In this case, we will use a new function to tidy our data. We will use `tidyr`'s `separate()` function to spread these values out across two columns:

```
baby_sleep %>%
  separate(
    col     = sleep_range,
    into    = c("min_hours", "max_hours"),
    sep     = "-",
    convert = TRUE
  )
```

```
# A tibble: 3 x 3
     id min_hours max_hours
  <dbl>     <dbl>     <dbl>
1  1001       0.5         2
2  1002       0.75        2.4
3  1003       1.1         3.8
```

Here's what we did above:

- We used `tidyr`'s `separate()` function to tidy the `baby_sleep` data frame.
- You can type `?separate` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `separate()` function is the `data` argument. You should pass the name of the data frame you want to restructure to the `data` argument. Above, we passed the `baby_sleep` data frame to the `data` argument using a pipe operator.
- The second argument to the `separate()` function is the `col` argument. You should pass the name of the column contain the data values that you want to split up to the `col` argument.
- The third argument to the `separate()` function is the `into` argument. You should pass the `into` argument a character vector of column names you want to give the new columns that will be created when you break apart the values in the `col` column.
- The fourth argument to the `separate()` function is the `sep` argument. You should pass the `sep` argument a character string that tells `separate()` what character separates the individual values in the `col` column.
- Finally, we passed the value `TRUE` to the `convert` argument. In doing so, we asked `separate()` to coerce the values in `min_hours` and `max_hours` from character type to numeric type.

32.6 The complete() function

The final function we're going to discuss in this chapter is `tidyr`'s `complete()` function. After we pivot data, we will sometimes notice "holes" in the data. This typically happens to me in the context of time data. When this happens, we can use the `complete()` function to fill-in the holes in our data.

This next example didn't actually involve pivoting, but it did come from another actual project that I was involved with, and nicely demonstrates the importance of filling-in holes in the data. As a part of this project, researchers were interested in increasing the number of reports of elder mistreatment that were being made to Adult Protective Services (APS) by emergency medical

technicians (EMTs) and paramedics. Each row in the raw data the researchers received from the emergency medical services provider represented a report to APS. Let's say that the data from the week of October 28th, 2019 to November 3rd, 2019 looked something like this:

```
reports <- tibble(
  date      = as.Date(c(
    "2019-10-29", "2019-10-29", "2019-10-30", "2019-11-02", "2019-11-02"
  )),
  emp_id    = c(5123, 2224, 5153, 9876, 4030),
  report_id = c("a8934", "af2as", "jzia3", "3293n", "dsf98")
) %>%
  print()
```

```
# A tibble: 5 x 3
  date       emp_id report_id
  <date>      <dbl> <chr>
1 2019-10-29   5123 a8934
2 2019-10-29   2224 af2as
3 2019-10-30   5153 jzia3
4 2019-11-02   9876 3293n
5 2019-11-02   4030 dsf98
```

Where:

- `date` is the date the report was made to APS.
- `emp_id` is a unique identifier for each EMT or paramedic.
- `report_id` is the unique identifier APS assigns to the incoming report.

Let's say that the researchers were interested in calculating the average number of reports per day. We would first need to count the number of reports made each day:

```
reports %>%
  count(date)
```

```
# A tibble: 3 x 2
  date           n
  <date>     <int>
1 2019-10-29     2
2 2019-10-30     1
3 2019-11-02     2
```

Next, we might naively go ahead and calculate the mean of n like this:

```
reports %>%
  count(date) %>%
  summarise(mean_reports_per_day = mean(n))
```

```
# A tibble: 1 x 1
  mean_reports_per_day
                 <dbl>
1                 1.67
```

And conclude that the mean number of reports made per day was 1.67. However, there is a problem with this strategy. Our study period wasn't three days long. It was seven days long (i.e., October 28th, 2019 to November 3rd, 2019). Because there weren't any reports made on 2019-10-28, 2019-10-31, 2019-11-01, or 2019-11-03 they don't exist in our count data. But, their absence doesn't represent a missing or unknown value. Their absence represents zero reports being made on that day. We need to explicitly encode that information in our count data if we want to accurately calculate the mean number of reports per day. In this tiny little simulated data frame, it's trivial to do this calculation manually. However, the real data set was collected over a three-year period. That's over 1,000 days that would have to be manually accounted for.

Luckily, we can use `tidyr`'s `complete()` function, along with the `seq.Date()` function we learned in the chapter on working with date variables, to fill-in the holes in our count data in an automated way:

```
reports %>%
  count(date) %>%
  complete(
    date = seq.Date(
      from = as.Date("2019-10-28"),
      to = as.Date("2019-11-03"),
      by = "days"
    )
  )
```

```
# A tibble: 7 x 2
  date           n
  <date>     <int>
1 2019-10-28    NA
2 2019-10-29     2
3 2019-10-30     1
```

```
4 2019-10-31    NA
5 2019-11-01    NA
6 2019-11-02     2
7 2019-11-03    NA
```

Here's what we did above:

- We used `tidyr`'s `complete()` function to fill-in the holes in the dates between 2019-10-28 and 2019-11-03.
- You can type `?complete` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to the `complete()` function is the `data` argument. You should pass the name of the data frame that contains the column you want to fill-in to the `data` argument. Above, we passed the `reports` data frame to the `data` argument using a pipe operator.
- The second argument to the `complete()` function is the `...` argument. This is where you tell the `complete()` function which column you want to fill-in, or expand, and give it instructions for doing so. Above, we asked `complete()` to make sure that each day between 2019-10-28 and 2019-11-03 was included in our `date` column. We did so by asking `complete()` to set the date column equal to the returned values from the `seq.Date()` function.

Notice that all the days during our period of interest are now included in our count data. However, by default, the value for each new row of the `n` column is set to `NA`. But, as we already discussed, `n` isn't missing for those days, it's zero. We can change those values from `NA` to zero by adjusting the value we pass to the `fill` argument. We'll do that next:

```
reports %>%
  count(date) %>%
  complete(
    date = seq.Date(
      from = as.Date("2019-10-28"),
      to = as.Date("2019-11-03"),
      by = "days"
    ),
    fill = list(n = 0)
  )
```

```
# A tibble: 7 x 2
  date           n
  <date>     <int>
```

```
1 2019-10-28     0
2 2019-10-29     2
3 2019-10-30     1
4 2019-10-31     0
5 2019-11-01     0
6 2019-11-02     2
7 2019-11-03     0
```

Now, we can finally calculate the correct value for mean number of reports made per day during the week of October 28th, 2019 to November 3rd, 2019:

```
reports %>%
  count(date) %>%
  complete(
    date = seq.Date(
      from = as.Date("2019-10-28"),
      to = as.Date("2019-11-03"),
      by = "days"
    ),
    fill = list(n = 0)
  ) %>%
  summarise(mean_reports_per_day = mean(n))
```

```
# A tibble: 1 x 1
  mean_reports_per_day
                 <dbl>
1                0.714
```

That concludes the chapter on restructuring data. For now, it also concludes the part of this book devoted to the basics of data management. At this point, you should have the tools you need to tackle the majority of the common data management tasks that you will come across. Further, there's a good chance that the packages we've used in this part of the book will contain a solution for the remaining data management challenges that we haven't explicitly covered. In the next part of the book, we will dive into repeated operations.

Part VI

Repeated Operations

33 Introduction to Repeated Operations

This part of the book is all about the DRY principle. We first discussed the DRY principle in the section on creating and modifying multiple columns. As a reminder, DRY is an acronym for "Don't Repeat Yourself." But, what does that mean?

Well, think back to the conditional operations chapter. In that chapter, we compared conditional statements in R with asking our daughters to wear a raincoat if it's raining. To extend the analogy, now imagine that we wake up one morning and say, "please wear your raincoat if it's raining today - July 1st." Then, we wake up the next morning and say, "please wear your raincoat if it's raining today - July 2nd." Then, we wake up the next morning and say, "please wear your raincoat if it's raining today - July 3rd." And, that pattern continues every morning until our daughters move out of the house. That's a ton of repetition!! Alternatively, wouldn't it be much more efficient to say, "please wear your raincoat on every day that it rains," just once?

The same logic applies to our R code. We often want to do the same (or very similar) thing multiple times. This can result in many lines of code that are very similar and unnecessarily repetitive, and this unnecessary repetition can occur in all phases of our projects.

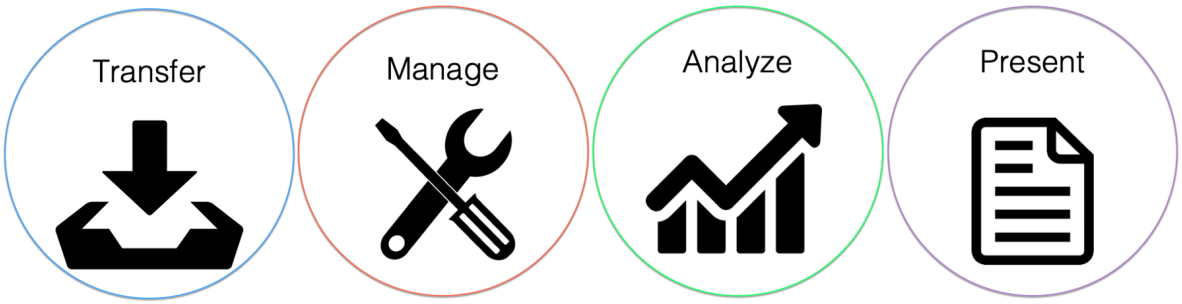


Figure 33.1: Project phases

For example:

- We may need to write R code to import many different data sets. In such a situation, it isn't uncommon for the code that imports the data to be the same for each data set – only the file name changes.
- We may need to recode certain values in multiple columns of our data frame to missing. In such a situation, it isn't uncommon for the code that recodes the values to be the same for each column – only the column name changes.
- We may need to calculate the same set of statistical measures for many different variables in our data frame. In such a situation, the code to calculate the statistical measures doesn't change – only the variables being passed to the code.
- We may need to create a table of results that includes statistical measures for many different variables in our data frame. In such a situation, the code to prepare and combine the statistical measures into a single table of results doesn't change – only the variables being passed to the code.

In all of these situations we are asking our R code to do something repeatedly, or iteratively, but with a slight change each time. We can write a separate chunk of code for each time we want to do that thing, or we can write one chunk of code that asks R to do that thing over and over. Writing code in the later way will often result in R programs that:

- *Are more concise.* In other words, we can write one line of code (or relatively few lines of code) instead of many lines of code. Further, such code generally removes "visual clutter" (i.e., the repetitive stuff) that can obscure what the overarching *intent* of the code.
- *Contain fewer typos.* Every keystroke we make is an opportunity to press the wrong key. If we are writing fewer lines of code, then it logically follows that we are making fewer keystrokes and creating fewer opportunities to hit the wrong key. Similarly, if we are repeatedly copying and pasting code, we are creating opportunities to accidentally forget to change a column name, date, file name, etc. in the pasted code.
- *Are easier to maintain.* If we want to change our code, we only have to change it in one place instead of many places. For example, let's say that we write R code to check the weather every morning. Later, we decide that we want our R code to check the weather *and* the traffic every morning. Would you rather add that additional request (i.e., check the traffic) to a separate line of code for each day or to the one line of code that asks R to check the weather every day?

> **Note**
>
> When we say "one line of code" above, we mean it figuratively. The code we use to remove unnecessary repetition will not necessarily be on one line; however, it should generally require less typing than code that includes unnecessary repetition.

So, writing code that is highly repetitive is usually not a great idea, and this part of the book is all about teaching you to recognize and remove unnecessary repetition from your code. As is often the case with R, there are multiple different methods we can use.

33.1 Multiple methods for repeated operations in R

In the chapters that follow, we will learn four different methods for removing unnecessary repetition from our code. They are:

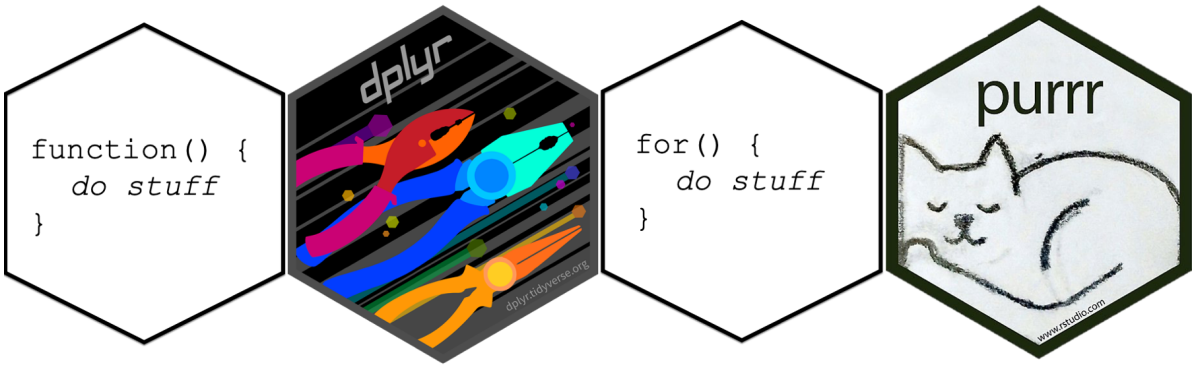


Figure 33.2: Four methods for removing unnecessary repetition

1. Writing our own functions that can be reused throughout our code.
2. Using `dplyr`'s column-wise operations.
3. Using for loops.
4. Using the `purrr` package.

It's also important to recognize that each of the methods above can be used independently or in combination with each other. We will see examples of both.

33.2 Tidy evaluation

In case it isn't obvious to you by now, we're fans of the `tidyverse` packages (i.e., `dplyr`, `ggplot2`, `tidyr`, etc.). We use `dplyr`, in particular, in virtually every single one of our R programs. The use of **non-standard evaluation** is just one of the many aspects of the `tidyverse` packages that we're fans of. As a reminder, among other things, non-standard evaluation is what allows us to refer to data frame columns without using dollar sign or bracket notation (i.e., data masking). However, non-standard evaluation will create some challenges for us when we try to use functions from `tidyverse` packages inside of functions and for loops that we write ourselves. Therefore, we will have to learn more about *tidy evaluation* if we

want to continue to use the `tidyverse` packages that we've been using throughout the book so far.

Tidy evaluation can be tricky even for experienced R programmers to wrap their heads around at first. Therefore, it might not be productive for us to try to learn a lot about the theory behind, or internals of, tidy evaluation as a standalone concept. Instead, in the chapters that follow, we plan to sprinkle in just enough tidy evaluation to accomplish the task at hand. As a little preview, a telltale sign that we are using tidy evaluation will be when you start seeing the `{{` (said, curly-curly) operator and the `!!` (said, bang bang) operator. Hopefully, this will all make more sense in the next chapter when we start to get into some examples.

We recommend the following resources for those of you who are interested in developing a deeper understanding of `rlang` and tidy evaluation:

1. Programming with dplyr. Accessed July 31, 2020. https://dplyr.tidyverse.org/articles/programming.html
2. Wickham H. Introduction. In: Advanced R. Accessed July 31, 2020. https://adv-r.hadley.nz/metaprogramming.html

Now, let's learn how to write our own functions!

34 Writing Functions

Have you noticed how we will often calculate the same statistical measures for many different variables in our data? For example, let's say that we have some pretty standard data about some study participants that looks like this:

```
library(dplyr)
```

```
study <- tibble(
  age       = c(32, 30, 32, 29, 24, 38, 25, 24, 48, 29, 22, 29, 24, 28, 24, 25,
                25, 22, 25, 24, 25, 24, 23, 24, 31, 24, 29, 24, 22, 23, 26, 23,
                24, 25, 24, 33, 27, 25, 26, 26, 26, 26, 26, 27, 24, 43, 25, 24,
                27, 28, 29, 24, 26, 28, 25, 24, 26, 24, 26, 31, 24, 26, 31, 34,
                26, 25, 27, NA),
  age_group = c(2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2,
                2, 1, 1, 1, NA),
  gender    = c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1,
                1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1,
                1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 1,
                1, 1, 2, 1, NA),
  ht_in     = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
                64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
                64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
                69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
                61, 69, 66, NA),
  wt_lbs    = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
                125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
                186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
                147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
                110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
                163, 141, NA),
  bmi       = c(30.99, 18.78, 26.52, 30.54, 22.39, 26.12, 23.69, 20.67, 26.29,
                25.39, 25.68, 45.15, 21.56, 20.17, 17.38, 20.8, 22.31, 22.75,
                26.62, 21.43, 19.14, 23.49, 22.66, 32.98, 25.05, 18.31, 29.13,
```

```
                27.07, 20.37, 25.01, 19.69, 25.97, 18.88, 20.07, NA, 26.76,
                26.97, 25.24, 20.68, 23.72, 24.82, 23.62, 18.65, 24.03, 23.86,
                10.63, 23.02, 23.72, 20.82, 28.24, NA, 37.55, 18.88, 18.3,
                19.13, 21.48, 22.59, 24.96, 21.63, NA, 29.35, 21.76, 17.97,
                22.31, 19.27, 24.07, 22.76, NA),
  bmi_3cat  = c(3, 1, 2, 3, 1, 2, 1, 1, 2, 2, 2, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1,
                1, 1, 3, 2, 1, 2, 2, 1, 2, 1, 2, 1, 1, NA, 2, 2, 2, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 2, NA, 3, 1, 1, 1, 1, 1, 1, 1, NA, 2, 1,
                1, 1, 1, 1, 1, NA)
) %>%
  mutate(
    age_group = factor(age_group, labels = c("Younger than 30", "30 and Older")),
    gender    = factor(gender, labels = c("Female", "Male")),
    bmi_3cat  = factor(bmi_3cat, labels = c("Normal", "Overweight", "Obese"))
  ) %>%
  print()
```

```
# A tibble: 68 x 7
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese
 2    30 30 and Older    Female    63    106  18.8 Normal
 3    32 30 and Older    Female    62    145  26.5 Overweight
 4    29 Younger than 30 Male      67    195  30.5 Obese
 5    24 Younger than 30 Female    67    143  22.4 Normal
 6    38 30 and Older    Female    58    125  26.1 Overweight
 7    25 Younger than 30 Female    64    138  23.7 Normal
 8    24 Younger than 30 Male      69    140  20.7 Normal
 9    48 30 and Older    Male      65    158  26.3 Overweight
10    29 Younger than 30 Male      68    167  25.4 Overweight
# i 58 more rows
```

When we have data like this, it's pretty common to calculate something like the number of missing values, mean, median, min, and max for all of the continuous variables. So, we might use the following code to calculate these measures:

```
study %>%
  summarise(
    n_miss = sum(is.na(age)),
    mean   = mean(age, na.rm = TRUE),
    median = median(age, na.rm = TRUE),
```

```
    min    = min(age, na.rm = TRUE),
    max    = max(age, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      1  26.9     26    22    48
```

Great! Next, we want to do the same calculations for `ht_in`. Of course, we don't want to type everything in that code chunk again, so we copy and paste. And change all the instances of `age` to `ht_in`:

```
study %>%
  summarise(
    n_miss = sum(is.na(ht_in)),
    mean   = mean(ht_in, na.rm = TRUE),
    median = median(ht_in, na.rm = TRUE),
    min    = min(ht_in, na.rm = TRUE),
    max    = max(ht_in, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      3  66.0     66    58    76
```

Now, let's do the same calculations for `wt_lbs` and `bmi`. Again, we will copy and paste, and change the variable name as needed:

```
study %>%
  summarise(
    n_miss = sum(is.na(wt_lbs)),
    mean   = mean(wt_lbs, na.rm = TRUE),
    median = median(wt_lbs, na.rm = TRUE),
    min    = min(ht_in, na.rm = TRUE),
    max    = max(wt_lbs, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      2  148.   142.    58   297
```

```
study %>%
  summarise(
    n_miss = sum(is.na(bmi)),
    mean   = mean(bmi, na.rm = TRUE),
    median = median(bmi, na.rm = TRUE),
    min    = min(bmi, na.rm = TRUE),
    max    = max(bmi, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      4  23.6   22.9  10.6  45.2
```

And, we're done!

However, there's a problem. Did you spot it? We accidentally forgot to change `ht_in` to `wt_lbs` in the min calculation above. Therefore, our results incorrectly indicate that the minimum weight was 58 lbs. Part of the reason for making this mistake in the first place is that there is a fair amount of visual clutter in each code chunk. In other words, it's hard to quickly scan each chunk and see only the elements that are *changing*.

Additionally, each code chunk was about 8 lines of code. Even with only 4 variables, that's still 32 lines. We can improve on this code by writing our own function. That's exactly what we will do in the code chunk below. For now, don't worry if you don't understand *how* the code works. We will dissect it later.

```
continuous_stats <- function(var) {
  study %>%
    summarise(
      n_miss = sum(is.na({{ var }})),
      mean   = mean({{ var }}, na.rm = TRUE),
      median = median({{ var }}, na.rm = TRUE),
      min    = min({{ var }}, na.rm = TRUE),
      max    = max({{ var }}, na.rm = TRUE)
    )
}
```

Now, let's *use* the function we just created above to once again calculate the descriptive measures we are interested in.

```
continuous_stats(age)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      1  26.9     26    22    48
```

```
continuous_stats(ht_in)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      3  66.0     66    58    76
```

```
continuous_stats(wt_lbs)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      2  148.   142.    60   297
```

```
continuous_stats(bmi)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      4  23.6   22.9  10.6  45.2
```

Pretty cool, right? We reduced 32 lines of code to 13 lines of code! Additionally, it's very easy to quickly scan our code and see that the only thing changing from chunk-to-chunk is the name of the variable that we are passing to our function and ensure that it is *actually* changing. As an added bonus, because we've strategically given our function an informative name, the intent behind what we are trying to accomplish is clearer now – we are calculating summary statistics about our continuous variables.

Hopefully, this little demonstration has left you feeling like writing your own functions can be really useful, and maybe even kind of fun. We're going to get into the nuts and bolts of *how* to write your own functions shortly, but first let's briefly discuss *when* to write your own functions.

34.1 When to write functions

Hadley Wickham, prolific R developer and teacher says, "You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)."[9] We completely agree with this general sentiment. We'll only amend our advice to you slightly. Specifically, you should consider using an appropriate method for repeating operations whenever you've copied and pasted a block of code more than twice. In other words, *writing a function* is not the *only* option available to us when we notice ourselves copying and pasting code.

34.2 How to write functions

Now, the fun part – writing our own functions. Writing functions can seem intimidating to many people at first. However, the basics are actually pretty simple.

34.2.1 The function() function

It all starts with the `function()` function. This is how you tell R that you are about to write your own function.

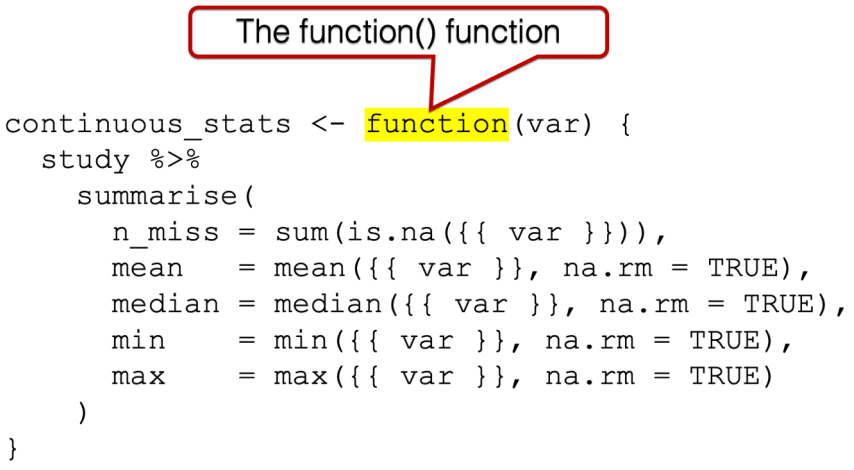


Figure 34.1: The function() function.

If you think back to the chapter on Speaking R's language, we talked about the analogy that is sometimes drawn between functions and factories.

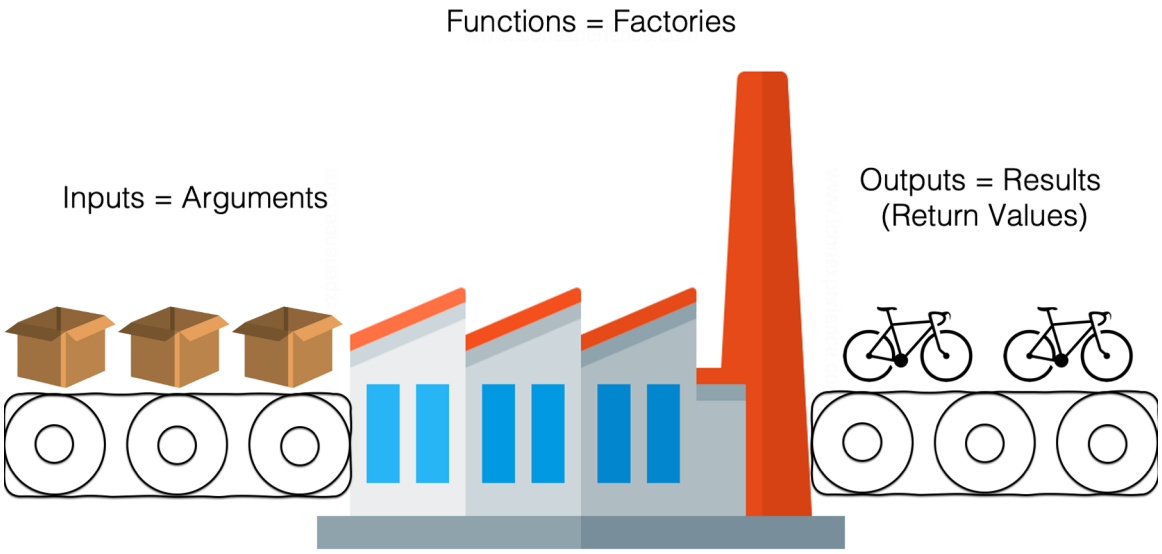


Figure 34.2: A factory making bicycles.

To build on that analogy, the`function()` function is sort of like the factory building. Without it, there is no factory, but an empty building alone doesn't do anything interesting:

```
function()
```

```
Error in parse(text = input): <text>:2:0: unexpected end of input
1: function()
   ^
```

In order to build our bicycles, we need to add some workers and equipment to our empty factory building. The R function equivalent to the workers and equipment is the **function body.**

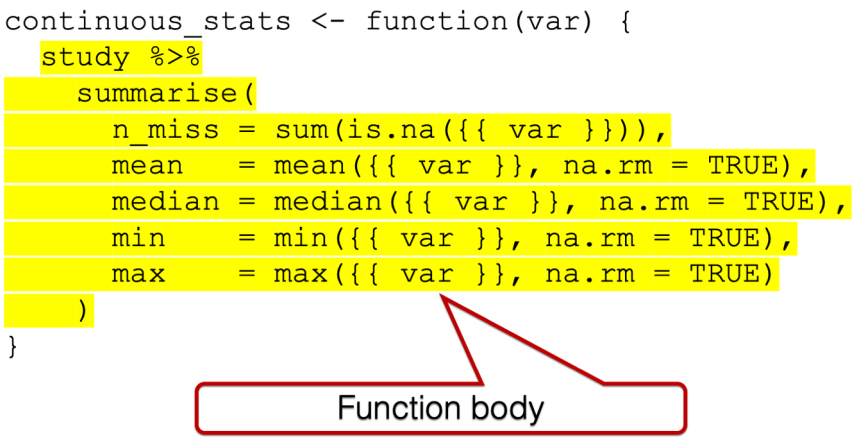


Figure 34.3: The function body.

And just like the factory needs doors to contain our workers and equipment and keep them safe (This is admittedly a bit of a reach, but just go with it), our function body needs to be wrapped with curly braces.

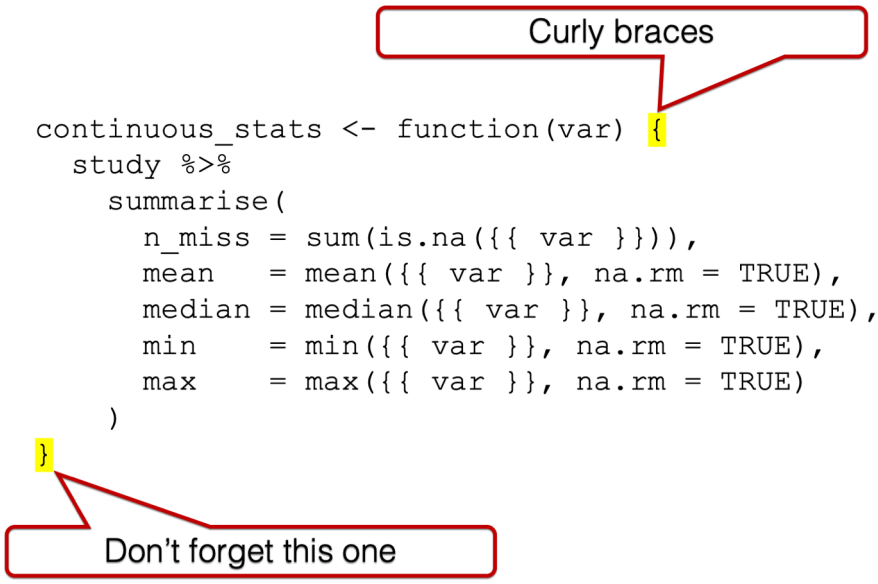


Figure 34.4: Curly braces around the function body.

We already talked about how the values we pass to arguments are raw material inputs that go into the factory.

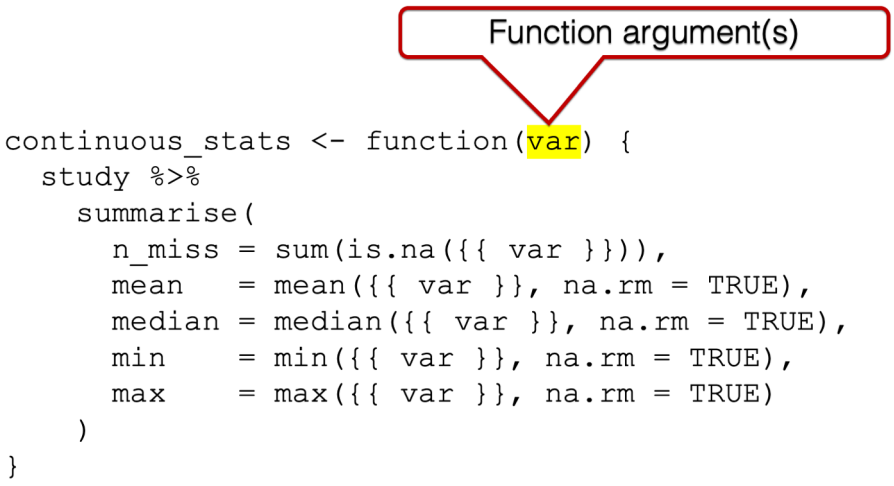


Figure 34.5: The function argument(s).

In the bicycle factory example, the raw materials were steel and rubber. In the function displayed above, the raw materials are variables.

If we want to be able to call our function (i.e., use it) later, then we have to have some way to refer to it. Therefore, we will assign our function a name.

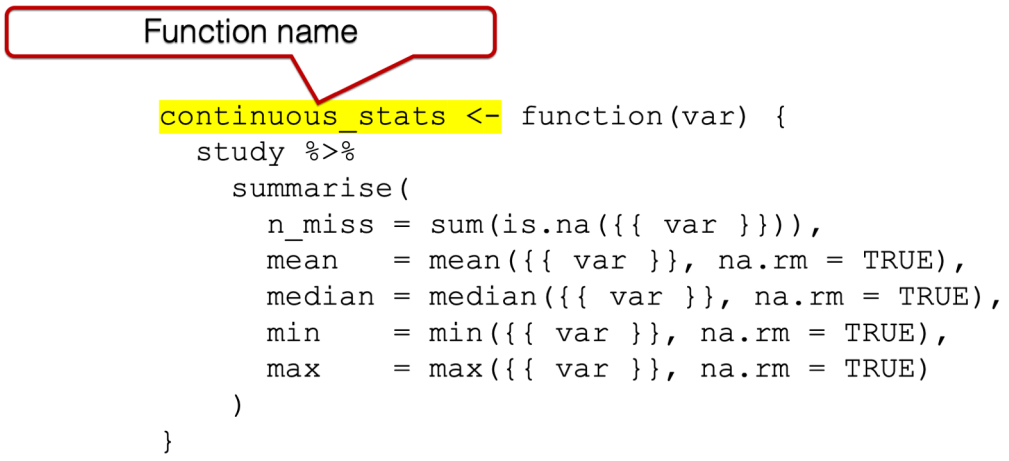


Figure 34.6: The named function.

34.2.2 The function writing process

So, we have some idea about *why* writing our own functions can be a good idea. We have some idea about *when* to write functions (i.e., don't repeat yourself… more than twice). And, we now know what the basic components of functions are. They are the `function()` function, the function body (wrapped in curly braces), the function argument(s), and the function name. But, if this is your first time being exposed to functions, then you may still be feeling like you aren't quite sure how to get started with writing your own. So, here's a little example of how a function writing workflow could go.

First, let's simulate some new data for this example. Let's say we have two data frames that contain first and last names:

```
people_1 <- tribble(
  ~id_1, ~name_first_1, ~name_last_1, ~street_1,
  1,     "Easton",      NA,           "Alameda",
  2,     "Elias",       "Salazar",    "Crissy Field",
  3,     "Colton",      "Fox",        "San Bruno",
  4,     "Cameron",     "Warren",     "Nottingham",
  5,     "Carson",      "Mills",      "Jersey",
  6,     "Addison",     "Meyer",      "Tingley",
```

```
  7,     "Aubrey",      "Rice",       "Buena Vista",
  8,     "Ellie",       "Schmidt",    "Division",
  9,     "Robert",      "Garza",      "Red Rock",
  10,    "Stella",      "Daniels",    "Holland"
) %>%
  print()
```

```
# A tibble: 10 x 4
    id_1 name_first_1 name_last_1 street_1
   <dbl> <chr>        <chr>       <chr>
 1     1 Easton       <NA>        Alameda
 2     2 Elias        Salazar     Crissy Field
 3     3 Colton       Fox         San Bruno
 4     4 Cameron      Warren      Nottingham
 5     5 Carson       Mills       Jersey
 6     6 Addison      Meyer       Tingley
 7     7 Aubrey       Rice        Buena Vista
 8     8 Ellie        Schmidt     Division
 9     9 Robert       Garza       Red Rock
10    10 Stella       Daniels     Holland
```

```
people_2 <- tribble(
  ~id_2, ~name_first_2, ~name_last_2, ~street_2,
  1,     "Easton",      "Stone",      "Alameda",
  2,     "Elas",        "Salazar",    "Field",
  3,     NA,            "Fox",        NA,
  4,     "Cameron",     "Waren",      "Notingham",
  5,     "Carsen",      "Mills",      "Jersey",
  6,     "Adison",      NA,           NA,
  7,     "Aubrey",      "Rice",       "Buena Vista",
  8,     NA,            "Schmidt",    "Division",
  9,     "Bob",         "Garza",      "Red Rock",
  10,    "Stella",      NA,           "Holland"
) %>%
  print()
```

```
# A tibble: 10 x 4
    id_2 name_first_2 name_last_2 street_2
   <dbl> <chr>        <chr>       <chr>
 1     1 Easton       Stone       Alameda
 2     2 Elas         Salazar     Field
```

```
 3     3 <NA>         Fox         <NA>
 4     4 Cameron      Waren       Notingham
 5     5 Carsen       Mills       Jersey
 6     6 Adison       <NA>        <NA>
 7     7 Aubrey       Rice        Buena Vista
 8     8 <NA>         Schmidt     Division
 9     9 Bob          Garza       Red Rock
10    10 Stella       <NA>        Holland
```

In this scenario, we want to see if first name, last name, and street name match at each ID between our data frames. More specifically, we want to combine the two data frames into a single data frame and create three new dummy variables that indicate whether first name, last name, and address match respectively. Let's go ahead and combine the data frames now:

```
people <- people_1 %>%
  bind_cols(people_2) %>%
  print()
```

```
# A tibble: 10 x 8
    id_1 name_first_1 name_last_1 street_1      id_2 name_first_2 name_last_2
   <dbl> <chr>        <chr>       <chr>        <dbl> <chr>        <chr>
 1     1 Easton       <NA>        Alameda          1 Easton       Stone
 2     2 Elias        Salazar     Crissy Field     2 Elas         Salazar
 3     3 Colton       Fox         San Bruno        3 <NA>         Fox
 4     4 Cameron      Warren      Nottingham       4 Cameron      Waren
 5     5 Carson       Mills       Jersey           5 Carsen       Mills
 6     6 Addison      Meyer       Tingley          6 Adison       <NA>
 7     7 Aubrey       Rice        Buena Vista      7 Aubrey       Rice
 8     8 Ellie        Schmidt     Division         8 <NA>         Schmidt
 9     9 Robert       Garza       Red Rock         9 Bob          Garza
10    10 Stella       Daniels     Holland         10 Stella       <NA>
# i 1 more variable: street_2 <chr>
```

Now, our first attempt at creating the dummy variables might look something like this:

```
people %>%
  mutate(
    name_first_match = name_first_1 == name_first_2,
    name_last_match  = name_last_1 == name_last_2,
    street_match     = street_1 == street_2
  ) %>%
```

```
  # Order like columns next to each other for easier comparison
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         NA               Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         NA               Schmidt     Schmidt
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

Let's take a moment to review the results we got. In row 1 we see that "Easton" and "Easton" match, and the value for `name_first_match` is `TRUE`. So far, so good. In row 2, we see that "Elias" and "Ela" do not match, and the value for `name_first_match` is `FALSE`. That is also the result we wanted. In row 3, we see that "Colton" and "NA" do not match; however, the value in `name_first_match` is `NA`. In this case, this is not the result we want. We have a problem. That brings us to the first step in this workflow.

34.2.2.1 Spotting a need for a function

In some cases, the need is purely repetitive code – like the example at the beginning of this chapter. In other cases, like this one, a built-in R function is not giving the the desired result.

Here is the basic problem in this particular case:

```
1 == 1
```

```
[1] TRUE
```

```
1 == 2
```

```
[1] FALSE
```

```
1 == NA
```

```
[1] NA
```

```
NA == 2
```

```
[1] NA
```

```
NA == NA
```

```
[1] NA
```

The equality operator (`==`) always returns `NA` when one, or both, of the values being tested is `NA`. Often, that is exactly the result we want. In this case, however, it is not. Fortunately, we can get the result we want by writing our own function. That brings us to step 2 in the workflow.

34.2.2.2 Making the code work for one specific case

Don't try to solve the entire problem for every case right out of the gate. Instead, solve one problem for a specific case, and then build on that win! Let's start by trying to figure out how to get the result we want for `name_first_match` in row 3 of our example data.

```
"Colton" == NA
```

```
[1] NA
```

This is essentially what we already had above. But, we want to change our result from `NA` to `FALSE`. Let's start by saving the result to an object that we can manipulate:

```
result <- "Colton" == NA
result
```

```
[1] NA
```

So, now the value returned by the equality comparison is saved to an object named `result`. Let's go ahead and use a conditional operation to change the value of `result` to `FALSE` when it is initially `NA`, and leave it alone otherwise:

```
result <- "Colton" == NA
result <- if_else(is.na(result), FALSE, result)
result
```

```
[1] FALSE
```

Alright! This worked! At least, it worked for this case. That brings us to step 3 in the workflow.

34.2.2.3 Making the solution into a "function"

How can this be done? Well, first we start with a skeleton of the function components we discussed above. They are the `function()` function, the function body (wrapped in curly braces), and the function name. At the moment, we don't have any arguments. We'll explain why soon.

```
is_match <- function() {

}
```

Then, we literally copy the solution from above and paste it into the function body, making sure to indent the code. Next, we need to run the code chunk to *create* the function. After doing so, you should see the function appear in your global environment. Keep in mind, this *creates* the function so that we can use it later, but the function isn't immediately *run*.

```
is_match <- function() {
  result <- "Colton" == NA
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

Now, let's test out our shiny new function. To *run* the function, we can simply type the function name, with the parentheses, and run the code chunk.

```
is_match()
```

```
[1] FALSE
```

And, it works! When we ask R to run a function we are really asking R to run the *code* in the *body* of the function. In this case, we know that the code in the body of the function results in the value `FALSE` because this results in `FALSE`:

```
result <- "Colton" == NA
result <- if_else(is.na(result), FALSE, result)
result
```

```
[1] FALSE
```

And all we did was stick that code in the function body. Said another way, this:

```
result <- "Colton" == NA
result <- if_else(is.na(result), FALSE, result)
result
```

and this:

```
is_match()
```

mean essentially the same thing to R now if that makes sense. Hang in there even if it still isn't quite clear. We'll get more practice soon.

At this point, you may be wondering about the function arguments, and why there aren't any. Well, we can try passing a value to our `is_match()` function. How about we pass the name "Easton" from the first row of our example data above:

```
is_match(name = "Easton")
```

```
Error in is_match(name = "Easton"): unused argument (name = "Easton")
```

But, we get an error. R doesn't know what the `name` argument is or what to do with the values we are passing to it. That's because we never said anything about any arguments when we created the `is_match()` function. We left the parentheses where the function arguments go empty.

```
is_match <- function() {
  result <- "Colton" == NA
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

Let's create `is_match()` again, but this time, let's add an argument:

```
is_match <- function(name) {
  result <- "Colton" == NA
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

```
is_match(name = "Easton")
```

```
[1] FALSE
```

Hmmm, let's add another argument and see what happens:

```
is_match <- function(name_1, name_2) {
  result <- "Colton" == NA
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

```
is_match(name_1 = "Easton", name_2 = "Easton")
```

```
[1] FALSE
```

It looks as though the arguments we are adding don't have any effect on our returned value. That's because they don't. We oversimplified how function arguments work just a little bit in our factory analogy earlier. When we add arguments to function our definition (i.e., when we create the function) it's really more like adding a loading dock to our factory. It's a place where our factory can *receive* raw materials. However, there still needs to be equipment inside the factory that can *use* those raw materials. If we drop off a load of rubber at our bicycle factory, but there's no machine inside our bicycle factory that uses rubber, then we wouldn't expect dropping off the rubber to have any effect on the outputs coming out of the factory.

We have similar situation above. We dropped the name "Easton" off at our `is_match()` function, but nothing *inside* our `is_match()` function can *use* the name "Easton". There's no machinery to plug that name into. That brings us to step 4 in the workflow.

34.2.2.4 Start generalizing the function

As it stands right now, our `is_match()` function can't accept any new names. The only result we will ever get from the current version of our `is_match()` function is the result of testing the equality between the values "Colton" and NA, and then converting that value to `FALSE`. This isn't a problem if the only values we care about comparing are "Colton" and NA, but of course, that isn't the case. We need a way to make our function work for other values too. Said another way, we need to make our function more general.

As you may have guessed already, that will require us creating an argument to receive input values *and* a place to use those input values in the function body. Let's start by adding a `first_name` argument:

```
is_match <- function(first_name) {
  result <- first_name == NA
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

```
is_match(first_name = "Easton")
```

```
[1] FALSE
```

Here's what we did above:

- We once again created our `is_match()` function. However, this time we created it with a single argument – `first_name`. We didn't have to name the argument `first_name`. We could have named it anything that we can name any other variable in R. But, `first_name` seemed like a reasonable choice since the value we want to pass to this argument is a person's first name. The `first_name` argument will *receive* the first name values that we want to pass to this function.
- We replaced the constant value "Colton" in the function body with the variable `first_name`. It isn't a coincidence that the name of the variable `first_name` matches the name of the argument `first_name`. R will take whatever value we give to the `first_name` argument and *pass* it to the variable with a matching name inside the function body. Then, R will run the code inside the function body as though the variable *is* the value we passed to it.

So, when we type:

```
is_match(first_name = "Easton")
```

```
[1] FALSE
```

R sees:

```
result <- "Easton" == NA
result <- if_else(is.na(result), FALSE, result)
result
```

```
[1] FALSE
```

It looks like our `is_match()` function is still going to return a value of `FALSE` no matter what value we pass to the `first_name` function. That's because no matter what value we pass to `result <- first_name == NA`, `result` will equal `NA`. Then, `result <- if_else(is.na(result), FALSE, result)` will change the value of `result` to `FALSE`. So, we still need to make our function more general. As you may have guessed, we can do that by adding a second argument:

```
is_match <- function(first_name, first_name) {
  result <- first_name == first_name
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

```
Error: repeated formal argument 'first_name' (<input>:1:34)
```

Uh, oh! We got an error. This error is telling us that each function argument must have a unique name. Let's try again:

```
is_match <- function(first_name_1, first_name_2) {
  result <- first_name_1 == first_name_2
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

```
is_match(first_name_1 = "Easton", first_name_2 = "Colton")
```

```
[1] FALSE
```

Is this working or is our function still just returning `FALSE` no matter what we pass to the arguments? Let's try to pass "Easton" to `first_name_1` and `first_name_2` and see what happens:

```
is_match(first_name_1 = "Easton", first_name_2 = "Easton")
```

```
[1] TRUE
```

We got a `TRUE`! That's exactly the result we wanted! Let's do one final check. Let's see what happens when we pass `NA` to our `is_match()` function:

```
is_match(first_name_1 = "Easton", first_name_2 = NA)
```

```
[1] FALSE
```

Perfect! It looks like our function is finally ready to help us solve the problem we identified way back at step one. But, while we are talking about *generalizing* our function, shouldn't we go ahead and use more general names for our function arguments? We were only using first names when we were *developing* our function, but we are going to use our function to compare last names and street names as well. In fact, our function will compare any two values and tell us whether or not they are a match. So, let's go ahead and change the argument names to `value_1` and `value_2`:

```
is_match <- function(value_1, value_2) {
  result <- value_1 == value_2  # Don't forget to change the variable names here!!
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

Now, we are ready to put our function to work testing whether or not the first name, last name, and street name match at each ID between our data frames:

```
people %>%
  mutate(
    name_first_match = is_match(name_first_1, name_first_2),
    name_last_match  = is_match(name_last_1, name_last_2),
    street_match     = is_match(street_1, street_2)
  ) %>%
  # Order like columns next to each other for easier comparison
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         FALSE            Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         FALSE            Schmidt     Schmidt
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

Works like a charm! Notice, however, that we still have a lot of repetition in the code above. Unfortunately, we still don't have all the tools we need to remove it. But, we will soon.

At this point in the chapter, the hope is that you're developing a feel for how to write your own functions and why that might be useful. With R, it's possible to write functions that are very complicated. But, hopefully, the examples above show you that functions don't have to be complicated to be useful. In that spirit, we will not dive too much deeper into the details and technicalities of function writing at this point. However, there are a few details that should be at least mentioned so that you aren't caught off guard by them as you begin to write your own functions. We will touch on each below, and then wrap up this chapter with resources for those of you who wish to dive deeper.

34.3 Giving your function arguments default values

We've been introducing new functions to you all throughout the book so far. Each time, we try to discuss some, or all, of the function's arguments – including the default values that are passed to the arguments. Most of you have probably developed some sort of intuitive understanding of just what it meant for the argument to have a default value. However, this seems like an appropriate point in the book to talk about default arguments a little more explicitly and show you how to add them to the functions you write.

Let's say that we want to write a function that will increase the value of a number, or set of numbers, incrementally. We may start with something like this:

```
increment <- function(x) {
  x + 1
}
```

Here's what we did above:

- We *created* our own function that will increase the value of a number, or set of numbers, incrementally. Specifically, when we pass a number to the `x` argument the value of that number plus one will be returned.

Let's go ahead and use our function now:

```
increment(2)
```

```
[1] 3
```

Here's what we did above:

- We passed the value `2` to the `x` argument of our `increment()` function. The `x` argument then passed the value `2` to the `x` variable in the function body. Said another way, R replaced the `x` variable in the function body with the value `2`. Then, R executed the code in the function body. In this case, the code in the function body added the values `2` and `1` together. Finally, the function returned the value `3`.

Believe it or not, our simple little `increment()` function is a full-fledged R function. It is just as legitimate as any other R function we've used in this book. But, let's go ahead and add a little more to its functionality. For example, maybe we want to be able to increment by values other than just one. How might we do that?

Hopefully, your first thought was to replace the constant value `1` in the function body with a variable that can have *any* number passed to it. That's exactly what we will do next:

```
increment <- function(x, by) {
  x + by
}
```

Here's what we did above:

- We *created* our own function that will increase the value of a number, or set of numbers, incrementally. Specifically, when we pass a number to the `x` argument the value of that number will be incremented by the value passed to the `by` argument.

What value should `increment()` return if we pass `2` to the `x` argument and `2` to the `by` argument?

```
increment(2, 2)
```

```
[1] 4
```

Hopefully, that's what you were expecting. But, now what happens if we don't pass any value to the `by` argument?

```
increment(2)
```

```
Error in increment(2): argument "by" is missing, with no default
```

We get an error saying that there wasn't any value passed to the `by` argument, and the `by` argument doesn't have a default value. But, we are really lazy, and it takes a lot of work to pass a value to the `by` argument every time we use the `increment()` function. Plus, we *almost* always only want to increment our numbers by one. In this case, our best course of action is to set the default value of `by` to `1`. Fortunately for us, doing so is really easy!

```
increment <- function(x, by = 1) {
  x + by
}
```

Here's what we did above:

- We *created* our own function that will increase the value of a number, or set of numbers, incrementally. Specifically, when we pass a number to the `x` argument the value of that number will be incremented by the value passed to the `by` argument. The default value passed to the `by` argument is `1`. Said another way, R will *pretend* that we passed the value `1` to the `by` argument if we don't explicitly pass a number other than `1` to the `by` argument.
- All we had to do to give `by` a default value was type `=` followed by the value (i.e., `1`) when we created the function.

Now let's try out our latest version of `increment()`:

```
# Default value
increment(2)
```

```
[1] 3
```

```
# Passing the value 1
increment(2, 1)
```

```
[1] 3
```

```
# Passing a value other than 1
increment(2, 2)
```

```
[1] 4
```

```
# Passing a vector of numbers to the x argument
increment(c(1, 2, 3), 2)
```

```
[1] 3 4 5
```

34.4 The values your functions return

When we run our functions, they typically execute each line of code in the function body, one after another, starting with the first line and ending at the last line. Therefore, the value that your function *returns* (i.e., the thing that comes out of the factory) is typically dictated by the last line of code in your function body.

To explain this further, let's take another look at our `is_match()` function:

```
is_match <- function(value_1, value_2) {
  result <- value_1 == value_2                    # Do this first
  result <- if_else(is.na(result), FALSE, result)  # Then this
  result                                           # Then this
}
```

Why did we type that third line of code? Afterall, that line of code isn't *doing* anything. Well, let's see what happens if we take it out:

```
is_match <- function(value_1, value_2) {
  result <- value_1 == value_2
  result <- if_else(is.na(result), FALSE, result)
}
```

```
is_match("Easton", "Easton")
```

It appears as though nothing happened! Did our function break?

Let's think about what typically happens when we use R's built-in functions. When we don't *assign* the value returned by the function to an object, then the returned value is printed to the screen:

```
sum(1, 1)
```

```
[1] 2
```

But, when we do assign the value returned by the function to an object, nothing is printed to the screen:

```
x <- sum(1, 1)
```

The same thing is happening in our function above. The last line of our function body is assigning a value (i.e., `TRUE` or `FALSE`) to the variable `result`. Just like `x <- sum(1, 1)` didn't print to the screen, `result <- if_else(is.na(result), FALSE, result)` doesn't print to the screen when we run `is_match("Easton", "Easton")` using this version of `is_match()`.

However, we can see in the example below that result of the operations being executed inside the function body can still be assigned to an object in our global environment, and we can print the contents of that object to screen:

```
x <- is_match("Easton", "Easton")
x
```

```
[1] TRUE
```

If all of that seems confusing, here is the bottom line. In general, it's a best practice for your function to print its return value to the screen. You can do this in one of three ways:

1 The value that results from the code in the last line of the function body isn't assigned to anything. We saw an example of this above with our `increment()` function:

```
increment <- function(x, by = 1) {
  x + by # Last line doesn't assign the value to an object
}
```

```
increment(2)
```

```
[1] 3
```

2 If you assign values to objects inside your function, then type the name of the object that contains the value you want your function to return on the last line of the function body. We saw an example of this with our `is_match()` function. We can also amend our `increment()` function follow this pattern:

```
increment <- function(x, by = 1) {
  out <- x + by # Now we assign the value to an object
  out           # Type object name on last line of the function body
}
```

```
increment(2)
```

```
[1] 3
```

3 Use the `return()` function.

```
increment <- function(x, by = 1) {
  out <- x + by
  return(out)
}
```

```
increment(2)
```

```
[1] 3
```

So, which method *should* you use? Well, for all but the simplest functions (like the one above) method 1 is not considered good coding practice. Method 3 may seem like it's the most explicit; however, it's actually considered best practice to use the `return()` function only when you want your function to return its value before R reaches the last line of the function body. For example, let's add another line of code to our function body that adds another `1` to the value of out:

```
increment <- function(x, by = 1) {
  out <- x + by
  out <- out + 1 # Adding an extra 1
  return(out)    # Return still in the last line
}
```

```
increment(2)
```

```
[1] 4
```

Now, let's move `return(out)` to the second line of the function body – above the line of code that adds an additional `1` to the value of `out`:

```
increment <- function(x, by = 1) {
  out <- x + by
  return(out)    # Return in the second line above adding an extra 1
  out <- out + 1 # Adding an extra 1
}
```

```
increment(2)
```

```
[1] 3
```

In the example above, the last `1` wasn't added to the value of `out` because we used the `return()` function. Said another way, `increment()` returned the value of `out` "early", and the last line of the function body was never executed.

In the example above, using the `return()` function in the way that we did obviously makes no sense. It was just meant to illustrate what the `return()` function *can* do. The `return()` function doesn't actually become useful until we start writing more complex functions. But, because the `return()` function has the special ability to end the execution of the function body early, it's considered a best practice to only use it for that purpose.

Therefore, in most situations, you will want to use method 2 (i.e., object name on last line) when writing your own functions.

One final note before we move on to the next section. Notice that we never used the `print()` function on the last line of our code. This was intentional. Using `print()` will give you the result you expect when you don't assign the value that your function returns to an object in your global environment:

```
increment <- function(x, by = 1) {
  out <- x + by
  print(out)
}
```

```
increment(2)
```

```
[1] 3
```

But, it will not give you the result you want if you do assign the value that your function returns to an object in your global environment:

```
increment <- function(x, by = 1) {
  out <- x + by
  print(out)
}
```

```
x <- increment(2)
```

```
[1] 3
```

```
x
```

```
[1] 3
```

34.5 Lexical scoping and functions

If you have been following along with the code above on your computer, you may have noticed that the objects we create inside our functions do not appear in our global environment. If you haven't been following along, you may want to jump on your computer really quickly for this section (or just take our word for it).

The reason the objects we created inside our functions do not appear in our global environment is that R actually has *multiple* environments were objects can live. Additionally, R uses something called lexical scoping rules to look for the objects you refer to in your R code. The vast majority of the time, we won't need to concern ourselves much with any of these other environments or the lexical scoping rules. However, function writing does require us to have some minimal understanding of these concepts. At the very least, you should be aware of the following when writing your own functions:

1 Objects we create inside of functions don't live in our global environment and we can't do anything with them outside of the function we created them in.

In the example below, we create an object named `out` inside of the `increment()` function:

```
increment <- function(x, by = 1) {
  out <- x + by # Assign the value to the out object inside the function
  out
}
```

We then use the function:

```
x <- increment(2)
x
```

```
[1] 3
```

However, the `out` object is not available to us:

```
out
```

```
Error: object 'out' not found
```

2 If the function we write can't find the object it's looking for inside the function body, then it will try to find it in the global environment.

For example, let's create a new function named `add` that adds the values of `x` and `y` together in its function body. Notice, however, that there is no `y` argument to pass a value to, and that `y` is never assigned a value inside of the `add()` function:

```
add <- function(x) {
  x + y
}
```

When we call the function:

```
add(2)
```

```
Error in add(2): object 'y' not found
```

We get an error. R can't find the object `y`. Now let's create a `y` object in our global environment:

```
y <- 100
```

And call the `add()` function again:

```
add(2)
```

```
[1] 102
```

As you can see, R wasn't able to find a value for `y` *inside* of the function body so it looked *outside* of the function in the global environment. This is definitely something to be aware of, but usually isn't an actual problem.

For starters, there is no obviously good reason to add a variable to your function body without assigning it a value inside the function body or matching it to a function argument. In other words, there's generally no good reason to have variables that serve no purpose floating around inside your functions.

If you do assign it a value inside the function, then R will not look outside of the function for a value:

```
add <- function(x) {
  y <- 1
  x + y
}
```

```
y <- 100
add(2)
```

```
[1] 3
```

Likewise, if you create the function with a matching argument, then R will not look outside of the function for a value:

```
add <- function(x, y) {
  x + y
}
```

```
y <- 100
add(2)
```

```
Error in add(2): argument "y" is missing, with no default
```

Again, this aspect of the lexical scoping rules is something to be aware of, but generally isn't a problem in practice.

34.6 Tidy evaluation

Now that you have all the basics of function writing under your belt, let's take look at what happens when we try to write functions that use `tidyverse` package functions in the function body.

For this section, let's return to our study data we used for the first example in this chapter. As a reminder, here's what the data looks like:

```
study
```

```
# A tibble: 68 x 7
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese
 2    30 30 and Older    Female    63    106  18.8 Normal
 3    32 30 and Older    Female    62    145  26.5 Overweight
 4    29 Younger than 30 Male      67    195  30.5 Obese
 5    24 Younger than 30 Female    67    143  22.4 Normal
 6    38 30 and Older    Female    58    125  26.1 Overweight
 7    25 Younger than 30 Female    64    138  23.7 Normal
 8    24 Younger than 30 Male      69    140  20.7 Normal
 9    48 30 and Older    Male      65    158  26.3 Overweight
10    29 Younger than 30 Male      68    167  25.4 Overweight
# i 58 more rows
```

We already calculated the number of missing values, mean, median, min, and max for all of the continuous variables. So, let's go ahead and calculate the number and percent of observations for each level of our categorical variables.

We know that we have 3 categorical variables (i.e., `age_group`, `gender`, and `bmi_3cat`), and we know that we want to perform the same calculation on all of them. So, we decide to write our own function. Following the workflow we discussed earlier, our next step is to make the code work for one specific case:

```
study %>%
  count(age_group) %>%
  mutate(percent = n / sum(n) * 100)
```

```
# A tibble: 3 x 3
  age_group           n percent
  <fct>           <int>   <dbl>
```

```
1 Younger than 30    56   82.4
2 30 and Older       11   16.2
3 <NA>                1    1.47
```

Great! Thanks to `dplyr`, we have the result we were looking for! The next step in the workflow is to make our solution into a function. Let's copy and paste our solution into a function skeleton like we did before:

```
cat_stats <- function(var) {
  study %>%
    count(age_group) %>%
    mutate(percent = n / sum(n) * 100)
}
```

```
cat_stats()
```

```
# A tibble: 3 x 3
  age_group           n percent
  <fct>           <int>   <dbl>
1 Younger than 30    56   82.4
2 30 and Older       11   16.2
3 <NA>                1    1.47
```

So far, so good! Now, let's replace `age_group` with `var` in the function body to generalize our function:

```
cat_stats <- function(var) {
  study %>%
    count(var) %>%
    mutate(percent = n / sum(n) * 100)
}
```

```
cat_stats(age_group)
```

```
Error in `count()`:
! Must group by variables found in `.data`.
x Column `var` is not found.
```

Unfortunately, this doesn't work. As we stated in the introduction to this part of the book, non-standard evaluation prevents us from using `dplyr` and other `tidyverse` packages inside of our functions in the same way that we might use other functions. Fortunately, the fix for this is pretty easy. All we need to do is **embrace** (i.e., wrap) the `var` variable with double curly braces:

```
cat_stats <- function(var) {
  study %>%
    count({{ var }}) %>%
    mutate(percent = n / sum(n) * 100)
}
```

```
cat_stats(age_group)
```

```
# A tibble: 3 x 3
  age_group           n percent
  <fct>           <int>   <dbl>
1 Younger than 30    56   82.4
2 30 and Older       11   16.2
3 <NA>                1    1.47
```

Now, we can use our new function on the rest of our categorical variables:

```
cat_stats(gender)
```

```
# A tibble: 3 x 3
  gender     n percent
  <fct>  <int>   <dbl>
1 Female    43   63.2
2 Male      24   35.3
3 <NA>       1    1.47
```

```
cat_stats(bmi_3cat)
```

```
# A tibble: 4 x 3
  bmi_3cat       n percent
  <fct>      <int>   <dbl>
1 Normal        43   63.2
2 Overweight    16   23.5
3 Obese          5    7.35
4 <NA>           4    5.88
```

This is working beautifully! However, we should probably make one final adjustment to our `cat_stats()` function. Let's say that we had another data frame with categorical variable we wanted to analyze:

```
other_study <- tibble(
  id = 1:10,
  age_group = c(rep("Younger", 9), "Older"),
) %>%
  print()
```

```
# A tibble: 10 x 2
      id age_group
   <int> <chr>
 1     1 Younger
 2     2 Younger
 3     3 Younger
 4     4 Younger
 5     5 Younger
 6     6 Younger
 7     7 Younger
 8     8 Younger
 9     9 Younger
10    10 Older
```

Now, let's pass age_group to our `cat_stats()` function again:

```
cat_stats(age_group)
```

```
# A tibble: 3 x 3
  age_group           n percent
  <fct>           <int>   <dbl>
1 Younger than 30    56   82.4
2 30 and Older       11   16.2
3 <NA>                1    1.47
```

Is that the result you expected? Hopefully not! That's the same result we got from the original study data. Have you figured out why this happened? Take another look at our function definition:

```
cat_stats <- function(var) {
  study %>%
    count({{ var }}) %>%
    mutate(percent = n / sum(n) * 100)
}
```

We have the `study` data frame hard coded into the first line of the function body. In the same way we need a matching argument-variable pair to pass multiple different columns into our function, we need a matching argument-variable pair to pass multiple different data frames into our function. We start by adding an argument to accept the data frame:

```
cat_stats <- function(data, var) {
  study %>%
    count({{ var }}) %>%
    mutate(percent = n / sum(n) * 100)
}
```

Again, we could name this argument almost anything, but `data` seems like a reasonable choice. Then, we replace `study` with `data` in the function body to generalize our function:

```
cat_stats <- function(data, var) {
  data %>%
    count({{ var }}) %>%
    mutate(percent = n / sum(n) * 100)
}
```

And now we can use our `cat_stats()` function on any data frame – including the `other_study` data frame we created above:

```
cat_stats(other_study, age_group)
```

```
# A tibble: 2 x 3
  age_group     n percent
  <chr>     <int>   <dbl>
1 Older         1      10
2 Younger       9      90
```

We can even use it with a pipe:

```
other_study %>%
  cat_stats(age_group)
```

```
# A tibble: 2 x 3
  age_group     n percent
  <chr>     <int>   <dbl>
1 Older         1      10
2 Younger       9      90
```

Some of you may be wondering why we didn't have to wrap `data` with double curly braces in the code above. Remember, we only have to use the curly braces with column names because of non-standard evaluation. More specifically, because of one aspect of non-standard evaluation called data masking. Data masking is what lets us refer to a column in a data frame without using dollar sign or bracket notation. For example, `age_group` doesn't exist in our global environment as a standalone object:

```
age_group
```

```
Error: object 'age_group' not found
```

It only exists as a part of (i.e. a column in) the `other_study` object:

```
other_study$age_group
```

```
 [1] "Younger" "Younger" "Younger" "Younger" "Younger" "Younger" "Younger"
 [8] "Younger" "Younger" "Older"
```

But the data frames themselves are not data masked. They do exist as standalone objects in our global environment:

```
other_study
```

```
# A tibble: 10 x 2
      id age_group
   <int> <chr>
 1     1 Younger
 2     2 Younger
 3     3 Younger
 4     4 Younger
```

```
 5     5 Younger
 6     6 Younger
 7     7 Younger
 8     8 Younger
 9     9 Younger
10    10 Older
```

Therefore, there is no need to wrap them with double curly braces. Having said that, it doesn't appear as though doing so will hurt anything:

```
cat_stats <- function(data, var) {
  {{data}} %>%
    count({{ var }}) %>%
    mutate(percent = n / sum(n) * 100)
}
```

```
cat_stats(other_study, age_group)
```

```
# A tibble: 2 x 3
  age_group     n percent
  <chr>     <int>   <dbl>
1 Older         1      10
2 Younger       9      90
```

That pretty much wraps up this chapter on the basics of writing function to reduce unnecessary repetition in your R code. If you're feeling good about writing your own functions, great! If you want to dig even deeper, take a look at the functions chapter of the Advanced R book.

If you're still feeling a little apprehensive or confused, don't feel bad. It takes most people (myself included) a while to get comfortable with writing functions. Just remember, functions *can* be complicated, but they don't *have* to be. Even very simple functions can sometimes be useful. So, start simple and get more complex as your skills and confidence grow.

If you find that you've written a function that is really useful, consider saving it for use again in the future. One way is saving functions as R scripts in a folder on your computer that can then be copied and pasted from the scripts into R programs as needed.

A much better way is using the `source()` function, which allows you to use use your saved functions without having to manually copy and paste them.

An even better way is learning how to make your own packages that contain groups of related functions and save them to your Github account. From there, you can use your functions on any computer, and even share them with others. Finally, you can even publish your packages on CRAN if you want to them with the broadest possible audience.

35 Column-wise Operations in dplyr

Throughout the chapters in this book we have learned to do a really vast array of useful data transformations and statistical analyses with the help of the `dplyr` package.

Figure 35.1: dplyr graphic

So far, however, we've always done these transformations and statistical analyses on one column of our data frame at a time. There isn't anything inherently "wrong" with this approach, but, for reasons we've already discussed, there are often advantages to telling R what you want to do one time, and then asking R to do that thing repeatedly *across* all, or a subset of, the columns in your data frame. That is exactly what `dplyr`'s `across()` function allows us to do.

There are so many ways we might want to use the `across()` function in our R programs. We can't begin to cover, or even imagine, them all. Instead, the goal of this chapter is just to provide you with an overview of the `across()` function and show you some examples of using it with `filter()`, `mutate()`, and `summarise()` to get you thinking about how you might want to use it in your R programs.

Before we discuss further, let's take a look at a quick example. The first thing we will need to do is load `dplyr`.

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

Then, we will simulate some data. In this case, we are creating a data frame that contains three columns of 10 random numbers:

```
set.seed(123)
df_xyz <- tibble(
  row = 1:10,
  x   = rnorm(10),
  y   = rnorm(10),
  z   = rnorm(10)
) %>%
  print()
```

```
# A tibble: 10 x 4
     row       x      y      z
   <int>   <dbl>  <dbl>  <dbl>
 1     1 -0.560   1.22  -1.07
 2     2 -0.230   0.360 -0.218
 3     3  1.56    0.401 -1.03
 4     4  0.0705  0.111 -0.729
 5     5  0.129  -0.556 -0.625
 6     6  1.72    1.79  -1.69
 7     7  0.461   0.498  0.838
 8     8 -1.27   -1.97   0.153
 9     9 -0.687   0.701 -1.14
10    10 -0.446  -0.473  1.25
```

Up to this point, if we wanted to find the mean of each column, we would probably have written code like this:

```
df_xyz %>%
  summarise(
    x_mean = mean(x),
    y_mean = mean(y),
    z_mean = mean(y)
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1 0.0746  0.209  0.209
```

With the help of the `across()` function, we can now get the mean of each column like this:

```
df_xyz %>%
  summarise(
    across(
      .cols  = c(x:z),
      .fns   = mean,
      .names = "{col}_mean"
    )
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1 0.0746  0.209 -0.425
```

Now, you might ask why this is a better approach. Fair question.

In this case, using `across()` doesn't actually reduce the number of lines of code we wrote. In fact, we wrote two additional lines when we used the `across()` function. However, imagine if we added 20 additional columns to our data frame. Using the first approach, we would have to write 20 additional lines of code inside the `summarise()` function. Using the `across()` approach, we wouldn't have to add any additional code at all. We would simply update the value we pass to the `.cols` argument.

Perhaps *more importantly*, did you notice that we "accidentally" forgot to replace `y` with `z` when we copied and pasted `z_mean = mean(y)` in the code chunk for the first approach? If not, go back and take a look. That mistake is fairly easy to catch and fix in this very simple example. But, in real-world projects, mistakes like this are easy to make, and not always so easy to catch. We are much less likely to make similar mistakes when we use `across()`.

35.1 The across() function

The `across()` function is part of the `dplyr` package. We will always use `across()` *inside* of one of the `dplyr` verbs we've been learning about. Specifically, `mutate()`, and `summarise()`. We will not use `across()` *outside* of the `dplyr` verbs. Additionally, we will always use `across()` within the context of a data frame (as opposed to a vector, matrix, or some other data structure).

To view the help documentation for `across()`, you can copy and paste `?dplyr::across` into your R console. If you do, you will see that `across()` has four arguments. They are:

1 `.cols`. The value we pass to this argument should be columns of the data frame we want to operate on. We can once again use tidy-select argument modifiers here. In the example above, we used `c(x:z)` to tell R that we wanted to operate on columns x through z (inclusive). If we had also wanted the mean of the `row` column for some reason, we could have used the `everything()` tidy-select modifier to tell R that we wanted to operate on all of the columns in the data frame.

2 `.fns`. This is where you tell `across()` what function, or functions, you want to apply to the columns you selected in `.cols`. In the example above, we passed the mean function to the `.fns` argument. Notice that we typed `mean` without the parentheses (i.e., `mean()`).

3 `...`. In this case, the `...` argument is where we pass any additional arguments to the function we passed to the `.fns` argument. For example, we passed the `mean` function to the `.fns` argument above. In the data frame above, none of the columns had any missing values. Let's go ahead and add some missing values so that we can take a look at how `...` works in `across()`.

```
df_xyz$x[2] <- NA_real_
df_xyz$y[4] <- NA_real_
df_xyz$z[6] <- NA_real_
df_xyz
```

```
# A tibble: 10 x 4
     row       x      y      z
   <int>   <dbl>  <dbl>  <dbl>
 1     1 -0.560   1.22  -1.07
 2     2 NA       0.360 -0.218
 3     3  1.56    0.401 -1.03
 4     4  0.0705 NA     -0.729
 5     5  0.129  -0.556 -0.625
 6     6  1.72    1.79  NA
 7     7  0.461   0.498  0.838
 8     8 -1.27   -1.97   0.153
 9     9 -0.687   0.701 -1.14
10    10 -0.446  -0.473  1.25
```

As we've already seen many times, R won't drop the missing values and carry out a complete case analysis by default:

```
df_xyz %>%
  summarise(
    x_mean = mean(x),
```

```
    y_mean = mean(y),
    z_mean = mean(y)
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1     NA     NA     NA
```

Instead, we have to explicitly tell R to carry out a complete case analysis. We can do so by filtering our rows with missing data (more on this later) or by changing the value of the `mean()` function's `na.rm` argument from `FALSE` (the default) to `TRUE`:

```
df_xyz %>%
  summarise(
    x_mean = mean(x, na.rm = TRUE),
    y_mean = mean(y, na.rm = TRUE),
    z_mean = mean(z, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1  0.108  0.220 -0.284
```

When we use `across()`, we will need to pass the `na.rm = TRUE` to the `mean()` function in `across()`'s `...` argument like this:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = mean,
      na.rm  = TRUE, # Passing na.rm = TRUE to the ... argument
      .names = "{col}_mean"
    )
  )
```

```
Warning: There was 1 warning in `summarise()`.
i In argument: `across(.cols = everything(), .fns = mean, na.rm = TRUE, .names
```

```
  = "{col}_mean")`.
Caused by warning:
! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.
Supply arguments directly to `.fns` through an anonymous function instead.

  # Previously
  across(a:b, mean, na.rm = TRUE)

  # Now
  across(a:b, \(x) mean(x, na.rm = TRUE))


# A tibble: 1 x 4
  row_mean x_mean y_mean z_mean
     <dbl>  <dbl>  <dbl>  <dbl>
1      5.5  0.108  0.220 -0.284
```

Notice that we do not actually type out `... =` or anything like that.

4 `.names`. You can use this argument to adjust the column names that will result from the operation you pass to `.fns`. In the example above, we used the special `{cols}` keyword to use each of the column names that were passed to the `.cols` argument as the first part of each of the new columns' names. Then, we asked R to add a literal underscore and the word "mean" because these are all mean values. That resulted in the new column names you see above. The default value for `.names` is just `{cols}`. So, if we hadn't modified the value passed to the `.names` argument, our results would have looked like this:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = mean,
      na.rm  = TRUE
    )
  )
```

```
# A tibble: 1 x 4
    row     x     y      z
  <dbl> <dbl> <dbl>  <dbl>
1   5.5 0.108 0.220 -0.284
```

There is also a special `{fn}` keyword that we can use to pass the name of each of the functions we used in `.fns` as part of the new column names. However, in order to get `{fn}` to work the

way we want it to, we have to pass a list of name-function pairs to the `.fns` argument. We'll explain further.

First, we will keep the code exactly as it was, but replace "mean" with "{fn}" in the `.names` argument:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = mean,
      na.rm  = TRUE,
      .names = "{col}_{fn}"
    )
  )
```

```
# A tibble: 1 x 4
  row_1   x_1   y_1    z_1
  <dbl> <dbl> <dbl>  <dbl>
1   5.5 0.108 0.220 -0.284
```

This is not the result we wanted. Because, we didn't *name* the function that we passed to `.fns`, `across()` essentially used "function number 1" as its name. In order to get the result we want, we need to pass a list of name-function pairs to the `.fns` argument like this:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = list(mean = mean),
      na.rm  = TRUE,
      .names = "{col}_{fn}"
    )
  )
```

```
# A tibble: 1 x 4
  row_mean x_mean y_mean z_mean
     <dbl>  <dbl>  <dbl>  <dbl>
1      5.5  0.108  0.220 -0.284
```

Although it may not be self-evident from just looking at the code above, the first `mean` in the `list(mean = mean)` name-function pair is a name that we are choosing to be passed to the new column names. Theoretically, we could have picked any name. For example:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = list(r4epi = mean),
      na.rm  = TRUE,
      .names = "{col}_{fn}"
    )
  )
```

```
# A tibble: 1 x 4
  row_r4epi x_r4epi y_r4epi z_r4epi
      <dbl>   <dbl>   <dbl>   <dbl>
1       5.5   0.108   0.220  -0.284
```

The second `mean` in the `list(mean = mean)` name-function pair is the name of the actual function we want to apply to the columns in `.cols`. This part of the name-function pair must be the name of the function that we actually want to apply to the columns in `.cols`. Otherwise, we will get an error:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = list(mean = r4epi),
      na.rm  = TRUE,
      .names = "{col}_{fn}"
    )
  )
```

```
Error in `summarise()`:
i In argument: `across(...)`.
Caused by error:
! object 'r4epi' not found
```

An additional advantage of passing a list of name-function pairs to the `.fns` argument is that we can pass *multiple* functions at once. For example, let's say that we want the minimum and maximum value of each column in our data frame. Without `across()` we might do that analysis like this:

```
df_xyz %>%
  summarise(
    x_min = min(x, na.rm = TRUE),
    x_max = max(x, na.rm = TRUE),
    y_min = min(y, na.rm = TRUE),
    y_max = max(y, na.rm = TRUE),
    z_min = min(z, na.rm = TRUE),
    z_max = max(z, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 6
  x_min x_max y_min y_max z_min z_max
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 -1.27  1.72 -1.97  1.79 -1.14  1.25
```

But, we can simply pass `min` and `max` as a list of name-function pairs if we use `across()`:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = list(min = min, max = max),
      na.rm  = TRUE,
      .names = "{col}_{fn}"
    )
  )
```

```
# A tibble: 1 x 8
  row_min row_max x_min x_max y_min y_max z_min z_max
    <int>   <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1       1      10 -1.27  1.72 -1.97  1.79 -1.14  1.25
```

How great is that?!?

So, we've seen how to pass an individual function to the `.fns` argument and we've seen how to pass a list containing multiple functions to the `.fns` argument. There is actually a third syntax for passing functions to the `.fns` argument. The `across()` documentation calls it "a purrr-style lambda". This can be a little bit confusing, so I'm going to show you an example, and then walk through it step by step.

```r
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = ~ mean(.x, na.rm = TRUE),
      .names = "{col}_mean"
    )
  )
```

```
# A tibble: 1 x 4
  row_mean x_mean y_mean z_mean
     <dbl>  <dbl>  <dbl>  <dbl>
1      5.5  0.108  0.220 -0.284
```

The purrr-style lambda always begins with the tilde symbol (~). Then we type out a function call behind the tilde symbol. We place the special `.x` symbol inside the function call where we would normally want to type the name of the column we want the function to operate on. The `across()` function will then substitute each column name we passed to the `.cols` argument for `.x` sequentially. In the example above, there isn't really any good reason to use this syntax. However, this syntax can be useful at times. We will see some examples below.

35.2 Across with mutate

We've already seen a number of examples of manipulating columns of our data frames using the `mutate()` function. In this section, we are going to take a look at two examples where using the `across()` function inside `mutate()` will allow us to apply the same manipulation to multiple columns in our data frame at once.

Let's go ahead and simulate the same `demographics` data frame we simulated for the recoding missing section of the conditional operations chapter. Let's also add two new columns: a four-category education column and a six-category income column. For all columns except `id` and `age`, a value of `7` represents "Don't know" and a value of `9` represents "refused."

```r
set.seed(123)
demographics <- tibble(
  id       = 1:10,
  age      = c(sample(1:30, 9, TRUE), NA),
  race     = c(1, 2, 1, 4, 7, 1, 2, 9, 1, 3),
  hispanic = c(7, 0, 1, 0, 1, 0, 1, 9, 0, 1),
  edu_4cat = c(4, 2, 9, 1, 2, 3, 4, 9, 3, 3),
```

```
  inc_6cat = c(1, 4, 1, 1, 5, 3, 2, 2, 7, 9)
) %>%
  print()
```

```
# A tibble: 10 x 6
      id   age  race hispanic edu_4cat inc_6cat
   <int> <int> <dbl>    <dbl>    <dbl>    <dbl>
 1     1    15     1        7        4        1
 2     2    19     2        0        2        4
 3     3    14     1        1        9        1
 4     4     3     4        0        1        1
 5     5    10     7        1        2        5
 6     6    18     1        0        3        3
 7     7    22     2        1        4        2
 8     8    11     9        9        9        2
 9     9     5     1        0        3        7
10    10    NA     3        1        3        9
```

When working with data like this, it's common to want to recode all the 7's and 9's to NA's. We saw how to do that one column at a time already:

```
demographics %>%
  mutate(
    race     = if_else(race == 7 | race == 9, NA_real_, race),
    hispanic = if_else(race == 7 | hispanic == 9, NA_real_, hispanic),
    edu_4cat = if_else(edu_4cat == 7 | edu_4cat == 9, NA_real_, edu_4cat)
  )
```

```
# A tibble: 10 x 6
      id   age  race hispanic edu_4cat inc_6cat
   <int> <int> <dbl>    <dbl>    <dbl>    <dbl>
 1     1    15     1        7        4        1
 2     2    19     2        0        2        4
 3     3    14     1        1       NA        1
 4     4     3     4        0        1        1
 5     5    10    NA       NA        2        5
 6     6    18     1        0        3        3
 7     7    22     2        1        4        2
 8     8    11    NA       NA       NA        2
 9     9     5     1        0        3        7
10    10    NA     3        1        3        9
```

In the code chunk above, we have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code.

Also, did you notice that we forgot to replace `race` with `hispanic` in `hispanic = if_else(race == 7 | hispanic == 9, NA_real_, hispanic)`? This time, we didn't write "forgot" in quotes because we *really did forget* and only noticed it later. In this case, the error caused a value of `1` to be recoded to `NA` in the `hispanic` column. These typos we've been talking about really do happen – even to me!

Here's how we can use `across()` in this situation:

```
demographics %>%
  mutate(
    across(
      .cols = c(-id, -age),
      .fns  = ~ if_else(.x == 7 | .x == 9, NA_real_, .x)
    )
  )
```

```
# A tibble: 10 x 6
      id   age  race hispanic edu_4cat inc_6cat
   <int> <int> <dbl>    <dbl>    <dbl>    <dbl>
 1     1    15     1       NA        4        1
 2     2    19     2        0        2        4
 3     3    14     1        1       NA        1
 4     4     3     4        0        1        1
 5     5    10    NA        1        2        5
 6     6    18     1        0        3        3
 7     7    22     2        1        4        2
 8     8    11    NA       NA       NA        2
 9     9     5     1        0        3       NA
10    10    NA     3        1        3       NA
```

Here's what we did above:

- We used a purrr-style lambda to replace `7`'s and `9`'s in all columns in our data frame, except `id` and `age`, with `NA`.
- Remember, the special `.x` symbol is just shorthand for each column passed to the `.cols` argument.

As another example, let's say that we are once again working with data from a drug trial that includes a list of side effects for each person:

```
set.seed(123)
drug_trial <- tibble(
  id           = 1:10,
  se_headache  = sample(0:1, 10, TRUE),
  se_diarrhea  = sample(0:1, 10, TRUE),
  se_dry_mouth = sample(0:1, 10, TRUE),
  se_nausea    = sample(0:1, 10, TRUE)
) %>%
 print()
```

```
# A tibble: 10 x 5
      id se_headache se_diarrhea se_dry_mouth se_nausea
   <int>       <int>       <int>        <int>     <int>
 1     1           0           1            0         0
 2     2           0           1            1         1
 3     3           0           1            0         0
 4     4           1           0            0         1
 5     5           0           1            0         1
 6     6           1           0            0         0
 7     7           1           1            1         0
 8     8           1           0            1         0
 9     9           0           0            0         0
10    10           0           0            1         1
```

Now, we want to create a factor version of each of the side effect columns. We've already learned how to do so one column at a time:

```
drug_trial %>%
  mutate(
    se_headache_f  = factor(se_headache, 0:1, c("No", "Yes")),
    se_diarrhea_f  = factor(se_diarrhea, 0:1, c("No", "Yes")),
    se_dry_mouth_f = factor(se_dry_mouth, 0:1, c("No", "Yes"))
  )
```

```
# A tibble: 10 x 8
      id se_headache se_diarrhea se_dry_mouth se_nausea se_headache_f
   <int>       <int>       <int>        <int>     <int> <fct>
 1     1           0           1            0         0 No
 2     2           0           1            1         1 No
 3     3           0           1            0         0 No
 4     4           1           0            0         1 Yes
```

```
 5     5           0           1           0         1 No
 6     6           1           0           0         0 Yes
 7     7           1           1           1         0 Yes
 8     8           1           0           1         0 Yes
 9     9           0           0           0         0 No
10    10           0           0           1         1 No
# i 2 more variables: se_diarrhea_f <fct>, se_dry_mouth_f <fct>
```

Once again, we have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code. Here's how we can use `across()` to do so:

```
drug_trial %>%
  mutate(
    across(
      .cols  = starts_with("se"),
      .fns   = ~ factor(.x, 0:1, c("No", "Yes")),
      .names = "{col}_f"
    )
  )
```

```
# A tibble: 10 x 9
      id se_headache se_diarrhea se_dry_mouth se_nausea se_headache_f
   <int>       <int>       <int>        <int>     <int> <fct>
 1     1           0           1            0         0 No
 2     2           0           1            1         1 No
 3     3           0           1            0         0 No
 4     4           1           0            0         1 Yes
 5     5           0           1            0         1 No
 6     6           1           0            0         0 Yes
 7     7           1           1            1         0 Yes
 8     8           1           0            1         0 Yes
 9     9           0           0            0         0 No
10    10           0           0            1         1 No
# i 3 more variables: se_diarrhea_f <fct>, se_dry_mouth_f <fct>,
#   se_nausea_f <fct>
```

Here's what we did above:

- We used a purrr-style lambda to create a factor version of all the side effect columns in our data frame.
- We used the `.names` argument to add an "_f" to the end of the new column names.

35.3 Across with summarise

Let's return to the `ehr` data frame we used in the chapter on working with character strings for our first example of using `across()` inside of `summarise`.

You may click here to download this file to your computer.

```
# We will need here, readr and stringr in the examples below
library(readr)
library(stringr)
library(here)
```

```
# Read in the data
ehr <- read_rds("ehr.Rds")
```

For this example, the only column we will concern ourselves with is the `symptoms` column:

```
symptoms <- ehr %>%
  select(symptoms) %>%
  print()
```

```
# A tibble: 15 x 1
   symptoms
   <chr>
 1 "\"Pain\", \"Headache\", \"Nausea\""
 2 "Pain"
 3 "Pain"
 4 "\"Nausea\", \"Headache\""
 5 "\"Pain\", \"Headache\""
 6 "\"Nausea\", \"Headache\""
 7 "Pain"
 8  <NA>
 9 "Pain"
10  <NA>
11 "\"Nausea\", \"Headache\""
12 "\"Headache\", \"Pain\", \"Nausea\""
13 "Headache"
14 "\"Headache\", \"Pain\", \"Nausea\""
15  <NA>
```

You may recall that we created dummy variables for each symptom like this:

```
symptoms <- symptoms %>%
  mutate(
    pain     = str_detect(symptoms, "Pain"),
    headache = str_detect(symptoms, "Headache"),
    nausea   = str_detect(symptoms, "Nausea")
  ) %>%
  print()
```

```
# A tibble: 15 x 4
   symptoms                                pain  headache nausea
   <chr>                                   <lgl> <lgl>    <lgl>
 1 "\"Pain\", \"Headache\", \"Nausea\""    TRUE  TRUE     TRUE
 2 "Pain"                                  TRUE  FALSE    FALSE
 3 "Pain"                                  TRUE  FALSE    FALSE
 4 "\"Nausea\", \"Headache\""              FALSE TRUE     TRUE
 5 "\"Pain\", \"Headache\""                TRUE  TRUE     FALSE
 6 "\"Nausea\", \"Headache\""              FALSE TRUE     TRUE
 7 "Pain"                                  TRUE  FALSE    FALSE
 8  <NA>                                   NA    NA       NA
 9 "Pain"                                  TRUE  FALSE    FALSE
10  <NA>                                   NA    NA       NA
11 "\"Nausea\", \"Headache\""              FALSE TRUE     TRUE
12 "\"Headache\", \"Pain\", \"Nausea\""    TRUE  TRUE     TRUE
13 "Headache"                              FALSE TRUE     FALSE
14 "\"Headache\", \"Pain\", \"Nausea\""    TRUE  TRUE     TRUE
15  <NA>                                   NA    NA       NA
```

> **Note**
>
> Some of you may have noticed that we repeated ourselves more than twice in the code chunk above and thought about using `across()` to remove it. Unfortunately, `across()` won't solve our problem in this situation. We will need some of the tools that we learn about in later chapters if we want to remove this repetition.

And finally, we used the `table()` function to get a count of how many people reported having a headache:

```
table(symptoms$headache)
```

```
FALSE  TRUE
    4     8
```

This is where the example stopped in the chapter on working with character strings. However, what if we wanted to know how many people reported the other symptoms as well? Well, we could repeatedly call the `table()` function:

```
table(symptoms$pain)
```

```
FALSE  TRUE
    4     8
```

```
table(symptoms$nausea)
```

```
FALSE  TRUE
    6     6
```

But, that would cause us to copy and paste repeatedly. Additionally, wouldn't it be nice to view these counts in a way that makes them easier to compare? One solution would be to use `summarise()` like this:

```
symptoms %>%
  summarise(
    had_headache = sum(headache, na.rm = TRUE),
    had_pain     = sum(pain, na.rm = TRUE),
    had_nausea   = sum(nausea, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 3
  had_headache had_pain had_nausea
         <int>    <int>      <int>
1            8        8          6
```

This works, but we can do better with `across()`:

```
symptoms %>%
  summarise(
    across(
      .cols  = c(headache, pain, nausea),
      .fns   = ~ sum(.x, na.rm = TRUE)
    )
  )
```

```
# A tibble: 1 x 3
  headache  pain nausea
     <int> <int>  <int>
1        8     8      6
```

Great! But, wouldn't it be nice to know the proportion of people with each symptom as well? You may recall that R treats `TRUE` and `FALSE` as `1` and `0` when used in a mathematical operation. Additionally, you may already be aware that the mean of a set of `1`'s and `0`'s is equal to the proportion of `1`'s in the set. For example, there are three ones and three zeros in the set `(1, 1, 1, 0, 0, 0)`. The proportion of `1`'s in the set is 3 out of 6, which is 0.5. Equivalently, the mean value of the set is (1 + 1 + 1 + 0 + 0 + 0) / 6, which equals 3 / 6, which is 0.5. So, when we have dummy variables like `headache`, `pain`, and `nausea` above, passing them to the `mean()` function returns the proportion of `TRUE` values. In this case, the proportion of people who had each symptom. We know we can do that calculation like this:

```
symptoms %>%
  summarise(
    had_headache = mean(headache, na.rm = TRUE),
    had_pain     = mean(pain, na.rm = TRUE),
    had_nausea   = mean(nausea, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 3
  had_headache had_pain had_nausea
         <dbl>    <dbl>      <dbl>
1        0.667    0.667        0.5
```

As before, we can do better with the `across()` function like this:

```
symptoms %>%
  summarise(
    across(
      .cols = c(pain, headache, nausea),
      .fns  = ~ mean(.x, na.rm = TRUE)
    )
  )
```

```
# A tibble: 1 x 3
   pain headache nausea
  <dbl>    <dbl>  <dbl>
1 0.667    0.667    0.5
```

Now, at this point, we might think, "wouldn't it be nice to see the count *and* the proportion in the same result?" Well, we can do that by supplying our purrr-style lambdas as functions in a list of name-function pairs like this:

```
symptom_summary <- symptoms %>%
  summarise(
    across(
      .cols = c(pain, headache, nausea),
      .fns  = list(
        count = ~ sum(.x, na.rm = TRUE),
        prop  = ~ mean(.x, na.rm = TRUE)
      )
    )
  ) %>%
  print()
```

```
# A tibble: 1 x 6
  pain_count pain_prop headache_count headache_prop nausea_count nausea_prop
       <int>     <dbl>          <int>         <dbl>        <int>       <dbl>
1          8     0.667              8         0.667            6         0.5
```

In this case, it's probably fine to stop here. But, what if we had 20 or 30 symptoms that we were analyzing? It would be really difficult to read and compare them arranged horizontally like this, wouldn't it?

Do you recall us discussing restructuring our results in the chapter on restructuring data frames? This is a circumstance where we might want to use `pivot_longer()` to make our results easier to read and interpret:

```
symptom_summary %>%
  tidyr::pivot_longer(
    cols      = everything(),
    names_to  = c("symptom", ".value"),
    names_sep = "_"
  )
```

```
# A tibble: 3 x 3
  symptom  count  prop
  <chr>    <int> <dbl>
1 pain         8 0.667
2 headache     8 0.667
3 nausea       6 0.5
```

There! Isn't that result much easier to read?

For our final example of this section, let's return the first example from the writing functions chapter. We started with some simulated study data:

```
study <- tibble(
  age       = c(32, 30, 32, 29, 24, 38, 25, 24, 48, 29, 22, 29, 24, 28, 24, 25,
                25, 22, 25, 24, 25, 24, 23, 24, 31, 24, 29, 24, 22, 23, 26, 23,
                24, 25, 24, 33, 27, 25, 26, 26, 26, 26, 26, 27, 24, 43, 25, 24,
                27, 28, 29, 24, 26, 28, 25, 24, 26, 24, 26, 31, 24, 26, 31, 34,
                26, 25, 27, NA),
  age_group = c(2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2,
                2, 1, 1, 1, NA),
  gender    = c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1,
                1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1,
                1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 1,
                1, 1, 2, 1, NA),
  ht_in     = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
                64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
                64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
                69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
                61, 69, 66, NA),
  wt_lbs    = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
                125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
                186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
                147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
                110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
                163, 141, NA),
  bmi       = c(30.99, 18.78, 26.52, 30.54, 22.39, 26.12, 23.69, 20.67, 26.29,
                25.39, 25.68, 45.15, 21.56, 20.17, 17.38, 20.8, 22.31, 22.75,
                26.62, 21.43, 19.14, 23.49, 22.66, 32.98, 25.05, 18.31, 29.13,
                27.07, 20.37, 25.01, 19.69, 25.97, 18.88, 20.07, NA, 26.76,
                26.97, 25.24, 20.68, 23.72, 24.82, 23.62, 18.65, 24.03, 23.86,
                10.63, 23.02, 23.72, 20.82, 28.24, NA, 37.55, 18.88, 18.3,
                19.13, 21.48, 22.59, 24.96, 21.63, NA, 29.35, 21.76, 17.97,
                22.31, 19.27, 24.07, 22.76, NA),
  bmi_3cat  = c(3, 1, 2, 3, 1, 2, 1, 1, 2, 2, 2, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1,
                1, 1, 3, 2, 1, 2, 2, 1, 2, 1, 2, 1, 1, NA, 2, 2, 2, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 2, NA, 3, 1, 1, 1, 1, 1, 1, 1, NA, 2, 1,
                1, 1, 1, 1, 1, NA)
) %>%
```

```
  mutate(
    age_group = factor(age_group, labels = c("Younger than 30", "30 and Older")),
    gender    = factor(gender, labels = c("Female", "Male")),
    bmi_3cat  = factor(bmi_3cat, labels = c("Normal", "Overweight", "Obese"))
  ) %>%
  print()
```

```
# A tibble: 68 x 7
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese
 2    30 30 and Older    Female    63    106  18.8 Normal
 3    32 30 and Older    Female    62    145  26.5 Overweight
 4    29 Younger than 30 Male      67    195  30.5 Obese
 5    24 Younger than 30 Female    67    143  22.4 Normal
 6    38 30 and Older    Female    58    125  26.1 Overweight
 7    25 Younger than 30 Female    64    138  23.7 Normal
 8    24 Younger than 30 Male      69    140  20.7 Normal
 9    48 30 and Older    Male      65    158  26.3 Overweight
10    29 Younger than 30 Male      68    167  25.4 Overweight
# i 58 more rows
```

And wrote our own function to calculate the number of missing values, mean, median, min, and max for all of the continuous variables:

```
continuous_stats <- function(var) {
  study %>%
    summarise(
      n_miss = sum(is.na({{ var }})),
      mean   = mean({{ var }}, na.rm = TRUE),
      median = median({{ var }}, na.rm = TRUE),
      min    = min({{ var }}, na.rm = TRUE),
      max    = max({{ var }}, na.rm = TRUE)
    )
}
```

We then used that function to calculate our statistics of interest for each continuous variable:

```
continuous_stats(age)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      1  26.9     26    22    48
```

```
continuous_stats(ht_in)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      3  66.0     66    58    76
```

```
continuous_stats(wt_lbs)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      2  148.   142.    60   297
```

```
continuous_stats(bmi)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      4  23.6   22.9  10.6  45.2
```

This is definitely an improvement over all the copying and pasting we were doing before we wrote our own function. However, there is still some unnecessary repetition above. One way we can remove this repetition is to use `across()` like this:

```
summary_stats <- study %>%
  summarise(
    across(
      .cols = c(age, ht_in, wt_lbs, bmi),
      .fns  = list(
        n_miss = ~ sum(is.na(.x)),
        mean   = ~ mean(.x, na.rm = TRUE),
        median = ~ median(.x, na.rm = TRUE),
        min    = ~ min(.x, na.rm = TRUE),
```

```
        max    = ~ max(.x, na.rm = TRUE)
      )
    )
  ) %>%
  print()
```

```
# A tibble: 1 x 20
  age_n_miss age_mean age_median age_min age_max ht_in_n_miss ht_in_mean
       <int>    <dbl>      <dbl>   <dbl>   <dbl>        <int>      <dbl>
1          1     26.9         26      22      48            3       66.0
# i 13 more variables: ht_in_median <dbl>, ht_in_min <dbl>, ht_in_max <dbl>,
#   wt_lbs_n_miss <int>, wt_lbs_mean <dbl>, wt_lbs_median <dbl>,
#   wt_lbs_min <dbl>, wt_lbs_max <dbl>, bmi_n_miss <int>, bmi_mean <dbl>,
#   bmi_median <dbl>, bmi_min <dbl>, bmi_max <dbl>
```

This method works, but it has the same problem that our symptom summaries had above. Our results are hard to read and interpret because they are arranged horizontally. We can once again pivot this data longer, but it won't be *quite* as easy as it was before. Our first attempt might look like this:

```
summary_stats %>%
  tidyr::pivot_longer(
    cols      = everything(),
    names_to  = c("characteristic", ".value"),
    names_sep = "_"
  )
```

```
Warning: Expected 2 pieces. Additional pieces discarded in 12 rows [1, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16].
```

```
# A tibble: 12 x 8
   characteristic     n  mean median   min   max  `in`   lbs
   <chr>          <int> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
 1 age                1  26.9   26    22    48    NA     NA
 2 ht                NA  NA     NA    NA    NA     3     NA
 3 ht                NA  NA     NA    NA    NA    66.0   NA
 4 ht                NA  NA     NA    NA    NA    66     NA
 5 ht                NA  NA     NA    NA    NA    58     NA
 6 ht                NA  NA     NA    NA    NA    76     NA
 7 wt                NA  NA     NA    NA    NA    NA      2
```

```
 8 wt                NA  NA     NA    NA    NA    NA    148.
 9 wt                NA  NA     NA    NA    NA    NA    142.
10 wt                NA  NA     NA    NA    NA    NA     60
11 wt                NA  NA     NA    NA    NA    NA    297
12 bmi                4  23.6   22.9  10.6  45.2  NA     NA
```

What do you think the problem is here?

Well, we passed an underscore to the `names_sep` argument. This tells `pivot_longer()` that that character string on the left side of the underscore should make up the values of the new `characteristic` column and each unique character string on the right side of the underscore should be used to create a new column name. In the symptoms data, this worked fine because all of the column names followed this pattern (e.g., `pain_count` and `pain_prop`). But, do the column names in `summary_stats` always follow this pattern? What about `age_n_miss` and `ht_in_n_miss`? All the extra underscores in the column names makes this pattern ineffective.

There are probably many ways we could address this problem. We think the most straightforward way is probably to go back to the code we used to create `summary_stats` and use the `.names` argument to separate the column name and statistic name with a character other than an underscore. Maybe a hyphen instead:

```
summary_stats <- study %>%
  summarise(
    across(
      .cols  = c(age, ht_in, wt_lbs, bmi),
      .fns   = list(
        n_miss = ~ sum(is.na(.x)),
        mean   = ~ mean(.x, na.rm = TRUE),
        median = ~ median(.x, na.rm = TRUE),
        min    = ~ min(.x, na.rm = TRUE),
        max    = ~ max(.x, na.rm = TRUE)
      ),
      .names = "{col}-{fn}" # This is the new part of the code
    )
  ) %>%
  print()
```

```
# A tibble: 1 x 20
  `age-n_miss` `age-mean` `age-median` `age-min` `age-max` `ht_in-n_miss`
         <int>      <dbl>        <dbl>     <dbl>     <dbl>          <int>
1            1       26.9           26        22        48              3
# i 14 more variables: `ht_in-mean` <dbl>, `ht_in-median` <dbl>,
```

```
#   `ht_in-min` <dbl>, `ht_in-max` <dbl>, `wt_lbs-n_miss` <int>,
#   `wt_lbs-mean` <dbl>, `wt_lbs-median` <dbl>, `wt_lbs-min` <dbl>,
#   `wt_lbs-max` <dbl>, `bmi-n_miss` <int>, `bmi-mean` <dbl>,
#   `bmi-median` <dbl>, `bmi-min` <dbl>, `bmi-max` <dbl>
```

Now, we can simply pass a hyphen to the `names_sep` argument to `pivot_longer()`:

```
summary_stats %>%
  tidyr::pivot_longer(
    cols      = everything(),
    names_to  = c("characteristic", ".value"),
    names_sep = "-"
  )
```

```
# A tibble: 4 x 6
  characteristic n_miss  mean median   min   max
  <chr>           <int> <dbl>  <dbl> <dbl> <dbl>
1 age                 1  26.9   26    22    48
2 ht_in               3  66.0   66    58    76
3 wt_lbs              2 148.   142.   60   297
4 bmi                 4  23.6   22.9  10.6  45.2
```

Look at how much easier those results are to read!

```
rm(study, summary_stats, continuous_stats)
```

35.4 Across with filter

We've already discussed complete case analysis multiple times in this book. That is, including only the rows from our data frame that don't have any missing values in our analysis. Additionally, we've already seen how we can use the `filter()` function to remove the rows of a *single* column where the data are missing. For example:

```
df_xyz %>%
  filter(!is.na(x))
```

```
# A tibble: 9 x 4
    row       x      y      z
  <int>   <dbl>  <dbl>  <dbl>
```

```
1     1 -0.560   1.22  -1.07
2     3  1.56    0.401 -1.03
3     4  0.0705 NA     -0.729
4     5  0.129  -0.556 -0.625
5     6  1.72    1.79  NA
6     7  0.461   0.498  0.838
7     8 -1.27   -1.97   0.153
8     9 -0.687   0.701 -1.14
9    10 -0.446  -0.473  1.25
```

Notice that row 2 – the row that had a missing value for `x` – is no longer in the data frame, and we can now easily calculate the mean value of `x`.

```
df_xyz %>%
  filter(!is.na(x)) %>%
  summarise(mean = mean(x))
```

```
# A tibble: 1 x 1
   mean
  <dbl>
1 0.108
```

However, we want to remove the rows that have a missing value in *any* column – not just `x`. We could get this result using multiple sequential `filter()` functions like this:

```
df_xyz %>%
  filter(!is.na(x)) %>%
  filter(!is.na(y)) %>%
  filter(!is.na(z))
```

```
# A tibble: 7 x 4
    row      x      y      z
  <int>  <dbl>  <dbl>  <dbl>
1     1 -0.560  1.22  -1.07
2     3  1.56   0.401 -1.03
3     5  0.129 -0.556 -0.625
4     7  0.461  0.498  0.838
5     8 -1.27  -1.97   0.153
6     9 -0.687  0.701 -1.14
7    10 -0.446 -0.473  1.25
```

As you can see, rows 2, 4, and 6 – the rows with a missing value for `x`, `y`, and `z` – were dropped.

Of course, in the code chunk above, we have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code.

At this point in the book, our first thought might be to use the `across()` function, inside the `filter()` function, to remove *all* of the rows rows with missing values from our data frame. However, as of `dplyr version 1.0.4`, using the `across()` function inside of `filter()` is deprecated. That means we shouldn't use it anymore. Instead, we should use the `if_any()` or `if_all()` functions, which take the exact same arguments as `across()`. In the code chunk below, we will show you how to solve this problem, then we will dissect the solution below.

```
df_xyz %>%
  filter(
    if_all(
      .cols = c(x:z),
      .fns  = ~ !is.na(.x)
    )
  )
```

```
# A tibble: 7 x 4
    row      x      y      z
  <int>  <dbl>  <dbl>  <dbl>
1     1 -0.560  1.22  -1.07
2     3  1.56   0.401 -1.03
3     5  0.129 -0.556 -0.625
4     7  0.461  0.498  0.838
5     8 -1.27  -1.97   0.153
6     9 -0.687  0.701 -1.14
7    10 -0.446 -0.473  1.25
```

Here's what we did above:

- You can type `?dplyr::if_any` or `?dplyr::if_all` into your R console to view the help documentation for this function and follow along with the explanation below.
- We used the `if_all()` function inside of the `filter()` function to keep only the rows in our data frame that had nonmissing values for *all* of the columns `x`, `y`, and `z`.
- We passed the value `c(x:z)` to the `.cols` argument. This told R to apply the function passed to the `.fns` argument to the columns `x` through `z` inclusive.

- We used a purrr-style lambda to test whether or not each value of each of the columns passed to `.cols` is NOT missing.
- Remember, the special `.x` symbol is just shorthand for each column passed to the `.cols` argument.

So, how does this work? Well, first let's remember that the `is.na()` function returns `TRUE` when the value of the vector passed to it is missing and `FALSE` when it is not missing. For example:

```
is.na(df_xyz$x)
```

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

We can then use the `!` operator to "flip" those results. In other words, to return `TRUE` when the value of the vector passed to it is *not* missing and `FALSE` when it is missing. For example:

```
!is.na(df_xyz$x)
```

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

The filter() function] then returns the rows from the data frame where the values returned by `!is.na()` are `TRUE` and drops the rows where they are `FALSE`. For example, we can copy and paste the TRUE/FALSE values above to keep only the rows with nonmissing values for `x`:

```
df_xyz %>%
  filter(c(TRUE, FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE))
```

```
# A tibble: 9 x 4
    row       x      y      z
  <int>   <dbl>  <dbl>  <dbl>
1     1 -0.560   1.22  -1.07
2     3  1.56    0.401 -1.03
3     4  0.0705 NA     -0.729
4     5  0.129  -0.556 -0.625
5     6  1.72    1.79  NA
6     7  0.461   0.498  0.838
7     8 -1.27   -1.97   0.153
8     9 -0.687   0.701 -1.14
9    10 -0.446  -0.473  1.25
```

Now, let's repeat this process for the columns `y` and `z` as well.

```
!is.na(df_xyz$y)
```

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

```
!is.na(df_xyz$z)
```

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

Next, let's stack these results next to each other to make them even easier to view.

```
not_missing <- tibble(
  row = 1:10,
  x   = !is.na(df_xyz$x),
  y   = !is.na(df_xyz$y),
  z   = !is.na(df_xyz$z)
) %>%
  print()
```

```
# A tibble: 10 x 4
     row x     y     z
   <int> <lgl> <lgl> <lgl>
 1     1 TRUE  TRUE  TRUE
 2     2 FALSE TRUE  TRUE
 3     3 TRUE  TRUE  TRUE
 4     4 TRUE  FALSE TRUE
 5     5 TRUE  TRUE  TRUE
 6     6 TRUE  TRUE  FALSE
 7     7 TRUE  TRUE  TRUE
 8     8 TRUE  TRUE  TRUE
 9     9 TRUE  TRUE  TRUE
10    10 TRUE  TRUE  TRUE
```

Here's what we did above:

- We created a data frame that contains the value `TRUE` in each position where `df_xyz` has a nonmissing value and `FALSE` in each position where `df_xyz` has a missing value. We wouldn't typically create this for our data analysis. We just created it here for teaching purposes.

You can think of the data frame of `TRUE` and `FALSE` values above as an intermediate product that `if_any()` and `if_all()` uses "under the hood" to decide which rows to keep. We think using this data frame as a conceptual model makes it a little easier to understand how `if_any()` and `if_all()` differ.

`if_any()` will keep the rows where *any* value of `x`, `y`, *or* `z` are `TRUE`. In this case, there is at least one `TRUE` value in every row. Therefore, we would expect `if_any()` to return all rows in our data frame. And, that's exactly what happens.

```
df_xyz %>%
  filter(
    if_any(
      .cols = c(x:z),
      .fns  = ~ !is.na(.x)
    )
  )
```

```
# A tibble: 10 x 4
     row       x      y      z
   <int>   <dbl>  <dbl>  <dbl>
 1     1 -0.560   1.22  -1.07
 2     2 NA       0.360 -0.218
 3     3  1.56    0.401 -1.03
 4     4  0.0705 NA     -0.729
 5     5  0.129  -0.556 -0.625
 6     6  1.72    1.79  NA
 7     7  0.461   0.498  0.838
 8     8 -1.27   -1.97   0.153
 9     9 -0.687   0.701 -1.14
10    10 -0.446  -0.473  1.25
```

On the other hand, `if_all()` will the keep the rows where *all* value of `x`, `y`, *and* `z` are `TRUE`. In this case, there is at least one `FALSE` value in rows 2, 4, and 6. Therefore, we would expect `if_all()` to return all rows in our data frame *except* rows 2, 4, and 6. That's exactly what happens, and it's exaclty the result we want.

```
df_xyz %>%
  filter(
    if_all(
      .cols = c(x:z),
      .fns  = ~ !is.na(.x)
    )
  )
```

```
# A tibble: 7 x 4
    row      x      y      z
  <int>  <dbl>  <dbl>  <dbl>
1     1 -0.560  1.22  -1.07
2     3  1.56   0.401 -1.03
3     5  0.129 -0.556 -0.625
4     7  0.461  0.498  0.838
5     8 -1.27  -1.97   0.153
6     9 -0.687  0.701 -1.14
7    10 -0.446 -0.473  1.25
```

Because this is a small, simple example, using `if_all()` doesn't actually reduce the number of lines of code we wrote. But again, try to imagine if we added 20 additional columns to our data frame. We would only need to update the value we pass to the `.cols` argument. This makes our code more concise, easier to maintain, and less error-prone.

35.5 Summary

We are big fans of using `across()`, `if_any()`, and `if_all()` in conjunction with the `dplyr` verbs. They allows us to remove a lot of the unnecessary repetition from our code in a way that integrates pretty seamlessly with the tools we are already using. Perhaps you will see value in using these functions as well. In the next chapter, we will learn about using for loops to remove unnecessary repetition from our code.

36 Writing For Loops

In this third chapter on repeated operations, we are going to discuss writing for loops.

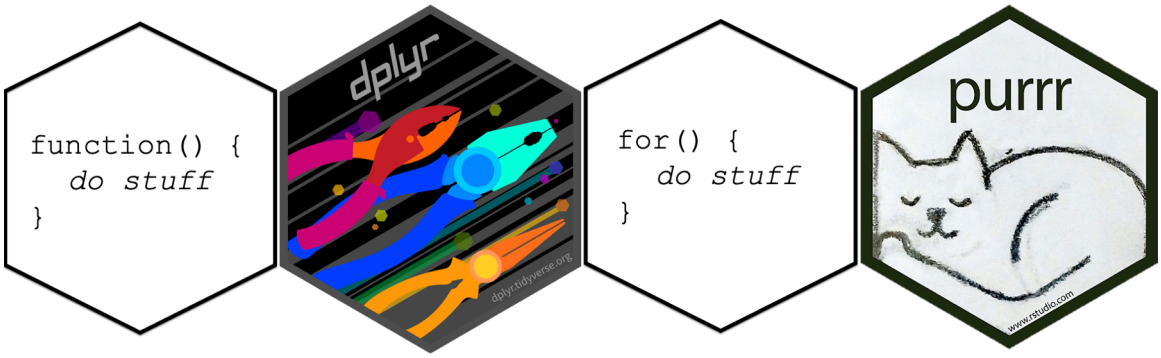


Figure 36.1: For loops graphic

In other documents you read, you may see for loops referred to as iterative processing, iterative operations, iteration, or just loops. Regardless of what you call them, for loops are not unique to R. Many if not all statistical software applications allow users to write for loops; although, the exact words and symbols used to construct them may differ slightly from one program to another.

Let's take a look at an example. After seeing a working example, we will take the code apart iteratively (do you see what we did there?) to figure out how it works.

We'll start by simulating some data. This is the same data we simulated at the beginning of the chapter on column-wise operations in `dplyr`. It's a data frame that contains three columns of 10 random numbers:

```
library(dplyr)
```

```
set.seed(123)
df_xyz <- tibble(
  x = rnorm(10),
  y = rnorm(10),
  z = rnorm(10)
) %>%
  print()
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 -0.230   0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705  0.111 -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  -1.69
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

As we previously discussed, if we wanted to find the mean of each column before learning about repeated operations, we would probably have written code like this:

```
df_xyz %>%
  summarise(
    x_mean = mean(x),
    y_mean = mean(y),
    z_mean = mean(y)
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1 0.0746  0.209  0.209
```

In the previous chapter, we learned how to use the `across()` function to remove unnecessary repetition from our code like this:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = mean,
      .names = "{col}_mean"
    )
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1 0.0746  0.209 -0.425
```

An alternative approach that would also work is to use a for loop like this:

```
xyz_means <- vector("double", ncol(df_xyz))

for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}
```

```
xyz_means
```

```
[1]  0.07462564  0.20862196 -0.42455887
```

Most people would agree that the for loop code is a little more complicated looking, and it's a little bit harder to quickly glance at it and figure out what's going on. It may even be a little bit intimidating for some of you.

Also, note that the result from the code that uses the `across()` function is a data frame with three columns and one row. The result from the code that uses a for loop is a character vector with three elements.

For the particular case above, we prefer to use the `across()` function instead of a for loop. However, as we will see below, there are some challenges that can be overcome with for loops that cannot currently be overcome with the `across()` function. But, before we jump into more examples, let's take a look at the basic structure of the for loop.

36.1 How to write for loops

For starters, *using* for loops in practice will generally require us to write code for two separate structures: An object to contain the results of our for loop and the for loop itself.

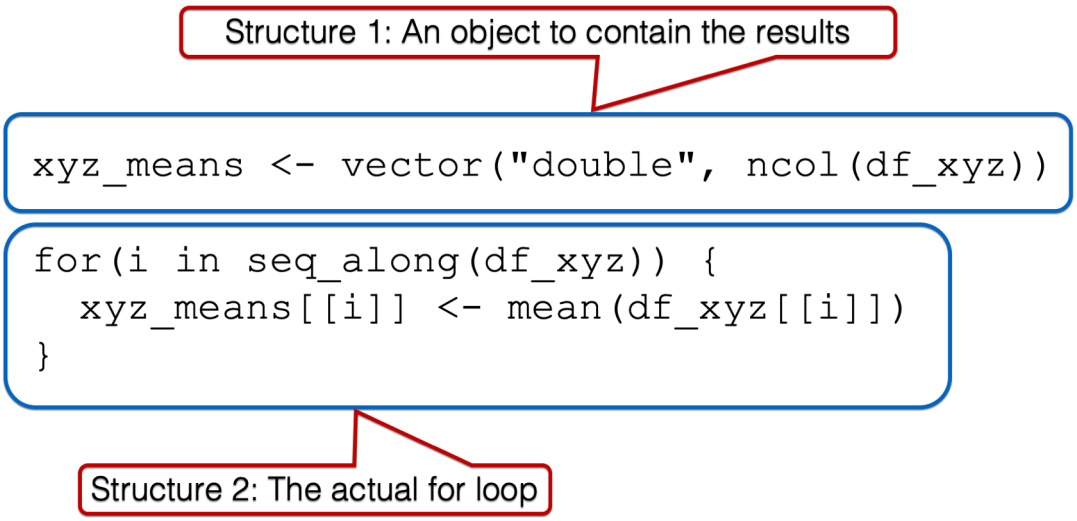


Figure 36.2: Two structures required for a for loop

In practice, we will generally write the code for structure 1 before writing the code for structure 2. However, it will be easier to understand why we need structure 1 if we first learn about the components of the for loop, and how they work together. Further, it will likely be easiest to understand the components of the for loop if we start on the inside and work our way out. Therefore, the first component of for loops that we are going to discuss is the body.

36.1.1 The for loop body

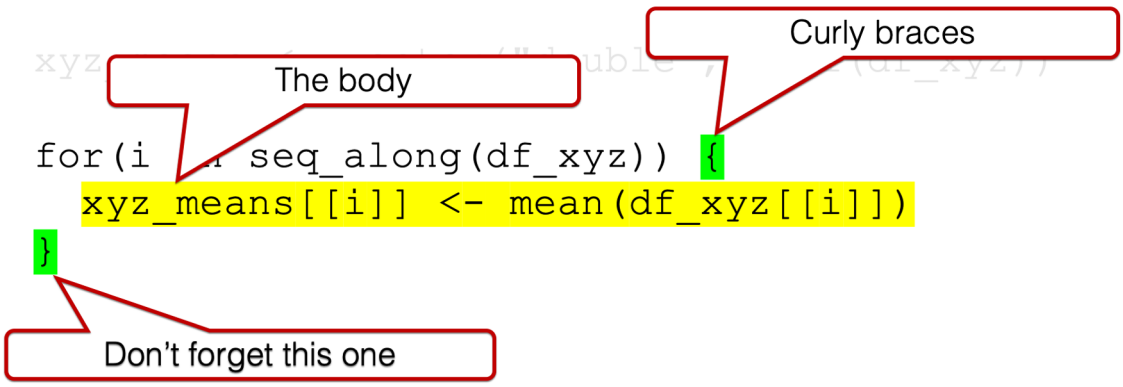


Figure 36.3: Body of the for loop

Similar to when we learned to write our own functions, the body of the for loop is where all the "stuff" happens. This is where we write the code that we want to be executed over and over. In our example, we want the mean value of the `x` column, the mean value of the `y` column, and the mean value of the `z` column of our data frame called `df_xyz`. We can do that manually like this using dollar sign notation:

```
mean(df_xyz$x)
```

```
[1] 0.07462564
```

```
mean(df_xyz$y)
```

```
[1] 0.208622
```

```
mean(df_xyz$z)
```

```
[1] -0.4245589
```

Or, we've also learned how to get the same result using bracket notation:

```
mean(df_xyz[["x"]])
```

```
[1] 0.07462564
```

```
mean(df_xyz[["y"]])
```

```
[1] 0.208622
```

```
mean(df_xyz[["z"]])
```

```
[1] -0.4245589
```

In the code above, we used the quoted column *names* inside the double brackets. However, we could have also used each column's *position* inside the double brackets. In other words, we can use `1` to refer to the `x` column because it is the first column in the data frame, we can use `2` to refer to the `y` column because it is the second column in the data frame, and we can use `3` to refer to the `z` column because it is the third column in the data frame:

```
mean(df_xyz[[1]])
```

```
[1] 0.07462564
```

```
mean(df_xyz[[2]])
```

```
[1] 0.208622
```

```
mean(df_xyz[[3]])
```

```
[1] -0.4245589
```

For reasons that will become clearer later, this will actually be the syntax we want to use inside of our for loop.

Notice, however, the we copied the same code more than twice above. For all of the reasons we've already discussed, we would like to just type `mean(df_xyz[[ # ]]` once and have R fill in the number inside the double brackets for us, one after the other. As you've probably guessed, that's exactly what the for loop does.

36.1.2 The for() function

All for loops start with the `for()` function. This is how you tell R that you are about to write a for loop.

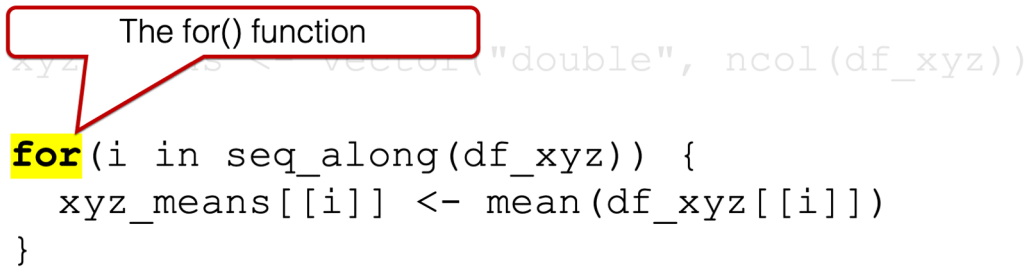


Figure 36.4: The for() function

In the examples in this book, the arguments to the `for()` function are generally going to follow this pattern:

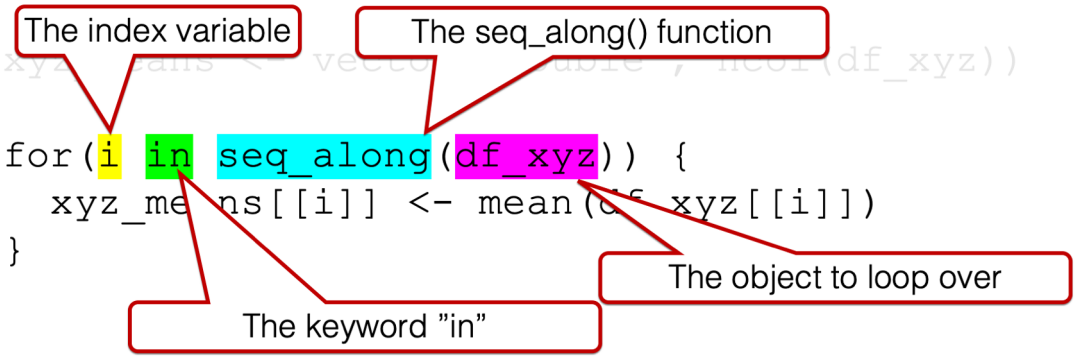


Figure 36.5: Pattern of the for() function arguments

1 An index variable, which is also sometimes called a "counter," to the left of the keyword `in`.

2 The keyword `in`.

3 The name of the object we want to loop (or iterate) over — often passed to the`seq_along()` function.

It can be a little intimidating to look at, but that's the basic structure. We will talk about all three arguments simultaneously because they all work together, and we will get an error if we are missing any one of them:

```
# No index variable
for(in 1) {
  print(i)
}
```

```
Error in parse(text = input): <text>:2:5: unexpected 'in'
1: # No index variable
2: for(in
       ^
```

```
# No keyword "in"
for(i 1) {
  print(i)
}
```

```
Error in parse(text = input): <text>:2:7: unexpected numeric constant
1: # No keyword "in"
2: for(i 1
         ^
```

```
# No object to loop over
for(i in ) {
  print(i)
}
```

```
Error in parse(text = input): <text>:2:10: unexpected ')'
1: # No object to loop over
2: for(i in )
            ^
```

So, what happens when we do have all three of these components? Well, the index variable will take on each value of the object to loop over *iteratively* (i.e., one at a time). If there is only one object to loop over, this is how R sees the index variable inside of the loop:

```
for(i in 1) {
  print(i)
}
```

```
[1] 1
```

If there are multiple objects to loop over, this is how R sees the index variable inside of the loop:

```
for(i in c(1, 2, 3)) {
  print(i)
}
```

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

Notice that the values being printed out are *not* a single numeric vector with three elements (e.g. `[1] 1, 2, 3`) like the object we started with to the right of the keyword `in`. Instead, three vectors with one element each are being printed out. One for 1 (i.e., `[1] 1`), one for 2 (i.e., `[1] 2`), and one for 3 (i.e., `[1] 3`). This is pointed out because it illustrates the *iterative* nature of a for loop. The index variable doesn't take on the values of the object to the right of the keyword `in` *simultaneously*. It takes them on *iteratively*, or separately, one after the other.

Further, it may not be immediately obvious at this point, but that's the basic "magic" of the for loop. *The index variable changes once for each element of whatever object is on the right side of the keyword* ***in***. Even the most complicated for loops generally start from this basic idea.

Note that the index variable does not have to be the letter `i`. It can be any letter:

```
for(x in c(1, 2, 3)) {
  print(x)
}
```

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

Or even a word:

```
for(number in c(1, 2, 3)) {
  print(number)
}
```

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

However, `i` is definitely the most common letter to use as the index variable and we suggest that you also use it in most cases. It's just what people will expect to see and easily understand.

Now, let's discuss the object to the right of the keyword `in`. In all of the examples above, we passed a vector to the right of the keyword `in`. As you saw, when there is a vector to the right of the keyword `in`, the index variable takes on the value of each element of the vector. However, the object to the right of the keyword `in` does not have to be a vector. In fact, it will often be a data frame.

When we ask the for loop to iterate over a data frame, what value do you think the index variable will take? The value of each cell of the data frame? The name or number of each column? The name or number of each row? Let's see:

```
for(i in df_xyz) {
  print(i)
}
```

```
 [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
 [1]  1.2240818  0.3598138  0.4007715  0.1106827 -0.5558411  1.7869131
 [7]  0.4978505 -1.9666172  0.7013559 -0.4727914
 [1] -1.0678237 -0.2179749 -1.0260044 -0.7288912 -0.6250393 -1.6866933
 [7]  0.8377870  0.1533731 -1.1381369  1.2538149
```

It may not be totally obvious to you, but inside the for loop above, the index variable took on three separate *vectors* of values – one for each column in the data frame. Of course, getting the mean value of each of these *vectors* is equivalent to getting the mean value of each *column* in our data frame. Remember, data frame columns *are* vectors. So, let's replace the `print()` function with the `mean()` function in the for loop body and see what happens:

```
for(i in df_xyz) {
  mean(i)
}
```

Hmmm, it doesn't seem as though anything happened. This is probably a good time to mention a little peculiarity about using for loops. As you can see in the example above, the return value of functions, and the contents of objects, referenced inside of the for loop body will not be printed to the screen unless we explicitly pass them to the `print()` function:

```
for(i in df_xyz) {
  print(mean(i))
}
```

```
[1] 0.07462564
[1] 0.208622
[1] -0.4245589
```

It worked! This is the exact same answer we got above. And, if all we want to do is print the mean values of `x`, `y`, and `z` to the screen, then we could stop here and call it a day. However, we often want to save our analysis results to an object. In the chapter on using column-wise

operations with `dplyr`, we saved our summary statistics to an object in the usual way (i.e., with the assignment arrow):

```
xyz_means <- df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = mean,
      .names = "{col}_mean"
    )
  )
```

From there, we can manipulate the results, save the results to a file, or print them to screen:

```
xyz_means
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1 0.0746  0.209 -0.425
```

At first, it may seem as though we can assign the results of our for loop to an object in a similar way:

```
xyz_means <- for(i in df_xyz) {
  mean(i)
}
```

```
xyz_means
```

```
NULL
```

Unfortunately, this doesn't work. Instead, we need to create an object that can store the results of our for loop. Then, we *update* (i.e., add to) that object at each iteration of the for loop. That brings us back to structure number 1.

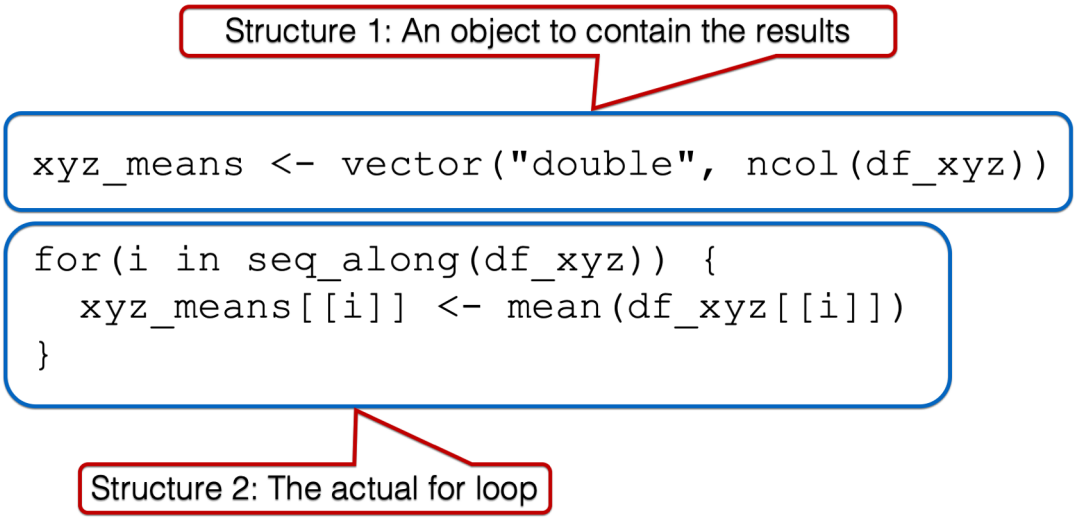


Figure 36.6: Two structures required for a for loop

Because the result of our for loop will be three numbers – the mean of `x`, the mean of `y`, and the mean of `z` – the most straightforward object to store them in is a numeric vector with a length of three (i.e., three "slots"). We can use the `vector()` function to create an empty vector:

```
my_vec <- vector()
my_vec
```

```
logical(0)
```

As you can see, by default, the `vector()` function creates a logical vector with length zero. We can change the vector type to numeric by passing `"numeric"` to the `mode` argument of the `vector()` function. We can also change the length to 3 by passing `3` to the length argument of the `vector()` function, and because we know we want this vector to hold the mean values of `x`, `y`, and `z`, let's name it `xyz_means`:

```
xyz_means <- vector("numeric", 3)
xyz_means
```

```
[1] 0 0 0
```

Finally, let's update `xyz_means` inside our for loop body:

```
for(i in df_xyz) {
  xyz_means <- mean(i)
}
```

```
xyz_means
```

```
[1] -0.4245589
```

Hmmm, we're getting closer, but that obviously still isn't the result we want. Below, we attempt to illustrate what's going on inside our loop.

R starts executing at the top of the for loop. In the first iteration, the value of i is set to a numeric vector with the same values as the `x` column in `df_xyz`. Then, the `i` in `mean(i)` inside the for loop body is replaced with those numeric values. Then, the mean of those numeric values is calculated and assigned to the object named `xyz_means`.

Iteration	i =	Statement Executed	xyz_mean =
One	-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564

```
for(i in df_xyz) {
  xyz_means <- mean(i)
}
```

Figure 36.7: Illustration of a for loop process - I

At this point, there is no more code left to execute inside of the for loop, so R returns to the top of the loop.

Iteration	i =	Statement Executed	xyz_mean =
One	-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564

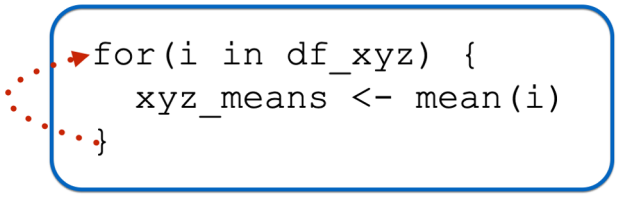


Figure 36.8: Illustration of a for loop process - II

`i` has not yet taken every value of the object to the right of the keyword `in`, so R starts another iteration of the for loop. In the second iteration, the value of `i` is set to a numeric vector with the same values as the `y` column in `df_xyz`. Then, the `i` in `mean(i)` inside the for loop body is replaced with those numeric values. Then, the mean of those numeric values is calculated and assigned to the object named `xyz_means`.

Iteration	i =	Statement Executed	xyz_mean =
One	-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564
Two	1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914	mean(1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914)	0.208622

```
for(i in df_xyz) {
  xyz_means <- mean(i)
}
```

Figure 36.9: Illustration of a for loop process - III

At this point, there is no more code left to execute inside of the for loop, so R returns to the top of the loop.

Iteration	i =	Statement Executed	xyz_mean =
One	-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564
Two	1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914	mean(1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914)	0.208622

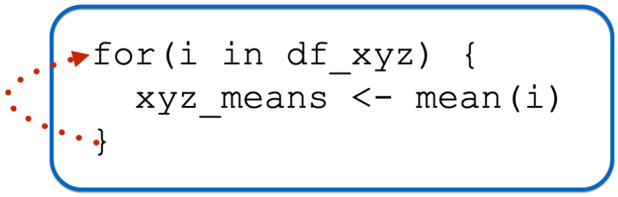


Figure 36.10: Illustration of a for loop process - IV

`i` still has not yet taken every value of the object to the right of the keyword `in`, so R starts another iteration of the for loop. In the third iteration, the value of `i` is set to a numeric vector with the same values as the `z` column in `df_xyz`. Then, the `i` in `mean(i)` inside the for loop body is replaced with those numeric values. Then, the mean of those numeric values is calculated and assigned to the object named `xyz_means`.

Iteration	i =	Statement Executed	xyz_mean =
One	-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564
Two	1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914	mean(1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914)	0.208622
Three	-1.0678237, -0.2179749, -1.0260044, -0.7288912, -0.6250393, -1.6866933, 0.8377870, 0.1533731, -1.1381369, 1.2538149	mean(-1.0678237, -0.2179749, -1.0260044, -0.7288912, -0.6250393, -1.6866933, 0.8377870, 0.1533731, -1.1381369, 1.2538149)	-0.4245589

```
for(i in df_xyz) {
  xyz_means <- mean(i)
}
```

Figure 36.11: Illustration of a for loop process - V

At this point, there is no more code left to execute inside of the for loop, so R returns to the top of the loop. However, this time, `i` has taken every value of the object to the right of the keyword `in`, so R does not start another iteration. It leaves the looping process, and the value of `xyz_means` remains `-0.4245589` – The result we got above.

You might be thinking, "wait, we made three slots in the `xyz_means` vector. Why does it only contain one number?" Well, remember that all we have to do to overwrite one object with another object is to assign the second object to the same name. For example, let's create a vector with three values called `my_vec`:

```
my_vec <- c(1, 2, 3)
my_vec
```

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

Now, let's assign another value to `my_vec`:

```
my_vec <- -0.4245589
my_vec
```

```
[1] -0.4245589
```

As you can see, assignment (<-) doesn't *add to* the vector, it *overwrites* (i.e., replaces) the vector. That's exactly what was happening inside of our for loop. To R, it basically looked like this:

```
xyz_means <- 0.07462564
xyz_means <- 0.208622
xyz_means <- -0.4245589
xyz_means
```

```
[1] -0.4245589
```

What we really want is to create the empty vector:

```
xyz_means <- vector("numeric", 3)
xyz_means
```

```
[1] 0 0 0
```

And then add a value to each *slot* in the vector. Do you remember how to do this?

We can do this using bracket notation:

```
xyz_means[[1]] <- 0.07462564
xyz_means[[2]] <- 0.208622
xyz_means[[3]] <- -0.4245589
xyz_means
```

```
[1]  0.07462564  0.20862200 -0.42455890
```

That's exactly the result we want.

Does that code above remind you of any other code we've already seen? How about this code:

```
mean(df_xyz[[1]])
```

```
[1] 0.07462564
```

```
mean(df_xyz[[2]])
```

```
[1] 0.208622
```

```
mean(df_xyz[[3]])
```

```
[1] -0.4245589
```

Hmmm, what if we combine the two? First, let's once again create our empty vector, and then try combining the two code chunks above to fill it:

```
xyz_means <- vector("numeric", 3)
xyz_means
```

```
[1] 0 0 0
```

```
xyz_means[[1]] <- mean(df_xyz[[1]])
xyz_means[[2]] <- mean(df_xyz[[2]])
xyz_means[[3]] <- mean(df_xyz[[3]])
xyz_means
```

```
[1]  0.07462564  0.20862196 -0.42455887
```

Again, that's exactly the result we want. Of course, there is still unnecessary repetition. If you look at the code carefully, you may notice that the only thing that changes from line to line is the number inside the double brackets. So, if we could just type `xyz_means[[ # ]] <- mean(df_xyz[[ # ]])` once, and update the number inside the double brackets, we should be able to get the result we want. We've actually already seen how to do that with a for loop too. Remember this for loop for the very beginning of the chapter:

```
for(i in c(1, 2, 3)) {
  print(i)
}
```

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

That looks promising, right? Let's once again create our empty vector, and then try combining the two code chunks above to fill it:

```
xyz_means <- vector("numeric", 3)
xyz_means
```

```
[1] 0 0 0
```

```
for(i in c(1, 2, 3)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}
```

```
xyz_means
```

```
[1]  0.07462564  0.20862196 -0.42455887
```

It works! We have used a for loop to successfully remove the unnecessary repetition from our code. However, there's still something we could do to make the code more robust. In the for loop above, we knew that we needed three iterations. Therefore, we passed `c(1, 2, 3)` as the object to the right of the keyword `in`. But, what if we didn't know exactly how columns there were? What if we just knew that we wanted to iterate over all the columns in the data frame passed to the right of the keyword `in`. How could we do that?

We can do that with the `seq_along()` function. When we pass a vector to the `seq_along()` function, it returns a sequence of integers with the same length as the vector being passed, starting at one. For example:

```
seq_along(c(4, 5, 6))
```

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

Or:

```
seq_along(c("a", "b", "c", "d"))
```

```
[1] 1 2 3 4
```

Similarly, when we pass a data frame to the `seq_along()` function, it returns a sequence of integers with a length equal to the number of columns in the data frame being passed, starting at one. For example:

```
seq_along(df_xyz)
```

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

Therefore, we can replace `for(i in c(1, 2, 3))` with `for(i in seq_along(df_xyz))` to make our code more robust (i.e., it will work in more situations):

```
xyz_means <- vector("numeric", 3)

for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}

xyz_means
```

```
[1]  0.07462564  0.20862196 -0.42455887
```

Just to make sure that we really understand what's going on in the code above, let's walk through the entire process one more time.

Iteration	i	df_xyz[[i]]	mean(df_xyz[[i]])	xyz_mean =
One	1	df_xyz[[1]]	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564

```
for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}
```

Figure 36.12: Illustration of a for loop process again - I

R starts executing at the top of the for loop. In the first iteration, the value of i is set to the first value in `seq_along(df_xyz)`, which is `1`. Then, the `i` in `df_xyz[[i]]` inside the for loop body is replaced with `1`. Then, R calculates the mean of `df_xyz[[1]]`, which is `x` column of the `df_xyz` data frame. Finally, the mean value is assigned to `xyz_means[[i]]`, which is `xyz_means[[1]]` in this iteration. So, the value of the first element in the `xyz_means` vector is `0.07462564`.

At this point, there is no more code left to execute inside of the for loop, so R returns to the top of the loop. `i` has not yet taken every value of the object to the right of the keyword `in`, so R starts another iteration of the for loop.

Iteration	i	df_xyz[[i]]	mean(df_xyz[[i]])	xyz_mean =
One	1	df_xyz[[1]]	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564
Two	2	df_xyz[[2]]	mean(1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914)	0.07462564, 0.208622

```
for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}
```

Figure 36.13: Illustration of a for loop process again - II

In the second iteration, the value of i is set to the second value in `seq_along(df_xyz)`, which is `2`. Then, the `i` in `df_xyz[[i]]` inside the for loop body is replaced with `2`. Then, R calculates the mean of `df_xyz[[2]]`, which is `y` column of the `df_xyz` data frame. Finally, the mean value is assigned to `xyz_means[[i]]`, which is `xyz_means[[2]]` in this iteration. So, the value of the second element in the `xyz_means` vector is `0.20862196`.

At this point, there is no more code left to execute inside of the for loop, so R returns to the top of the loop. `i` still has not yet taken every value of the object to the right of the keyword `in`, so R starts another iteration of the for loop.

Iteration	i	df_xyz[[i]]	mean(df_xyz[[i]])	xyz_mean =
One	1	df_xyz[[1]]	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564
Two	2	df_xyz[[2]]	mean(1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914)	0.07462564, 0.208622
Three	3	df_xyz[[3]]	mean(-1.0678237, -0.2179749, -1.0260044, -0.7288912, -0.6250393, -1.6866933, 0.8377870, 0.1533731, -1.1381369, 1.2538149)	0.07462564, 0.208622, -0.4245589

```
for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}
```

Figure 36.14: Illustration of a for loop process again - III

In the third iteration, the value of i is set to the third value in `seq_along(df_xyz)`, which is `3`. Then, the `i` in `df_xyz[[i]]` inside the for loop body is replaced with `3`. Then, R calculates the mean of `df_xyz[[3]]`, which is `z` column of the `df_xyz` data frame. Finally, the mean value is assigned to `xyz_means[[i]]`, which is `xyz_means[[3]]` in this iteration. So, the value of the third element in the `xyz_means` vector is `-0.42455887`.

Iteration	i =	Statement Executed	xyz_mean =
One	-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197	mean(-0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197)	0.07462564
Two	1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914	mean(1.2240818, 0.3598138, 0.4007715, 0.1106827, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914)	0.208622
Three	-1.0678237, -0.2179749, -1.0260044, -0.7288912, -0.6250393, -1.6866933, 0.8377870, 0.1533731, -1.1381369, 1.2538149	mean(-1.0678237, -0.2179749, -1.0260044, -0.7288912, -0.6250393, -1.6866933, 0.8377870, 0.1533731, -1.1381369, 1.2538149)	-0.4245589

```
for(i in df_xyz) {
  xyz_means <- mean(i)
}
```

Figure 36.15: Illustration of a for loop process - V

At this point, there is no more code left to execute inside of the for loop, so R returns to the top of the loop. However, this time, `i` has taken every value of the object to the right of the keyword `in`, so R does not start another iteration. It leaves the looping process, and the value of `xyz_means` remains `0.07462564, 0.20862196, -0.4245589`.

There's one final adjustment we should probably make to the code above. Did you notice that when we create the empty vector to contain our results, we're still hard coding its length to 3? For the same reason we replaced `for(i in c(1, 2, 3))` with `for(i in seq_along(df_xyz))`, we want to replace `vector("numeric", 3)` with `vector("numeric", length(df_xyz))`.

Now, let's add a fourth column to our data frame:

```
df_xyz <- df_xyz %>%
  mutate(a = rnorm(10)) %>%
  print()
```

```
# A tibble: 10 x 4
         x      y      z       a
     <dbl>  <dbl>  <dbl>   <dbl>
 1 -0.560   1.22  -1.07   0.426
 2 -0.230   0.360 -0.218 -0.295
```

```
 3  1.56    0.401 -1.03   0.895
 4  0.0705  0.111 -0.729  0.878
 5  0.129  -0.556 -0.625  0.822
 6  1.72    1.79  -1.69   0.689
 7  0.461   0.498  0.838  0.554
 8 -1.27   -1.97   0.153 -0.0619
 9 -0.687   0.701 -1.14  -0.306
10 -0.446  -0.473  1.25  -0.380
```

And see what happens when we pass it to our new, robust for loop code:

```
xyz_means <- vector("numeric", length(df_xyz)) # Using length() instead of 3

for(i in seq_along(df_xyz)) { # Using seq_along() instead of c(1, 2, 3)
  xyz_means[[i]] <- mean(df_xyz[[i]])
}

xyz_means
```

```
[1]  0.07462564  0.20862196 -0.42455887  0.32204455
```

Our for loop now gives us the result we want no matter how many columns are in the data frame. Having the flexibility to loop over an arbitrary number of columns wasn't that important in this case – we knew exactly how many columns we wanted to loop over. However, what if we wanted to add more columns in the future? Using the second method, we wouldn't have to make any changes to our code. This is often an important consideration when we embed for loops inside of functions that we write ourselves.

For example, maybe we think, "that for loop above was really useful. I want to write it into a function so that I can use it again in my other projects." Well, we've already seen how to take our working code, embed it inside of a function, make it more general, and assign it a name. If you forgot how to do this, please review the function writing process. In this case, that process would result in something like this:

```
multi_means <- function(data) {
  # Create a structure to contain results
  result <- vector("numeric", length(data))

  # Iterate over each column of data
  for(i in seq_along(data)) {
    result[[i]] <- mean(data[[i]])
  }
```

```
  # Return the result
  result
}
```

Which we can easily apply to our data frame like this:

```
multi_means(df_xyz)
```

```
[1]  0.07462564  0.20862196 -0.42455887  0.32204455
```

Further, because we've made the for loop code inside of the function body flexible with `length()` and `seq_along()` we can easily pass any other data frame (with all numeric columns) to our function like this:

```
set.seed(123)
new_df <- tibble(
  age    = rnorm(10, 50, 10),
  height = rnorm(10, 65, 5),
  weight = rnorm(10, 165, 10)
) %>%
  print()
```

```
# A tibble: 10 x 3
     age height weight
   <dbl>  <dbl>  <dbl>
 1  44.4   71.1   154.
 2  47.7   66.8   163.
 3  65.6   67.0   155.
 4  50.7   65.6   158.
 5  51.3   62.2   159.
 6  67.2   73.9   148.
 7  54.6   67.5   173.
 8  37.3   55.2   167.
 9  43.1   68.5   154.
10  45.5   62.6   178.
```

```
multi_means(new_df)
```

```
[1]  50.74626  66.04311 160.75441
```

If we want our for loop to return the results with informative names, similar those that are returned when we use the `across()` method, we can simply add one line of code to our for loop body that names each result:

```
xyz_means <- vector("numeric", length(df_xyz))

for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
  names(xyz_means)[[i]] <- paste0(names(df_xyz)[[i]], "_mean") # Name results here
}

xyz_means
```

```
     x_mean      y_mean      z_mean      a_mean
 0.07462564  0.20862196 -0.42455887  0.32204455
```

If it isn't quite clear to you why that code works, try picking it apart, replacing `i` with a number, and figuring out how it works.

We can make our results resemble those returned by the `across()` method even more by converting our named vector to a data frame like this:

```
xyz_means %>%
  as.list() %>%
  as_tibble()
```

```
# A tibble: 1 x 4
  x_mean y_mean z_mean a_mean
   <dbl>  <dbl>  <dbl>  <dbl>
1 0.0746  0.209 -0.425  0.322
```

Finally, we can update our `multi_means()` function with changes we made above so that our results are returned as a data frame with informative column names:

```
multi_means <- function(data) {
  # Create a structure to contain results
  result <- vector("numeric", length(data))

  # Iterate over each column of data
  for(i in seq_along(data)) {
    result[[i]] <- mean(data[[i]])
```

```
    names(result)[[i]] <- paste0(names(data)[[i]], "_mean")
  }

  # Return the result as a tibble
  as_tibble(as.list(result))
}
```

```
multi_means(new_df)
```

```
# A tibble: 1 x 3
  age_mean height_mean weight_mean
     <dbl>       <dbl>       <dbl>
1     50.7        66.0        161.
```

36.2 Using for loops for data transfer

In the previous section, we used an example that wasn't really all that realistic, but it was useful (hopefully) for learning the mechanics of for loops. As mentioned at the beginning of the chapter, rather than using a for loop for the analysis above, using `across()` with `summarise()` might be preferable.

However, keep in mind that `across()` is designed specifically for repeatedly applying functions column-wise (i.e., across columns) of a *single* data frame in conjunction with `dplyr` verbs. By definition, if we are repeating code outside of `dplyr`, or if we are applying code across *multiple data frames*, then we probably aren't going to be able to use `across()` to complete our coding task.

For example, let's say that we have data stored across multiple sheets of an Excel workbook. This simulated data contains some demographic information about three different cities: Houston, Atlanta, and Charlotte. We need to import each sheet, clean the data, and combine them into a single data frame in order to complete our analysis. First, we will load the `readxl`package:

```
library(readxl)
```

You may click here to download this file to your computer.

Then, we may import each sheet like this:

```
houston <- read_excel(
  "city_ses.xlsx",
  sheet = "Houston"
) %>%
  print()
```

```
# A tibble: 5 x 4
  pid     age sex   ses_score
  <chr> <dbl> <chr>     <dbl>
1 001      13 F            88
2 003      13 F            78
3 007      14 M            83
4 014      12 F            76
5 036      13 M            84
```

```
atlanta <- read_excel(
  "city_ses.xlsx",
  sheet = "Atlanta"
) %>%
  print()
```

```
# A tibble: 5 x 4
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

```
charlotte <- read_excel(
  "city_ses.xlsx",
  sheet = "Charlotte"
) %>%
  print()
```

```
# A tibble: 5 x 4
  pid     age sex     ses
  <chr> <dbl> <chr> <dbl>
1 004      13 F        84
```

```
2 011      14 M        66
3 018      12 M        92
4 023      12 M        89
5 030      13 F        83
```

In the code chunks above, we have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code. Of course, we could write our own function to reduce some of the repetition:

```
import_cities <- function(sheet) {
  df <- read_excel(
    "city_ses.xlsx",
    sheet = sheet
  )
}
```

```
houston <- import_cities("Houston") %>% print()
```

```
# A tibble: 5 x 4
  pid     age sex   ses_score
  <chr> <dbl> <chr>     <dbl>
1 001      13 F            88
2 003      13 F            78
3 007      14 M            83
4 014      12 F            76
5 036      13 M            84
```

```
atlanta <- import_cities("Atlanta") %>% print()
```

```
# A tibble: 5 x 4
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

```
charlotte <- import_cities("Charlotte") %>% print()
```

```
# A tibble: 5 x 4
  pid     age sex     ses
  <chr> <dbl> <chr> <dbl>
1 004      13 F        84
2 011      14 M        66
3 018      12 M        92
4 023      12 M        89
5 030      13 F        83
```

That method is better. And depending on the circumstances of your project, it may be the best approach. However, an alternative approach would be to use a for loop. Using the for loop approach might look something like this:

```
# Save the file path to an object so we don't have to type it repeatedly
# or hard-code it in.
path <-  "city_ses.xlsx"

# Use readxl::excel_sheets to get the name of each sheet in the workbook.
# this makes our code more robust.
sheets <- excel_sheets(path)

for(i in seq_along(sheets)) {
  # Convert sheet name to lowercase before using it to name the df
  new_nm <- tolower(sheets[[i]])
  assign(new_nm, read_excel(path, sheet = sheets[[i]]))
}
```

```
houston
```

```
# A tibble: 5 x 4
  pid     age sex   ses_score
  <chr> <dbl> <chr>     <dbl>
1 001      13 F            88
2 003      13 F            78
3 007      14 M            83
4 014      12 F            76
5 036      13 M            84
```

```
atlanta
```

```
# A tibble: 5 x 4
```

```
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

```
charlotte
```

```
# A tibble: 5 x 4
  pid     age sex     ses
  <chr> <dbl> <chr> <dbl>
1 004      13 F        84
2 011      14 M        66
3 018      12 M        92
4 023      12 M        89
5 030      13 F        83
```

Here's what we did above:

- We used a for loop to import every sheet from an Excel workbook.
- First, we saved the path to the Excel workbook to a separate object. We didn't have to do this. However, doing so prevented us from having to type out the full file path repeatedly in the rest of our code. Additionally, if the file path ever changed, we would only have to update it in one place.
- Second, we used the `excel_sheets()` function to create a character vector containing each sheet name. We didn't have to do this. We could have typed each sheet name manually. However, there shouldn't be any accidental typos if we use the `excel_sheets()` function, and we don't have to make any changes to our code if more sheets are added to the Workbook in the future.
- Inside the for loop, we assigned each data frame created by the `read_excel()` function to our global environment using the `assign()` function. We haven't used the `assign()` function before, but you can read the help documentation by typing `?assign` in your R console.
  - The first argument to the `assign()` function is `x`. The value you pass to `x` should be the name of the object you want to create. Above, we passed `new_nm` (for new name) to the `x` argument. At each iteration of the for loop, `new_nm` contained the name of each sheet in `sheets`. So, `Houston` at the first iteration, `Atlanta` at the second iteration, and `Charlotte` at the third iteration. Of course, we like using

lowercase names for our data frames, so we used `tolower()` to convert `Houston`, `Atlanta`, and `Charlotte` to `houston`, `atlanta`, and `charlotte`. These will be the names used for each data frame assigned to our global environment inside of the for loop.

- The second argument to the `assign()` function is `value`. The value you pass to `value` should be the contents you want to assign the object with the name you passed to the `x` argument. Above, we passed the code that imports each sheet of the `city_ses.xlsx` data frame to the `value` argument.

For loops can often be helpful for data transfer tasks. In the code above, we looped over sheets of a single Excel workbook. However, we could have similarly looped over file paths to import multiple different Excel workbooks instead. We could have even used nested for loops to import multiple sheets from multiple Excel workbooks. The code would not have looked drastically different.

36.3 Using for loops for data management

In the chapter on writing functions, we created an `is_match()` function. In that scenario, we wanted to see if first name, last name, and street name matched at each ID between our data frames. More specifically, we wanted to combine the two data frames into a single data frame and create three new dummy variables that indicated whether first name, last name, and address matched respectively.

Here are the data frames we simulated and combined:

```
people_1 <- tribble(
  ~id_1, ~name_first_1, ~name_last_1, ~street_1,
  1,     "Easton",      NA,           "Alameda",
  2,     "Elias",       "Salazar",    "Crissy Field",
  3,     "Colton",      "Fox",        "San Bruno",
  4,     "Cameron",     "Warren",     "Nottingham",
  5,     "Carson",      "Mills",      "Jersey",
  6,     "Addison",     "Meyer",      "Tingley",
  7,     "Aubrey",      "Rice",       "Buena Vista",
  8,     "Ellie",       "Schmidt",    "Division",
  9,     "Robert",      "Garza",      "Red Rock",
  10,    "Stella",      "Daniels",    "Holland"
)
```

```
people_2 <- tribble(
  ~id_2, ~name_first_2, ~name_last_2, ~street_2,
  1,     "Easton",      "Stone",      "Alameda",
  2,     "Elas",        "Salazar",    "Field",
  3,     NA,            "Fox",        NA,
  4,     "Cameron",     "Waren",      "Notingham",
  5,     "Carsen",      "Mills",      "Jersey",
  6,     "Adison",      NA,           NA,
  7,     "Aubrey",      "Rice",       "Buena Vista",
  8,     NA,            "Schmidt",    "Division",
  9,     "Bob",         "Garza",      "Red Rock",
  10,    "Stella",      NA,           "Holland"
)
```

```
people <- people_1 %>%
  bind_cols(people_2) %>%
  print()
```

```
# A tibble: 10 x 8
    id_1 name_first_1 name_last_1 street_1      id_2 name_first_2 name_last_2
   <dbl> <chr>        <chr>       <chr>        <dbl> <chr>        <chr>
 1     1 Easton       <NA>        Alameda          1 Easton       Stone
 2     2 Elias        Salazar     Crissy Field     2 Elas         Salazar
 3     3 Colton       Fox         San Bruno        3 <NA>         Fox
 4     4 Cameron      Warren      Nottingham       4 Cameron      Waren
 5     5 Carson       Mills       Jersey           5 Carsen       Mills
 6     6 Addison      Meyer       Tingley          6 Adison       <NA>
 7     7 Aubrey       Rice        Buena Vista      7 Aubrey       Rice
 8     8 Ellie        Schmidt     Division         8 <NA>         Schmidt
 9     9 Robert       Garza       Red Rock         9 Bob          Garza
10    10 Stella       Daniels     Holland         10 Stella       <NA>
# i 1 more variable: street_2 <chr>
```

Here is the function we wrote to help us create the dummy variables:

```
is_match <- function(value_1, value_2) {
  result <- value_1 == value_2
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

And here is how we applied the function we wrote to get our results:

```
people %>%
  mutate(
    name_first_match = is_match(name_first_1, name_first_2),
    name_last_match  = is_match(name_last_1, name_last_2),
    street_match     = is_match(street_1, street_2)
  ) %>%
  # Order like columns next to each other for easier comparison
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         FALSE            Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         FALSE            Schmidt     Schmidt
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

However, in the code chunk above, we still have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code. Because we are using dplyr, and all of our data resides inside of a single data frame, your first instinct might be to use `across()` inside of `mutate()` to perform column-wise operations. Unfortunately, that method won't work in this scenario.

The `across()` function will apply the function we pass to the `.fns` argument to each column passed to the `.cols` argument, one at a time. But, we need to pass two columns at a time to the `is_match()` function. For example, `name_first_1` and `name_first_2`. There's really no good way to accomplish this task using `is_match()` inside of `across()`. However, it is fairly simple to accomplish this task with a for loop:

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
```

```
  col_1   <- paste0(cols[[i]], "_1")
  col_2   <- paste0(cols[[i]], "_2")
  new_col <- paste0(cols[[i]], "_match")
  people[[new_col]] <- is_match(people[[col_1]], people[[col_2]])
}
```

```
people %>%
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         FALSE            Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         FALSE            Schmidt     Schmidt
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

Here's what we did above:

- We used our `is_match()` function inside of a for loop to create three new dummy variables that indicated whether first name, last name, and address matched respectively.

Let's pull the code apart piece-by-piece to see how it works.

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
  col_1   <- paste0(cols[[i]], "_1")
  col_2   <- paste0(cols[[i]], "_2")
  new_col <- paste0(cols[[i]], "_match")
  print(col_1)
  print(col_2)
  print(new_col)
}
```

```
[1] "name_first_1"
[1] "name_first_2"
[1] "name_first_match"
[1] "name_last_1"
[1] "name_last_2"
[1] "name_last_match"
[1] "street_1"
[1] "street_2"
[1] "street_match"
```

First, we created a character vector that contained the base name (i.e., no `_1` or `_2`) of each of the columns we wanted to compare. Then, we iterated over that character vector by passing it as the object to the right of the keyword `in`.

At each iteration, we used `paste0()` to create three column names from the character string in `cols`. For example, in the first iteration of the loop, the value of `cols` was `name_first`. The first line of code in the for loop body combined `name_first` with `_1` to make the character string `name_first_1` and save it as an object named `col_1`. The second line of code in the for loop body combined `name_first` with `_2` to make the character string `name_first_2` and save it as an object named `col_2`. And, the third line of code in the for loop body combined `name_first` with `_match` to make the character string `name_first_match` and save it as an object named `new_col`.

This will allow us to use `col_1`, `col_2`, and `new_col` in the code that compares the columns and creates each dummy variable. For example, here is what `people[[col_1]]` looks like at each iteration:

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
  col_1 <- paste0(cols[[i]], "_1")
  col_2 <- paste0(cols[[i]], "_2")
  print(people[[col_1]])
}
```

```
 [1] "Easton"  "Elias"   "Colton"  "Cameron" "Carson"  "Addison" "Aubrey"
 [8] "Ellie"   "Robert"  "Stella"
 [1] NA        "Salazar" "Fox"     "Warren"  "Mills"   "Meyer"   "Rice"
 [8] "Schmidt" "Garza"   "Daniels"
 [1] "Alameda"      "Crissy Field" "San Bruno"    "Nottingham"   "Jersey"
 [6] "Tingley"      "Buena Vista"  "Division"     "Red Rock"     "Holland"
```

It is a vector that matches `people[["name_first_1"]]`, `people[["name_last_1"]]`, and `people[["street_1"]]` respectively.

And here is what `col_2` looks like at each iteration:

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
  col_1 <- paste0(cols[[i]], "_1")
  col_2 <- paste0(cols[[i]], "_2")
  print(people[[col_2]])
}
```

```
 [1] "Easton"  "Elas"    NA        "Cameron" "Carsen"  "Adison"  "Aubrey"
 [8] NA        "Bob"     "Stella"
 [1] "Stone"   "Salazar" "Fox"     "Waren"   "Mills"   NA        "Rice"
 [8] "Schmidt" "Garza"   NA
 [1] "Alameda"     "Field"       NA            "Notingham"   "Jersey"
 [6] NA            "Buena Vista" "Division"    "Red Rock"    "Holland"
```

Now, we can pass each vector to our `is_match()` function at each iteration like this:

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
  col_1 <- paste0(cols[[i]], "_1")
  col_2 <- paste0(cols[[i]], "_2")
  print(is_match(people[[col_1]], people[[col_2]]))
}
```

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

These logical vectors are the results we want to go into our new dummy variables. Therefore, the last step is to assign each logical vector above to a new variable in our data frame called `people[["name_first_match"]]`, `people[["name_last_match"]]`, and `people[["street_match"]]` respectively. We do so by allowing `people[[new_col]]` to represent those values at each iteration of the loop:

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
  col_1   <- paste0(cols[[i]], "_1")
  col_2   <- paste0(cols[[i]], "_2")
  new_col <- paste0(cols[[i]], "_match")
  people[[new_col]] <- is_match(people[[col_1]], people[[col_2]])
}
```

And here is our result:

```
people %>%
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         FALSE            Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         FALSE            Schmidt     Schmidt
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

In the code above, we used roughly the same amount of code to complete the task with a loop that we used to complete it without a loop. However, this code still has some advantages. We only typed "name_first", "name_last", and "street" once at the beginning of the code chunk. Therefore, we didn't have to worry about forgetting to change a column name after copying and pasting code. Additionally, if we later decide that we also want to compare other columns (e.g., middle name, birth date, city, state, zip code), we only have to update the code in one place – where we create the `cols` vector.

36.4 Using for loops for analysis

For our final example of this chapter, let's return to the final example from the column-wise operations chapter. We started with some simulated study data:

```
study <- tibble(
  age       = c(32, 30, 32, 29, 24, 38, 25, 24, 48, 29, 22, 29, 24, 28, 24, 25,
                25, 22, 25, 24, 25, 24, 23, 24, 31, 24, 29, 24, 22, 23, 26, 23,
                24, 25, 24, 33, 27, 25, 26, 26, 26, 26, 26, 27, 24, 43, 25, 24,
                27, 28, 29, 24, 26, 28, 25, 24, 26, 24, 26, 31, 24, 26, 31, 34,
                26, 25, 27, NA),
  age_group = c(2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2,
                2, 1, 1, 1, NA),
  gender    = c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1,
                1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1,
                1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 1,
                1, 1, 2, 1, NA),
  ht_in     = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
                64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
                64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
                69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
                61, 69, 66, NA),
  wt_lbs    = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
                125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
                186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
                147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
                110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
                163, 141, NA),
  bmi       = c(30.99, 18.78, 26.52, 30.54, 22.39, 26.12, 23.69, 20.67, 26.29,
                25.39, 25.68, 45.15, 21.56, 20.17, 17.38, 20.8, 22.31, 22.75,
                26.62, 21.43, 19.14, 23.49, 22.66, 32.98, 25.05, 18.31, 29.13,
                27.07, 20.37, 25.01, 19.69, 25.97, 18.88, 20.07, NA, 26.76,
                26.97, 25.24, 20.68, 23.72, 24.82, 23.62, 18.65, 24.03, 23.86,
                10.63, 23.02, 23.72, 20.82, 28.24, NA, 37.55, 18.88, 18.3,
                19.13, 21.48, 22.59, 24.96, 21.63, NA, 29.35, 21.76, 17.97,
                22.31, 19.27, 24.07, 22.76, NA),
  bmi_3cat  = c(3, 1, 2, 3, 1, 2, 1, 1, 2, 2, 2, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1,
                1, 1, 3, 2, 1, 2, 2, 1, 2, 1, 2, 1, 1, NA, 2, 2, 2, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 2, NA, 3, 1, 1, 1, 1, 1, 1, 1, NA, 2, 1,
                1, 1, 1, 1, 1, NA)
```

```
) %>%
  mutate(
    age_group = factor(age_group, labels = c("Younger than 30", "30 and Older")),
    gender    = factor(gender, labels = c("Female", "Male")),
    bmi_3cat  = factor(bmi_3cat, labels = c("Normal", "Overweight", "Obese"))
  ) %>%
  print()
```

```
# A tibble: 68 x 7
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese
 2    30 30 and Older    Female    63    106  18.8 Normal
 3    32 30 and Older    Female    62    145  26.5 Overweight
 4    29 Younger than 30 Male      67    195  30.5 Obese
 5    24 Younger than 30 Female    67    143  22.4 Normal
 6    38 30 and Older    Female    58    125  26.1 Overweight
 7    25 Younger than 30 Female    64    138  23.7 Normal
 8    24 Younger than 30 Male      69    140  20.7 Normal
 9    48 30 and Older    Male      65    158  26.3 Overweight
10    29 Younger than 30 Male      68    167  25.4 Overweight
# i 58 more rows
```

Then we saw how to use `across()` with `pivot_longer()` to remove repetition and get our results into a format that were easier to read an interpret:

```
summary_stats <- study %>%
  summarise(
    across(
      .cols  = c(age, ht_in, wt_lbs, bmi),
      .fns   = list(
        n_miss = ~ sum(is.na(.x)),
        mean   = ~ mean(.x, na.rm = TRUE),
        median = ~ median(.x, na.rm = TRUE),
        min    = ~ min(.x, na.rm = TRUE),
        max    = ~ max(.x, na.rm = TRUE)
      ),
      .names = "{col}-{fn}" # This is the new part of the code
    )
  ) %>%
  print()
```

```
# A tibble: 1 x 20
  `age-n_miss` `age-mean` `age-median` `age-min` `age-max` `ht_in-n_miss`
         <int>      <dbl>        <dbl>     <dbl>     <dbl>          <int>
1            1       26.9           26        22        48              3
# i 14 more variables: `ht_in-mean` <dbl>, `ht_in-median` <dbl>,
#   `ht_in-min` <dbl>, `ht_in-max` <dbl>, `wt_lbs-n_miss` <int>,
#   `wt_lbs-mean` <dbl>, `wt_lbs-median` <dbl>, `wt_lbs-min` <dbl>,
#   `wt_lbs-max` <dbl>, `bmi-n_miss` <int>, `bmi-mean` <dbl>,
#   `bmi-median` <dbl>, `bmi-min` <dbl>, `bmi-max` <dbl>
```

```
summary_stats %>%
  tidyr::pivot_longer(
    cols      = everything(),
    names_to  = c("characteristic", ".value"),
    names_sep = "-"
  )
```

```
# A tibble: 4 x 6
  characteristic n_miss  mean median   min   max
  <chr>           <int> <dbl>  <dbl> <dbl> <dbl>
1 age                 1  26.9   26    22    48
2 ht_in               3  66.0   66    58    76
3 wt_lbs              2 148.   142.   60   297
4 bmi                 4  23.6   22.9  10.6  45.2
```

I think that method works really nicely for our continuous variables; however, the situation is slightly more complicated for categorical variables. To illustrate the problem as simply as possible, let's start by just getting counts for each of our categorical variables:

```
study %>%
  count(age_group)
```

```
# A tibble: 3 x 2
  age_group           n
  <fct>           <int>
1 Younger than 30    56
2 30 and Older       11
3 <NA>                1
```

```
study %>%
  count(gender)
```

```
# A tibble: 3 x 2
  gender     n
  <fct>  <int>
1 Female    43
2 Male      24
3 <NA>       1
```

```
study %>%
  count(bmi_3cat)
```

```
# A tibble: 4 x 2
  bmi_3cat       n
  <fct>      <int>
1 Normal        43
2 Overweight    16
3 Obese          5
4 <NA>           4
```

You are, of course, and old pro at this by now, and you quickly spot all the unnecessary repetition. So, you decide to pass `count` to the `.fns` argument like this:

```
study %>%
  summarise(
    across(
      .cols = c(age_group, gender, bmi_3cat),
      .fns  = count
    )
  )
```

```
Error in `summarise()`:
i In argument: `across(.cols = c(age_group, gender, bmi_3cat), .fns =
  count)`.
Caused by error in `across()`:
! Can't compute column `age_group`.
Caused by error in `UseMethod()`:
! no applicable method for 'count' applied to an object of class "factor"
```

Unfortunately, this won't work. At least not currently. There are a couple reasons why this won't work, but the one that is probably easiest to wrap your head around is related to the number of results produced by `count()`. What does this mean? Well, when we pass each continuous variable to `mean()` (or `median`, `min`, or `max`) we get *one* result back for each column:

```
study %>%
  summarise(
    across(
      .cols = c(age, ht_in),
      .fns  = ~ mean(.x, na.rm = TRUE)
    )
  )
```

```
# A tibble: 1 x 2
    age ht_in
  <dbl> <dbl>
1  26.9  66.0
```

It's easy for `dplyr` to arrange those results into a data frame. However, the results from `count()` are much less predictable. For example, `study %>% count(age_group)` had three results, `study %>% count(gender)` had three results, and `study %>% count(bmi_3cat)` had four results. Also, remember that every column of a data frame has to have the same number of rows. So, if the code we used to try to pass `count` to the `.fns` argument above would actually run, it might look something like this:

age_group <fctr>	n <int>	gender <fctr>	n <int>	bmi_3cat <fctr>	n <int>
Younger than 30	56	Female	43	Normal	43
30 and Older	11	Male	24	Overweight	16
NA	1	NA	1	Obese	5
?	?	?	?	NA	4

Figure 36.16: Representation of result from the above code if it could be run successfully

Because `summarise()` lays the results out side-by-side, it's not clear what would go into the 4 cells in the bottom-left corner of the results data frame. Therefore, it isn't necessarily straightforward for `dplyr` to figure out how it should return such results to us.

However, when we use a for loop, *we can create our own structure* to hold the results. And, that structure can be pretty much any structure that meets our needs. In this case, one option would be to create a data frame to hold our categorical counts that looks like this:

variable <chr>	category <chr>	n <int>

Figure 36.17: Empty data frame structure to hold for loop results

Then, we can use a for loop to fill in the empty data frame so that we end up with results that look like this:

variable <chr>	category <chr>	n <int>
age_group	Younger than 30	56
age_group	30 and Older	11
age_group	NA	1
gender	Female	43
gender	Male	24
gender	NA	1
bmi_3cat	Normal	43
bmi_3cat	Overweight	16
bmi_3cat	Obese	5
bmi_3cat	NA	4

Figure 36.18: Data frame structure with for loop results

The process for getting to our finished product is a little bit involved (and probably a little intimidating for some of you) and will require us to cover a couple new topics. So, we'll start by giving you the complete code for accomplishing this task. Then, we'll pick the code apart, piece-by-piece, to make sure we understand *how* it works.

Here is the complete solution:

```
# Structure 1. An object to contain the results.
  # Create the data frame structure that will contain our results
cat_table <- tibble(
  variable = vector("character"),
  category = vector("character"),
  n        = vector("numeric")
)

# Structure 2. The actual for loop.
  # For each column, get the column name, category names, and count.
  # Then, add them to the bottom of the results data frame we created above.
for(i in c("age_group", "gender", "bmi_3cat")) {
  cat_stats <- study %>%
    count(.data[[i]]) %>% # Use .data to refer to the current data frame.
    mutate(variable = names(.)[1]) %>% # Use . to refer to the result to this point.
```

```
    rename(category = 1)

  # Here is where we update cat_table with the results for each column
  cat_table <- bind_rows(cat_table, cat_stats)
}
```

```
cat_table
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <chr>           <dbl>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

We'll use the rest of this chapter section to walk through the code above and make sure we understand how it works. For starters, we will create our results data frame structure like this:

```
cat_table <- tibble(
  variable = vector("character"),
  category = vector("character"),
  n        = vector("numeric")
)
```

```
str(cat_table)
```

```
tibble [0 x 3] (S3: tbl_df/tbl/data.frame)
 $ variable: chr(0)
 $ category: chr(0)
 $ n       : num(0)
```

As you can see, we created an empty data frame with three columns. One to hold the variable names, one to hold the variable categories, and one to hold the count of occurrences of each category. Now, we can use a for loop to iteratively add results to our empty data frame structure. This works similarly to the way we added mean values to the `xyz_means` vector in the first example above. As a reminder, here is what the for loop code looks like:

```
for(i in c("age_group", "gender", "bmi_3cat")) {
  cat_stats <- study %>%
    count(.data[[i]]) %>%
    mutate(variable = names(.)[1]) %>%
    rename(category = 1)

  cat_table <- bind_rows(cat_table, cat_stats)
}
```

For our next step, let's walk through the first little chunk of code inside the for loop body. Specifically:

```
cat_stats <- study %>%
  count(.data[[i]]) %>%
  mutate(variable = names(.)[1]) %>%
  rename(category = 1)
```

If we were using this code to analyze a single variable, as opposed to using it in a for loop, this is what the result would look like:

```
cat_stats <- study %>%
  count(age_group) %>%
  mutate(variable = names(.)[1]) %>%
  rename(category = 1) %>%
  print()
```

```
# A tibble: 3 x 3
  category            n variable
  <fct>           <int> <chr>
1 Younger than 30    56 age_group
2 30 and Older       11 age_group
3 <NA>                1 age_group
```

We've already seen what the `study %>% count(age_group)` part of the code does, and we already know that we can use `mutate()` to create a new column in our data frame. In this case,

the name of the new column is `variable`. But, you may be wondering what the `names(.)[1]` after the equal sign does. Let's take a look. Here, we can see the data frame that is getting passed to `mutate()`:

```
cat_stats <- study %>%
  count(age_group) %>%
  print()
```

```
# A tibble: 3 x 2
  age_group           n
  <fct>           <int>
1 Younger than 30    56
2 30 and Older       11
3 <NA>                1
```

It's a data frame with two columns. The first column actually has two different kinds of information that we need. It contains the name of the variable being analyzed as the column name, and it contains all the categories of that variable as the column values. We want to separate those two pieces of information into two columns. This task is similar to some of the "tidy data" tasks we worked through in the chapter on restructuring data frames. In fact, we can also use `pivot_longer()` to get the result we want:

```
study %>%
  count(age_group) %>%
  tidyr::pivot_longer(
    cols      = "age_group",
    names_to  = "variable",
    values_to = "category"
  )
```

```
# A tibble: 3 x 3
      n variable  category
  <int> <chr>     <fct>
1    56 age_group Younger than 30
2    11 age_group 30 and Older
3     1 age_group <NA>
```

In this solution for this task, however, we're not going to use `pivot_longer()` for a couple of reasons. First, it's an opportunity for us to learn about the special use of dot (`.`) inside of `dplyr` verbs. Second, this solution will use `dplyr` only. It will not require us to use the `tidyr` package.

Before we talk about the dot, however, let's make sure we know what the `names()[1]` is doing. There aren't any new concepts here, but we may not have used them this way before. The `name()` function just returns a character vector containing the column names of the data frame we pass to it. So, when we pass the `cat_stats` data frame to it, this is what it returns:

```
names(cat_stats)
```

```
[1] "age_group" "n"
```

We want to use the first value, `"age_group"` to fill-in our the new `variable` column we want to create. We can use bracket notation to subset the first element of the character vector of column names above like this:

```
names(cat_stats)[1]
```

```
[1] "age_group"
```

What does the dot do? Well, outside of our `dplyr` pipeline, it doesn't do anything useful:

```
names(.)[1]
```

```
Error: object '.' not found
```

Inside of our `dplyr` pipeline, you can think of it as a placeholder for whatever is getting passed to the `dplyr` verb – `mutate()` in this case. So, what is getting passed to mutate? The result of everything that comes before `mutate()` in the pipeline. And what does that result look like in this case? It looks like this:

```
study %>%
  count(age_group)
```

```
# A tibble: 3 x 2
  age_group           n
  <fct>           <int>
1 Younger than 30    56
2 30 and Older       11
3 <NA>                1
```

So, we can use the dot inside of mutate as a substitute for the results data frame getting passed to `mutate()`. Said another way. To `dplyr`, this:

```
names(study %>% count(age_group))
```

and this:

```
study %>% count(age_group) %>% names(.)
```

are the exact same thing in this context:

```
cat_stats <- study %>%
  count(age_group) %>%
  mutate(variable = names(.)[1]) %>%
  print()
```

```
# A tibble: 3 x 3
  age_group           n variable
  <fct>           <int> <chr>
1 Younger than 30    56 age_group
2 30 and Older       11 age_group
3 <NA>                1 age_group
```

Now, we have all the variables we wanted for our final results table. Keep in mind, however, that we will eventually be stacking similar results from our other variables (i.e., `gender` and `bmi_3cat`) below these results using `bind_rows()`. You may remember from the chapter on working with multiple data frames that the `bind_rows()` function matches columns together by *name*, not by position. So, we need to change the `age_group` column name to `category`. If we don't, we will end up with something that looks like this:

```
study %>%
  count(age_group) %>%
  bind_rows(study %>% count(gender))
```

```
# A tibble: 6 x 3
  age_group           n gender
  <fct>           <int> <fct>
1 Younger than 30    56 <NA>
2 30 and Older       11 <NA>
3 <NA>                1 <NA>
4 <NA>               43 Female
5 <NA>               24 Male
6 <NA>                1 <NA>
```

Not what we want, right? Again, if we were doing this analysis one variable at a time, our code might look like this:

```
cat_stats <- study %>%
  count(age_group) %>%
  mutate(variable = names(.)[1]) %>%
  rename(category = age_group) %>%
  print()
```

```
# A tibble: 3 x 3
  category            n variable
  <fct>           <int> <chr>
1 Younger than 30    56 age_group
2 30 and Older       11 age_group
3 <NA>                1 age_group
```

We used the `rename()` function above to change the name of the first column from `age_group` to `category`. Remember, the syntax for renaming columns with the `rename()` function is `new_name = old_name`. But, inside our for loop we will actually have 3 old names, right? In the first iteration `old_name` will be `age_group`, in the second iteration `old_name` will be `gender`, and in the third iteration `old_name` will be `bmi_cat`. We could loop over the names, but there's an even easier solution. Instead of asking `rename()` to rename our column by name using this syntax, `new_name = old_name`, we can also ask `rename()` to rename our column by *position* using this syntax, `new_name = column_number`. So, in our example above, we could get the same result by replacing `age_group` with `1` because `age_group` is the first column in the data frame:

```
cat_stats <- study %>%
  count(age_group) %>%
  mutate(variable = names(.)[1]) %>%
  rename(category = 1) %>% # Replace age_group with 1
  print()
```

```
# A tibble: 3 x 3
  category            n variable
  <fct>           <int> <chr>
1 Younger than 30    56 age_group
2 30 and Older       11 age_group
3 <NA>                1 age_group
```

And, using this method, we don't have to make any changes to the value being passed to `rename()` when we are analyzing our other variables. For example:

```r
cat_stats <- study %>%
  count(gender) %>% # Changed the column from age_group to gender
  mutate(variable = names(.)[1]) %>%
  rename(category = 1) %>% # Still have 1 here
  print()
```

```
# A tibble: 3 x 3
  category     n variable
  <fct>    <int> <chr>
1 Female      43 gender
2 Male        24 gender
3 <NA>         1 gender
```

At this point, we have all the elements we need manually create the data frame of final results we want. First, we create the empty results table:

```r
cat_table <- tibble(
  variable = vector("character"),
  category = vector("character"),
  n        = vector("numeric")
) %>%
  print()
```

```
# A tibble: 0 x 3
# i 3 variables: variable <chr>, category <chr>, n <dbl>
```

Then, we get the data frame of results for `age_group`:

```r
cat_stats <- study %>%
  count(age_group) %>%
  mutate(variable = names(.)[1]) %>%
  rename(category = 1) %>%
  print()
```

```
# A tibble: 3 x 3
  category            n variable
  <fct>           <int> <chr>
1 Younger than 30    56 age_group
2 30 and Older       11 age_group
3 <NA>                1 age_group
```

Then, we use `bind_rows()` to add those results to our `cat_table` data frame:

```
cat_table <- cat_table %>%
  bind_rows(cat_stats) %>%
  print()
```

```
# A tibble: 3 x 3
  variable  category            n
  <chr>     <chr>           <dbl>
1 age_group Younger than 30    56
2 age_group 30 and Older       11
3 age_group <NA>                1
```

Then, we copy and paste the last two steps above, replacing `age_group` with `gender`:

```
cat_stats <- study %>%
  count(gender) %>% # Change to gender
  mutate(variable = names(.)[1]) %>%
  rename(category = 1)

cat_table <- cat_table %>%
  bind_rows(cat_stats) %>%
  print()
```

```
# A tibble: 6 x 3
  variable  category            n
  <chr>     <chr>           <dbl>
1 age_group Younger than 30    56
2 age_group 30 and Older       11
3 age_group <NA>                1
4 gender    Female             43
5 gender    Male               24
6 gender    <NA>                1
```

Then, we copy and paste the two steps above, replacing `gender` with `bmi_3cat`:

```
cat_stats <- study %>%
  count(bmi_3cat) %>%  # Change to bmi_3cat
  mutate(variable = names(.)[1]) %>%
  rename(category = 1)
```

```
cat_table <- cat_table %>%
  bind_rows(cat_stats) %>%
  print()
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <chr>           <dbl>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

That is exactly the final result we wanted, and you might have noticed that the only elements of the code chunks above that changed were the column names being passed to `count()`. If we can just figure out how to loop over the column names, then we can remove a ton of unnecessary repetition from our code. Our first attempt might look like this:

```
for(i in c(age_group, gender, bmi_3cat)) {
  study %>%
    count(i) %>%
    mutate(variable = names(.)[1]) %>%
    rename(category = 1)
}
```

```
Error: object 'age_group' not found
```

However, it doesn't work. In the code above, R is looking for and *object* in the global environment called `age_group`. Of course, there is no object in the global environment named `age_group`. Rather, there is an object in the global environment named `study` that has a column named `age_group`.

We can get rid of that error by wrapping each column name in quotes:

```
for(i in c("age_group", "gender", "bmi_3cat")) {
  study %>%
    count(i) %>%
    mutate(variable = names(.)[1]) %>%
    rename(category = 1)
}
```

```
Error in `count()`:
! Must group by variables found in `.data`.
x Column `i` is not found.
```

Unfortunately, that just gives us a different error. In the code above, `count()` is looking for a column named `i` in the `study` data frame. You may be wondering why `i` is not being converted to `"age_group"`, `"gender"`, and `"bmi_3cat"` in the code above. The short answer is that it's because of tidy evaluation and data masking.

So, we need a way to iteratively pass each quoted column name to the `count()` function inside our for loop body, but also let `dplyr` know that they *are column names*, not just random character strings. Fortunately, the `rlang` package (which is partially imported with `dplyr`), provides us with a special construct that can help us solve this problem. It's called the `.data` pronoun. Here's how we can use it:

```
for(i in c("age_group", "gender", "bmi_3cat")) {
  study %>%
    count(.data[[i]]) %>%
    mutate(variable = names(.)[1]) %>%
    rename(category = 1) %>%
    print()
}
```

```
# A tibble: 3 x 3
  category            n variable
  <fct>           <int> <chr>
1 Younger than 30    56 age_group
2 30 and Older       11 age_group
3 <NA>                1 age_group
# A tibble: 3 x 3
  category     n variable
  <fct>    <int> <chr>
1 Female      43 gender
2 Male        24 gender
```

```
3 <NA>         1 gender
# A tibble: 4 x 3
  category       n variable
  <fct>      <int> <chr>
1 Normal        43 bmi_3cat
2 Overweight    16 bmi_3cat
3 Obese          5 bmi_3cat
4 <NA>           4 bmi_3cat
```

Here's how it works. Remember that data masking allows us to write column names directly in `dplyr` code without having to use dollar sign or bracket notation to tell R which data frame that column lives in. For example, in the following code, `dplyr` just "knows" that `age_group` is a column in the `study` data frame:

```
study %>%
  count(age_group)
```

```
# A tibble: 3 x 2
  age_group           n
  <fct>           <int>
1 Younger than 30    56
2 30 and Older       11
3 <NA>                1
```

The same is not true for base R functions. For example, we can't pass `age_group` directly to the `table()` function:

```
table(age_group)
```

```
Error: object 'age_group' not found
```

We have to use dollar sign or bracket notation to tell R that `age_group` is a column in `study`:

```
table(study[["age_group"]])
```

```
Younger than 30    30 and Older
             56              11
```

This is a really nice feature of `dplyr` when we're using dplyr interactively. But, as we've already discussed, it does present us with some challenges when we use `dplyr` functions inside of the functions we write ourselves and inside of for loops.

As you can see in the code below, the tidy evaluation essentially blocks the `i` inside of `count()` from being replaced with each of the character strings we are looping over. Instead, `dplyr` looks for a literal `i` as a column name in the `study` data frame.

```
for(i in c("age_group", "gender", "bmi_3cat")) {
  study %>%
    count(i)
}
```

```
Error in `count()`:
! Must group by variables found in `.data`.
x Column `i` is not found.
```

So, we need a way to tell `dplyr` that `"age_group"` is a column *in* the `study` data frame. Well, we know how to use quoted column names inside bracket notation. So, we could write code like this:

```
study %>%
  count(study[["age_group"]])
```

```
# A tibble: 3 x 2
  `study[["age_group"]]`     n
  <fct>                  <int>
1 Younger than 30           56
2 30 and Older              11
3 <NA>                       1
```

However, the column name (i.e., `study[["age_group"]]`) in the results data frame above isn't ideal to work with. Additionally, the code above isn't very flexible because we have the `study` data frame hard-coded into it (i.e., `study[["age_group"]]`). That's where the `.data` pronoun comes to the rescue:

```
study %>%
  count(.data[["age_group"]])
```

```
# A tibble: 3 x 2
  age_group           n
  <fct>           <int>
1 Younger than 30    56
2 30 and Older       11
3 <NA>                1
```

The .data pronoun "is not a data frame; it's a special construct, a pronoun, that allows you to access the current variables either directly, with .data$x or indirectly with .data[[var]]."[10]

When we put it all together, our code looks like this:

```
# Create the data frame structure that will contain our results
cat_table <- tibble(
  variable = vector("character"),
  category = vector("character"),
  n        = vector("numeric")
)

# For each column, get the column name, category names, and count.
# Then, add them to the bottom of the results data frame we created above.
for(i in c("age_group", "gender", "bmi_3cat")) {
  cat_stats <- study %>%
    count(.data[[i]]) %>% # Use .data to refer to the current data frame.
    mutate(variable = names(.)[1]) %>% # Use . to refer to the current data frame.
    rename(category = 1)

  # Here is where we update cat_table with the results for each column
  cat_table <- bind_rows(cat_table, cat_stats)
}
```

```
cat_table
```

```
# A tibble: 10 x 3
   variable  category           n
   <chr>     <chr>          <dbl>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
```

```
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

And, we can do other interesting things with our results now that we have it in this format. For example, we can easily add percentages along with our counts like this:

```
cat_table %>%
  group_by(variable) %>%
  mutate(
    percent = n / sum(n) * 100
  )
```

```
# A tibble: 10 x 4
# Groups:   variable [3]
   variable  category            n percent
   <chr>     <chr>           <dbl>   <dbl>
 1 age_group Younger than 30    56   82.4
 2 age_group 30 and Older       11   16.2
 3 age_group <NA>                1    1.47
 4 gender    Female             43   63.2
 5 gender    Male               24   35.3
 6 gender    <NA>                1    1.47
 7 bmi_3cat  Normal             43   63.2
 8 bmi_3cat  Overweight         16   23.5
 9 bmi_3cat  Obese               5    7.35
10 bmi_3cat  <NA>                4    5.88
```

Finally, we could also write our own function that uses the code above. That way, we can easily reuse this code in the future:

```
cat_stats <- function(data, ...) {
  # Create the data frame structure that will contain our results
  cat_table <- tibble(
    variable = vector("character"),
    category = vector("character"),
    n        = vector("numeric")
  )

  # For each column in ..., get the column name, category names, and count.
```

```
  # Then, add them to the bottom of the results data frame we created above.
  for(i in c(...)) {
    stats <- data %>%
      count(.data[[i]]) %>% # Use .data to refer to the current data frame.
      mutate(variable = names(.)[1]) %>% # Use . to refer to the current data frame.
      rename(category = 1)

    # Here is where we update cat_table with the results for each column
    cat_table <- bind_rows(cat_table, stats)
  }
  # Return results
  cat_table
}
```

```
cat_stats(study, "age_group", "gender", "bmi_3cat")
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <chr>           <dbl>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

We covered a lot of material in this chapter. For loops tend to be confusing for people who are just learning to program. When you throw in the tidy evaluation stuff, it can be really confusing – even for experienced R programmers. So, if you are still feeling a little confused, don't beat yourself up. Also, trying to memorize everything we covered in this chapter is not recommended. Instead, we recommend that you read it until you have understood what for loops are and when they might be useful at a high level. Then, refer back to this chapter (or other online references that discuss for loops) if you find yourself in a situation where you believe that for loops might be the right tool to help you complete a given programming task. Having said that, also keep in mind that for loops are rarely the *only* tool you will have at your disposal to complete the task. In the next chapter, we will learn how to use functionals, specifically the `purrr` package, in place of for loops. You may find this approach to iteration more intuitive.

37 Using the purrr Package

In this final chapter of the repeated operations part of the book, we are going to discuss the purrr package.

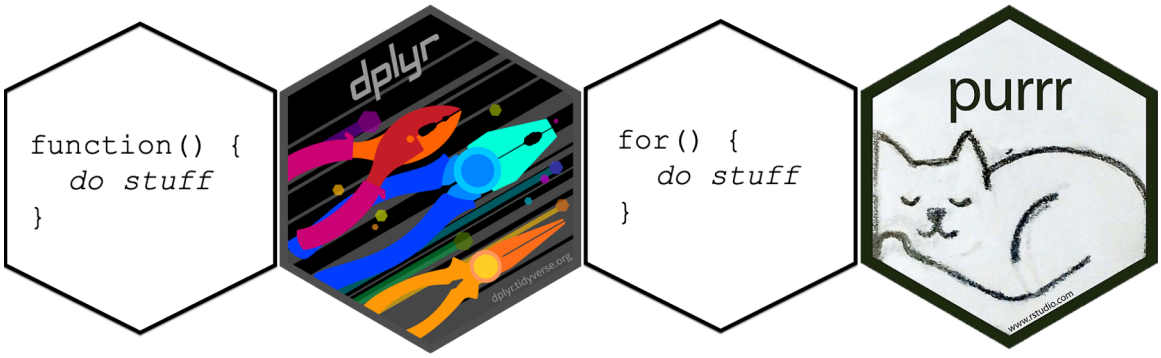


Figure 37.1: For loops graphic

The `purrr` package provides a really robust set of functions that can help us more efficiently complete a bunch of different tasks in R. For the purposes of this chapter, however, we are going to focus on using the `purrr::map` functions as an alternative approach to removing unnecessary repetition from the various different code chunks we've already seen in other chapters.

For our purposes, you can think of the `purrr::map` functions as a replacement for for loops. In other words, you can think of them as *doing* the same thing as a for loop, but writing the code in a different way.

> **Note**
>
> We also want to mention that the `purrr` package is closely related to base R's `apply` functions (i.e., `apply()`, `lapply()`, `sapply()`, `tapply()`). We aren't going to discuss those functions any further, but you will often see them mentioned side-by-side as solutions to a given coding challenge on websites like Stack Overflow. The `purrr` package is partially meant to be an improved replacement for the `apply` functions.

As usual, let's start by taking a look at a simple example – the same one we used to start the chapter on column-wise operations and the chapter on writing for loops. Afterwards, we will compare the basic structure of `purrr::map` functions to the basic structure of for loops. Finally, we will work through a number of the examples we've already worked through in this part of the book using the `purrr` approach.

At this point, we will go ahead and load `dplyr` and `purrr` and simulate our data:

```
library(dplyr)
library(purrr)
```

```
set.seed(123)
df_xyz <- tibble(
  x = rnorm(10),
  y = rnorm(10),
  z = rnorm(10)
) %>%
  print()
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 -0.230   0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705  0.111 -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  -1.69
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

In the chapter on column-wise operations we used `dplyr`'s `across()` function to efficiently find the mean of each column in the `df_xyz` data frame:

```
df_xyz %>%
  summarise(
    across(
      .cols  = everything(),
      .fns   = mean,
      .names = "{col}_mean"
    )
  )
```

```
# A tibble: 1 x 3
  x_mean y_mean z_mean
   <dbl>  <dbl>  <dbl>
1 0.0746  0.209 -0.425
```

In the chapter on writing for loops, we learned an alternative approach that would also work:

```
xyz_means <- vector("double", ncol(df_xyz))

for(i in seq_along(df_xyz)) {
  xyz_means[[i]] <- mean(df_xyz[[i]])
}

xyz_means
```

```
[1]  0.07462564  0.20862196 -0.42455887
```

An alternative way to complete the analysis above is with the `map_dbl()` function from the `purrr` package like this:

```
xyz_means <- map_dbl(
  .x = df_xyz,
  .f = mean
)

xyz_means
```

```
          x           y           z
 0.07462564  0.20862196 -0.42455887
```

Here's what we did above:

- We used `purr`'s `map_dbl()` function to iteratively calculate the mean of each column in the `df_xyz` data frame. There are other `map` functions beside `map_dbl()`. We will eventually discuss them all.
- You can type `?purrr::map_dbl` into your R console to view the help documentation for this function and follow along with the explanation below.
- The first argument to all of the `map` functions is the `.x` argument. You should pass the name of a list, data frame, or vector that you want to iterate over to the `.x` argument. If the object passed to the `.x` argument is a vector, then `map` will apply the function passed to the `.f` argument (see below) to each element of the vector. If the object passed to the `.x` argument is a data frame, then `map` will apply the function passed to the `.f` argument to each column of the data frame. Above, we passed the `df_xyz` data frame to the `.x`.
- The second argument to all of the `map` functions is the `.f` argument. You should pass the name of function, or functions, you want to apply iteratively to the object you passed to the `.x` argument. In the example above, we passed the mean function to the `.f` argument. Notice that we typed `mean` without the parentheses.
- The third argument to all of the `map` functions is the `...` argument. In this case, the `...` argument is where we pass any additional arguments to the function we passed to the `.f` argument. For example, we passed the `mean` function to the `.f` argument above. If the data frame above had missing values, we could have passed `na.rm = TRUE` to the `mean()` function using the `...` argument. We saw a similar example of this when we were learning about `across()`.

As you can see, using the `map_dbl()` package requires far less code than the for loop did, which has at least two potential advantages. First, it's less typing, which means less opportunity for typos. Second, many people in the R community feel as though this *functional* (i.e., use of a function) approach to iteration is much easier to read and understand than the traditional for loop approach.

Additionally, you may have also noticed that we were able to assign the returned results of `map_dbl(df_xyz, mean)` to an object in our global environment in the usual way (i.e., with the assignment arrow). This eliminates the need for creating a structure to hold our results ahead of time as we had to do with the for loop.

Finally, when we use `map_dbl()` there isn't a leftover index variable (i.e., `i`) floating around our global environment the way there was when we were writing for loops.

For those reasons, and possibly others, it's been an observation that the majority of R users prefer the functional approach to iteration – either `purrr` or the `apply` functions – over using for loops in most situations.

However, some of you reading this text book might have had their first experiences with programming in a language other than R that relied more heavily on for loops. For this reason, you might tend to think first in terms of a for loop and then mentally convert the for

loop to a `map` function before writing your code. Therefore, the next section is going to compare and contrast the *basic* for loop with the `map` functions. You may find this section instructive or interesting even if you aren't someone who first learned iteration using for loops.

37.1 Comparing for loops and the map functions

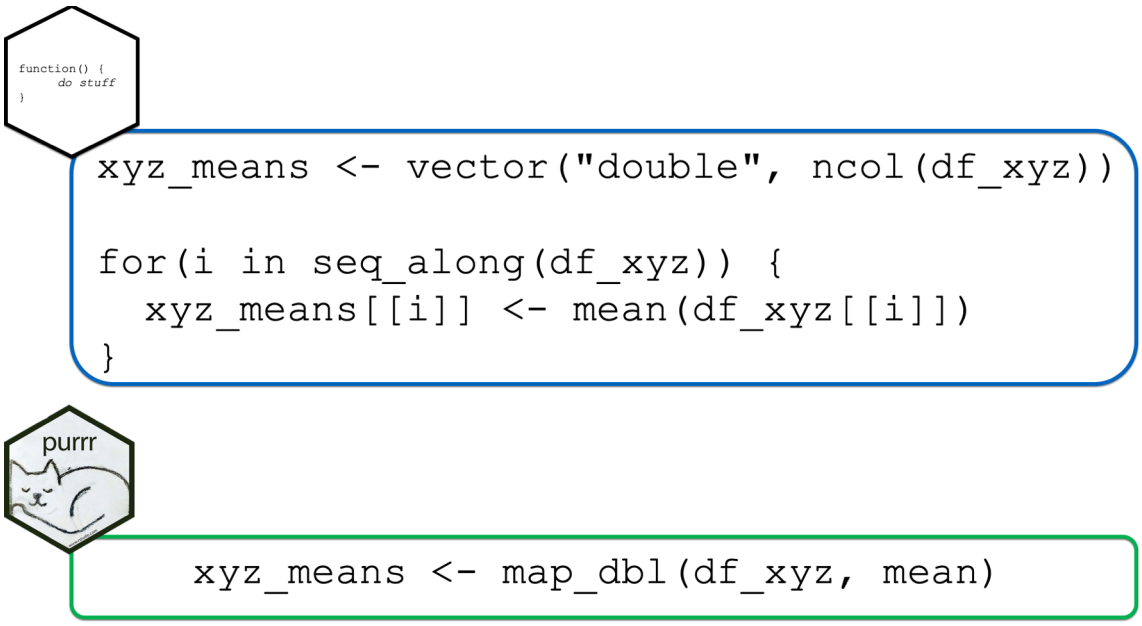


Figure 37.2: Comparing for loops and map functions

In this section, we will compare for loops and the `purrr::map` functions using the example from the beginning of the chapter.

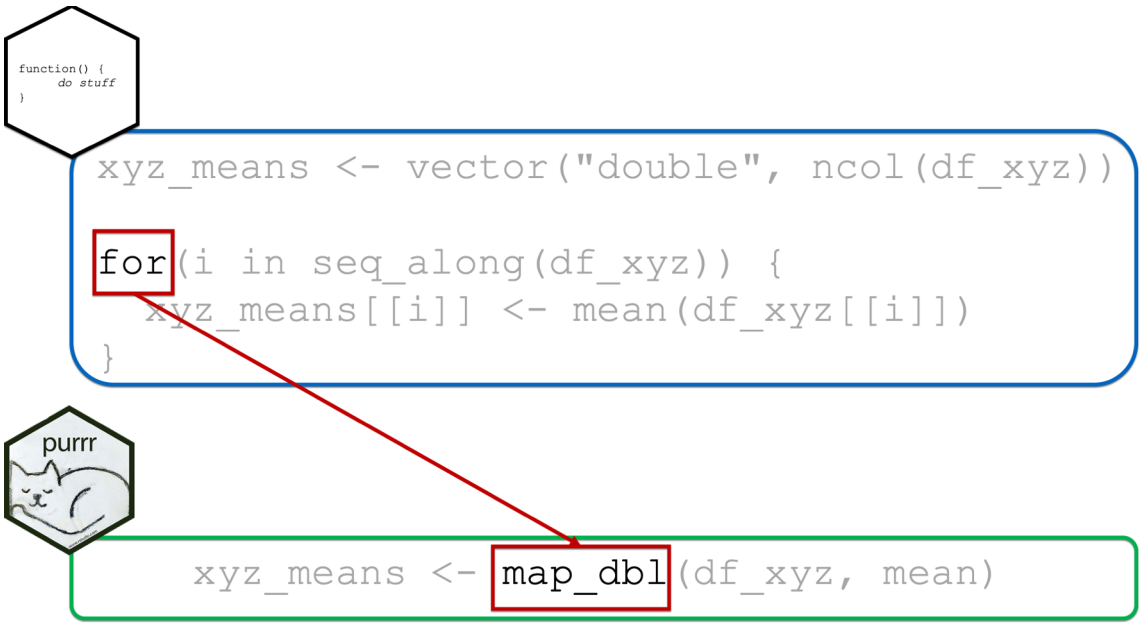


Figure 37.3: Comparing for loops and map functions using example

It's probably obvious to you at this point, but when using `purrr::map` instead of a for loop, we will be using one of the `map` functions instead of the `for()` function.

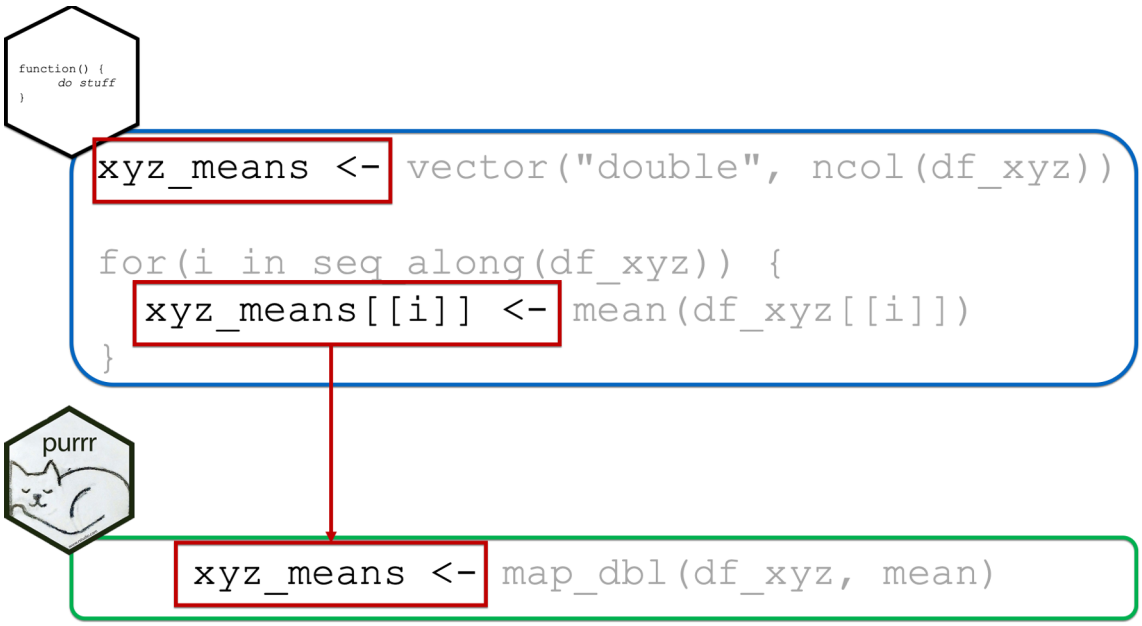


Figure 37.4: Assigning the results of map function to object

Next, as previously discussed above, we are able to assign the returned results of `map_dbl(df_xyz, mean)` to an object in our global environment in the usual way (i.e., with the assignment arrow). This eliminates the need for creating a structure to hold our results ahead of time as we had to do with the for loop. It also eliminates the need to write code that explicitly updates the returned results structure at each iteration (i.e., `xyz_means[[i]]`) as we had to do with the for loop.

However, one nice byproduct of creating the structure to hold our returned results ahead of time was that doing so made it obvious what form and type we expected our results to take.

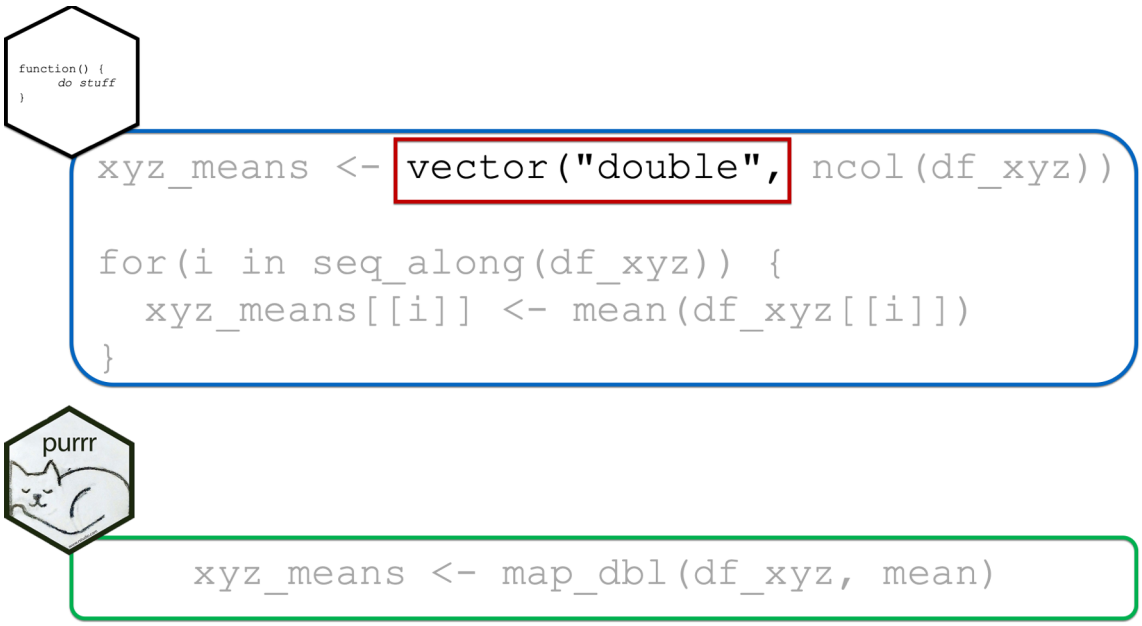


Figure 37.5: Setting the type of the returned results structure I

In the `xyz_means` example above, it's obvious that we expected our returned results to be a vector of numbers because the structure we created to contain our results was a vector of type double.

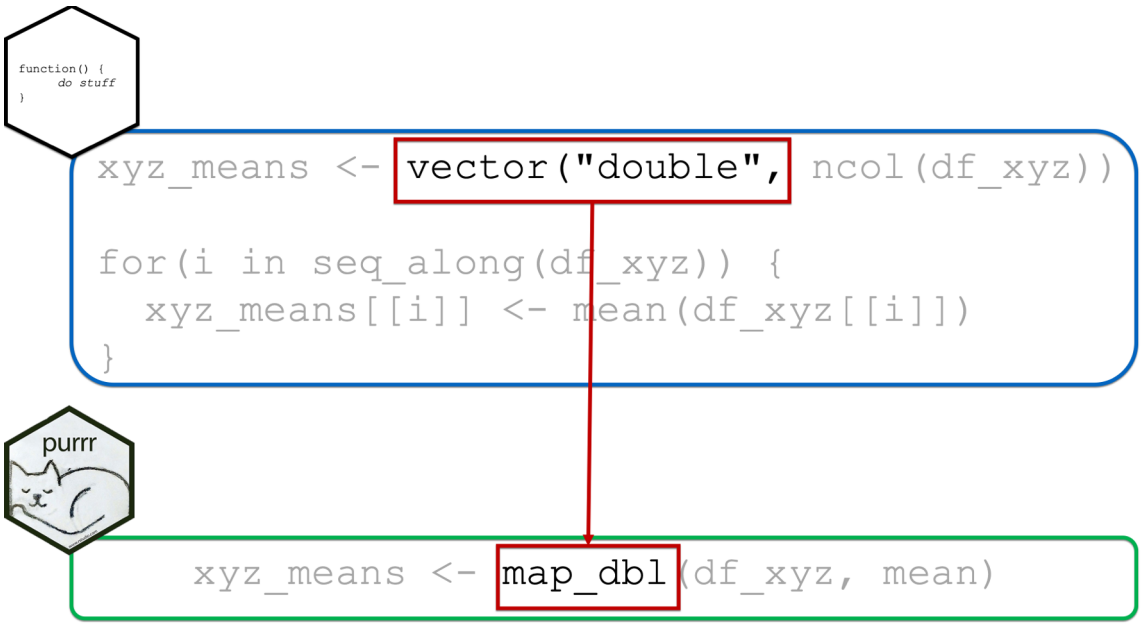


Figure 37.6: Setting the type of the returned results structure II

When using the `purrr::map` functions, which `map` function we choose will serve the same purpose. In the example above, we used `map_dbl()`, which implied that we expected our results to be a vector of type double. In fact, it not only implied that our results *should* be a vector of type double, but it *guaranteed* that our results *would* be a vector of type double (or we would get an error). In this sense, the `map` functions are much safer to use than for loops – we don't get unexpected results.

As a silly example, let's say that we want to extract the number of letters in each name contained in a vector of names. We'll start by creating a vector that contains three random names:

```
names <- c("Avril", "Joe", "Whitney")
```

Next, let's create a structure to contain our results:

```
n_letters <- vector("double", length(names)) # Expecting double
```

The code above (i.e., `vector("double", length(names))`) implies that we expect our results to be type double, which make sense if we expect our results to be the number of letters in some names.

Finally, let's write our for loop:

```
for(i in seq_along(names)) {
  n_letters[[i]] <- stringr::str_extract(names[[i]], "\\w") # Returns character
}

n_letters
```

```
[1] "A" "J" "W"
```

Uh, oh! Those "counts" are letters! What happened? Well, apparently we thought that `stringr::str_extract(names[[i]], "\\w")` would return the count of letters in each name. In actuality, it returns the first letter in each name.

Again, this is a silly example. In this case, it's easy to see and fix our mistake. However, it could be very difficult to debug this problem if the code were buried in a long script or inside of other functions.

Now, let's see what happens when we use `purrr`. We still start with the names:

```
names <- c("Avril", "Joe", "Whitney")
```

We also still imply our expectations that the returned result should be a numeric vector. However, this time we do so by using the `map_dbl` function:

```
n_letters <- map_dbl(
  .x = names,
  .f = stringr::str_extract, "\\w{1}"
)
```

```
Error in `map_dbl()`:
i In index: 1.
Caused by error:
! Can't coerce from a string to a double.
```

But, this time, we don't get an unexpected result. This time, we get an error. This may seem like a pain if you are newish to programming. But it's much better to get an error that you can go fix than an incorrect result that you are totally unaware of!

While we are discussing return types, let's go ahead and introduce some of the other `map` functions. They are:

- `map_dbl()`, which we've already seen. The `map_dbl()` function always returns a numeric vector or an error.

- `map_int()`, which always returns an integer vector or an error.
- `map_lgl()`, which always returns a logical vector or an error.
- `map_chr()`, which always returns a character vector or an error.
- `map_dfr()`, which always returns a data frame created by row-binding results or an error.
- `map_dfc()`, which always returns a data frame created by column-binding results or an error.
- `map()`, which is the most generic, and always returns a list (or an error). We've haven't discussed lists much in this book, but whenever something won't fit into any other kind of object, it will fit into a list.
- `walk()`, which is the only `map` function without a `map` name. We will use `walk()` when we are more interested in the "side-effects" of the function passed to `.f` than its return value. What in the world does that mean? It means that the only thing `walk()` "returns" is exactly what was passed to its `.x` argument. No matter what you pass to the `.f` argument, the object passed to `.x` will be returned by `walk()` unmodified. Your next question might be, "then what's the point? How could that ever be useful?" Typically, `walk()` will only be useful to us for plotting (e.g., where you are interested in viewing the plots, but not saving them as an object) and/or data transfer (we will see an example of this below).

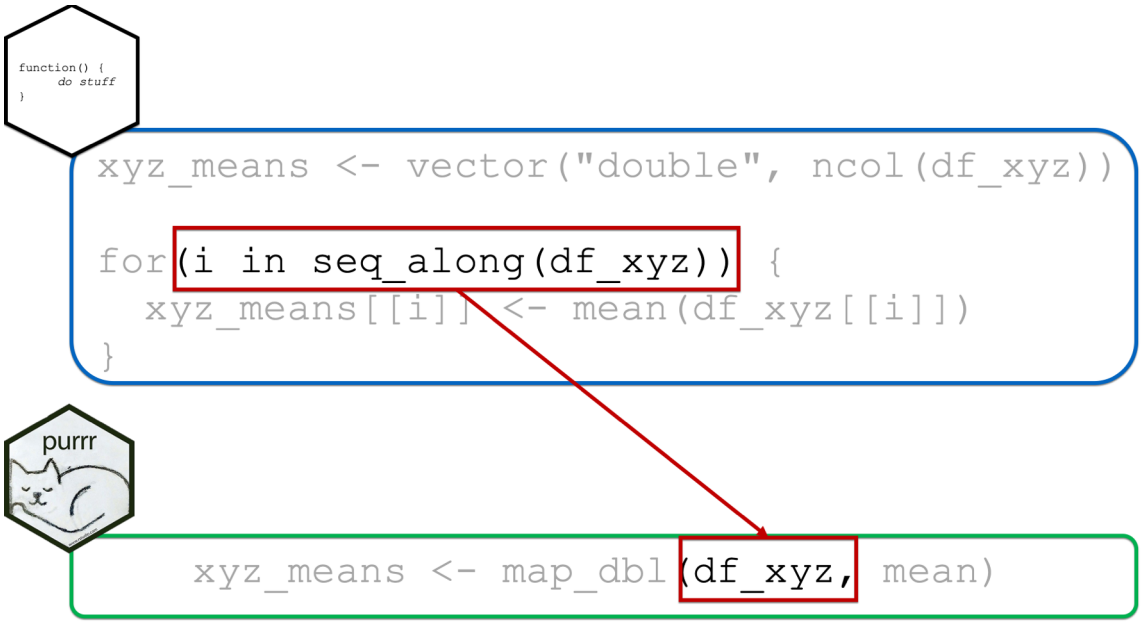


Figure 37.7: Replacing the arguments in the for function with the object we pass to the .x argument of the map function

Next, the object we pass to the `.x` function of the `map` function replaces the entire `i in seq_along(object)` pattern that is passed to the for loop. Again, if the object passed to the `.x` argument is a vector, then `map` will apply the function passed to the `.f` argument to each element of the vector. If the object passed to the `.x` argument is a data frame, then `map` will apply the function passed to the `.f` argument to each column of the data frame.

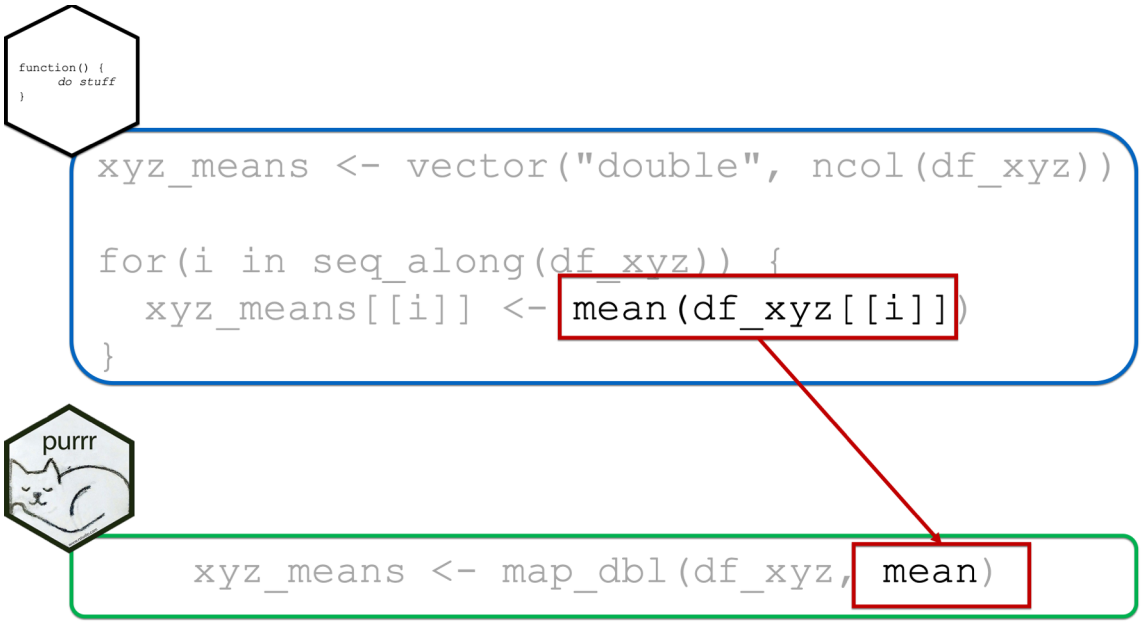


Figure 37.8: Replacing the body of the for function with the function we pass to the .f argument of the map function

Finally, the function passed to the `.f` argument can replace the rest of the "stuff" going on in the for loop body. We can pass a single function (e.g., mean) to the `.f` argument as we did above. However, we can also pass anonymous functions to the `.f` argument. We pass anonymous functions to the `.f` function in basically the exact same way passed anonymous functions to the `.fns` argument of the `across()` function in the chapter on column-wise operations. And, yes, we can also write our anonymous functions using purrr-style lambdas. In fact, the purrr-style lambda syntax is called the purrr-style lambda syntax because it was first created for the `purrr` package and later adopted by `dplyr::across()`. That name probably makes a lot more sense than it did a couple of chapters ago!

That pretty much covers the basics of using the `purrr::map` functions. If you've been reading this book in sequence, there won't really be any conceptually new material in this chapter. We're basically going to do the same things we've been doing for the last couple of chapters. We'll just be using a slightly different (and perhaps preferable) syntax. If you haven't been

reading the book in sequence, you might want to read the chapters on writing functions, column-wise operations, and writing for loops to get the most of the examples below.

37.2 Using purrr for data transfer

37.2.1 Example 1: Importing multiple sheets from an Excel workbook

In the chapter on writing functions we used a for loop to help us import data from an Excel workbook that was stored across multiple sheets. We will once again go through this example using the `purrr` approach.

The simulated data contains some demographic information about three different cities: Houston, Atlanta, and Charlotte. In this scenario, we need to import each sheet, clean the data, and combine them into a single data frame in order to complete our analysis. First, we will load the `readxl` package:

```
library(readxl)
```

You may click here to download this file to your computer.

Then, we may import each sheet like this:

```
houston <- read_excel(
  "city_ses.xlsx",
  sheet = "Houston"
)

atlanta <- read_excel(
  "city_ses.xlsx",
  sheet = "Atlanta"
)

charlotte <- read_excel(
  "city_ses.xlsx",
  sheet = "Charlotte"
)
```

In the code chunks above, we have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code. So, our next step was to write a function to remove some of the unnecessary repetition:

```
import_cities <- function(sheet) {
  df <- read_excel(
    "city_ses.xlsx",
    sheet = sheet
  )
}

houston   <- import_cities("Houston")
atlanta   <- import_cities("Atlanta")
charlotte <- import_cities("Charlotte")
```

However, that approach still has some repetition. So, we next learned how to use a for loop as an alternative approach:

```
path <- "city_ses.xlsx"
sheets <- excel_sheets(path)

for(i in seq_along(sheets)) {
  new_nm <- tolower(sheets[[i]])
  assign(new_nm, read_excel(path, sheet = sheets[[i]]))
}
```

That works just fine! However, we could alternatively use `purrr::walk()` instead like this:

```
# Save the file path to an object so we don't have to type it repeatedly
# or hard-code it in.
path <- "city_ses.xlsx"

walk(
  .x = excel_sheets(path),
  .f = function(x) {
    new_nm <- tolower(x)
    assign(new_nm, read_excel(path, sheet = x), envir = .GlobalEnv)
  }
)
```

```
houston
```

```
# A tibble: 5 x 4
  pid     age sex   ses_score
  <chr> <dbl> <chr>     <dbl>
```

```
1 001      13 F            88
2 003      13 F            78
3 007      14 M            83
4 014      12 F            76
5 036      13 M            84
```

```
atlanta
```

```
# A tibble: 5 x 4
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

```
charlotte
```

```
# A tibble: 5 x 4
  pid     age sex     ses
  <chr> <dbl> <chr> <dbl>
1 004      13 F        84
2 011      14 M        66
3 018      12 M        92
4 023      12 M        89
5 030      13 F        83
```

Here's what we did above:

- We used the `walk()` function from the `purrr` package to import every sheet from an Excel workbook.
- First, we saved the path to the Excel workbook to a separate object. We didn't have to do this. However, doing so prevented us from having to type out the full file path repeatedly in the rest of our code. Additionally, if the file path ever changed, we would only have to update it in one place.
- Second, we passed the return value of the `excel_sheets()` function, which is a character vector containing each sheet name, to the `.x` argument of the `walk()` function. We didn't have to do this. We could have typed each sheet name manually. However, there shouldn't be any accidental typos if we use the `excel_sheets()` function, and we don't have to make any changes to our code if more sheets are added to the Workbook in the future.

- Third, we passed an anonymous function to the `walk()`'s `.f` argument. Inside the anonymous function, we assigned each data frame created by the `read_excel()` function to our global environment using the `assign()` function. Notice that because we are using the `assign()` *inside* of another function, we have to explicitly tell the `assign()` function to assign the data frames being imported to the global environment using `envir = .GlobalEnv`. Without getting too technical, keep in mind that functions create their own little enclosed environments (see a discussion here), which makes the `envir = .GlobalEnv` part necessary.

Additionally, you may have some questions swirling around your head right now about the `walk()` function itself. In particular, you might be wondering why we used `walk()` instead of `map()` and why we didn't assign the return value of `walk()` to an object. We'll answer both questions next.

37.2.2 Why walk instead of map?

The short answer is that `map` functions return *one* thing (i.e., a vector, list, or data frame). In this situation, we wanted to "return" *three* things (i.e., the `houston` data frame, the `atlanta` data frame, and the `charlotte` data frame).

Technically, we could have used the `map()` function to return a list of data frames like this:

```
list_of_df <- map(
  .x = excel_sheets(path),
  .f = ~ read_excel(path, sheet = .x)
)
```

```
str(list_of_df)
```

```
List of 3
 $ : tibble [5 x 4] (S3: tbl_df/tbl/data.frame)
  ..$ pid      : chr [1:5] "001" "003" "007" "014" ...
  ..$ age      : num [1:5] 13 13 14 12 13
  ..$ sex      : chr [1:5] "F" "F" "M" "F" ...
  ..$ ses_score: num [1:5] 88 78 83 76 84
 $ : tibble [5 x 4] (S3: tbl_df/tbl/data.frame)
  ..$ id       : chr [1:5] "002" "009" "012" "013" ...
  ..$ age      : num [1:5] 14 15 13 13 12
  ..$ gender   : chr [1:5] "M" "M" "F" "F" ...
  ..$ ses_score: num [1:5] 64 35 70 66 59
 $ : tibble [5 x 4] (S3: tbl_df/tbl/data.frame)
  ..$ pid: chr [1:5] "004" "011" "018" "023" ...
```

```
  ..$ age: num [1:5] 13 14 12 12 13
  ..$ sex: chr [1:5] "F" "M" "M" "M" ...
  ..$ ses: num [1:5] 84 66 92 89 83
```

From there, we could extract and modify each data frame from the list like this:

```
houston <- list_of_df[[1]]
houston
```

```
# A tibble: 5 x 4
  pid     age sex   ses_score
  <chr> <dbl> <chr>     <dbl>
1 001      13 F            88
2 003      13 F            78
3 007      14 M            83
4 014      12 F            76
5 036      13 M            84
```

```
atlanta <- list_of_df[[2]]
atlanta
```

```
# A tibble: 5 x 4
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

```
charlotte <- list_of_df[[2]]
charlotte
```

```
# A tibble: 5 x 4
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

Of course, now we have a bunch of repetition again! Alternatively, we could have also used the `map_dfr()`, which always returns a data frame created by row-binding results or an error. You can think of `map_dfr()` as taking the three data frames above and passing them to the `bind_rows()` function and returning that result:

```
# Passing list_of_df to bind_rows()
bind_rows(list_of_df)
```

```
# A tibble: 15 x 7
   pid     age sex   ses_score id    gender   ses
   <chr> <dbl> <chr>     <dbl> <chr> <chr>  <dbl>
 1 001      13 F            88 <NA>  <NA>      NA
 2 003      13 F            78 <NA>  <NA>      NA
 3 007      14 M            83 <NA>  <NA>      NA
 4 014      12 F            76 <NA>  <NA>      NA
 5 036      13 M            84 <NA>  <NA>      NA
 6 <NA>     14 <NA>         64 002   M         NA
 7 <NA>     15 <NA>         35 009   M         NA
 8 <NA>     13 <NA>         70 012   F         NA
 9 <NA>     13 <NA>         66 013   F         NA
10 <NA>     12 <NA>         59 022   F         NA
11 004      13 F            NA <NA>  <NA>      84
12 011      14 M            NA <NA>  <NA>      66
13 018      12 M            NA <NA>  <NA>      92
14 023      12 M            NA <NA>  <NA>      89
15 030      13 F            NA <NA>  <NA>      83
```

```
# Using map_dfr() to directly produce the same result
cities <- map_dfr(
  .x = excel_sheets(path),
  .f = ~ read_excel(path, sheet = .x)
)

cities
```

```
# A tibble: 15 x 7
   pid     age sex   ses_score id    gender   ses
   <chr> <dbl> <chr>     <dbl> <chr> <chr>  <dbl>
 1 001      13 F            88 <NA>  <NA>      NA
 2 003      13 F            78 <NA>  <NA>      NA
 3 007      14 M            83 <NA>  <NA>      NA
```

```
 4 014      12 F            76 <NA>  <NA>      NA
 5 036      13 M            84 <NA>  <NA>      NA
 6 <NA>     14 <NA>         64 002   M         NA
 7 <NA>     15 <NA>         35 009   M         NA
 8 <NA>     13 <NA>         70 012   F         NA
 9 <NA>     13 <NA>         66 013   F         NA
10 <NA>     12 <NA>         59 022   F         NA
11 004      13 F            NA <NA>  <NA>      84
12 011      14 M            NA <NA>  <NA>      66
13 018      12 M            NA <NA>  <NA>      92
14 023      12 M            NA <NA>  <NA>      89
15 030      13 F            NA <NA>  <NA>      83
```

There would be absolutely nothing wrong with taking this approach and then cleaning up the combined data you see above. However, in this case, the preference was to import each sheet as a separate data frame, clean up each separate data frame, and then combine ourselves. If your preference is to use `map_dfr()` instead, then you definitely should.

37.2.3 why we didn't assign the return value of `walk()` to an object?

As we discussed above, the only thing `walk()` "returns" is exactly what was passed to its `.x` argument. No matter what you pass to the `.f` argument, the object passed to `.x` will be returned by `walk()` unmodified. In this case, that would just be the sheet names:

```
returned_by_walk <- walk(
  .x = excel_sheets(path),
  .f = function(x) {
    new_nm <- tolower(x)
    assign(new_nm, read_excel(path, sheet = x), envir = .GlobalEnv)
  }
)

returned_by_walk
```

```
[1] "Houston"   "Atlanta"   "Charlotte"
```

Don't be confused, the data frames are still being imported and assigned to the global environment via the anonymous function we passed to `.f` above. But, but those data frames aren't the values *returned by* `walk()` – They are a *side-effect* of the operations taking place inside of `walk()`.

Finally, we could make our original `walk()` code slightly more concise by using the purrr-style lambda syntax to write our anonymous function like this:

```
path <- "city_ses.xlsx"

walk(
  .x = excel_sheets(path),
  .f = ~ assign(tolower(.), read_excel(path, sheet = .), envir = .GlobalEnv)
)
```

```
houston
```

```
# A tibble: 5 x 4
  pid     age sex   ses_score
  <chr> <dbl> <chr>     <dbl>
1 001      13 F            88
2 003      13 F            78
3 007      14 M            83
4 014      12 F            76
5 036      13 M            84
```

```
atlanta
```

```
# A tibble: 5 x 4
  id      age gender ses_score
  <chr> <dbl> <chr>      <dbl>
1 002      14 M             64
2 009      15 M             35
3 012      13 F             70
4 013      13 F             66
5 022      12 F             59
```

```
charlotte
```

```
# A tibble: 5 x 4
  pid     age sex     ses
  <chr> <dbl> <chr> <dbl>
1 004      13 F        84
2 011      14 M        66
3 018      12 M        92
4 023      12 M        89
5 030      13 F        83
```

37.3 Using purrr for data management

37.3.1 Example 1: Adding NA at multiple positions

We'll start this section with a relatively simple example using the same data we used to start the chapter on column-wise operations and the chapter on writing for loops.

```
set.seed(123)
df_xyz <- tibble(
  x = rnorm(10),
  y = rnorm(10),
  z = rnorm(10)
) %>%
  print()
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 -0.230   0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705  0.111 -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  -1.69
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

In those chapters, we used the code below to add missing values to our data frame:

```
df_xyz$x[2] <- NA_real_
df_xyz$y[4] <- NA_real_
df_xyz$z[6] <- NA_real_
df_xyz
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 NA       0.360 -0.218
```

```
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

Dealing with those missing values, rather than *adding* those missing values was the point of the previously mentioned examples. So, we ignored the unnecessary repetition in the code above. But, for all the reasons we've been discussing, we should strive to write more robust code. Imagine, for example, that you were adding missing data to hundreds or thousands of rows as part of a simulation study. Using the method above would become problematic pretty quickly.

In this case, it might be useful to start our solution with writing a function (click here to review function writing). Let's name our function `add_na_at()` because it helps us add an `NA` value to a vector at a position of our choosing. Logically, then, it follows that we will need to be able to pass our function a *vector* that we want to add the `NA` value to, and a *position* to add the `NA` value at. So, our first attempt might look something like this:

```
add_na_at <- function(vect, pos) {
  vect[[pos]] <- NA
}
```

Let's test it out:

```
add_na_at(df_xyz$x, 2) %>% print()
```

```
[1] NA
```

Is a single `NA` the result we wanted? Nope! If this result is surprising to you, please review the section of the writing functions chapter on return values. Briefly, the last line of our function body is the single value `df_xyz$x[[2]]`, which was set to be equal to `NA`. But, we don't want our function to return just one position of the vector – we want it to return the entire vector. So, let's reference the entire vector on the last line of the function body:

```
add_na_at <- function(vect, pos) {
  vect[[pos]] <- NA
  vect
}
```

```
add_na_at(df_xyz$x, 2)
```

```
 [1] -0.56047565          NA  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

That's better! Again, we know that data frame columns *are* vectors, so we can use our new function inside of mutate to add `NA` values to each column in our data frame at a position of our choosing:

```
df_xyz %>%
  mutate(
    x = add_na_at(x, 2),
    y = add_na_at(y, 4),
    z = add_na_at(z, 6)
  )
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 NA       0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

I can hear what you are saying now. "Sure, that's the result we wanted, but we didn't eliminate very much repetitive code." You are not wrong. A case could be made that this code is easier to quickly glance at and understand, but it isn't much less repetitive. That's where `purrr` comes in. Let's try using `purrr` to come up with a better solution now.

The first question we might ask ourselves is, "which `map` function should we choose?" Well, we know we want our end result to be a data frame, so it makes sense for us to choose either `map_dfr` or `map_dfc`. However, it might be useful to start with the plain `map()` function that returns a list as we begin to experiment with solving a problem using `purrr`. This is because R can put almost anything into a list, and therefore, we will almost always get *something*

returned to us (as opposed to an error) by `map()`. Further, the thing returned to us typically can provide us with some insight into what's going on inside `.f`.

Next, we know that we want to iterate over every column of the `df_xyz` data frame. So, we can pass it to the `.x` argument.

We also know that we want each column to get passed to the `vect` argument of `add_na_at()` iteratively. So, we want to pass `add_na_at` (without parentheses) to the `.f` argument.

Finally, we can't supply `add_na_at()` with just one argument – the vector – can we?

```
add_na_at(df_xyz$x)
```

```
Error in vect[[pos]] <- NA: missing subscript
```

No way! We have to give it position as well. Do you remember which argument allows us to pass any additional arguments to the function we passed to the `.f` argument?

The `...` argument is where we pass any additional arguments to the function we passed to the `.f` argument. But remember, we don't actually type out `... =`. We simply type additional arguments, separated by commas, after the function name supplied to `.f`:

```
map(
  .x = df_xyz,
  .f = add_na_at, 2
)
```

```
$x
 [1] -0.56047565          NA  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197

$y
 [1]  1.2240818         NA  0.4007715         NA -0.5558411  1.7869131
 [7]  0.4978505 -1.9666172  0.7013559 -0.4727914

$z
 [1] -1.0678237         NA -1.0260044 -0.7288912 -0.6250393         NA
 [7]  0.8377870  0.1533731 -1.1381369  1.2538149
```

Or alternatively, we can use the purrr-style lambda to pass our function to `.f`:

```
map(
  .x = df_xyz,
  .f = ~ add_na_at(.x, 2)
)
```

```
$x
 [1] -0.56047565          NA  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197

$y
 [1]  1.2240818         NA  0.4007715         NA -0.5558411  1.7869131
 [7]  0.4978505 -1.9666172  0.7013559 -0.4727914

$z
 [1] -1.0678237         NA -1.0260044 -0.7288912 -0.6250393         NA
 [7]  0.8377870  0.1533731 -1.1381369  1.2538149
```

Notice that we have to use the special `.x` symbol inside the function call where we would normally want to type the name of the column we want the function to operate on. We saw something similar before in the chapter on column-wise operations.

Now, let's discuss the result we are getting. The result you see above is a list, which is what `map()` will always return to us. Specifically, this is a list with three elements – `x`, `y`, and `z`. Each element of the list is a vector of numbers. Does this feel familiar? Does it seem sort of similar to a data frame? If so, good intuition! In R, a data frame *is* a list. It's simply a special case of a list. It's a special case because all vectors in the data frame must have the length, and because R knows to print each vector to the screen as a column. In fact, we can easily convert the list above to a data frame by passing it to the `as.data.frame()` function:

```
map(
  .x = df_xyz,
  .f = ~ add_na_at(.x, 2)
) %>%
  as.data.frame()
```

```
            x          y          z
1  -0.56047565  1.2240818 -1.0678237
2           NA         NA         NA
3   1.55870831  0.4007715 -1.0260044
4   0.07050839         NA -0.7288912
5   0.12928774 -0.5558411 -0.6250393
```

```
6   1.71506499  1.7869131         NA
7   0.46091621  0.4978505  0.8377870
8  -1.26506123 -1.9666172  0.1533731
9  -0.68685285  0.7013559 -1.1381369
10 -0.44566197 -0.4727914  1.2538149
```

Alternatively, we could just use `map_dfc` as a shortcut instead:

```
map_dfc(
  .x = df_xyz,
  .f = ~ add_na_at(.x, 2)
)
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 NA      NA     NA
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

Why `map_dfc` instead of `map_dfr`? Because we want to combine `x`, `y`, and `z` together as *columns*, not as rows.

Ok, so we almost have the solution we want. There's just one problem. In the code above, the `NA` is always being put into the second position because we have `2` hard-coded into `add_na_at(.x, 2)`. We need a way to iterate over our columns *and* a set of numbers *simultaneously* in order to get the final result we want. Fortunately, that's exactly what the `map2` variants (i.e., `map2_dbl()`, `map2_int()`, `map2_lgl()`, etc.) of each of the `map` functions allows us to do.

Instead of supplying map a single object to iterate over (i.e., `.x`) we can supply it with *two* objects to iterate over (i.e., `.x` and `.y`):

```r
map2_dfc(
  .x = df_xyz,
  .y = c(2, 4, 6),
  .f = ~ add_na_at(.x, .y)
)
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 NA       0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

This can sometimes take a second to wrap your mind around. Here's an illustration that may help:

Iteration	.x	.y	.f	result
One	df_xyz$x	2	add_na_at(df_xyz$x, 2)	-0.56047565, NA, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197

```
map2_dfc(
   .x = df_xyz,
   .y = c(2, 4, 6),
   .f = ~ add_na_at(.x, .y)
)
```

Figure 37.9: Illustrating how map iterates over two objects simultaneously - I

In the first iteration, `.x` took on the value of the first column in the `df_xyz` data frame (i.e., `x`) and `.y` took on the value of the first element in the numeric vector that we passed to the `.y` argument (i.e., `2`). Then, the `.x` and `.y` were replaced with `df_xyz$x` and `2` respectively in the function we passed to `.f`. The result of that iteration was a vector of numbers that was identical to `df_xyz$x` except that its second element was an `NA`.

Iteration	.x	.y	.f	result
One	df_xyz$x	2	add_na_at(df_xyz$x, 2)	-0.56047565, NA, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197
Two	y	4	add_na_at(df_xyz$y, 4)	1.2240818, 0.3598138, 0.4007715, NA, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914

```
map2_dfc(
   .x = df_xyz,
   .y = c(2, 4, 6),
   .f = ~ add_na_at(.x, .y)
)
```

Figure 37.10: Illustrating how map iterates over two objects simultaneously - II

In the second iteration, `.x` took on the value of the second column in the `df_xyz` data frame (i.e., `y`) and `.y` took on the value of the second element in the numeric vector that we passed to the `.y` argument (i.e., `4`). Then, the `.x` and `.y` were replaced with `df_xyz$y` and `4` respectively in the function we passed to `.f`. The result of that iteration was a vector of numbers that was identical to `df_xyz$y` except that its fourth element was an `NA`.

Iteration	.x	.y	.f	result
One	df_xyz$x	2	add_na_at(df_xyz$x, 2)	-0.56047565, NA, 1.55870831, 0.07050839, 0.12928774, 1.71506499, 0.46091621, -1.26506123, -0.68685285, -0.44566197
Two	y	4	add_na_at(df_xyz$y, 4)	1.2240818, 0.3598138, 0.4007715, NA, -0.5558411, 1.7869131, 0.4978505, -1.9666172, 0.7013559, -0.4727914
Three	z	6	add_na_at(df_xyz$z, 6)	-1.0678237, -0.2179749, -1.0260044, -0.7288912, -0.6250393, NA, 0.8377870, 0.1533731, -1.1381369, 1.2538149

```
map2_dfc(
  .x = df_xyz,
  .y = c(2, 4, 6),
  .f = ~ add_na_at(.x, .y)
)
```

Figure 37.11: Illustrating how map iterates over two objects simultaneously - III

In the third iteration, `.x` took on the value of the third column in the `df_xyz` data frame (i.e., `z`) and `.y` took on the value of the third element in the numeric vector that we passed to the `.y` argument (i.e., `6`). Then, the `.x` and `.y` were replaced with `df_xyz$z` and `6` respectively in the function we passed to `.f`. The result of that iteration was a vector of numbers that was identical to `df_xyz$z` except that its sixth element was an `NA`.

Finally, `map2_dfc()` passed all of these vectors to `bind_cols()` (invisibly to us) and returned them as a data frame.

The code above gives us our entire solution. But, if we really were using this code in a simulation with hundreds or thousands of columns, we probably wouldn't want to manually supply a vector of column positions to the `.y` argument. Instead, we could use the `sample()` function to supply random column positions to the `.y` argument like this:

```
set.seed(8142020)

map2_dfc(
  .x = df_xyz,
  .y = sample(1:10, 3, TRUE),
  .f = ~ add_na_at(.x, .y)
)
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560  NA     -1.07
 2 NA       0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97  NA
 9 NA       0.701 -1.14
10 -0.446  -0.473  1.25
```

Pretty nice, right?

Before moving on, note that we did not have to create the `add_na_at()` function ahead of time the way we did. If we didn't think we would need to use `add_na_at()` in any other part of our program, we might have decided to pass the code inside of `add_na_at()` to the `.f` argument as an anonymous function instead.

As a reminder, here is what our *named* function looks like:

```
add_na_at <- function(vect, pos) {
  vect[[pos]] <- NA
  vect
}
```

And here is what our `purrr` code would look like if we used an *anonymous* function instead:

```
map2_dfc(
  .x = df_xyz,
  .y = c(2, 4, 6),
  .f = function(vect, pos) {
    vect[[pos]] <- NA
    vect
  }
)
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
```

```
 1 -0.560   1.22  -1.07
 2 NA       0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

Or, if we used a purrr-style lambda anonymous function instead:

```
map2_dfc(
  .x = df_xyz,
  .y = c(2, 4, 6),
  .f = ~ {
    .x[[.y]] <- NA
    .x
  }
)
```

```
# A tibble: 10 x 3
         x      y      z
     <dbl>  <dbl>  <dbl>
 1 -0.560   1.22  -1.07
 2 NA       0.360 -0.218
 3  1.56    0.401 -1.03
 4  0.0705 NA     -0.729
 5  0.129  -0.556 -0.625
 6  1.72    1.79  NA
 7  0.461   0.498  0.838
 8 -1.27   -1.97   0.153
 9 -0.687   0.701 -1.14
10 -0.446  -0.473  1.25
```

Whichever style you choose to use is largely just a matter of preference in this case (as it is in many cases).

37.3.2 Example 2. Detecting matching values by position

In the chapter on writing functions, we created an `is_match()` function. In that scenario, we wanted to see if first name, last name, and street name matched at each ID between our data frames. More specifically, we wanted to combine the two data frames into a single data frame and create three new dummy variables that indicated whether first name, last name, and address matched respectively.

Here are the data frames we simulated and combined:

```
people_1 <- tribble(
  ~id_1, ~name_first_1, ~name_last_1, ~street_1,
  1,     "Easton",      NA,           "Alameda",
  2,     "Elias",       "Salazar",    "Crissy Field",
  3,     "Colton",      "Fox",        "San Bruno",
  4,     "Cameron",     "Warren",     "Nottingham",
  5,     "Carson",      "Mills",      "Jersey",
  6,     "Addison",     "Meyer",      "Tingley",
  7,     "Aubrey",      "Rice",       "Buena Vista",
  8,     "Ellie",       "Schmidt",    "Division",
  9,     "Robert",      "Garza",      "Red Rock",
  10,    "Stella",      "Daniels",    "Holland"
)

people_2 <- tribble(
  ~id_2, ~name_first_2, ~name_last_2, ~street_2,
  1,     "Easton",      "Stone",      "Alameda",
  2,     "Elas",        "Salazar",    "Field",
  3,     NA,            "Fox",        NA,
  4,     "Cameron",     "Waren",      "Notingham",
  5,     "Carsen",      "Mills",      "Jersey",
  6,     "Adison",      NA,           NA,
  7,     "Aubrey",      "Rice",       "Buena Vista",
  8,     NA,            "Schmidt",    "Division",
  9,     "Bob",         "Garza",      "Red Rock",
  10,    "Stella",      NA,           "Holland"
)

people <- people_1 %>%
  bind_cols(people_2) %>%
  print()
```

```
# A tibble: 10 x 8
```

```
    id_1 name_first_1 name_last_1 street_1      id_2 name_first_2 name_last_2
   <dbl> <chr>        <chr>       <chr>        <dbl> <chr>        <chr>
 1     1 Easton       <NA>        Alameda          1 Easton       Stone
 2     2 Elias        Salazar     Crissy Field     2 Elas         Salazar
 3     3 Colton       Fox         San Bruno        3 <NA>         Fox
 4     4 Cameron      Warren      Nottingham       4 Cameron      Waren
 5     5 Carson       Mills       Jersey           5 Carsen       Mills
 6     6 Addison      Meyer       Tingley          6 Adison       <NA>
 7     7 Aubrey       Rice        Buena Vista      7 Aubrey       Rice
 8     8 Ellie        Schmidt     Division         8 <NA>         Schmidt
 9     9 Robert       Garza       Red Rock         9 Bob          Garza
10    10 Stella       Daniels     Holland         10 Stella       <NA>
# i 1 more variable: street_2 <chr>
```

Here is the function we wrote to help us create the dummy variables:

```
is_match <- function(value_1, value_2) {
  result <- value_1 == value_2
  result <- if_else(is.na(result), FALSE, result)
  result
}
```

And here is how we applied the function we wrote to get our results:

```
people %>%
  mutate(
    name_first_match = is_match(name_first_1, name_first_2),
    name_last_match  = is_match(name_last_1, name_last_2),
    street_match     = is_match(street_1, street_2)
  ) %>%
  # Order like columns next to each other for easier comparison
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         FALSE            Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
```

```
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         FALSE            Schmidt     Schmidt
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

However, in the code chunk above, we still have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition from our code. Because we are using dplyr, and all of our data resides inside of a single data frame, your first instinct might be to use `across()` inside of `mutate()` to perform column-wise operations. Unfortunately, that method won't work in this scenario.

The `across()` function will apply the function we pass to the `.fns` argument to each column passed to the `.cols` argument, one at a time. But, we need to pass two columns at a time to the `is_match()` function. For example, `name_first_1` *and* `name_first_2`. That makes this task a little trickier than most. But, here's how we accomplished it using a for loop:

```
cols <- c("name_first", "name_last", "street")

for(i in seq_along(cols)) {
  col_1   <- paste0(cols[[i]], "_1")
  col_2   <- paste0(cols[[i]], "_2")
  new_col <- paste0(cols[[i]], "_match")
  people[[new_col]] <- is_match(people[[col_1]], people[[col_2]])
}

people %>%
  select(id_1, starts_with("name_f"), starts_with("name_l"), starts_with("s"))
```

```
# A tibble: 10 x 10
    id_1 name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <dbl> <chr>        <chr>        <lgl>            <chr>       <chr>
 1     1 Easton       Easton       TRUE             <NA>        Stone
 2     2 Elias        Elas         FALSE            Salazar     Salazar
 3     3 Colton       <NA>         FALSE            Fox         Fox
 4     4 Cameron      Cameron      TRUE             Warren      Waren
 5     5 Carson       Carsen       FALSE            Mills       Mills
 6     6 Addison      Adison       FALSE            Meyer       <NA>
 7     7 Aubrey       Aubrey       TRUE             Rice        Rice
 8     8 Ellie        <NA>         FALSE            Schmidt     Schmidt
```

```
 9     9 Robert       Bob          FALSE            Garza       Garza
10    10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

Now, let's go over one way to get the same result using `purrr`. The first method very closely resembles our for loop. In fact, we will basically just copy and paste our for loop body into an anonymous function being passed to the `.f` argument:

```
map_dfc(
  .x = c("name_first", "name_last", "street"),
  .f = function(col, data = people) {
    col_1 <- paste0(col, "_1")
    col_2 <- paste0(col, "_2")
    new_nm <- paste0(col, "_match")
    data[[new_nm]] <- data[[col_1]] == data[[col_2]]
    data[[new_nm]] <- if_else(is.na(data[[new_nm]]), FALSE, data[[new_nm]])
    data[c(col_1, col_2, new_nm)]
  }
)
```

```
# A tibble: 10 x 9
   name_first_1 name_first_2 name_first_match name_last_1 name_last_2
   <chr>        <chr>        <lgl>            <chr>       <chr>
 1 Easton       Easton       TRUE             <NA>        Stone
 2 Elias        Elas         FALSE            Salazar     Salazar
 3 Colton       <NA>         FALSE            Fox         Fox
 4 Cameron      Cameron      TRUE             Warren      Waren
 5 Carson       Carsen       FALSE            Mills       Mills
 6 Addison      Adison       FALSE            Meyer       <NA>
 7 Aubrey       Aubrey       TRUE             Rice        Rice
 8 Ellie        <NA>         FALSE            Schmidt     Schmidt
 9 Robert       Bob          FALSE            Garza       Garza
10 Stella       Stella       TRUE             Daniels     <NA>
# i 4 more variables: name_last_match <lgl>, street_1 <chr>, street_2 <chr>,
#   street_match <lgl>
```

In the code above, we used roughly the same amount of code to complete the task with a loop that we used to complete it without a loop. However, this code still has some advantages. We only typed "name_first", "name_last", and "street" once at the beginning of the code chunk. Therefore, we didn't have to worry about forgetting to change a column name after copying

and pasting code. Additionally, if we later decide that we also want to compare other columns (e.g., middle name, birth date, city, state, zip code), we only have to update the code in one place – where we create the `cols` vector.

37.4 Using purrr for analysis

Let's return to the examples from the column-wise operations chapter and the chapter on writing for loops for our discussion of using the `purrr` package to remove unnecessary repetition from our analyses. We will once again use the simulated for the examples below.

```
study <- tibble(
  age       = c(32, 30, 32, 29, 24, 38, 25, 24, 48, 29, 22, 29, 24, 28, 24, 25,
                25, 22, 25, 24, 25, 24, 23, 24, 31, 24, 29, 24, 22, 23, 26, 23,
                24, 25, 24, 33, 27, 25, 26, 26, 26, 26, 26, 27, 24, 43, 25, 24,
                27, 28, 29, 24, 26, 28, 25, 24, 26, 24, 26, 31, 24, 26, 31, 34,
                26, 25, 27, NA),
  age_group = c(2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2,
                2, 1, 1, 1, NA),
  gender    = c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1,
                1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1,
                1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 1,
                1, 1, 2, 1, NA),
  ht_in     = c(70, 63, 62, 67, 67, 58, 64, 69, 65, 68, 63, 68, 69, 66, 67, 65,
                64, 75, 67, 63, 60, 67, 64, 73, 62, 69, 67, 62, 68, 66, 66, 62,
                64, 68, NA, 68, 70, 68, 68, 66, 71, 61, 62, 64, 64, 63, 67, 66,
                69, 76, NA, 63, 64, 65, 65, 71, 66, 65, 65, 71, 64, 71, 60, 62,
                61, 69, 66, NA),
  wt_lbs    = c(216, 106, 145, 195, 143, 125, 138, 140, 158, 167, 145, 297, 146,
                125, 111, 125, 130, 182, 170, 121, 98, 150, 132, 250, 137, 124,
                186, 148, 134, 155, 122, 142, 110, 132, 188, 176, 188, 166, 136,
                147, 178, 125, 102, 140, 139, 60, 147, 147, 141, 232, 186, 212,
                110, 110, 115, 154, 140, 150, 130, NA, 171, 156, 92, 122, 102,
                163, 141, NA),
  bmi       = c(30.99, 18.78, 26.52, 30.54, 22.39, 26.12, 23.69, 20.67, 26.29,
                25.39, 25.68, 45.15, 21.56, 20.17, 17.38, 20.8, 22.31, 22.75,
                26.62, 21.43, 19.14, 23.49, 22.66, 32.98, 25.05, 18.31, 29.13,
                27.07, 20.37, 25.01, 19.69, 25.97, 18.88, 20.07, NA, 26.76,
                26.97, 25.24, 20.68, 23.72, 24.82, 23.62, 18.65, 24.03, 23.86,
                10.63, 23.02, 23.72, 20.82, 28.24, NA, 37.55, 18.88, 18.3,
```

```
                19.13, 21.48, 22.59, 24.96, 21.63, NA, 29.35, 21.76, 17.97,
                22.31, 19.27, 24.07, 22.76, NA),
  bmi_3cat  = c(3, 1, 2, 3, 1, 2, 1, 1, 2, 2, 2, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1,
                1, 1, 3, 2, 1, 2, 2, 1, 2, 1, 2, 1, 1, NA, 2, 2, 2, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 2, NA, 3, 1, 1, 1, 1, 1, 1, 1, NA, 2, 1,
                1, 1, 1, 1, 1, NA)
) %>%
  mutate(
    age_group = factor(age_group, labels = c("Younger than 30", "30 and Older")),
    gender    = factor(gender, labels = c("Female", "Male")),
    bmi_3cat  = factor(bmi_3cat, labels = c("Normal", "Overweight", "Obese"))
  ) %>%
  print()
```

```
# A tibble: 68 x 7
     age age_group       gender ht_in wt_lbs   bmi bmi_3cat
   <dbl> <fct>           <fct>  <dbl>  <dbl> <dbl> <fct>
 1    32 30 and Older    Male      70    216  31.0 Obese
 2    30 30 and Older    Female    63    106  18.8 Normal
 3    32 30 and Older    Female    62    145  26.5 Overweight
 4    29 Younger than 30 Male      67    195  30.5 Obese
 5    24 Younger than 30 Female    67    143  22.4 Normal
 6    38 30 and Older    Female    58    125  26.1 Overweight
 7    25 Younger than 30 Female    64    138  23.7 Normal
 8    24 Younger than 30 Male      69    140  20.7 Normal
 9    48 30 and Older    Male      65    158  26.3 Overweight
10    29 Younger than 30 Male      68    167  25.4 Overweight
# i 58 more rows
```

37.4.1 Example 1: Continuous statistics

In this first example, we will use `purrr` to calculate a set of statistics for multiple continuous variables in our data frame. We will start by creating the same function we created at the beginning of the chapter on writing functions.

```
continuous_stats <- function(var) {
  study %>%
    summarise(
      n_miss = sum(is.na({{ var }})),
      mean   = mean({{ var }}, na.rm = TRUE),
      median = median({{ var }}, na.rm = TRUE),
```

```
      min    = min({{ var }}, na.rm = TRUE),
      max    = max({{ var }}, na.rm = TRUE)
    )
}
```

Now, let's once again *use* the function we just created above to calculate the descriptive measures we are interested in.

```
continuous_stats(age)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      1  26.9     26    22    48
```

```
continuous_stats(ht_in)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      3  66.0     66    58    76
```

```
continuous_stats(wt_lbs)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      2  148.   142.    60   297
```

```
continuous_stats(bmi)
```

```
# A tibble: 1 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      4  23.6   22.9  10.6  45.2
```

Once again, you notice that we have essentially the same code copied more than twice. That's a red flag that we should be thinking about removing unnecessary repetition. We've already seen how to accomplish this goal using the `across()` function. Now, let's learn how to accomplish this goal using the `purrr` package.

```
map_dfr(
  .x = quos(age, ht_in, wt_lbs, bmi),
  .f = continuous_stats
)
```

```
# A tibble: 4 x 5
  n_miss  mean median   min   max
   <int> <dbl>  <dbl> <dbl> <dbl>
1      1  26.9   26    22    48
2      3  66.0   66    58    76
3      2 148.   142.   60   297
4      4  23.6   22.9  10.6  45.2
```

Here's what we did above:

- We used the `map_dfr()` function from the `purrr` package to iteratively pass the columns `age`, `ht_in`, `wt_lbs`, and `bmi` to our `continuous_stats` function *and* row-bind the results into a single results data frame.
- We haven't seen the `quos()` function before. It's another one of those tidy evaluation functions. You can type `?rlang::quos` in your console to read more about it. When we can wrap a single column name with the `quo()` function, or a list of column names with the `quos()` function, we are telling R to look for them in the data frame being passed to a `dplyr` verb rather than looking for them as objects in the global environment.

At this point, you may be wondering which row in the results data frame above corresponds to which variable? Great question! When we were using `continuous_stats()` to analyze one variable at a time, it didn't really matter that the variable name wasn't part of the output. However, now that we are apply `continuous_stats()` to multiple columns, it would really be nice to have the column name in the results. Luckily, we can easily make that happen with one small tweak to our `continuous_stats()` function.

```
continuous_stats <- function(var) {
  study %>%
    summarise(
      variable = quo_name(var), # Add variable name to the output
      n_miss   = sum(is.na({{ var }})),
      mean     = mean({{ var }}, na.rm = TRUE),
      median   = median({{ var }}, na.rm = TRUE),
      min      = min({{ var }}, na.rm = TRUE),
      max      = max({{ var }}, na.rm = TRUE)
    )
}
```

Here's what we did above:

- We used the `quo_name()` function to grab the name of the column being passed to the `summarise()` function and turn it into a character string. Then, we assigned that character string to column in the results data frame called `variable`. So, when the `age` column is passed to `summarise()` inside of the function body, `quo_name(var)` returns the value `"age"` and then that value is assigned to the `variable` column in the expression `variable = quo_name(var)`.

Let's try out our new and improved `continuous_stats()` function:

```
map_dfr(
  .x = quos(age, ht_in, wt_lbs, bmi),
  .f = continuous_stats
)
```

```
# A tibble: 4 x 6
  variable n_miss  mean median   min   max
  <chr>     <int> <dbl>  <dbl> <dbl> <dbl>
1 age           1  26.9   26    22    48
2 ht_in         3  66.0   66    58    76
3 wt_lbs        2 148.   142.   60   297
4 bmi           4  23.6   22.9  10.6  45.2
```

That works great, but we'd probably like to be able to use `continuous_stats()` with other data frames too. Currently, we can't do that because we have the `study` data frame hard-coded into our function. Luckily, we've already seen how to replace a hard-coded data frame by adding a `data` argument to our function like this:

```
continuous_stats <- function(data, var) {
  data %>%  # Don't forget to replace "study" with "data" here too!
    summarise(
      variable = quo_name(var),
      n_miss   = sum(is.na({{ var}} )),
      mean     = mean({{ var }}, na.rm = TRUE),
      median   = median({{ var }}, na.rm = TRUE),
      min      = min({{ var }}, na.rm = TRUE),
      max      = max({{ var }}, na.rm = TRUE)
    )
}
```

And now, we can analyze all the continuous variables in the `study` data:

```
map_dfr(
  .x = quos(age, ht_in, wt_lbs, bmi),
  .f = continuous_stats, data = study
)
```

```
# A tibble: 4 x 6
  variable n_miss  mean median   min   max
  <chr>     <int> <dbl>  <dbl> <dbl> <dbl>
1 age           1  26.9   26    22    48
2 ht_in         3  66.0   66    58    76
3 wt_lbs        2 148.   142.   60   297
4 bmi           4  23.6   22.9  10.6  45.2
```

And all the continuous variables in the `df_xyz` data:

```
map_dfr(
  .x = quos(x, y, z),
  .f = continuous_stats, data = df_xyz
)
```

```
# A tibble: 3 x 6
  variable n_miss   mean  median   min   max
  <chr>     <int>  <dbl>   <dbl> <dbl> <dbl>
1 x             1  0.108  0.0705 -1.27  1.72
2 y             1  0.220  0.401  -1.97  1.79
3 z             1 -0.284 -0.625  -1.14  1.25
```

37.4.2 Example 2: Categorical statistics

For our second example of using the `purrr` package for analysis, we'll once again write some code to iteratively analyze all the categorical variables in our `study` data frame. In the last chapter, we learned how to use a for loop to do this analysis. As a refresher, here is the final solution we arrived at:

```
# Structure 1. An object to contain the results.
  # Create the data frame structure that will contain our results
cat_table <- tibble(
  variable = vector("character"),
  category = vector("character"),
  n        = vector("numeric")
```

```
)

# Structure 2. The actual for loop.
  # For each column, get the column name, category names, and count.
  # Then, add them to the bottom of the results data frame we created above.
for(i in c("age_group", "gender", "bmi_3cat")) {
  cat_stats <- study %>%
    count(.data[[i]]) %>% # Use .data to refer to the current data frame.
    mutate(variable = names(.)[1]) %>% # Use . to refer to the current data frame.
    rename(category = 1)

  # Here is where we update cat_table with the results for each column
  cat_table <- bind_rows(cat_table, cat_stats)
}
```

```
cat_table
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <chr>           <dbl>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

To use **purrr** instead, we can pretty much copy and paste the code from the for loop body above as an anonymous function to the `.f` argument to `map_dfr()` like this:

```
map_dfr(
  .x = c("age_group", "gender", "bmi_3cat"),
  .f = function(x) {
    study %>%
      count(.data[[x]]) %>%
      mutate(variable = names(.)[1]) %>%
      rename(category = 1) %>%
```

```
      select(variable, category, n)
  }
)
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <fct>           <int>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

As you can see, the code that is *doing the analysis* is exactly the same in our for loop solution and our `purrr` solution. However, in this case, the `purrr` solution requires a lot less code *around* the analysis code. And, for some, the `purrr` code is easier to read.

If we didn't want to type our column names in quotes, we could use tidy evaluation again. All we have to do is pass the column names to the `quos()` function in the `.x` argument and change the `.data[[x]]` being passed to the `count()` function to `{ x }` like this:

```
map_dfr(
  .x = quos(age_group, gender, bmi_3cat), # Change c() to quos()
  .f = function(x) {
    study %>%
      count({{ x }}) %>% # Change .data[[x]] to {{ x }}
      mutate(variable = names(.)[1]) %>%
      rename(category = 1) %>%
      select(variable, category, n)
  }
)
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <fct>           <int>
 1 age_group Younger than 30    56
```

```
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

And as before, we'd probably like to be able to use this code with other data frames too. So, we will once again replace a hard-coded data frame by adding a `data` argument to our function:

```
map_dfr(
  .x = quos(age_group, gender, bmi_3cat),
  .f = function(x, data = study) {
    data %>% # Don't forget to replace "study" with "data" here too!
      count({{ x }}) %>%
      mutate(variable = names(.)[1]) %>%
      rename(category = 1) %>%
      select(variable, category, n)
  }
)
```

```
# A tibble: 10 x 3
   variable  category            n
   <chr>     <fct>           <int>
 1 age_group Younger than 30    56
 2 age_group 30 and Older       11
 3 age_group <NA>                1
 4 gender    Female             43
 5 gender    Male               24
 6 gender    <NA>                1
 7 bmi_3cat  Normal             43
 8 bmi_3cat  Overweight         16
 9 bmi_3cat  Obese               5
10 bmi_3cat  <NA>                4
```

And that concludes the chapter! You might feel a little bit like your head is swimming at this point. It was a lot to take in! As was stated at the end of the for loop chapter, it is not recommended to memorize everything we covered in this chapter. Instead, we recommend

that you read it until you sort of get the general idea of the `purrr` package and when it might be useful. Then, refer back to this chapter, or other online references that discuss the `purrr` package (there are many good ones out there), if you find yourself in a situation where you believe that the `purrr` package might be the right tool to help you complete a given programming task.

If you feel as though you want to take a deeper dive into the `purrr` package right away, then we suggest checking out the iteration chapter of R for Data Science. For an even deeper dive, the functionals chapter of Advanced R is recommended.

This concludes the repeated operations part of the book. If you aren't feeling totally comfortable with the material we covered in this part of the book right now, that's ok. You're not expected to yet. It takes *time* and *practice* for most people to be able to wrap their head around repeated operations. You are on track at this point as long as you understand why unnecessary repetition in your code is generally something you want to avoid. Then, slowly start using any of the methods *you feel most comfortable with* to remove the unnecessary repetition from your code. Start by doing so in very simple cases and gradually work your way up to more complicated cases. With some practice, you may eventually think this stuff is even fun!

Part VII

Collaboration

38 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.

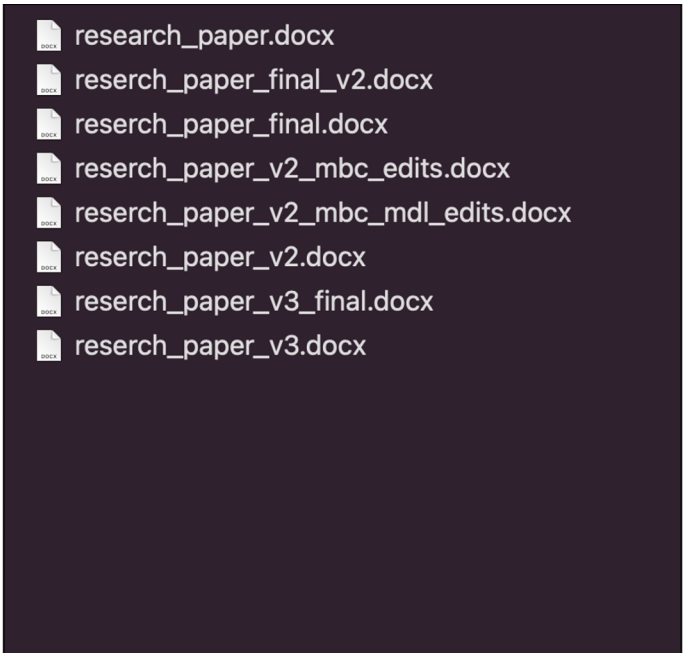
But, why should you be interested in the first place? Well, there are at least four overarching reasons why you should consider learning to use git and GitHub as part of your workflow when your projects include data and/or coding:

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.

38.1 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.

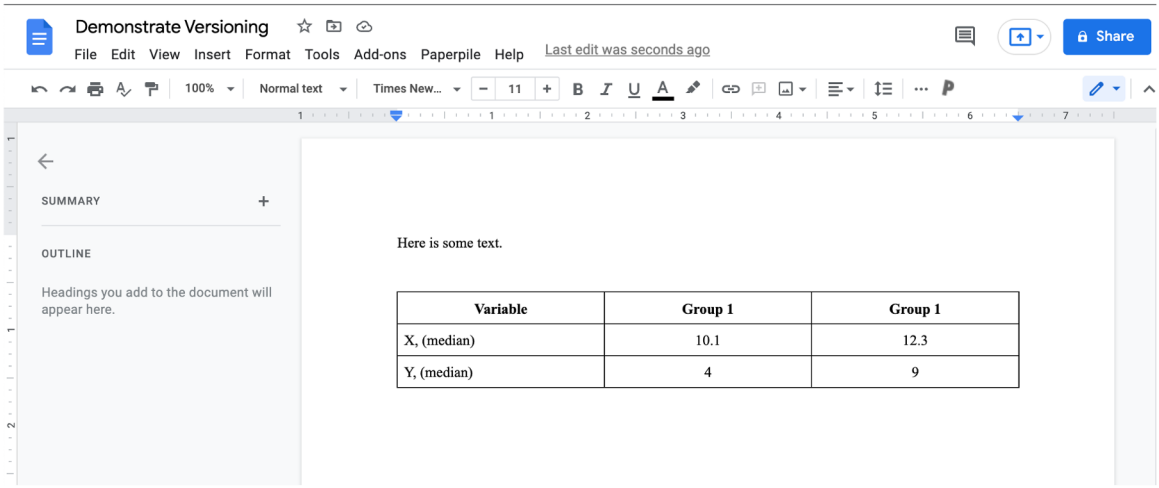


Figure 38.1: A gif about switching back to an old version in Google Docs.

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.

38.2 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.

38.3 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:[11]

> 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:[11] [12]

> 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.

38.4 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**
>
> 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.

38.5 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.

39 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.

39.1 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.

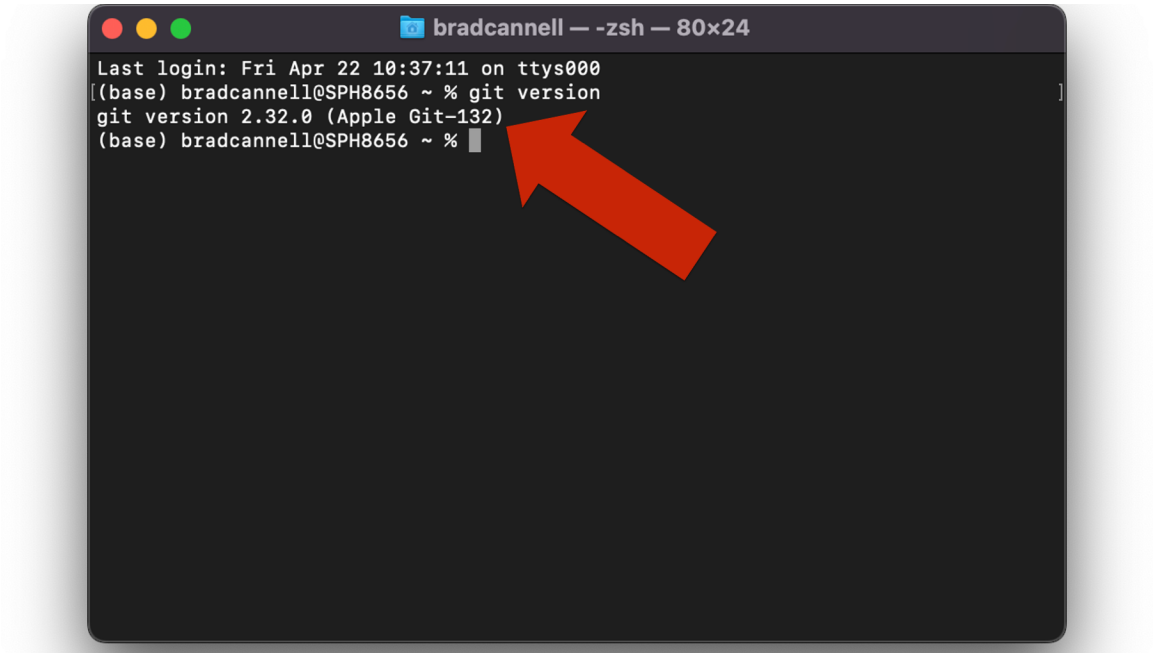
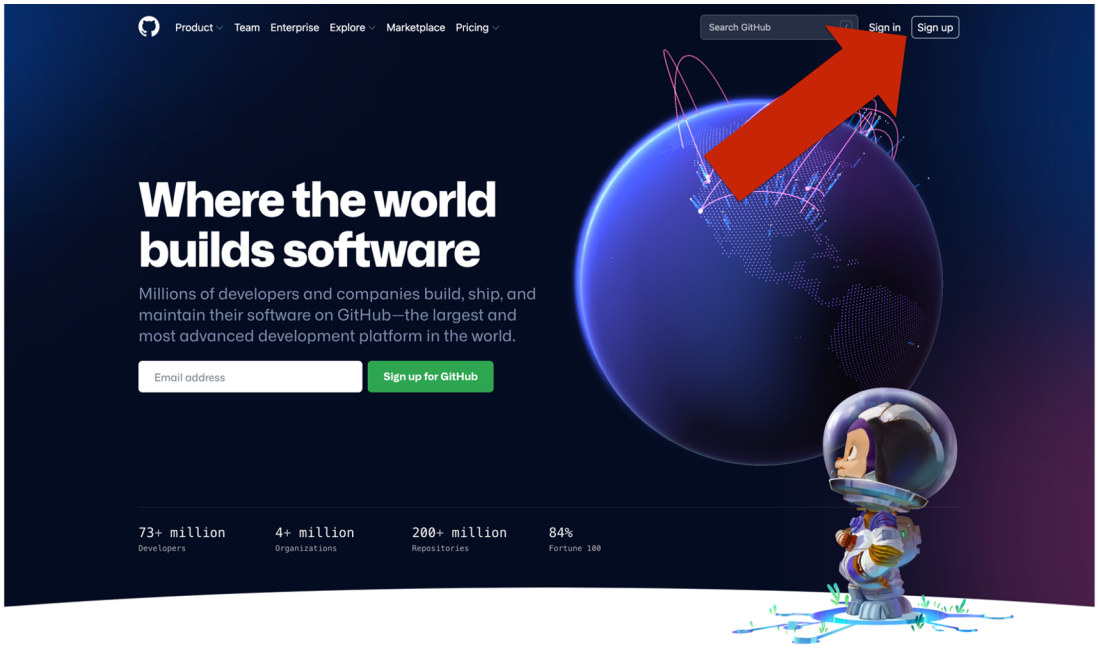


Figure 39.1: Checking git version in the MacOS terminal.

39.2 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/



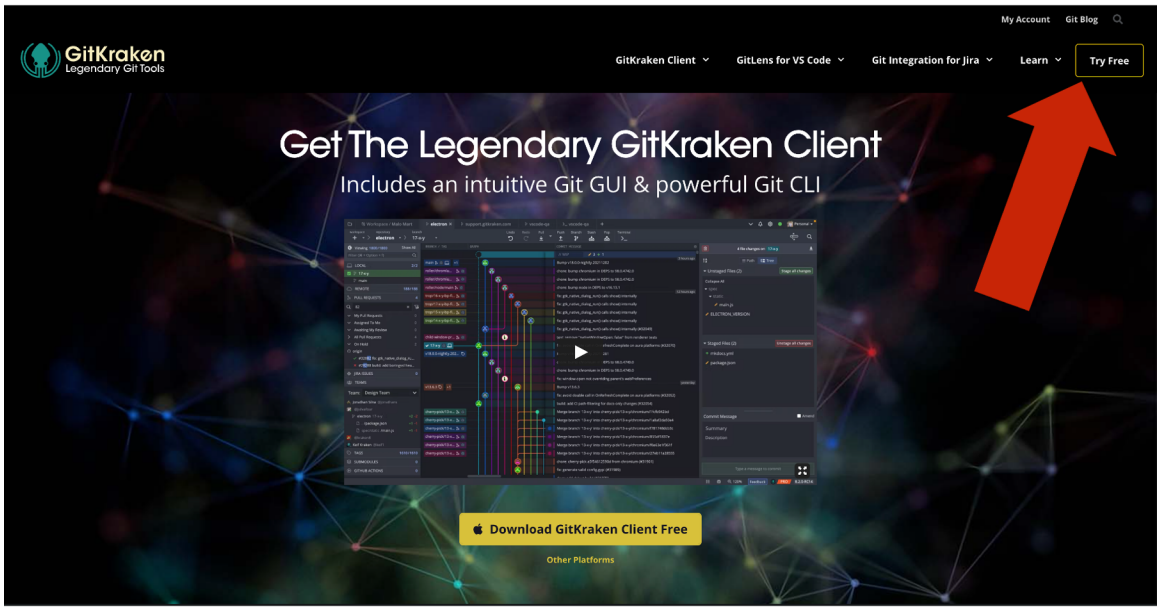
39.3 Install GitKraken

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 39.1. 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.

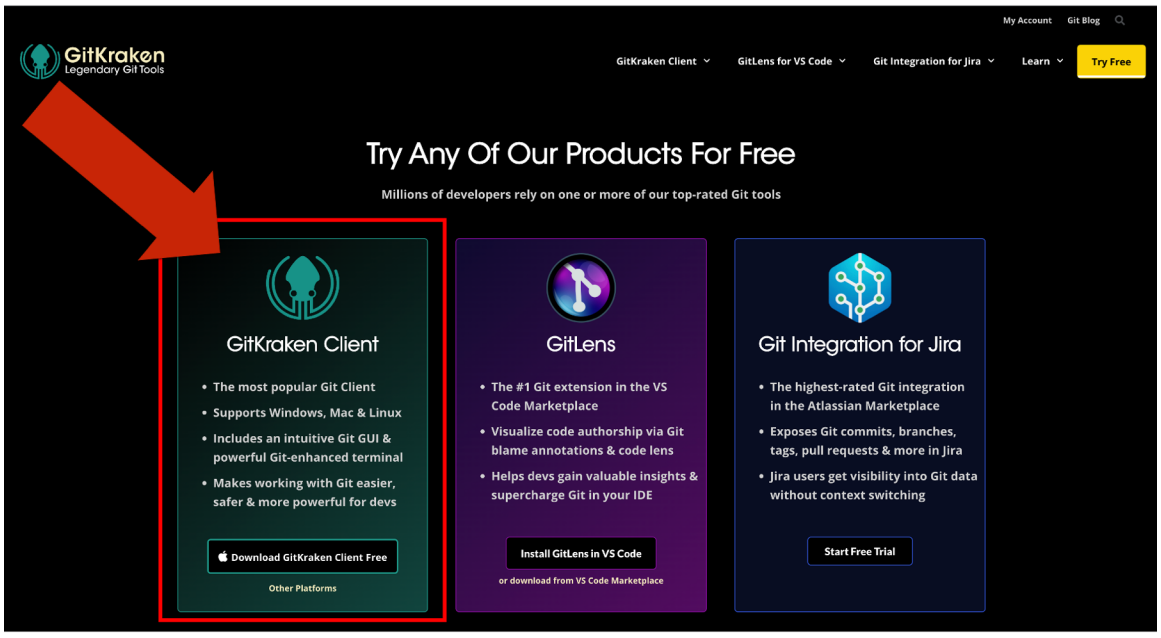
Thankfully, other third-party vendors have made excellent GUI's for git that we can download and use for free. Our current favorite is called **GitKraken**. To use GitKraken, you will first need to navigate to the GitKraken website (https://www.gitkraken.com/). If it helps, you can think of git and GitKraken 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"

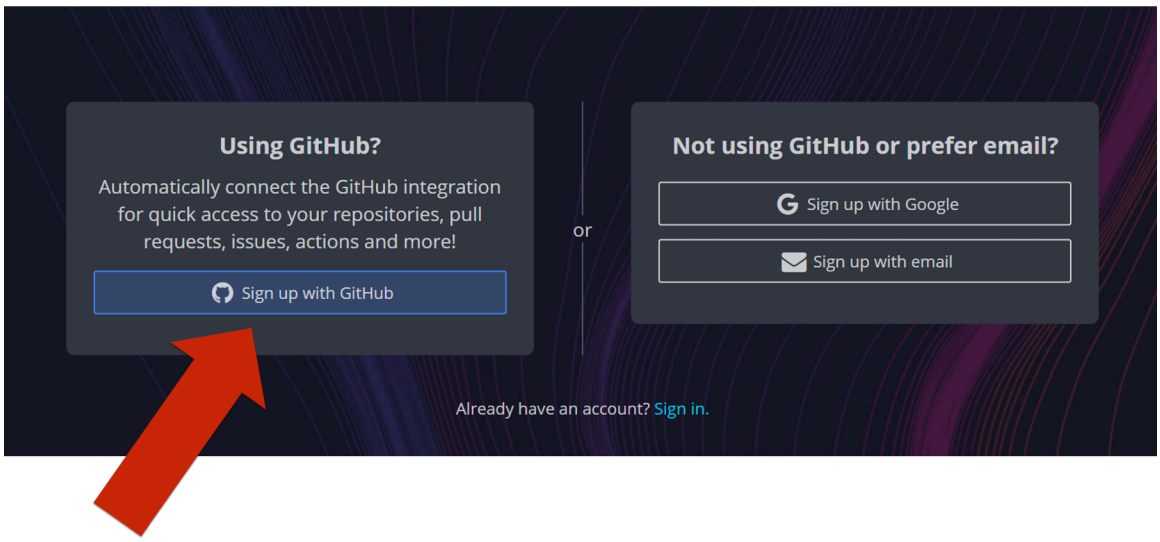
version, which is… free. And, the free version will do everything you need to do to follow along with this book.



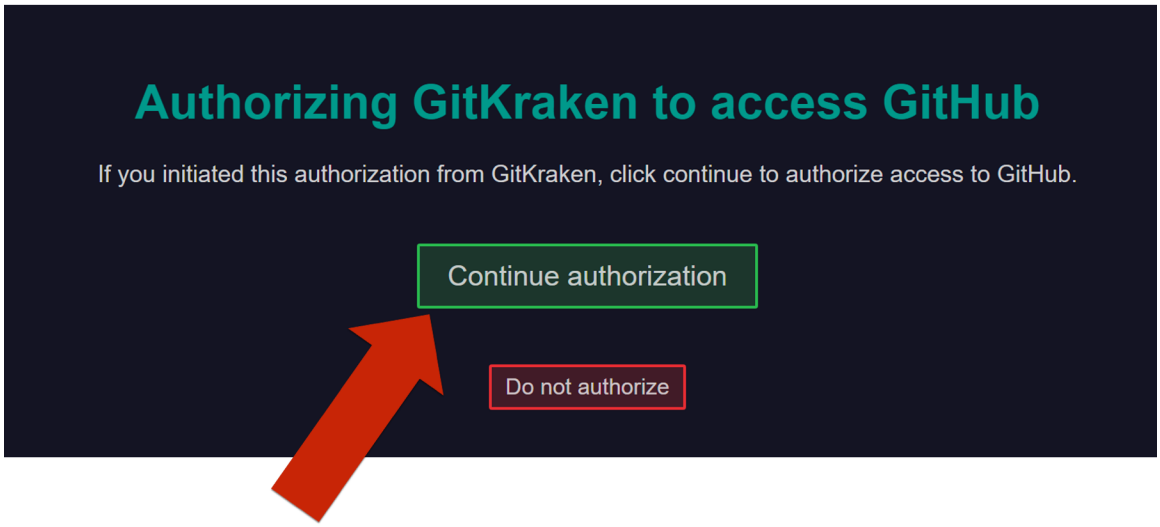
Next, you will need to click on the "Try Free" button. Then, download and install the GitKraken Client to your computer.



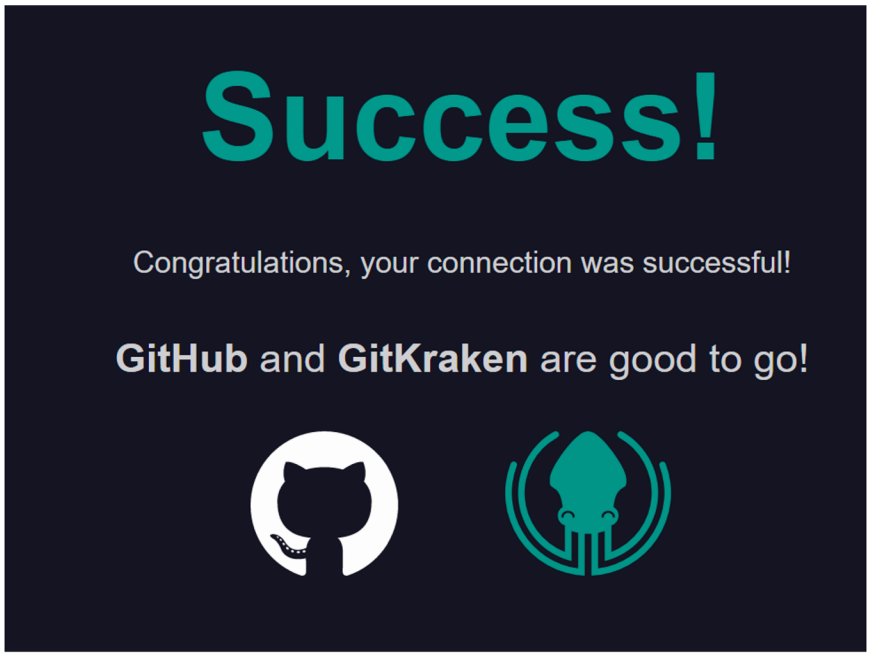
As you are installing GitKraken, it should ask you if you want to sign up with your GitHub account. Yes, you do! It will make your life much easier down the road. If you didn't sign up for a GitHub account in the previous step, please go back and do so.



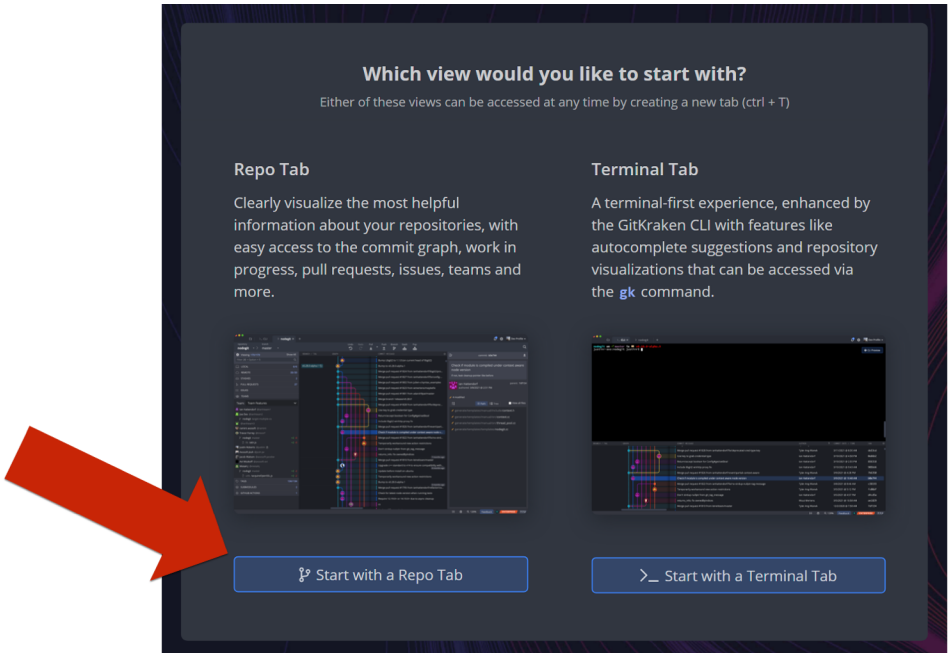
Then click the green `Continue authorization` button.



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.

39.4 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.

39.5 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

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.

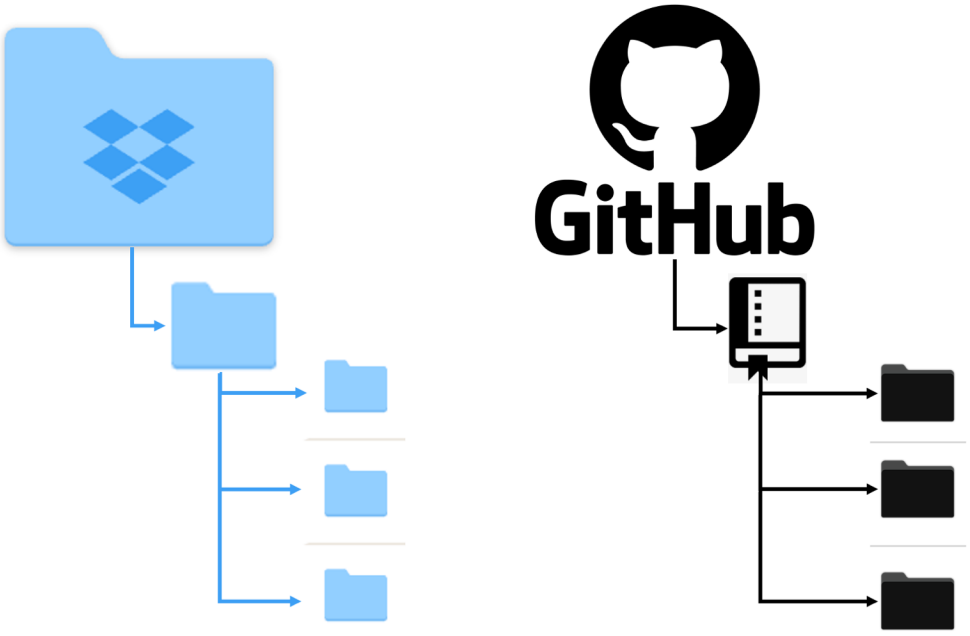
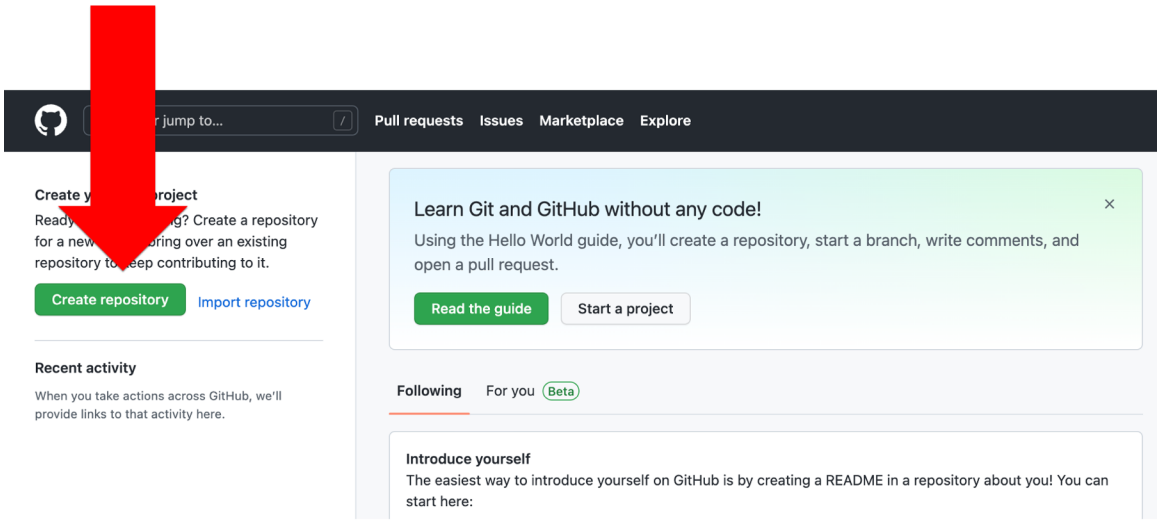


Figure 39.2: GitHub repositories compared to Dropbox.

> **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/).

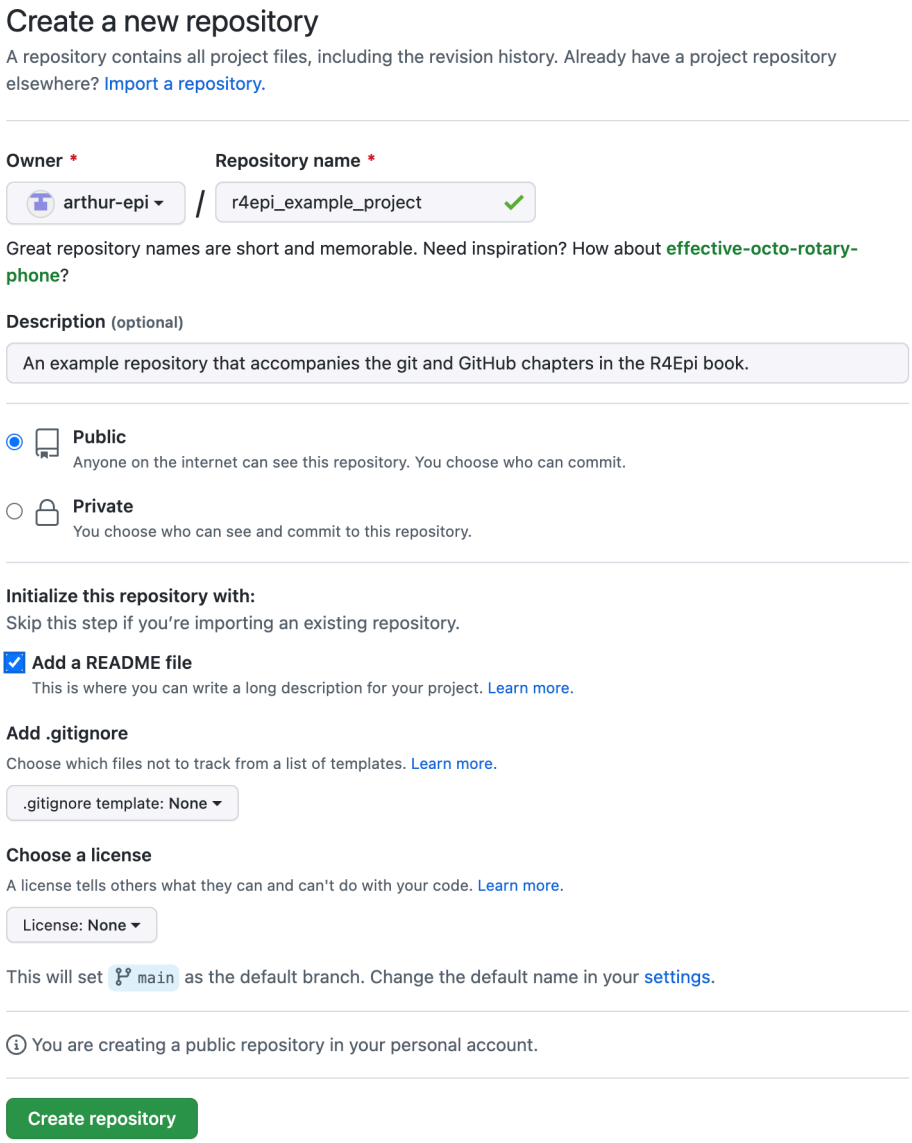


After clicking the green `Create repository` button, the next page Arthur will see is the setup page for his repository. For the purposes of this example, he will use the following information to set it up.

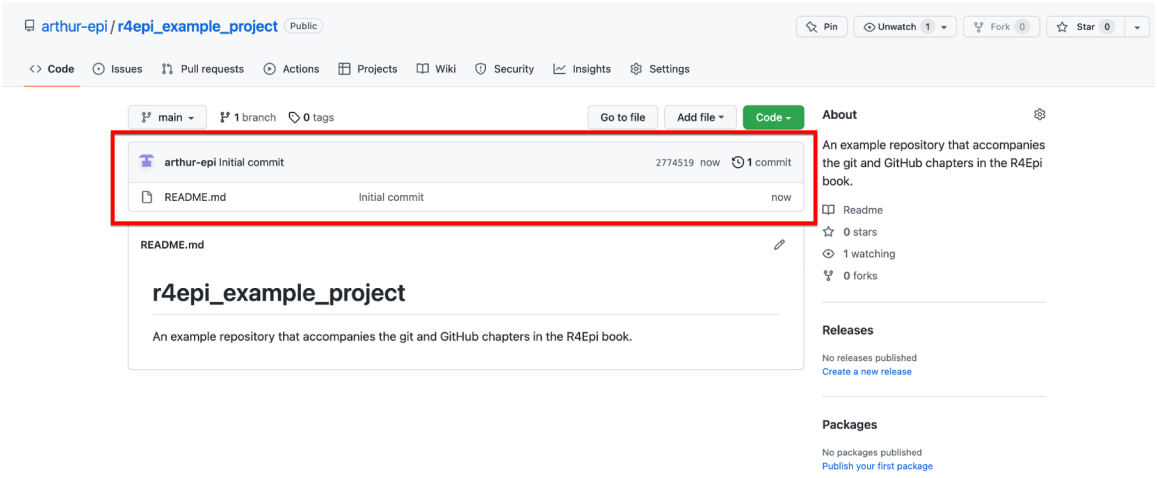
- **Repository name**: As the on-screen prompt says, great repository names are short and memorable. Further, the repository name must be unique to his account (i.e., he can't have two repositories with the same name), and it can only include letters, numbers, dashes (`-`), underscores (`_`), and periods (`.`). 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.

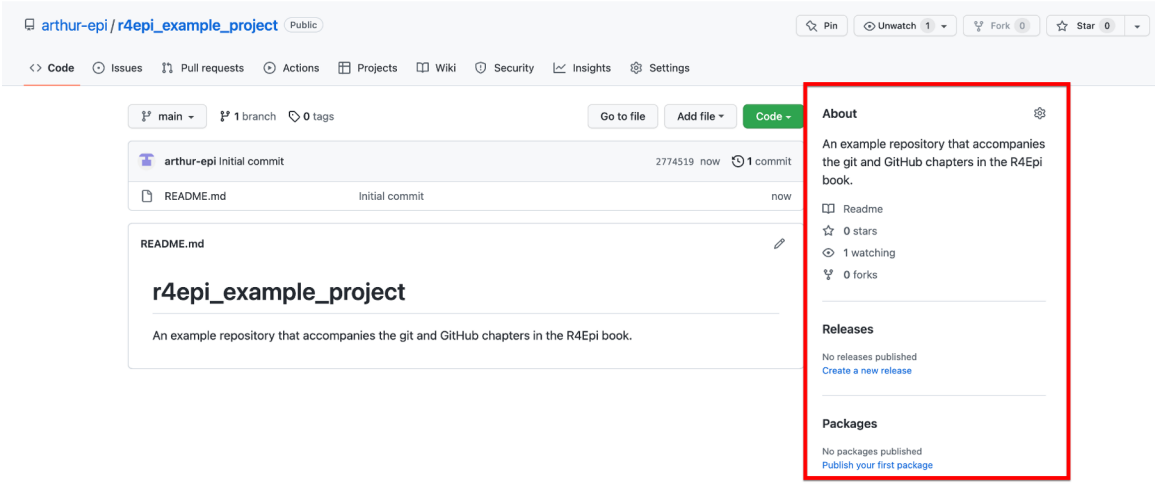
- **Add a README file**: A **README** file has a special place in GitHub. Under the hood, it is just a markdown file. No different than the Quarto files we learned about in the chapter on Quarto files (. However, naming it `README` gives it a special status. When we include a README file in our repository, 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."[13] 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.



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.

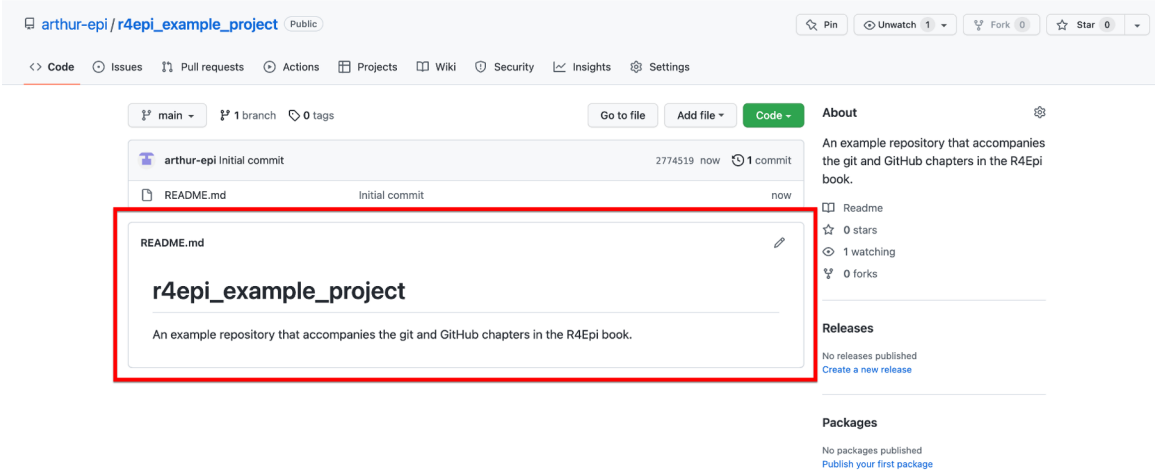


To the right of files and folders section of the homepage is the `About` section of the page. This section (outlined in red below) contains the repository's description, tags, and other information that we will ignore for now.



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, figured, 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**.

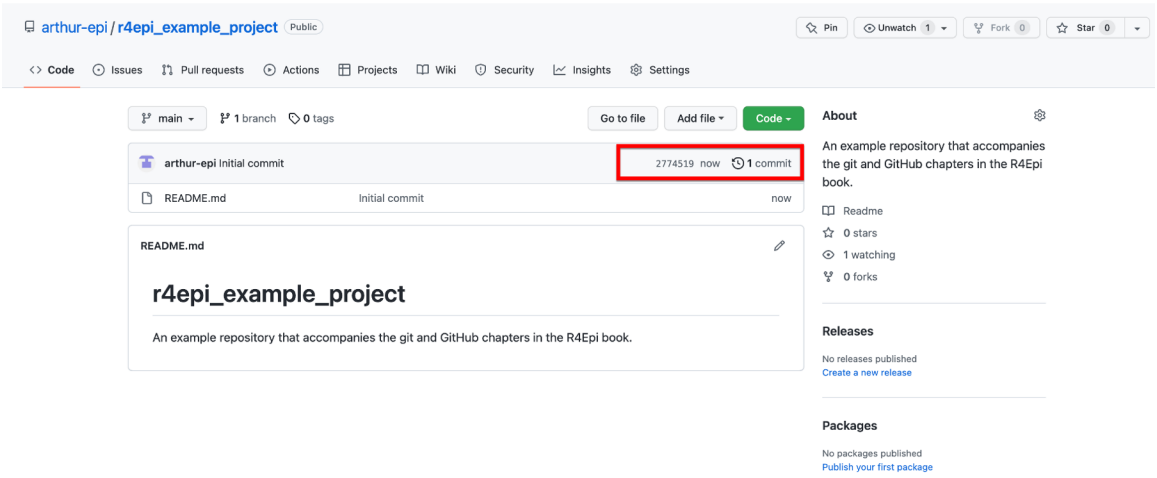
2. In the process, he will **preserve** his files, and by extension, his project.

3. Doing so will help to make his research more **reproducible**.

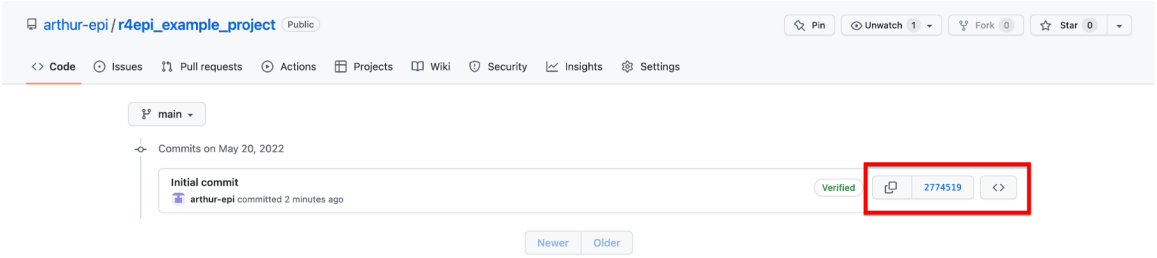
4. And make it easier for him to **collaborate** with others – including his future self.

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).



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.



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.

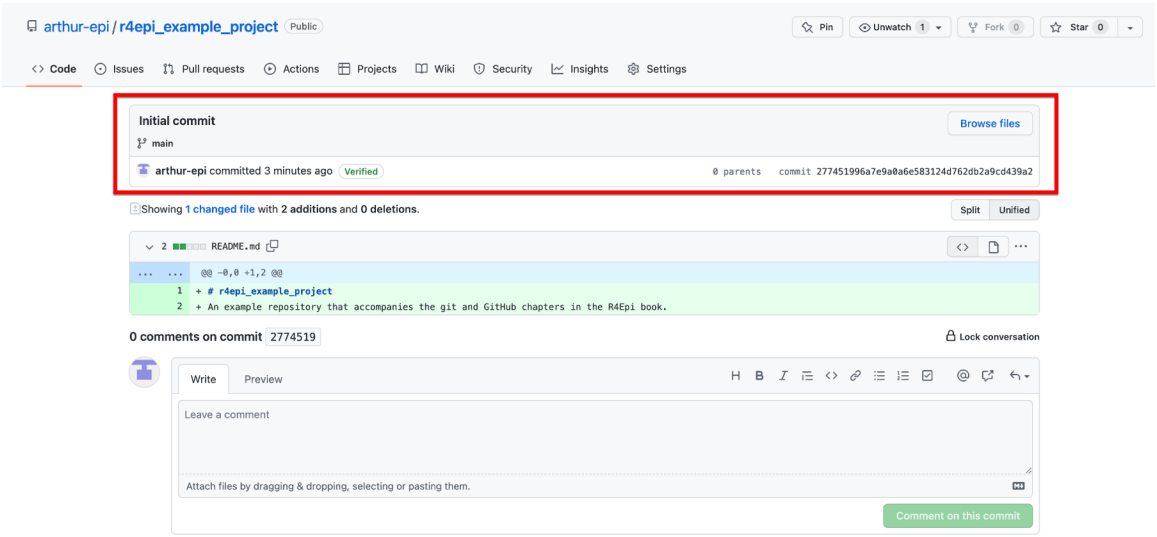
2. As noted above, the middle button is labeled with the first 7 characters of this commit's ID - `274519`. Clicking on it will take us to a new screen with the details of what this commit does to the files in the repository (i.e., additions, edits, and deletions). Arthur will click it so we take a look 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.

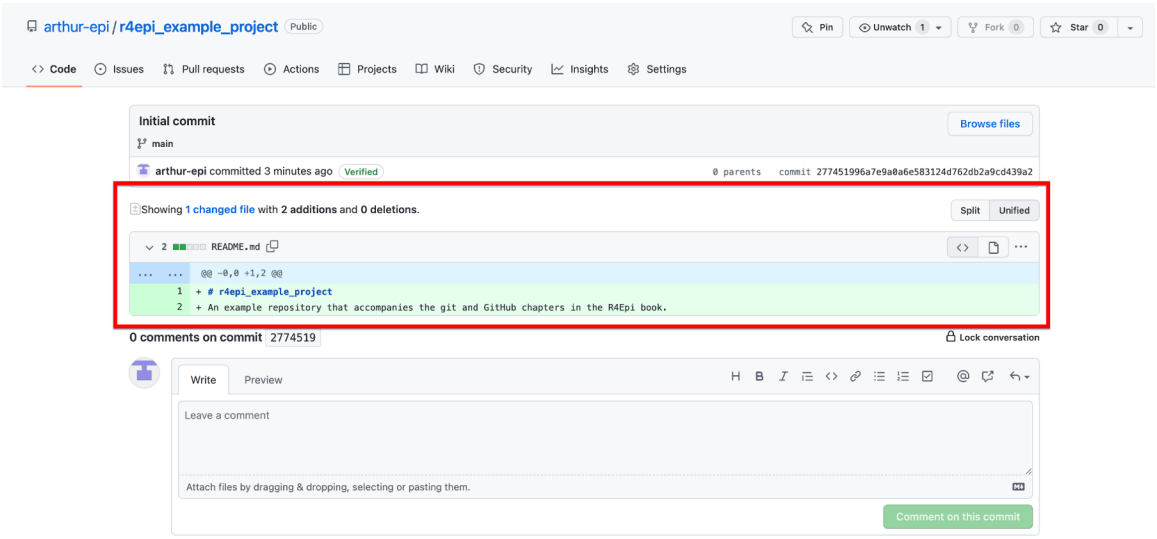
Now, Arthur will click the middle button labeled with the short version of the commit ID.

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

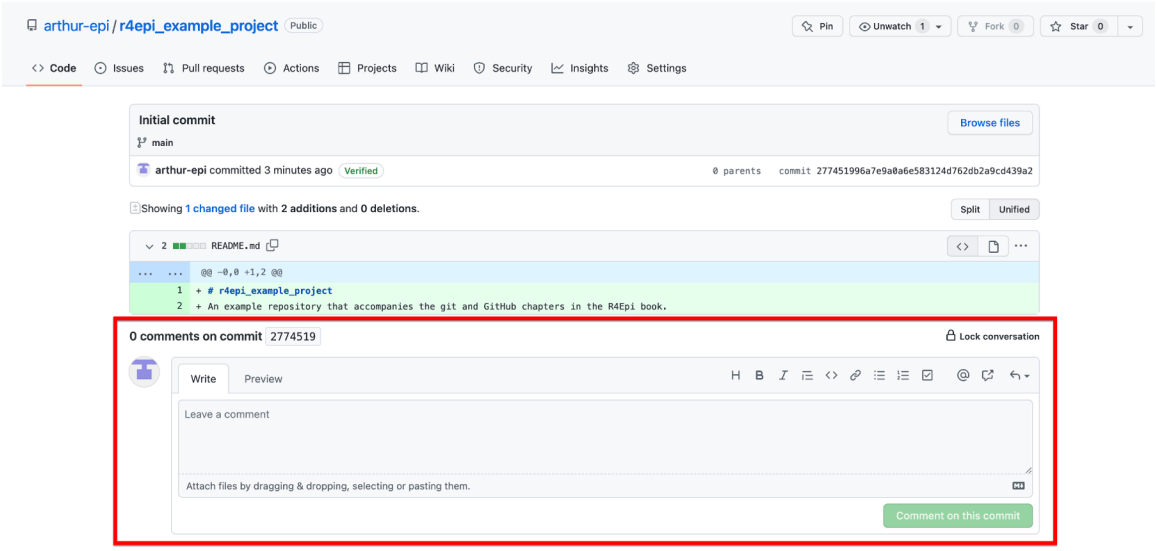
tells us which branch we are operating on. Branches are a more advanced topic that we will discuss later. Currently, our repository only has one branch – the default `main` branch – and the symbol followed by the word "main" is telling us that this commit is on the main branch. To the far right of this 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.



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.

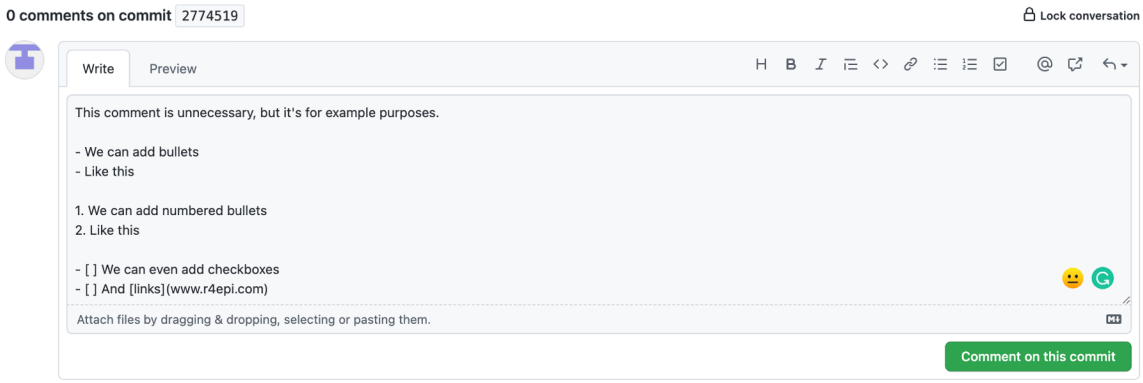


The final section of the commit details page shows us any existing comments that Arthur, or others, made about this commit. It also allows us, or others to create a new comment, using the text box.

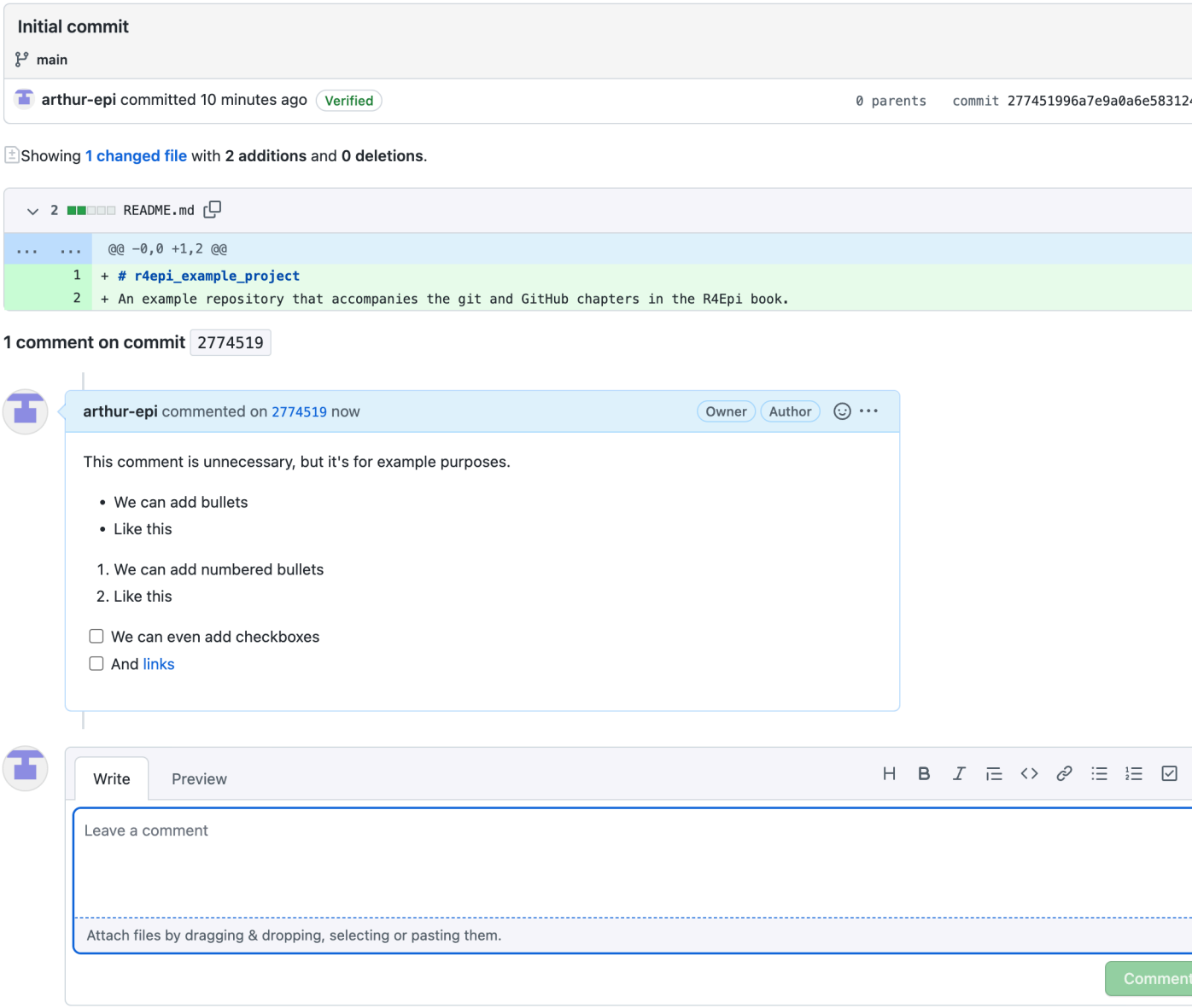


In the screenshot below, we can see an example comment. Note all the cool things features GitHub comments allow us to use. We can format the text, add bullets, add links, and even

add clickable checkboxes.



Finally, clicking the green `Comment on this commit` button adds our comment to the commit details page.



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**
>
> 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).

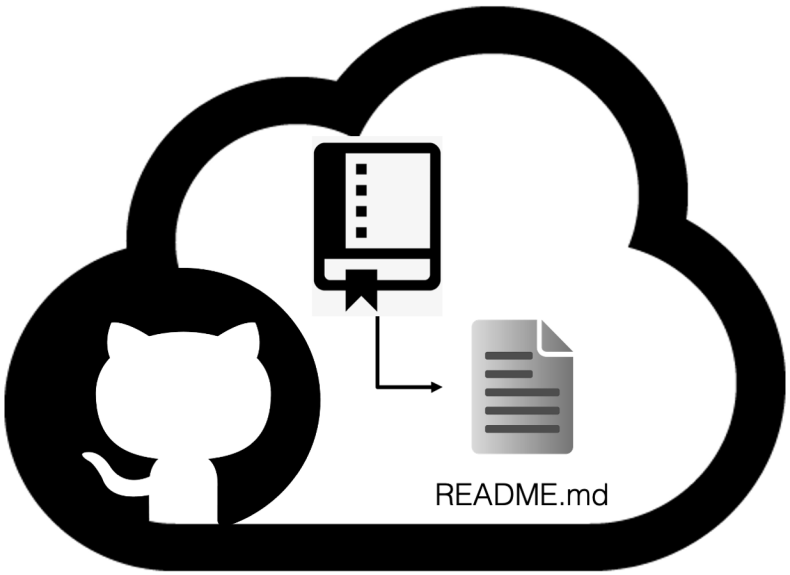


Figure 39.3: GitHub Cloud.

So, how does he get the repository from the GitHub cloud to his computer so that he can start making changes to it?

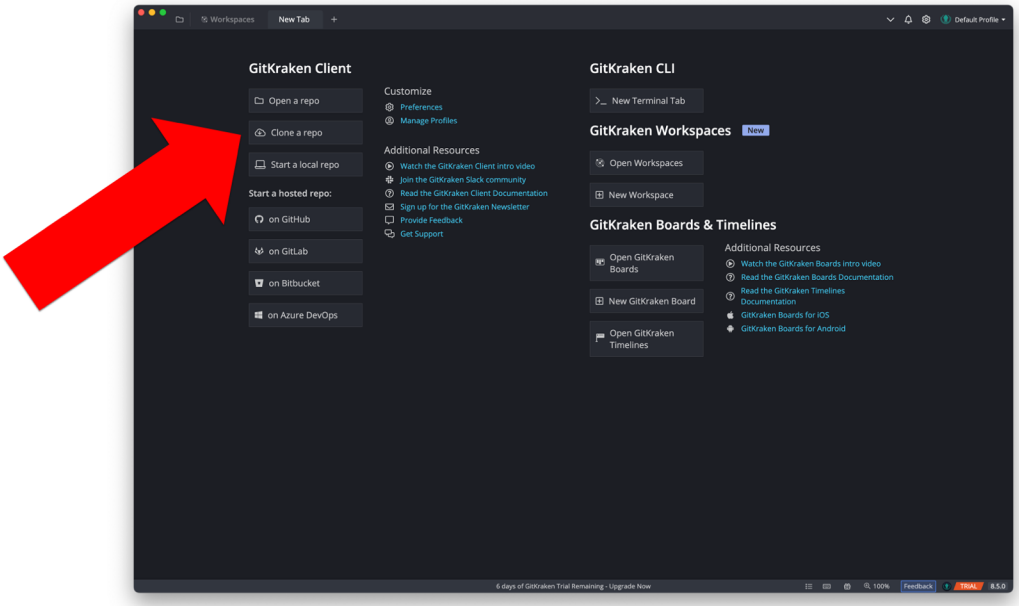
He will clone the repository to his computer. Don't get thrown off by the funny name. You can simply think "make a copy of" whenever you see the word "clone" for now. So, he will "make a copy of" the repository on his computer. However, cloning the repository actually does two very useful things at once:

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.

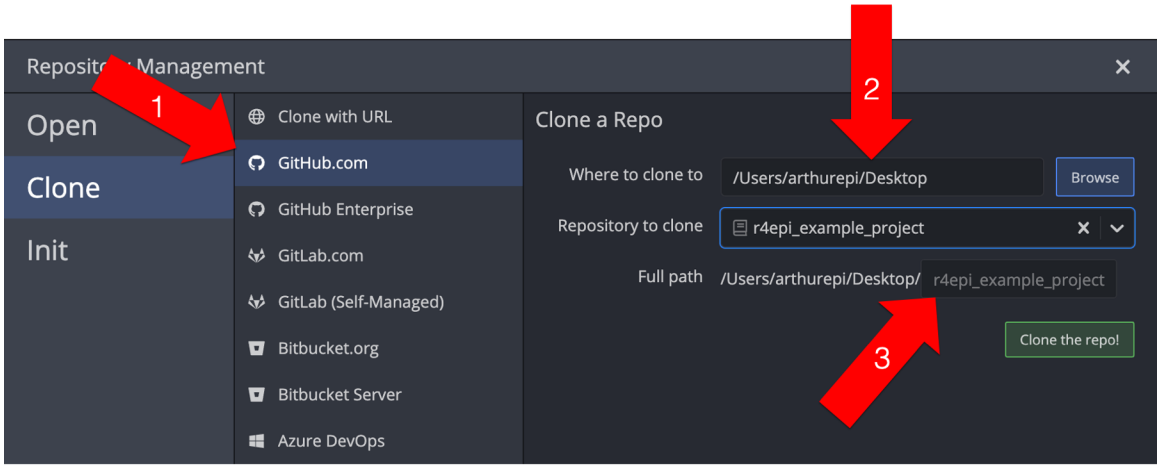


When the Repository Management dialogue box opens, he will need to make 3 changes.

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.

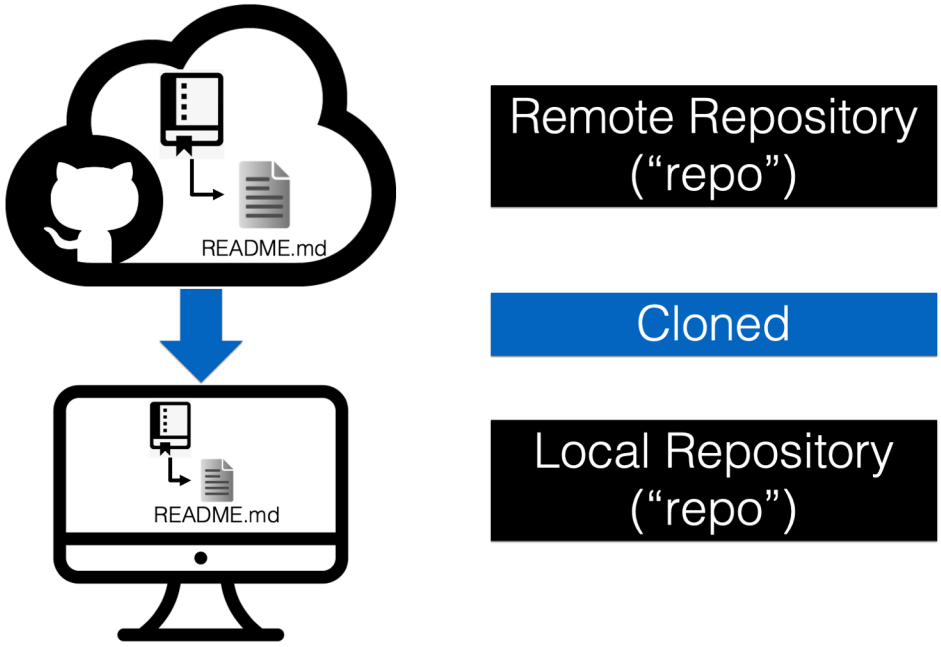
2. Set the path where he wants the repository to be cloned to. Remember, the repository is a just a folder with some files in it. When we clone the repository to our computer, those files and folders will live on our computer somewhere. We need to tell GitKraken where we want them to live. In the screenshot 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.



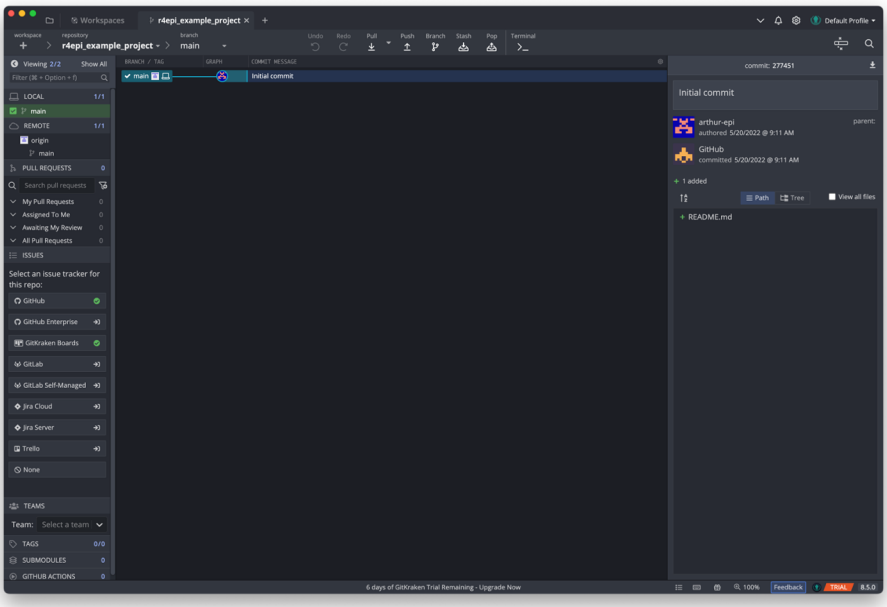
As we discussed above, Arthur's repository already existed on the GitHub cloud see Figure 39.3.

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).

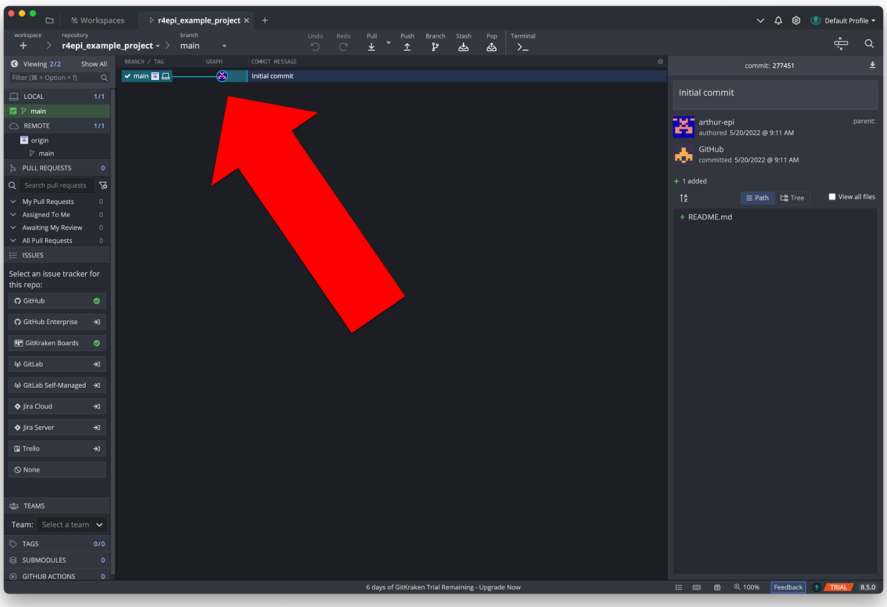
Second, he should be able to open a tab in GitKraken with all the versioning information about his repository.



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.

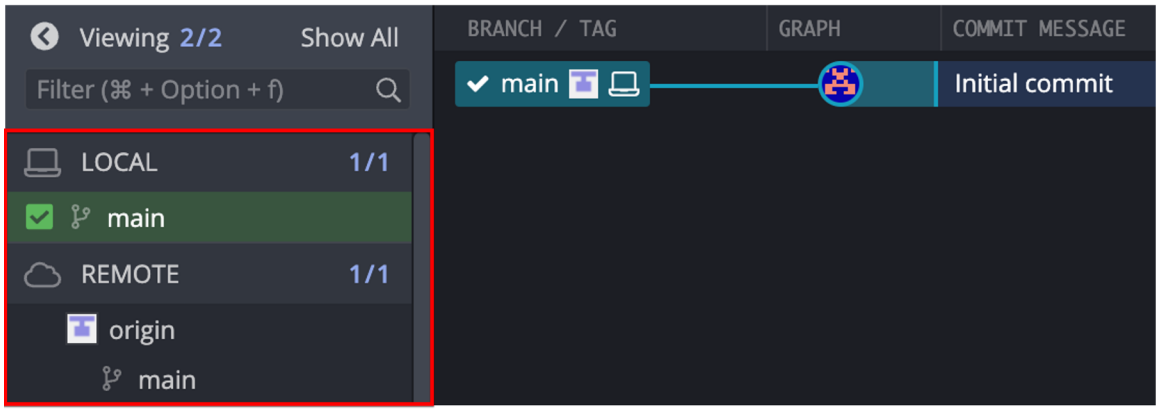
https://www.youtube.com/embed/RiAeNSFjjLc

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.

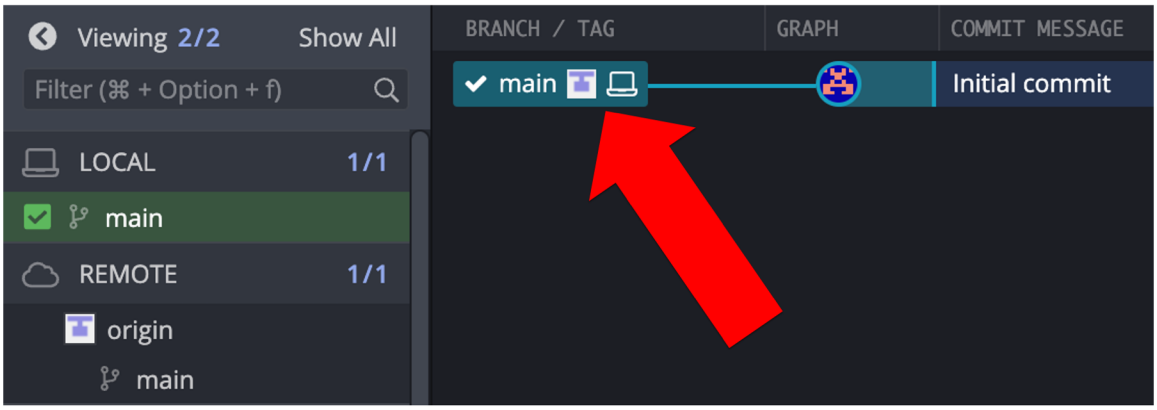


If we zoom in on the upper left corner of the left sidebar menu (outlined in red below), we can see that GitKraken is aware of two different places where the repository lives. First, it tells us that Arthur has a local repository on his computer with one branch – the main branch. Next, it tells us that 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.

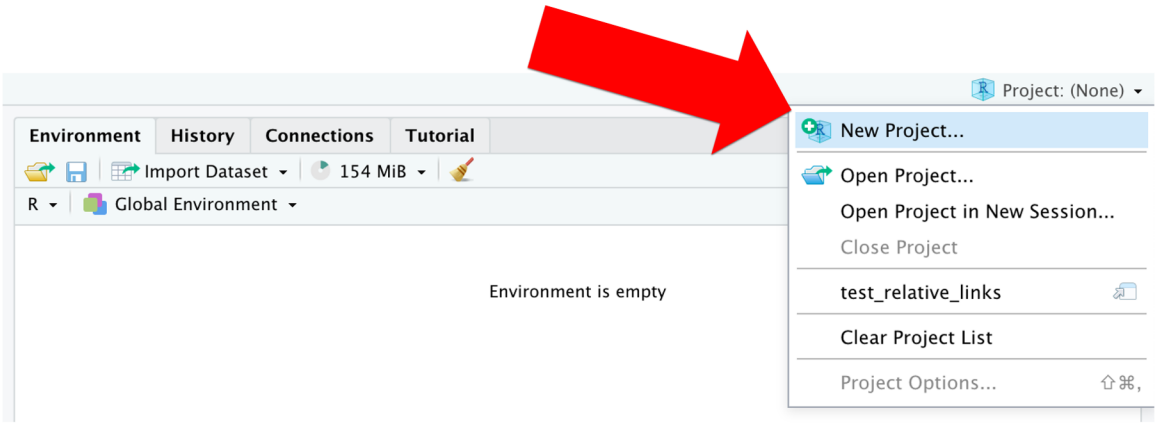


Another useful thing we can see in the current view, is that the local repository and the remote repository on GitHub are in sync. Meaning, the files and folders in the repository on Arthur's computer are identical to the files and folders in the repository on the GitHub cloud. We know this because the 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.

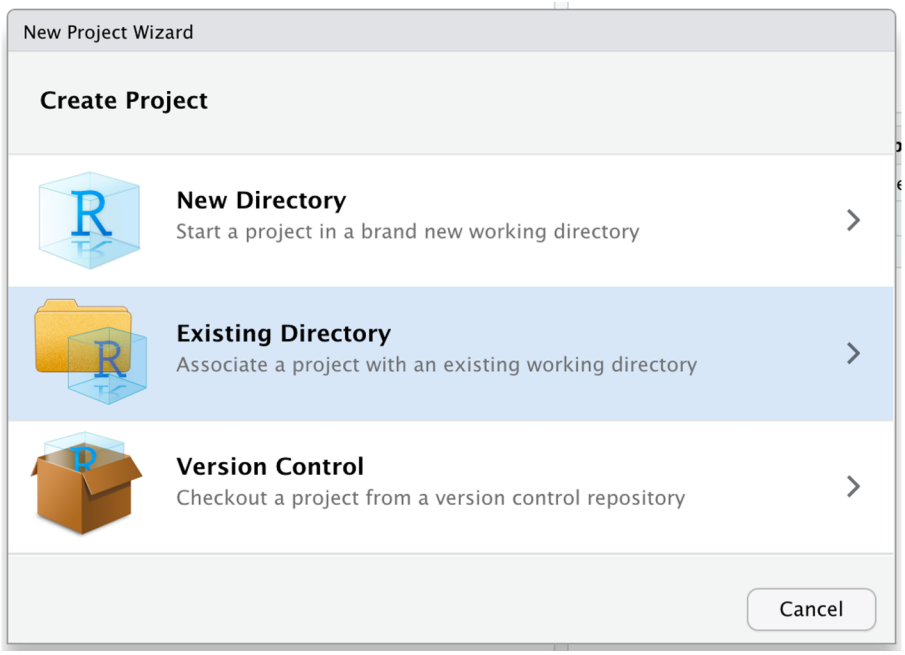


Step 3: Add an R project file to the repository

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.

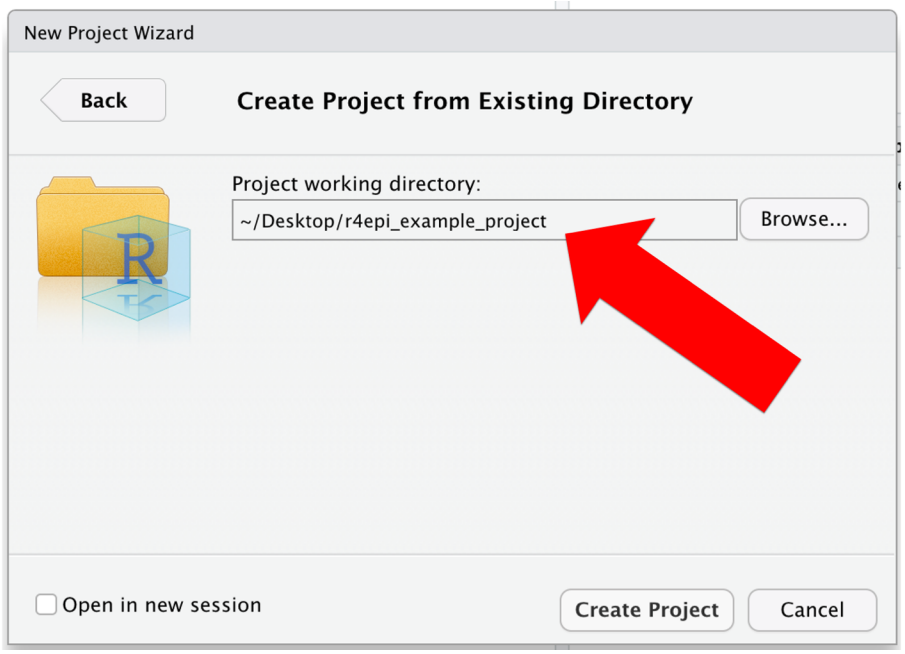


That will open the new project dialogue box. This time, he will click the `Existing Directory` option instead of clicking the `New Directory` option. Why? Because the directory (i.e., folder) he wants to contain his R project already exists on his computer. Arthur cloned it to his desktop in [step 2][Step 2: Clone the repository] above.



All Arthur has to do now, is tell RStudio where to find the `r4epi_example_project` directory

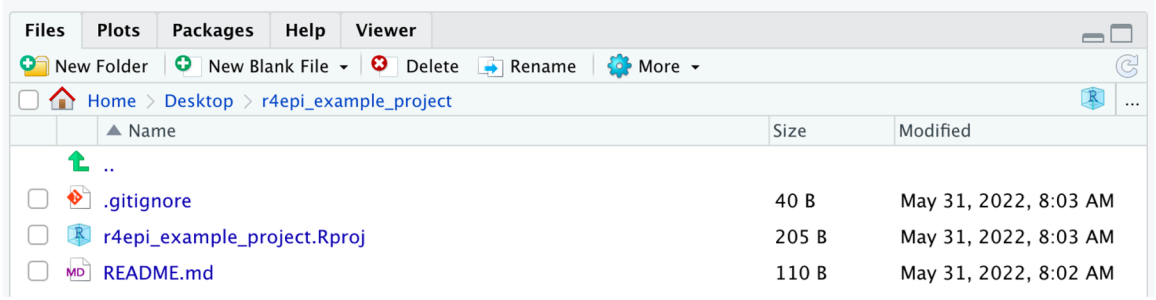
on his computer using the `Browse...` button. In this case, on his desktop. Finally, he will click the `Create Project` button.



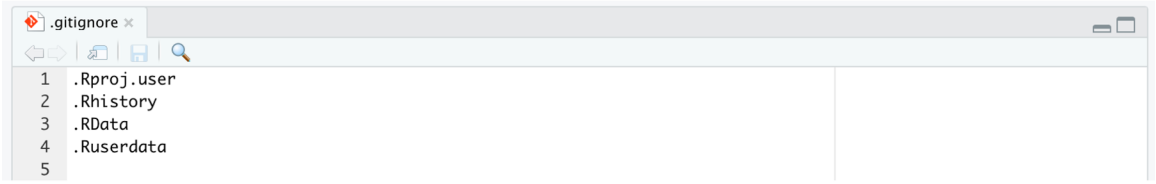
Step 4: Update and commit gitignore

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.

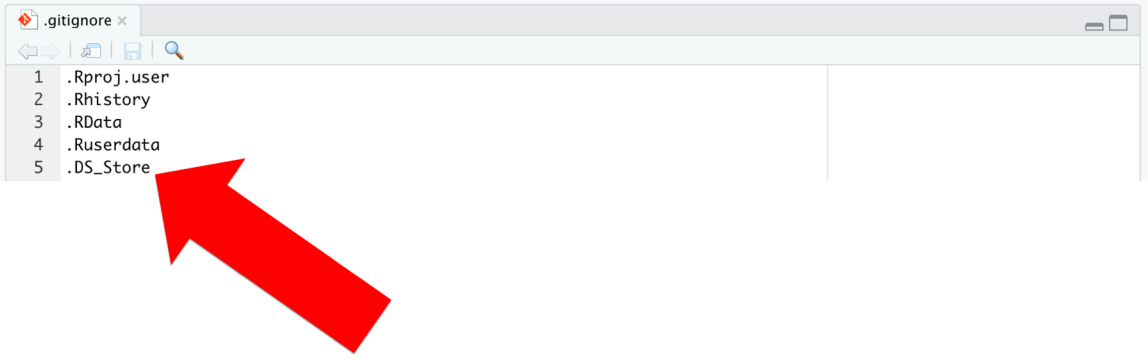


Currently, there are four files on the `.gitignore` list. These files were added automatically by RStudio to try to help him out. Tracking versions of these files typically isn't useful. Because these files are on the `.gitignore` list, git and GitHub won't even notice if Arthur creates, edits, or deletes 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`.



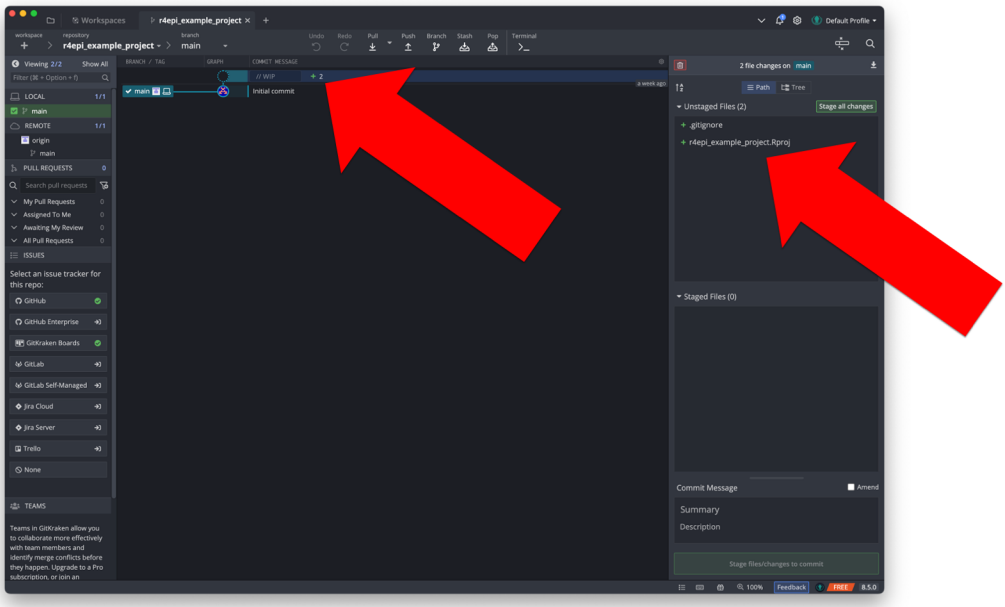
Typically, the next thing we would do after creating our repository is to start creating and adding the files we need to complete our analyses.

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.

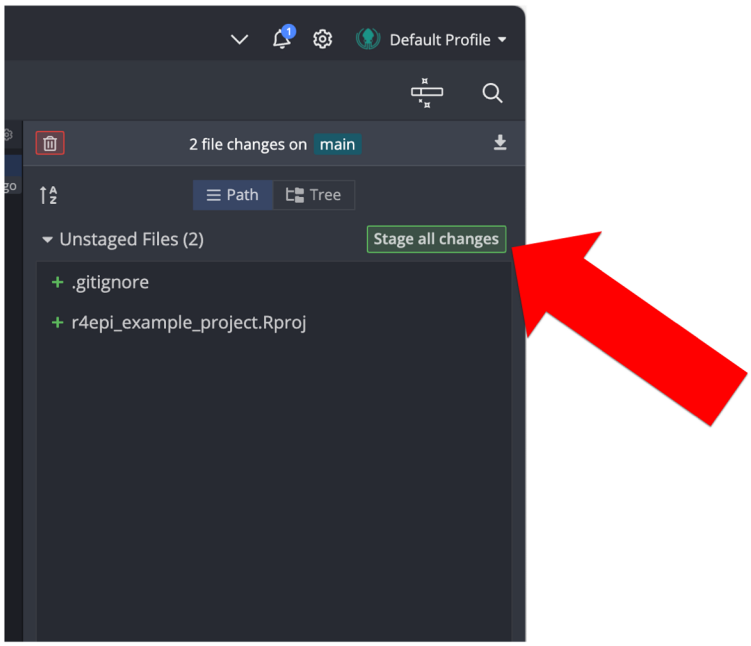
First, the repository graph in the middle section of the user interface has now has two rows. The bottom row is still the initial commit, but now there is a row above it that says `// WIP`

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.

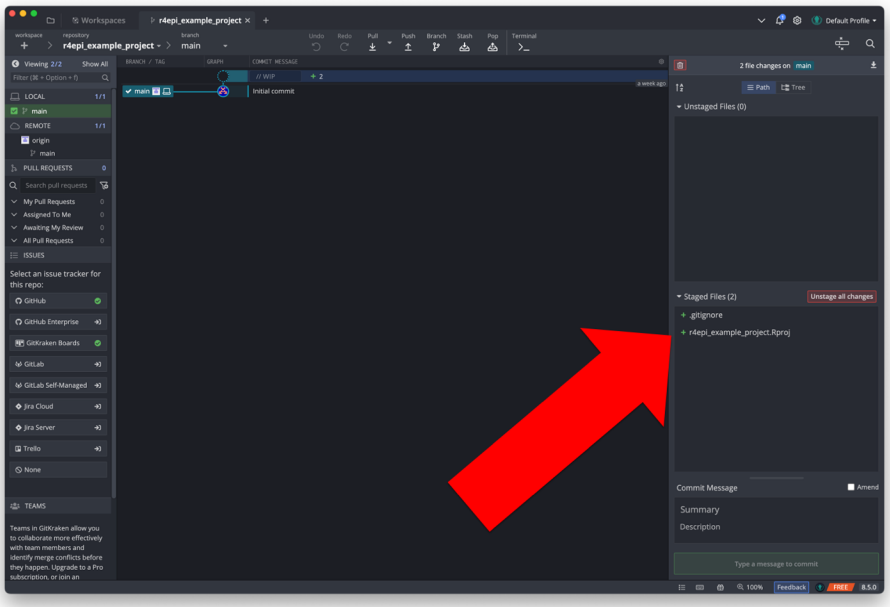
Additionally, the commit panel on the right side of the screen shows that there are two new uncommitted and unstaged files in the directory. They are `.gitignore` and `r4epi_example_project.Rproj`.



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.

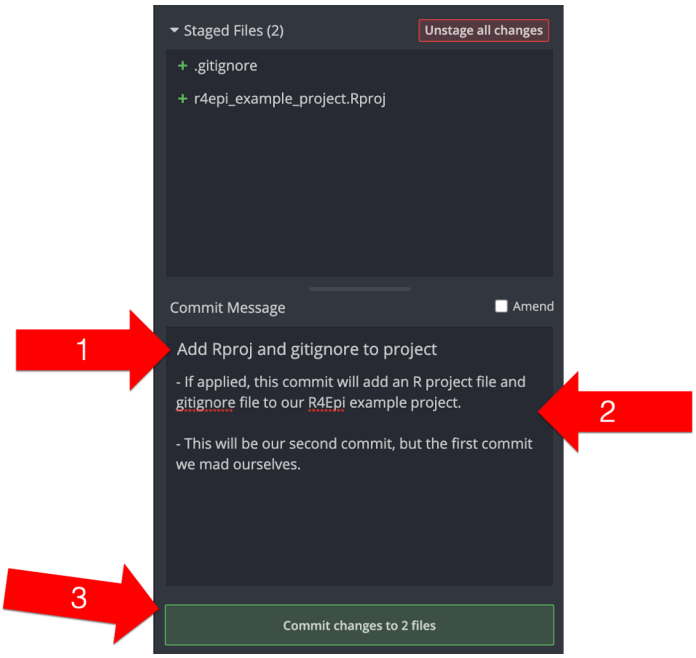


Next, Arthur will write a commit message. Just like there are best practices for writing R code, there are also best practices for writing commit messages. Here is a link to a blog post that we think does a good job of explaining these best practices: https://cbea.ms/git-commit.

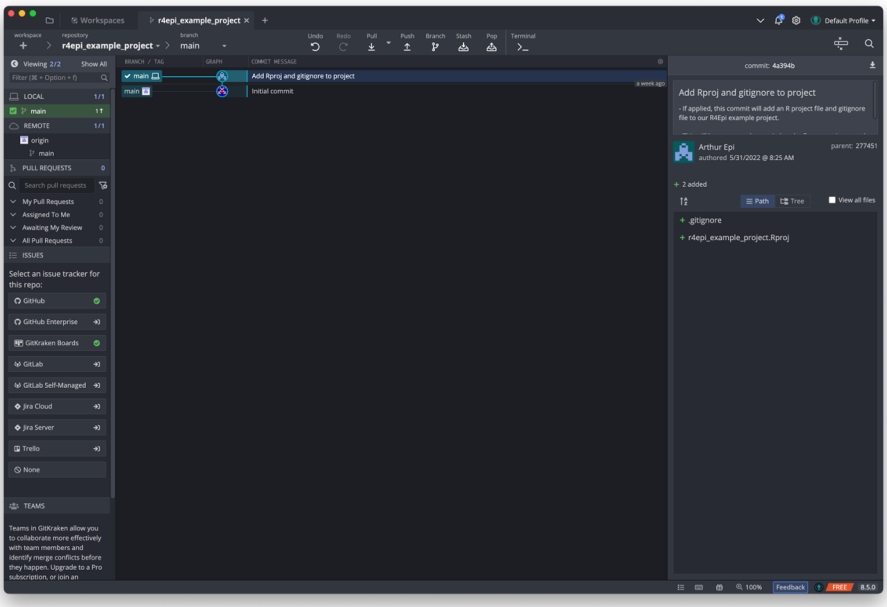
The first line is called the **commit message**. You can think of the commit message as a brief summary of what this commit does to the repository. This message will help Arthur and his collaborators find key commits later in the future. In this context, "brief" means 72 characters or less. GitKraken 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.

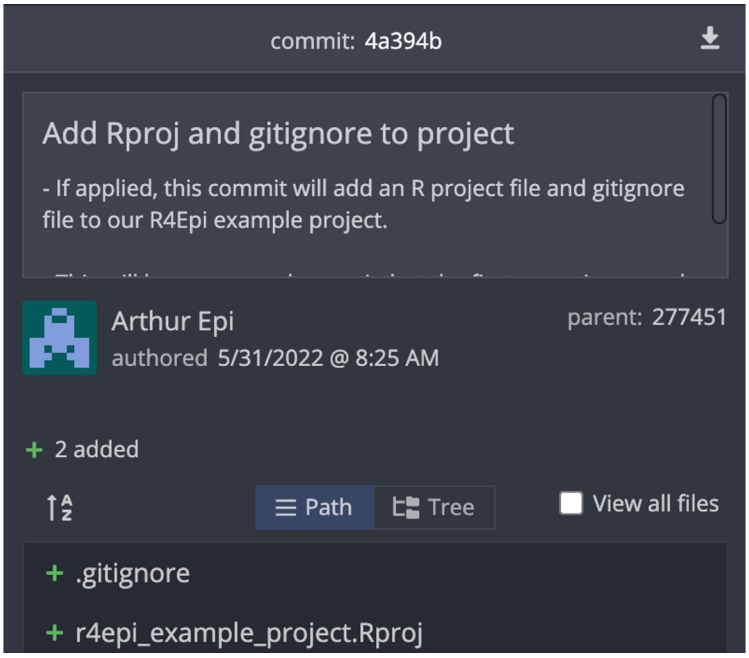


And here is what his GitKraken screen looks like after committing.

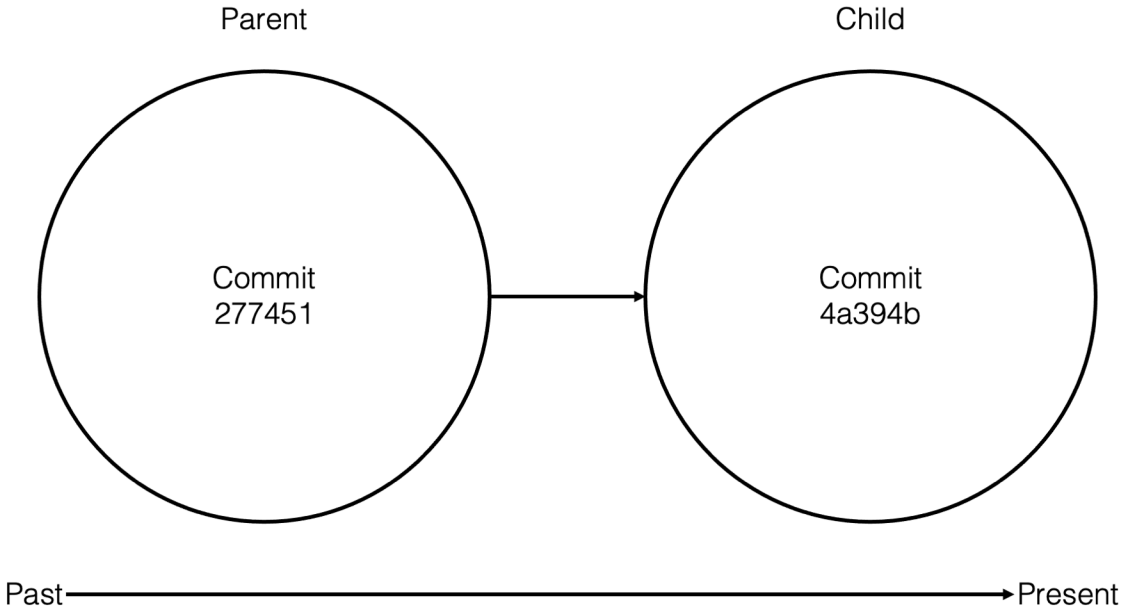


Let's pay special attention to what is being displayed in a couple of different areas. We'll start by zooming in on the commit panel.

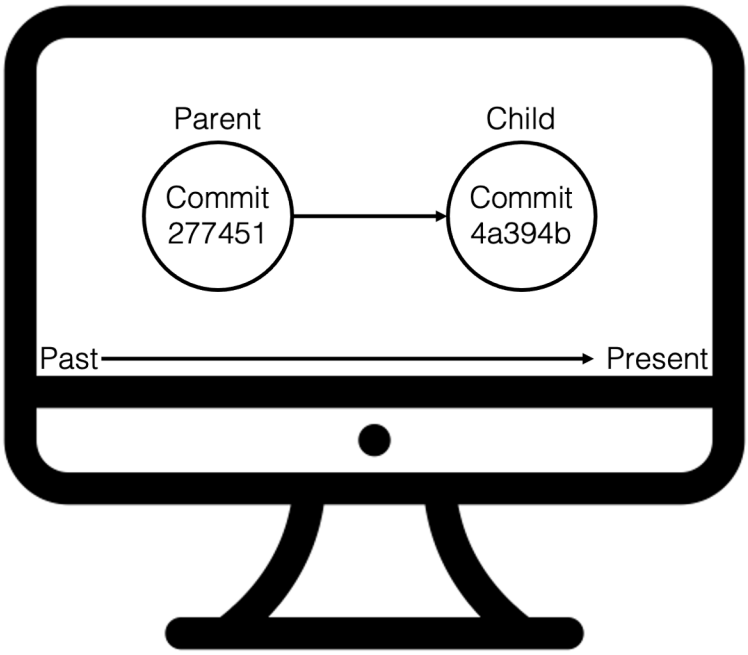
At the top of the commit panel, we can see the short version of the commit ID – `4a394b`. Below that, we can see the commit message and description. Below that, we can see who created the commit and when. This tends to be more useful when we are collaborating with others. To the right of that information, 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`.



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.



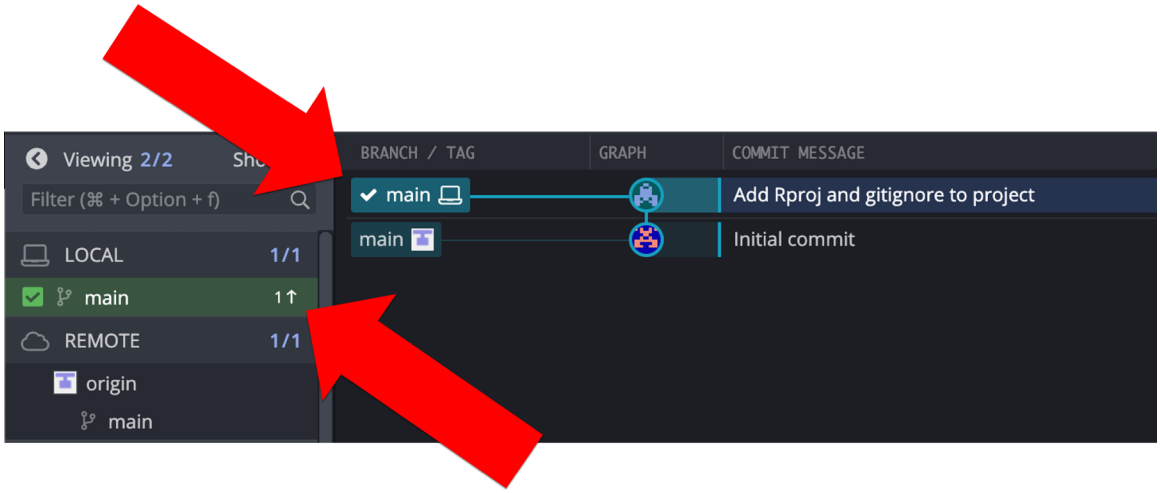
It is also important to point out that Arthur's most recent commit (`4a394b`) **only exists in his local repository**. That is, the repository on his computer. He has not yet shared the commit – or the new files associated with the commit – to the remote repository on GitHub.



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

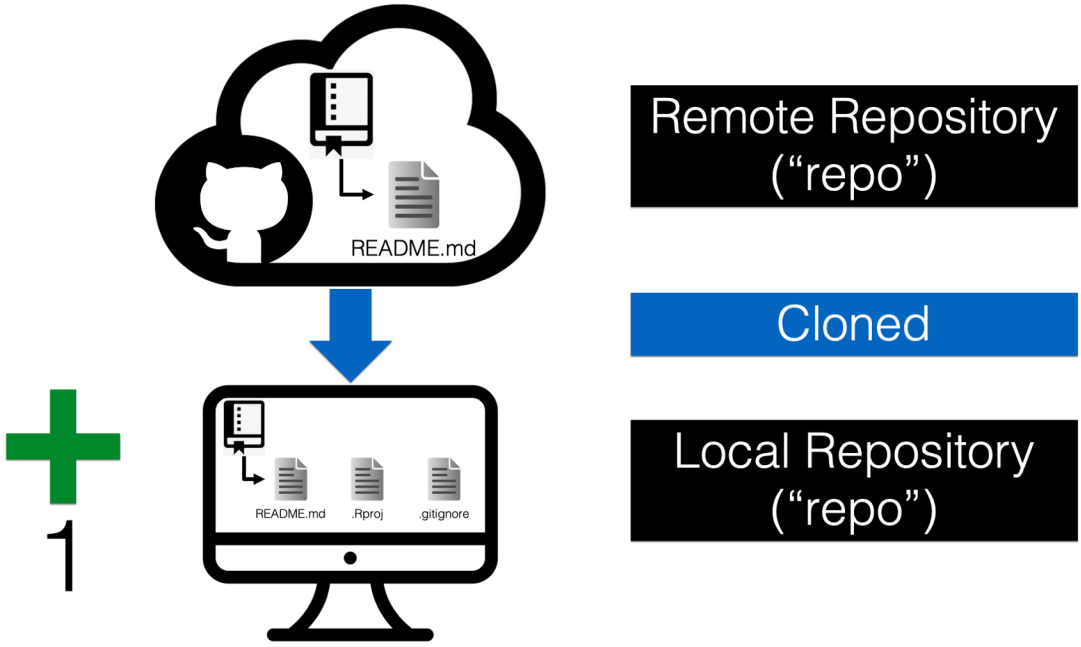
(i.e., the little gray and white icon) are on different rows. Additionally, there is a little `1` next to an up arrow displayed to the left of the main branch of our local repository in the left panel of GitKraken. Both of these indicate that the most recent commits contained in each repository are different. 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.

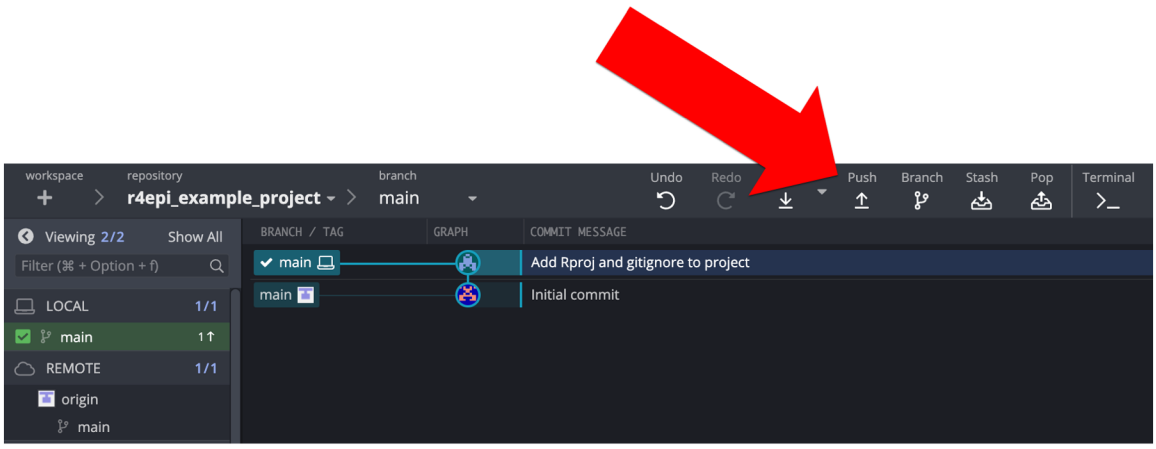
Let's quickly pause and recap what Arthur has done so far.



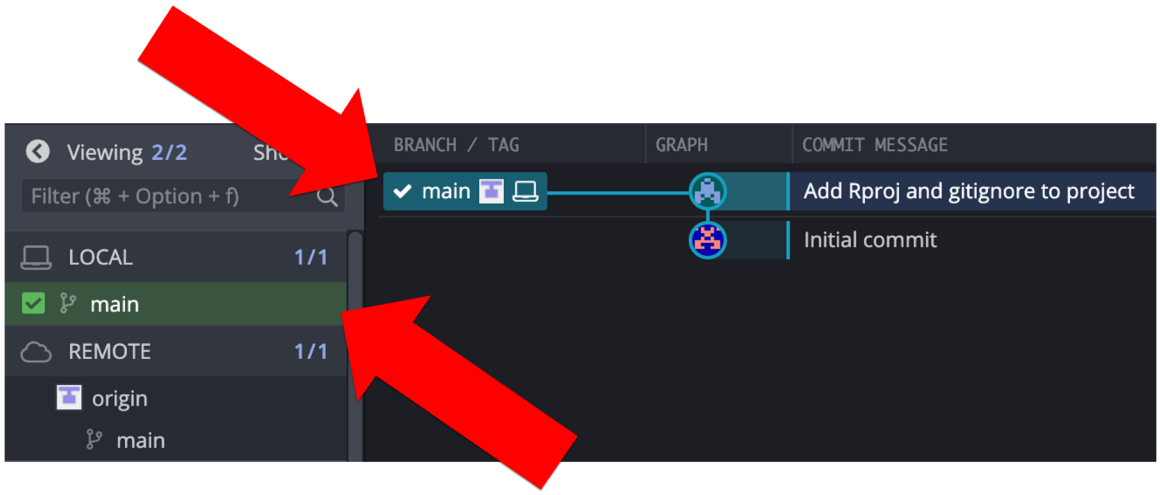
First, Arthur created a repository on GitHub. It was a remote repository because he accesses it over the internet. Then, he cloned (i.e., made a copy of) the remote repository to his computer. This copy is referred to as a local repository. Next, Arthur made some changes to the repository locally and 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).

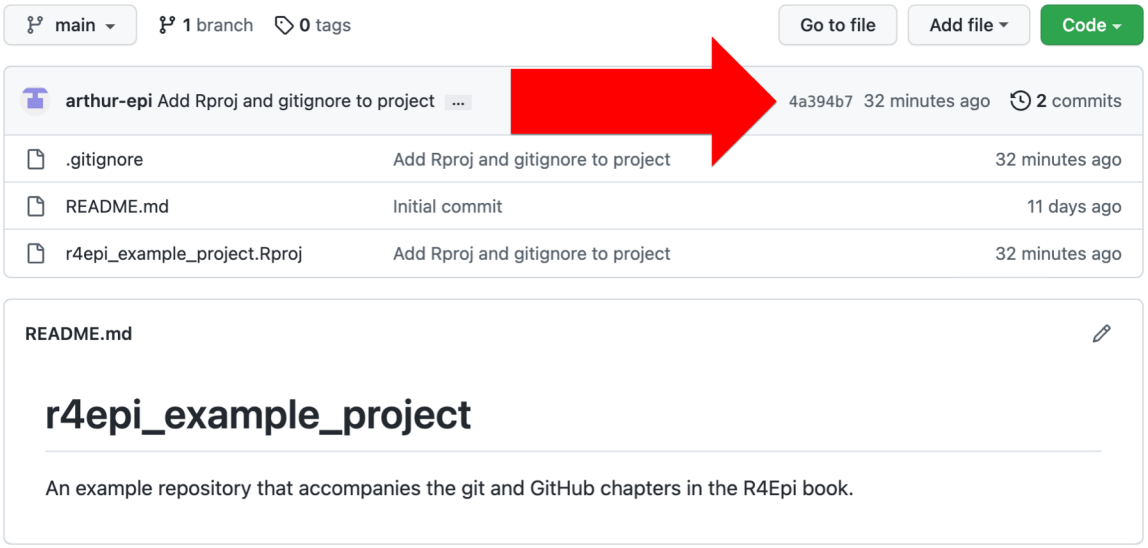


After doing so, we will once again see some changes. What changes do you notice in the screenshot below?

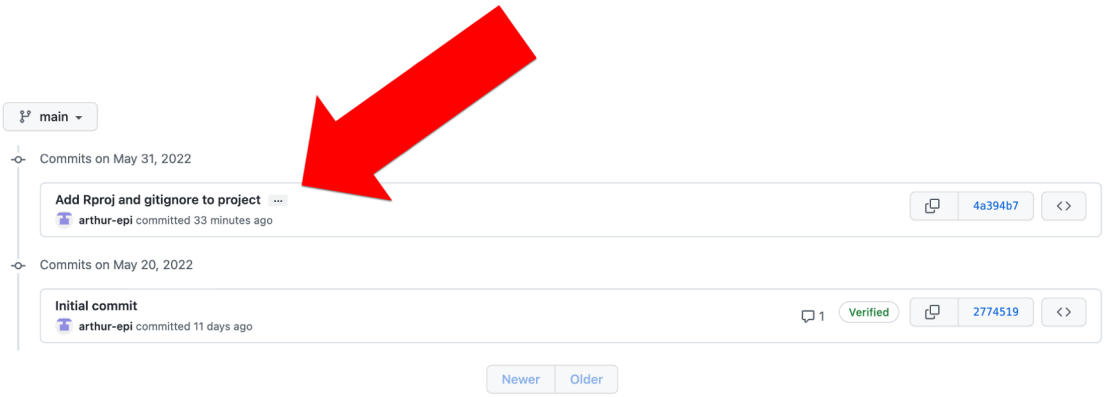


In the repository graph, the local repository (i.e., the little laptop icon) and the remote repository (i.e., the little gray and white icon) are back on the same row. Additionally, the little `1` next to an up arrow is no longer displayed in the left panel. Both of these changes indicate that the most recent commits contained in each repository are the same.

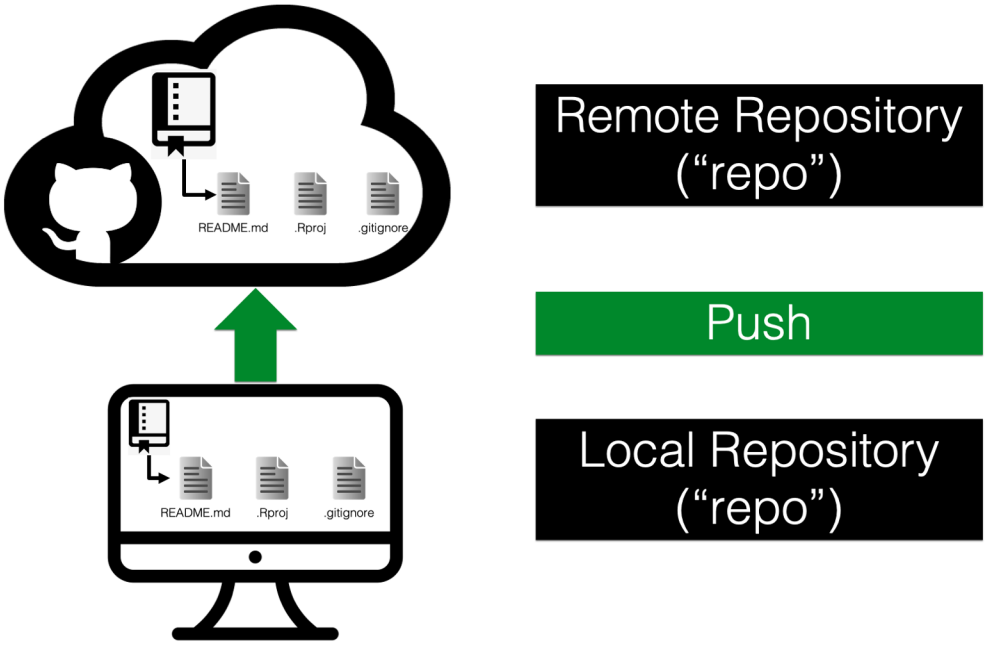
And if Arthur once again checks GitHub...



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.



In the commit history, he can now see commit `4a394b7`. Let's take another pause here and recap.

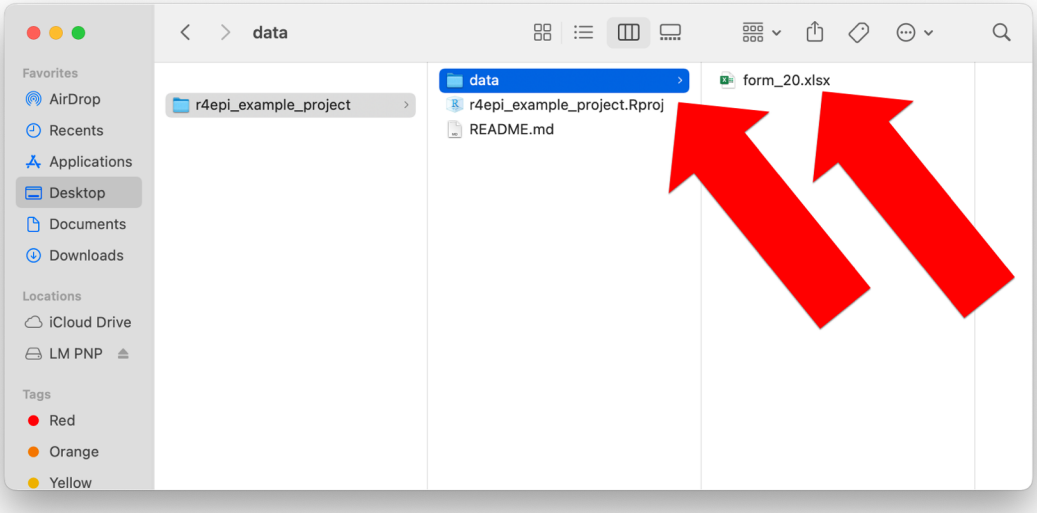


First, Arthur created a repository on GitHub. Then, he **cloned** the remote (i.e., GitHub) repository to his computer. Next, Arthur made some changes to the repository locally and **committed** them locally. Finally, he **pushed** the local commit up to GitHub. Now, his GitHub repository and local repository 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.

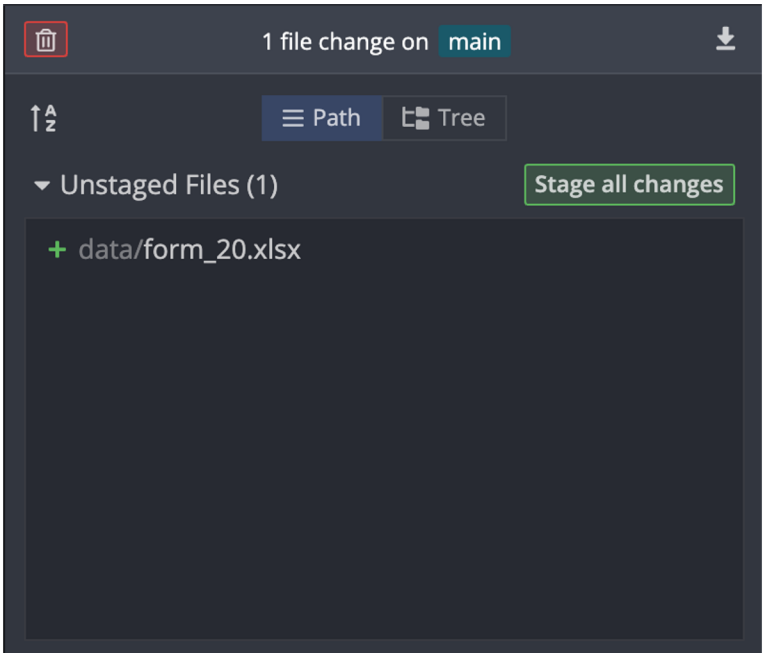
Step 5: Keep adding and committing files

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.

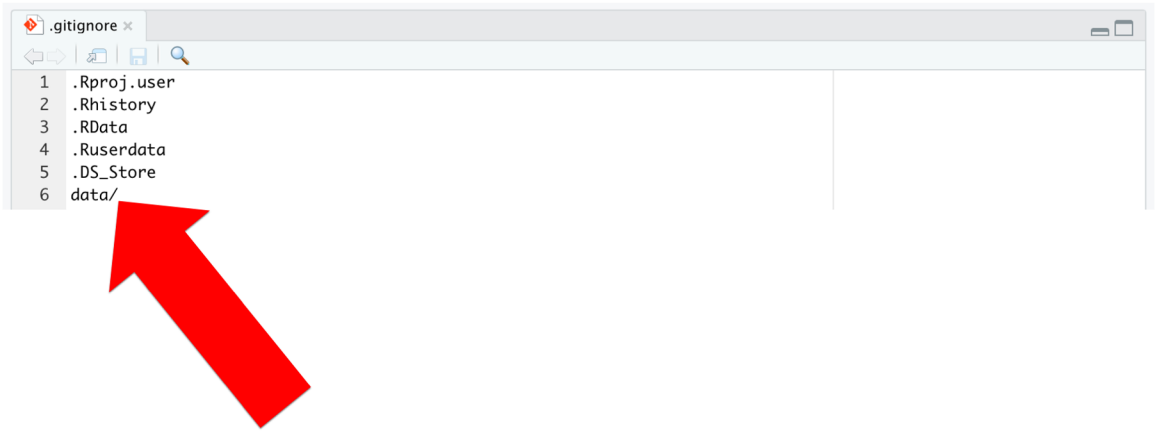


Just like before, GitHub is telling Arthur that he has a new unstaged file in the repository. Stop for a moment and think. What should Arthur do next?

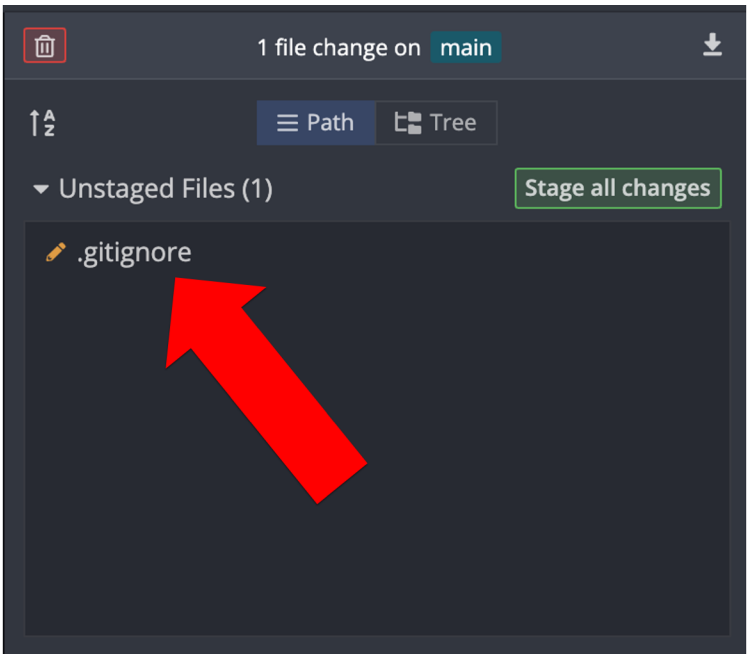
Was your answer, "stage and commit the new file"? If so, slow down and think again. Remember, in general, we don't *ever* want to commit our research data to our GitHub repository. GitHub is not typically considered secure or private. So, how can Arthur keep the data in his local repository so that he 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.

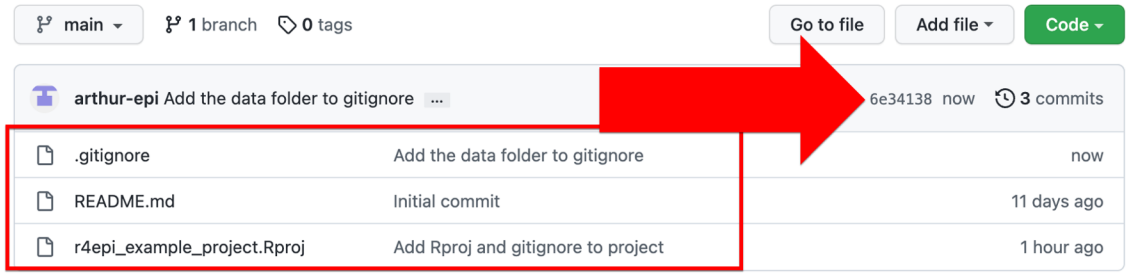


The new file `data/form_20.xlsx` is no longer showing up as an unstaged change. Instead, the only unstaged change showing up is the edited `.gitignore` file. We can tell that the changes to the `.gitignore` file are edits – as opposed to adding the file for the first time – because

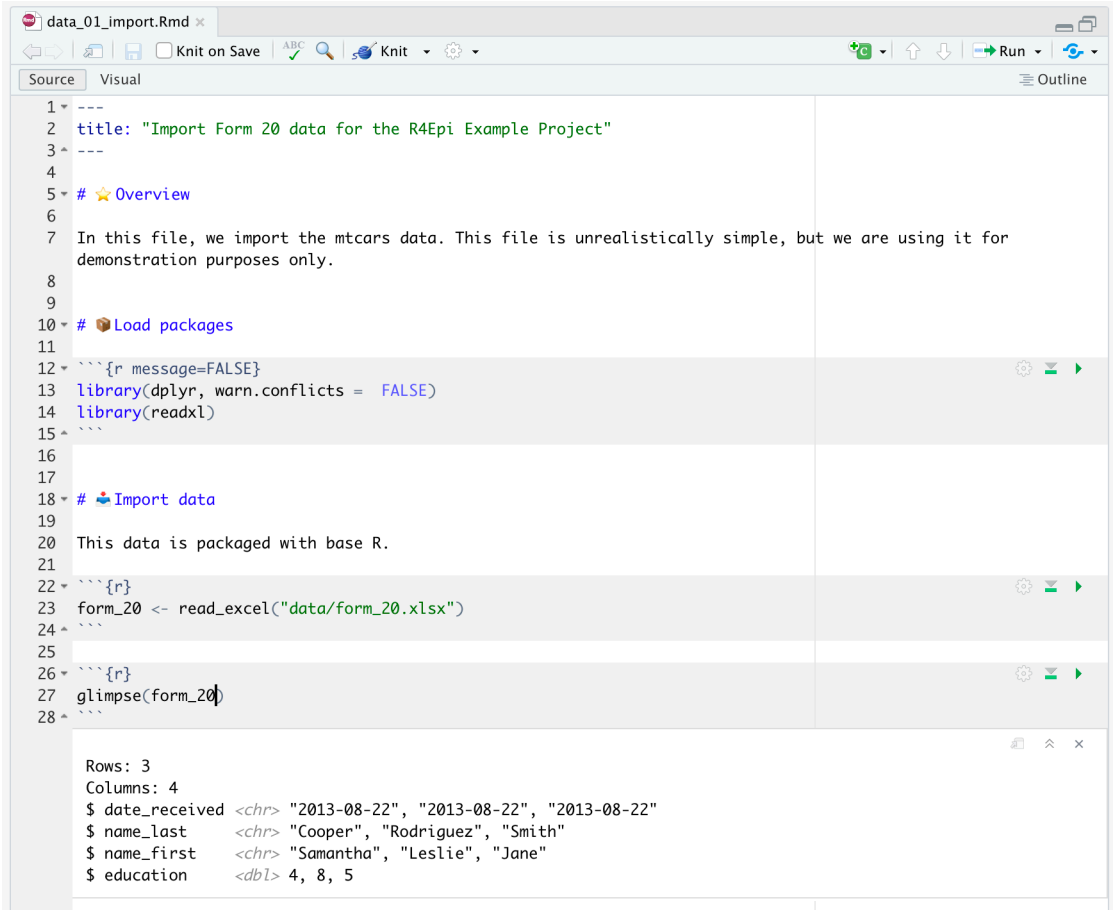
there is a little pencil icon to the left of the file name instead of a little green plus icon. Now what should Arthur do next?

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.



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.



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.

39.6 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."[14] 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."[14] 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!

39.7 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**
>
> 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.

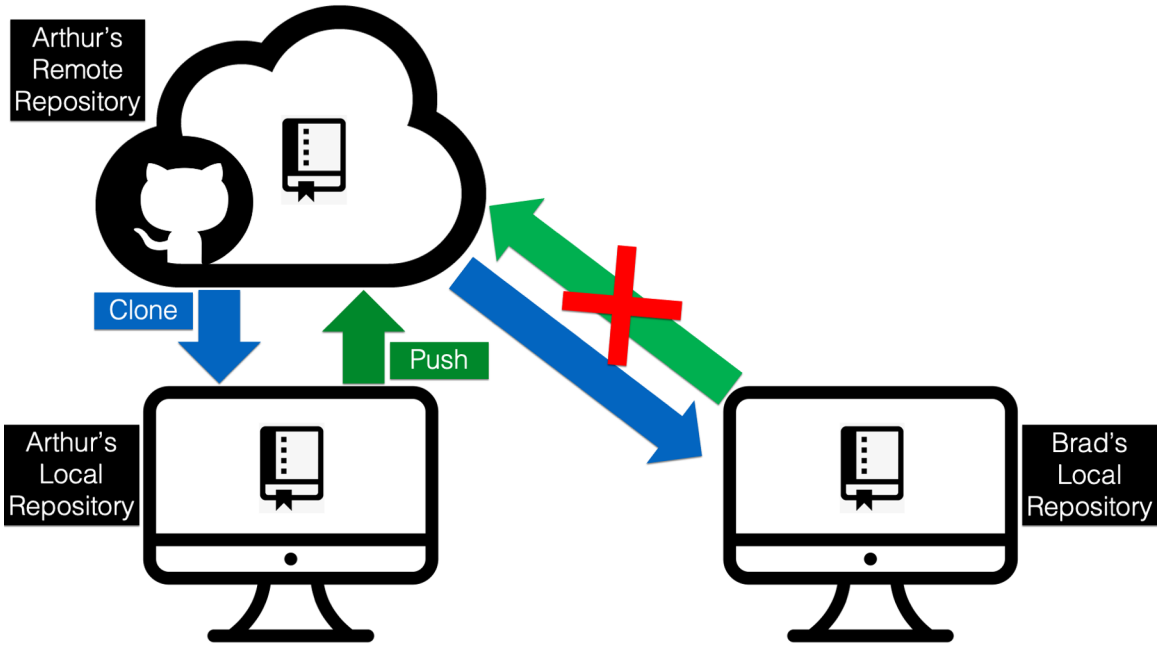
In this example, we're going to work collaboratively with Arthur on the `r4epi_example_project`. Arthur could have just emailed us all of the project files, but sometimes that might be *many* files, some of them may be very large, and he runs the risk of forgetting to send some of them by accident. Further, 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?

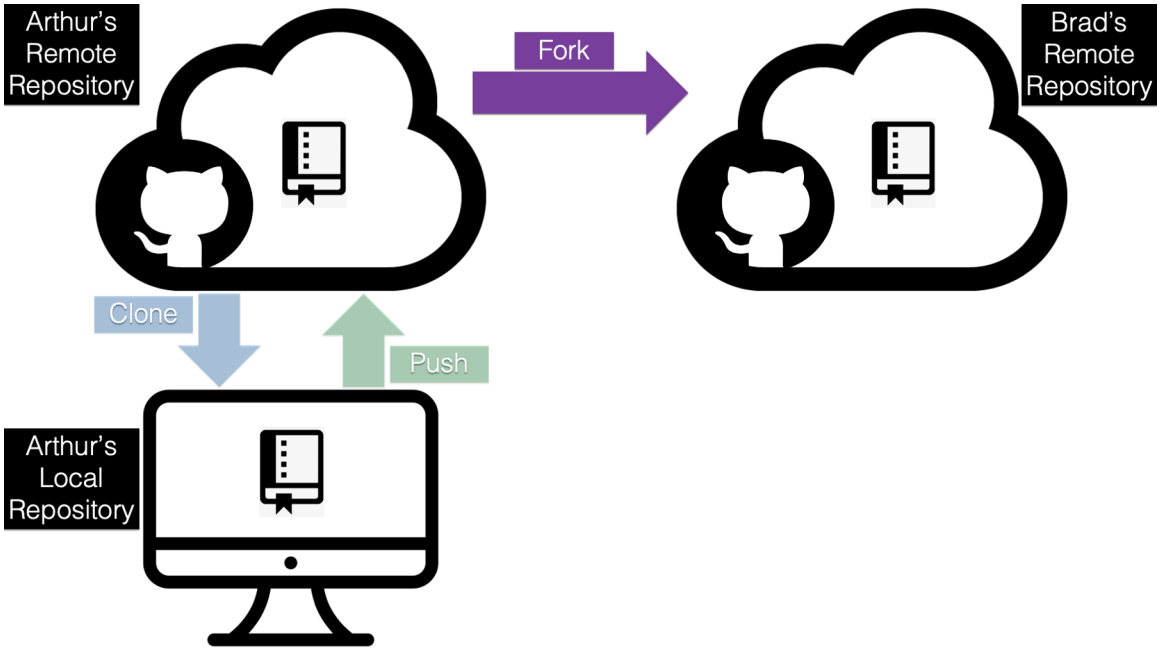
While it is technically possible for us to edit code files directly on GitHub (see [Contributing to R4Epi]), this is typically *only* a good idea for *extremely minor edits* (e.g., a typo in the documentation). Typically, we will want to make a copy of all the code files on our computer so that we can 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?

39.7.1 Forking a repository

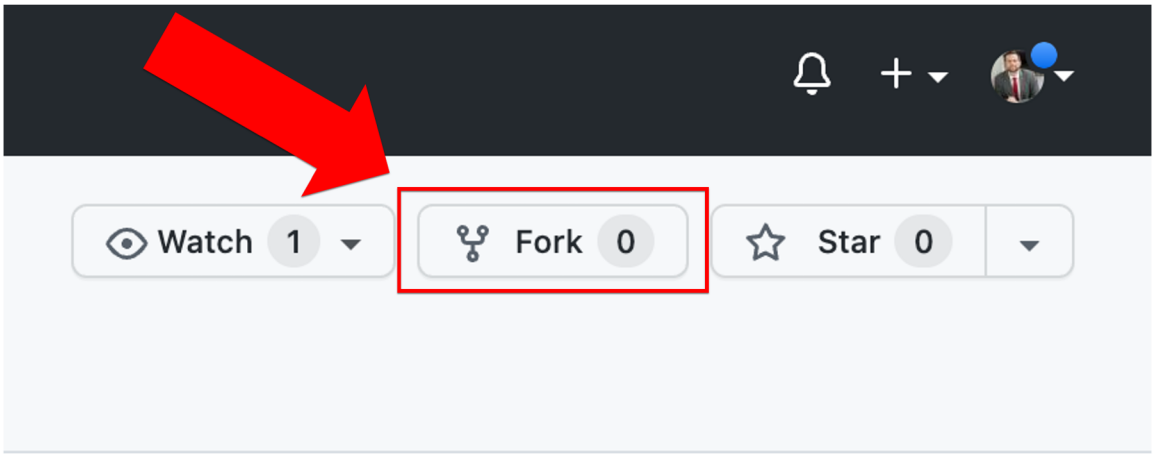
If your answer the question above was, "we **clone** the `r4epi_example_project` repository to our computer" you were close, but that isn't our best option here. While we technically *can* clone public repositories that aren't on our account, we *can't* push any changes to them. And this is a **good 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!



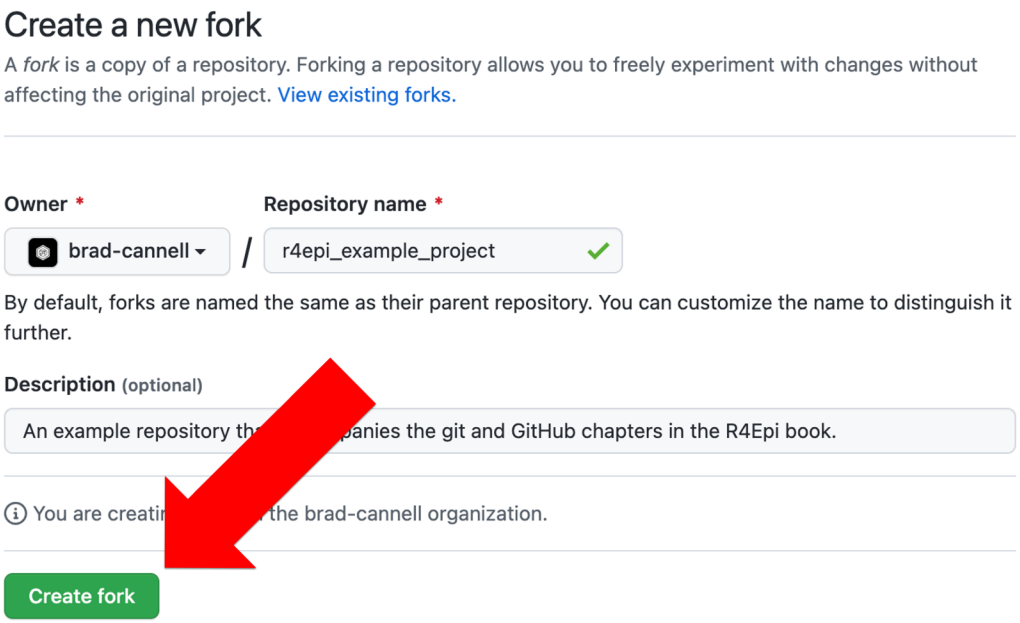
In this case, **forking** the repository is going to be the better option. This is another funny name, but we are once again just talking about making a copy of the repository. However, this time we are copying the repository from the original *GitHub account* (i.e., Arthur's) to our *GitHub account*. 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.



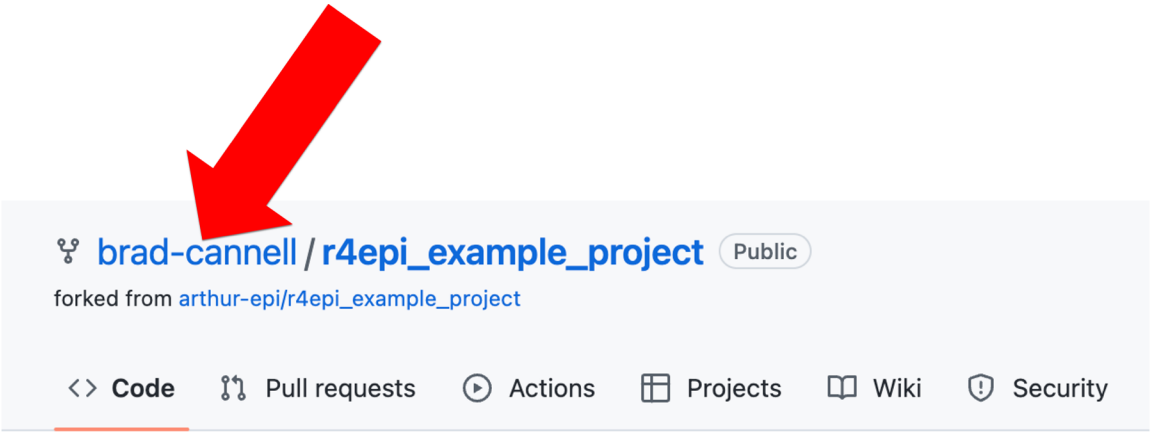
The purple arrow above indicates that we are forking (i.e., making a copy of) the original `r4epi_example_project` repository on Arthur's GitHub account to Brad's GitHub account. And doing so is really easy. All Brad has to do is log in to GitHub and navigate to *Arthur's* `r4epi_example_project` repository 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.



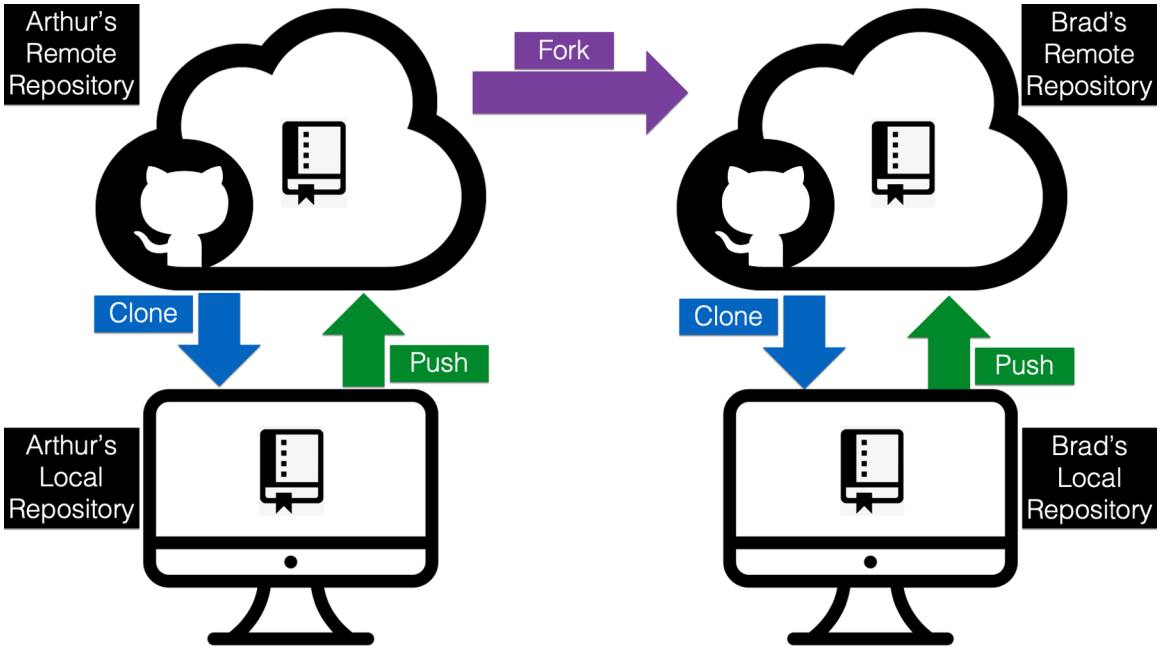
And after a few moments, this will create an entirely new repository on Brad's GitHub account. It will contain an exact copy of the all the files that were on the repository in Arthur's GitHub account, but *Brad* is the owner of *this* repository on his account (shown in the screenshot below).



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.

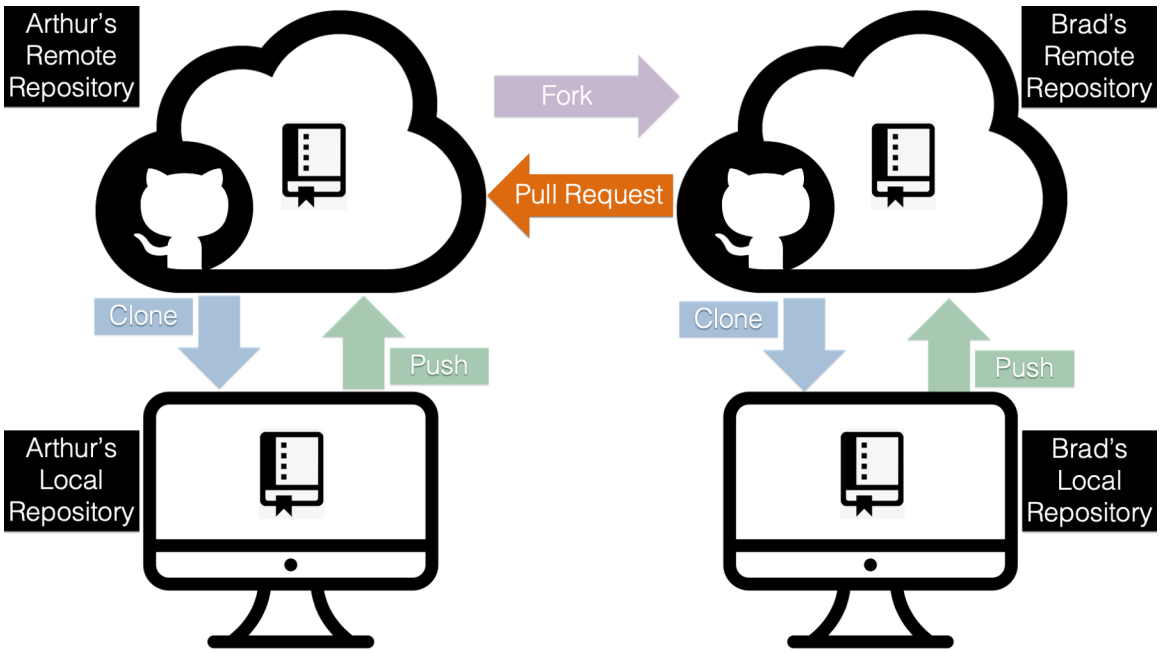
> **Note**
>
> As we've pointed out multiple times in this chapter, we generally do not want to upload research data to GitHub. Why? Because it isn't typically considered private or secure. However, in order for Brad to do work on this project, he will need to access the data somehow. This will require Arthur to share to 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**.



39.7.2 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

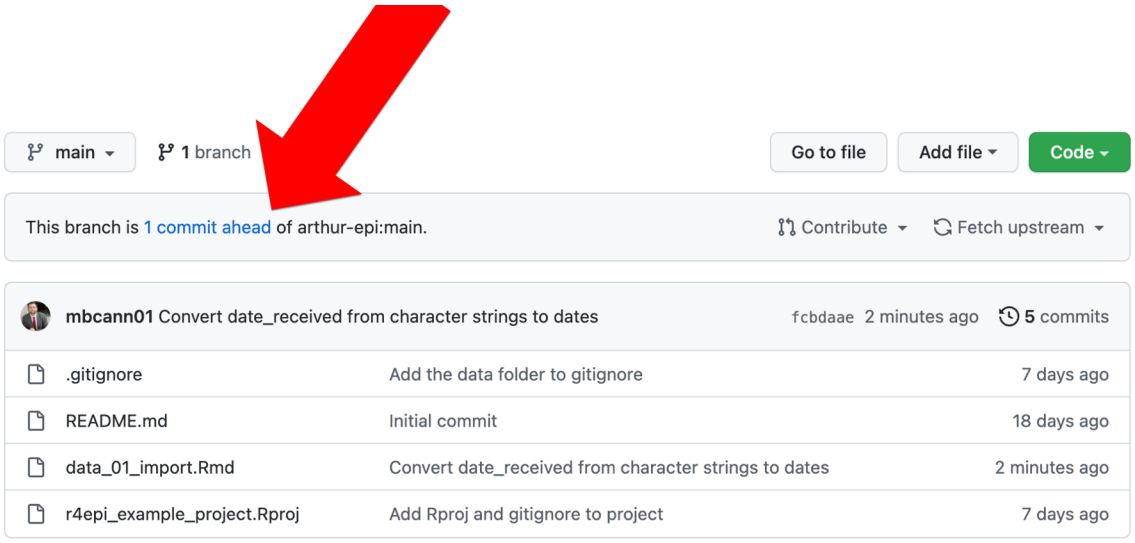
Convert date_received from character strings to dates.

```{r}
form_20 <- form_20 %>%
  mutate(date_received = as.Date(date_received))
```

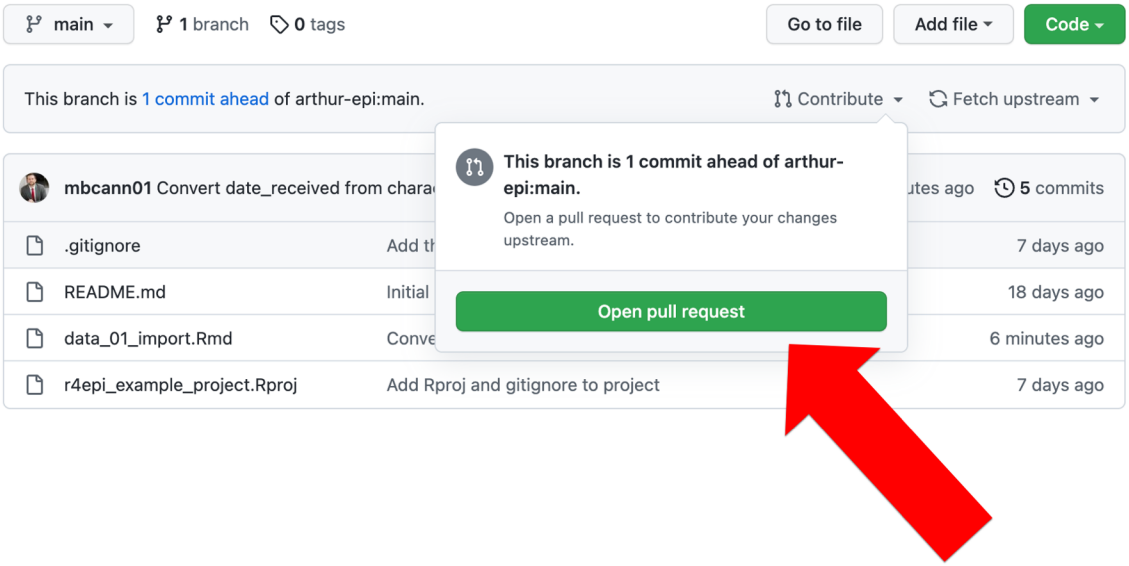
```{r}
glimpse(form_20)
```

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 changed, and pushed the commit to his GitHub account, but nothing has changed in the repository on Arthur's GitHub account.



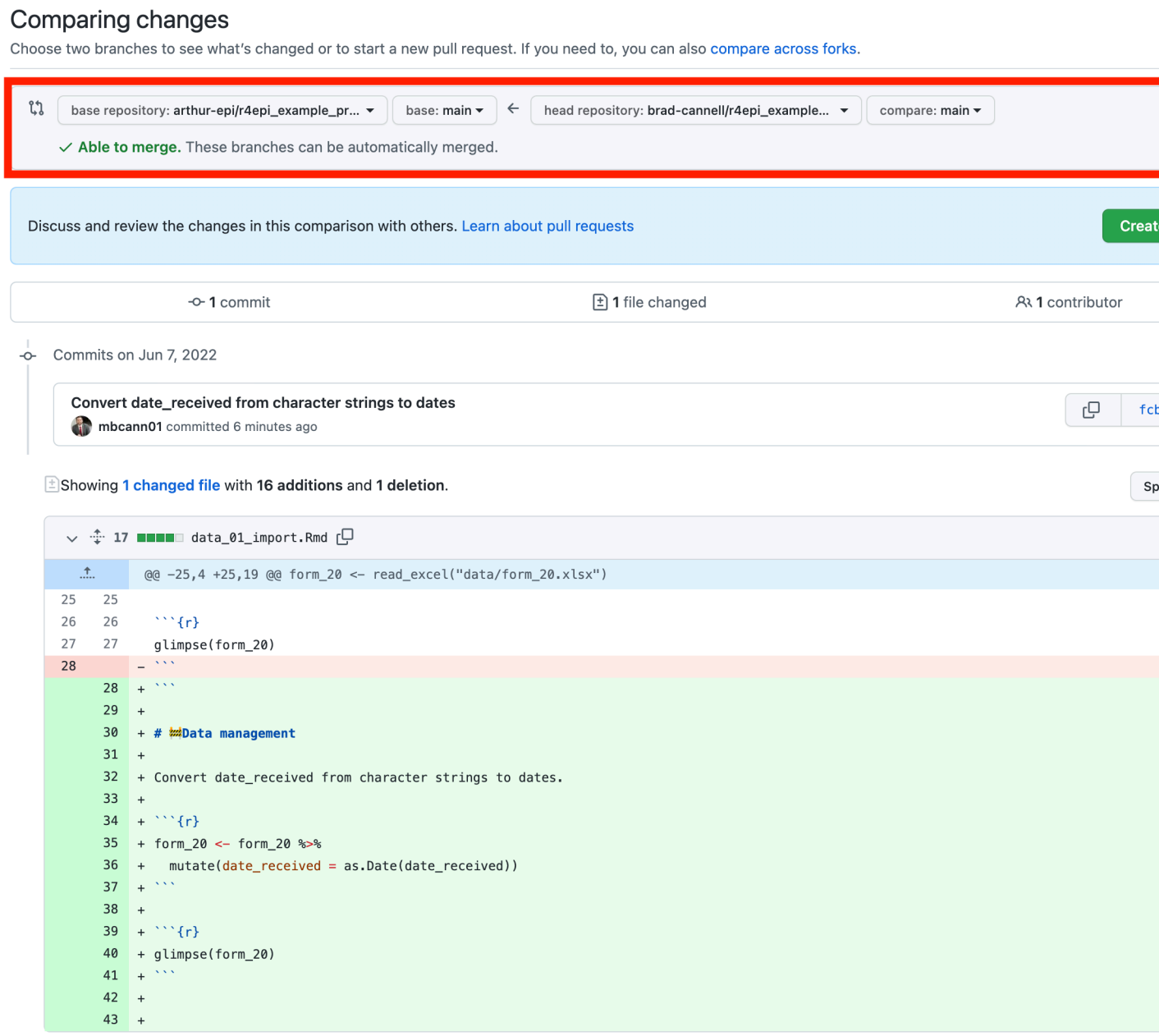
Now, Brad needs to create a pull request. This pull request will let Arthur know that Brad has made some changes to the code that he wants to share with Arthur. To do so, Brad will click `Contribute` and then click the green `Open pull request` button as shown below.



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.



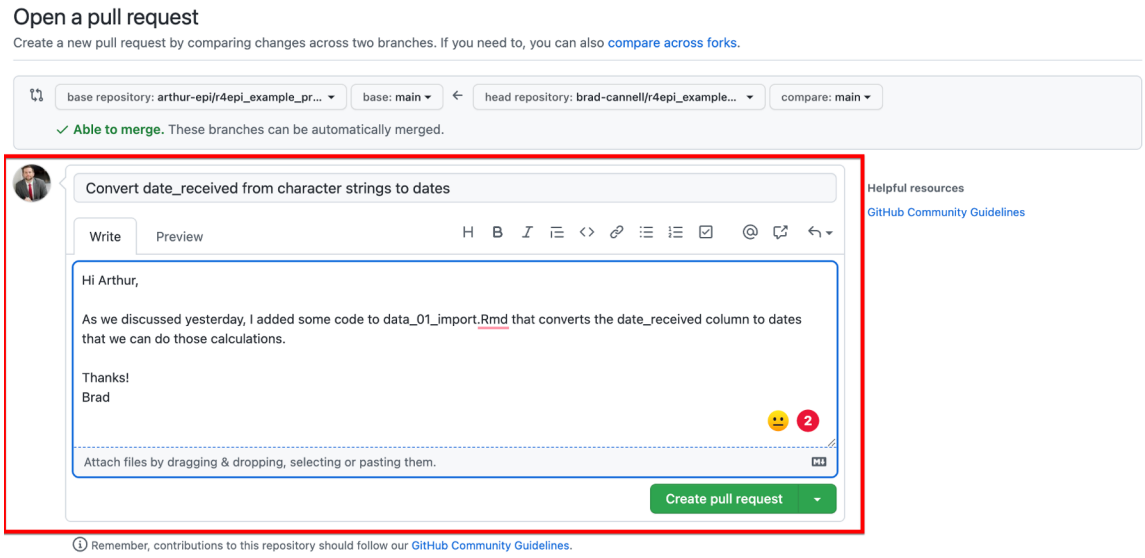
Let's pause here and get explicit about two things.

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.

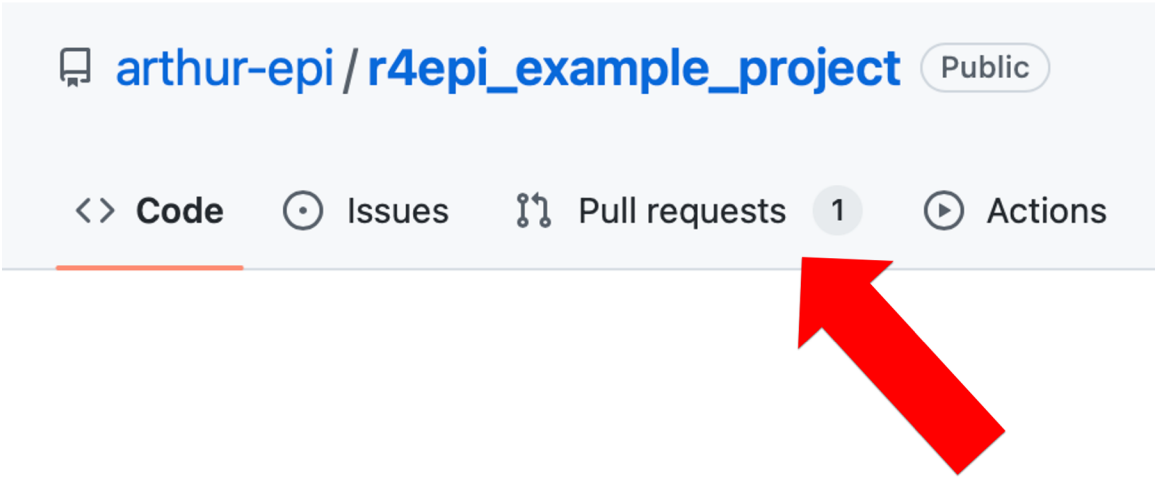
2. Pull requests are sent at the **branch level** not at the **file level**. Meaning, if Arthur accepts Brad's pull request, it will make *all* of the files on his main branch identical to *all* of the files on Brad's main branch (the main branch because that is the branch Brad chose in the screenshot 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.

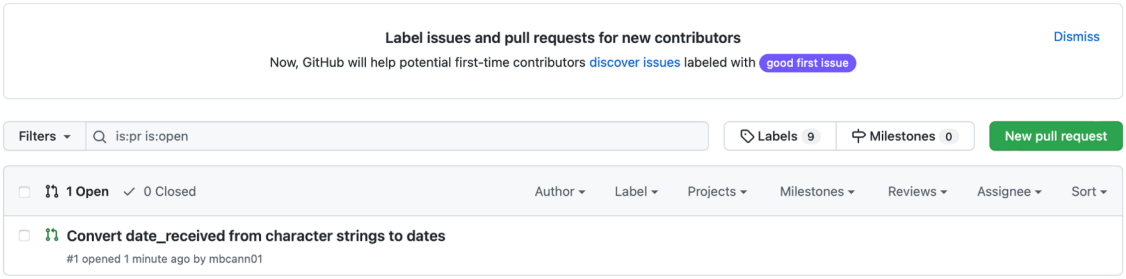
After filling out the commit message, Brad will click the green `Create pull request` button on last time, and he is done. This will send Arthur the pull request.



The next time Arthur checks the `r4epi_example_project` on GitHub, he will see that he has a new pull request.

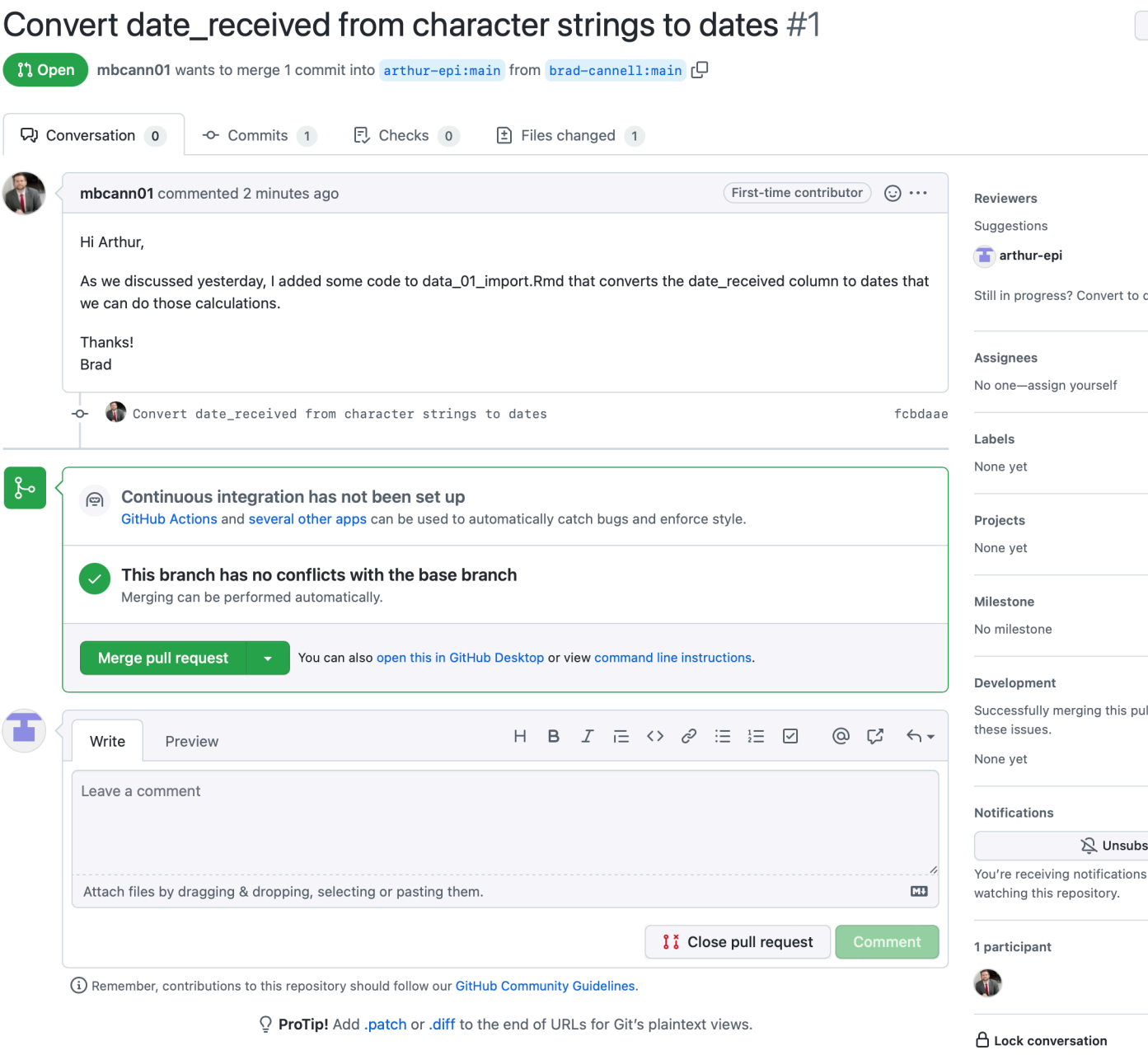


If he clicks on the text `Pull requests` text, he will be taken to his pull requests page. It will show him all pending pull requests. In this case, there is just the one pull request that Brad sent.



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

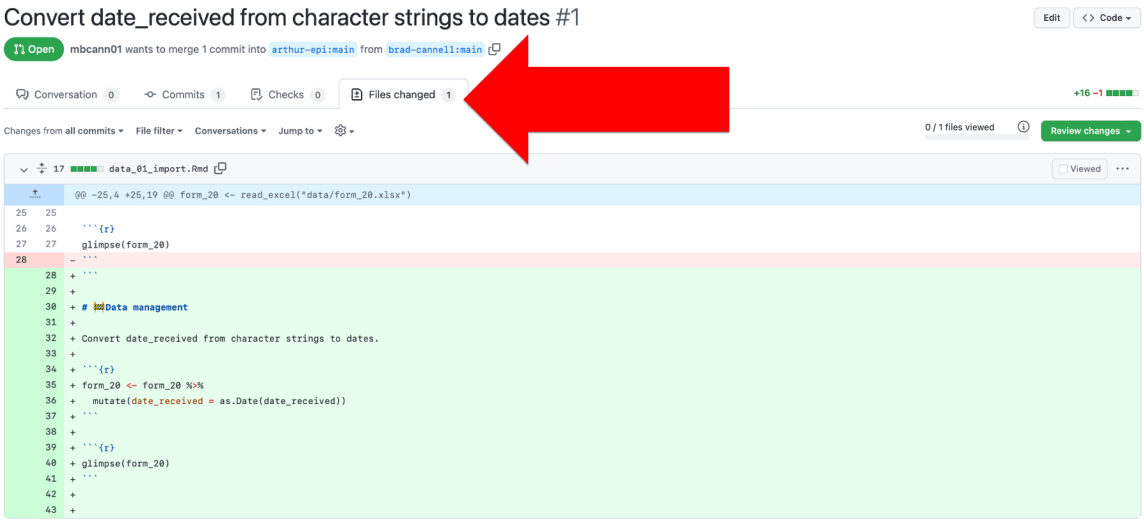
him the message Brad wrote, tell him that he can merge this branch without any conflicts if he so chooses, and give him an opportunity to write a message back to Brad before deciding whether to merge this pull request or close it.



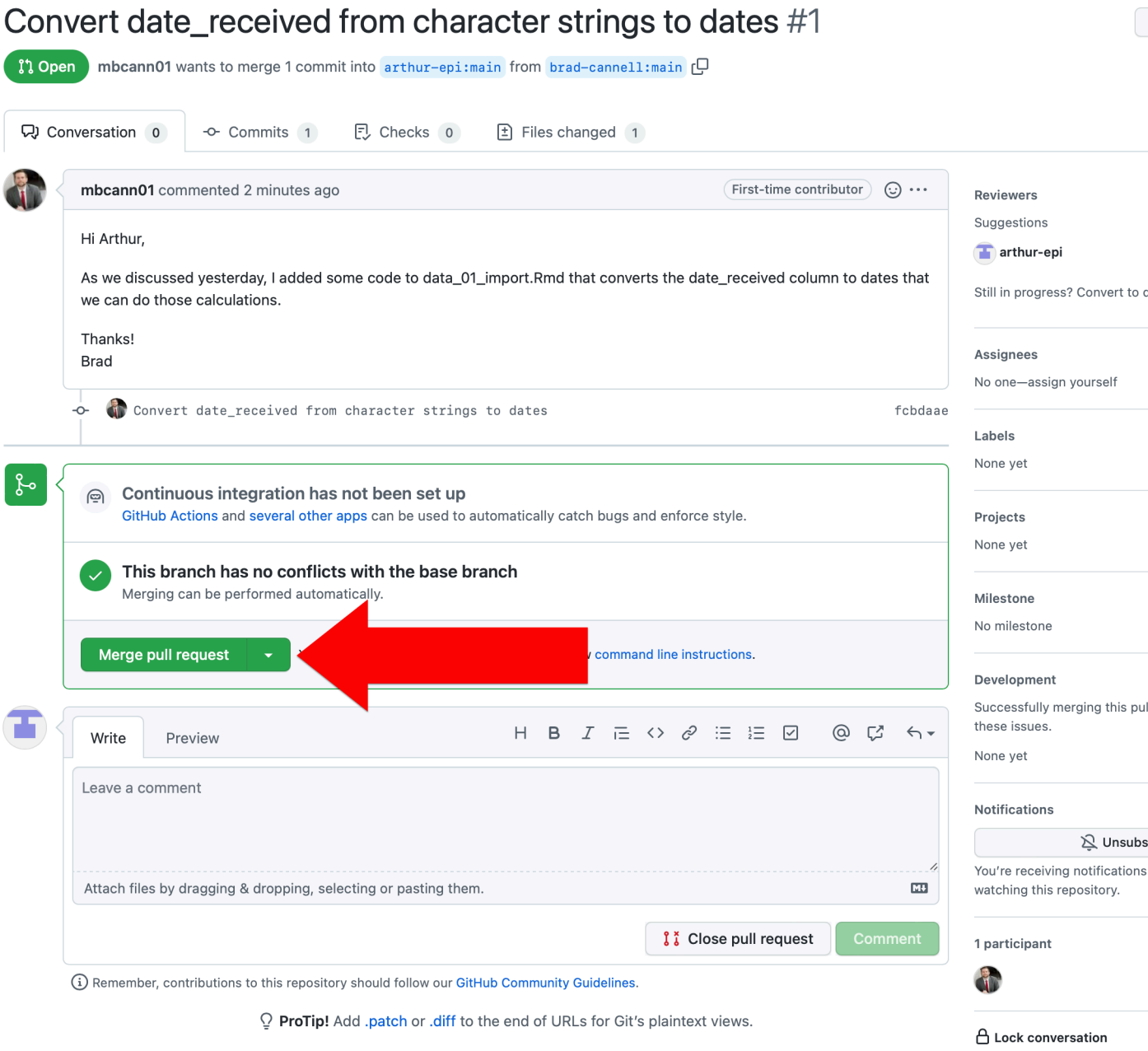
He also has the option to view some additional details by clicking the `Commits` tab, `Checks` tab, and/or `Files changed` tab towards the top of the screen. Let's say he decides to click on

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.

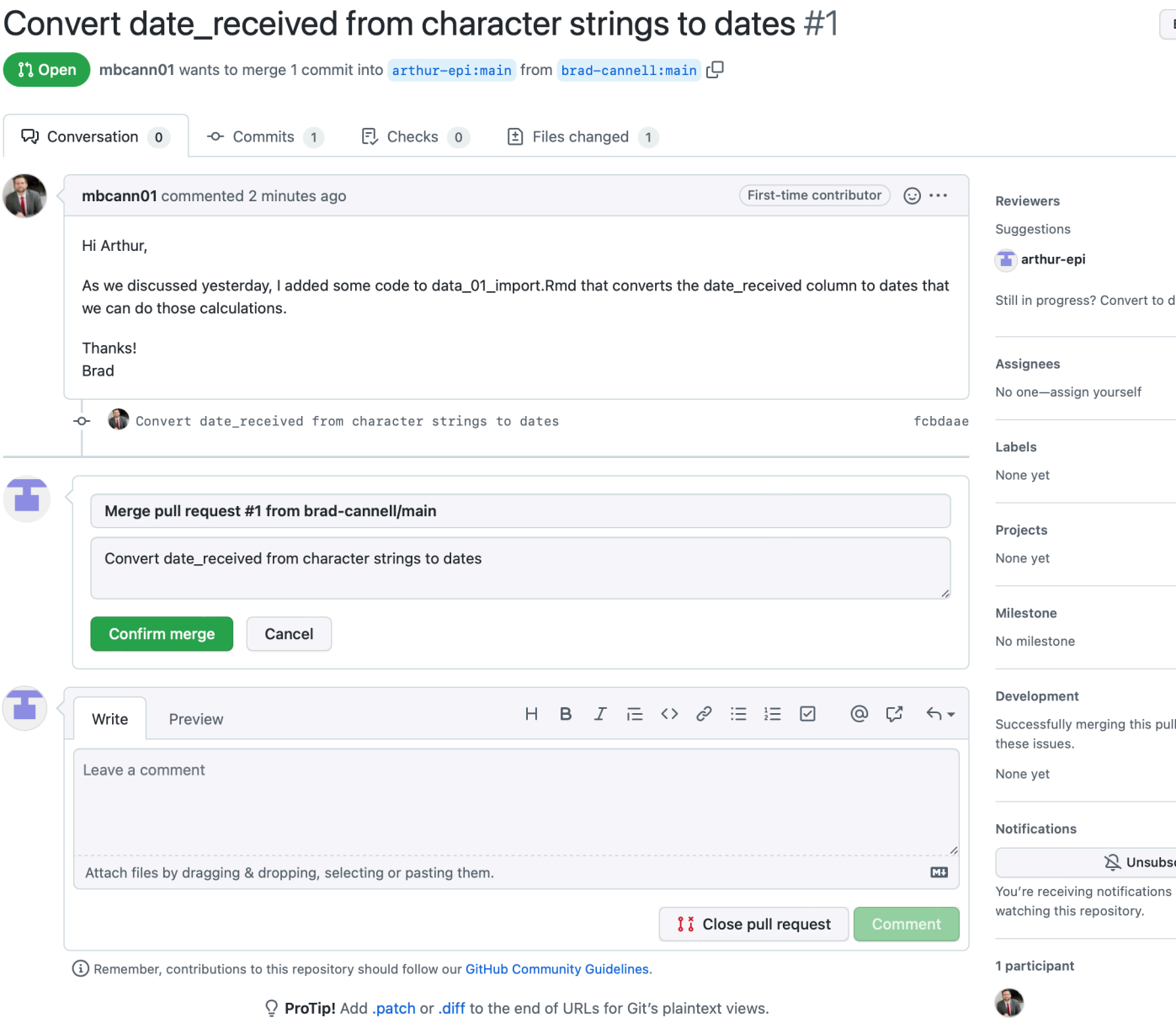


Back on the `Conversation` tab (see screenshot below), Arthur has some options. If he wants more clarification about the pull request, he can send leave a comment for Brad using the comment box near the bottom of the screen. If he knows that he does **NOT** want to merge this pull request into his code, 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.

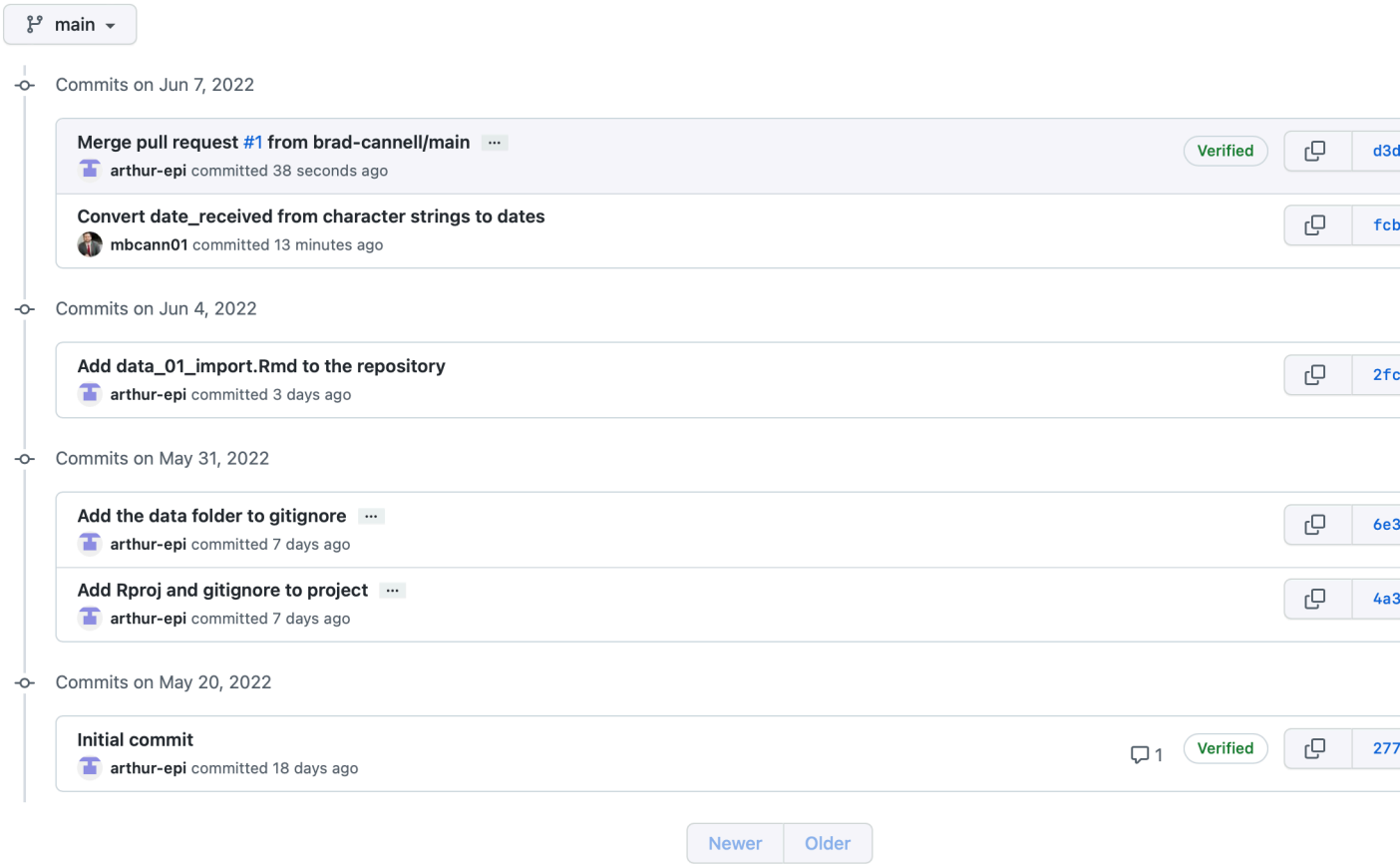


Then, he is given an opportunity to add some details about the changes this merge will make to the repository once it is committed. You can once again think of this message as having a very similar purpose to commit messages, which were discussed above. In fact, it will appear as a commit in the repository's commit history.

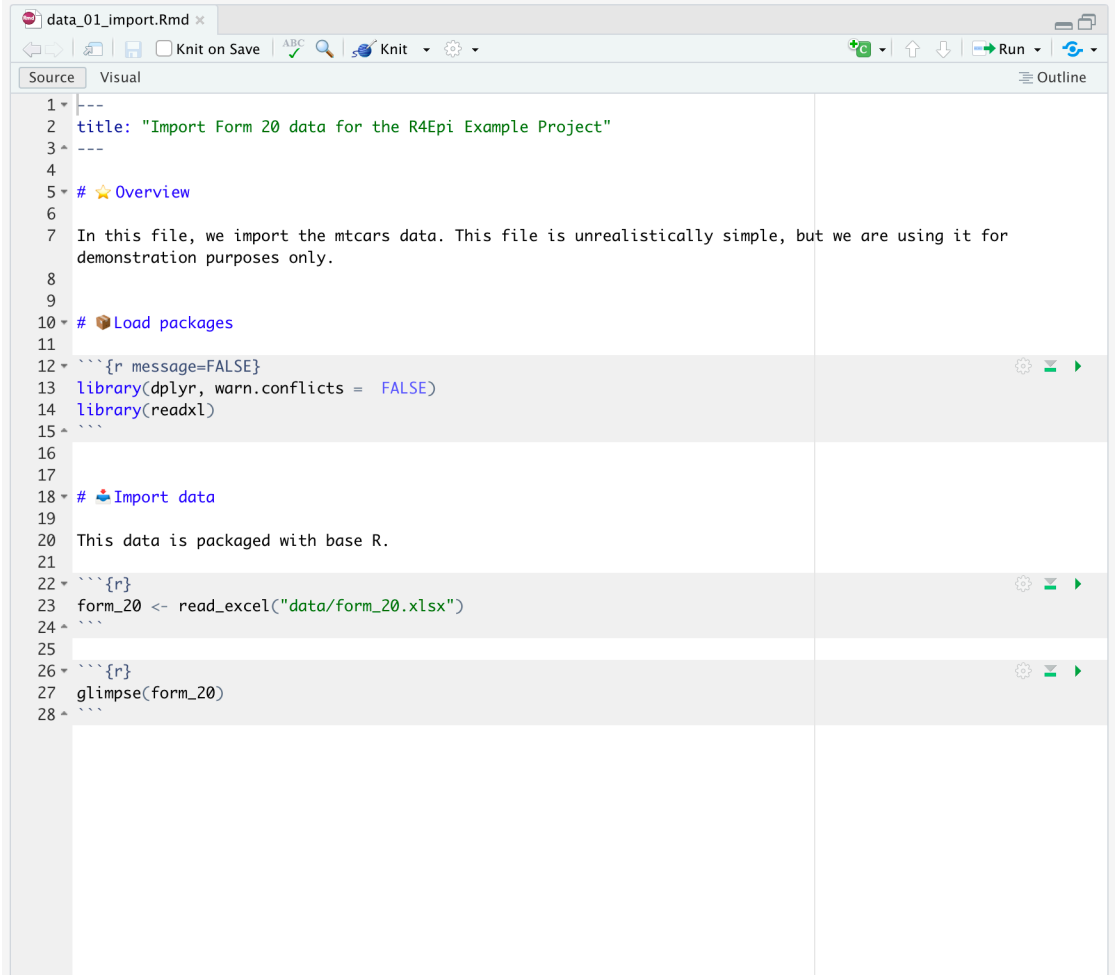
Finally, he clicks the green `Confirm merge` button.



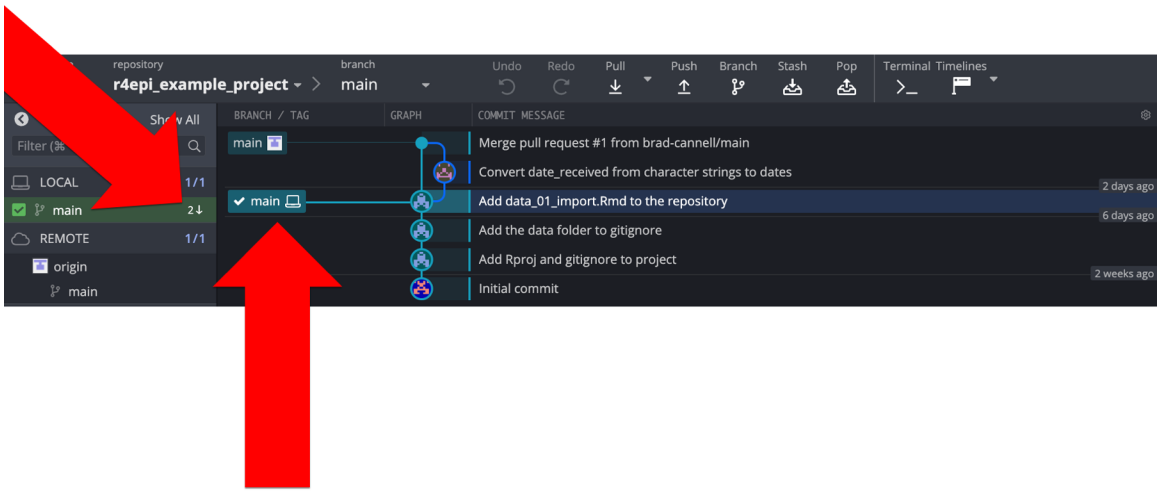
And if Arthur navigates back to his commit history page, he can see two new commits. Brad's commit with the updated `data_01_import.Qmd` file, and the commit that was automatically created when Arthur merged the branches together.



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?

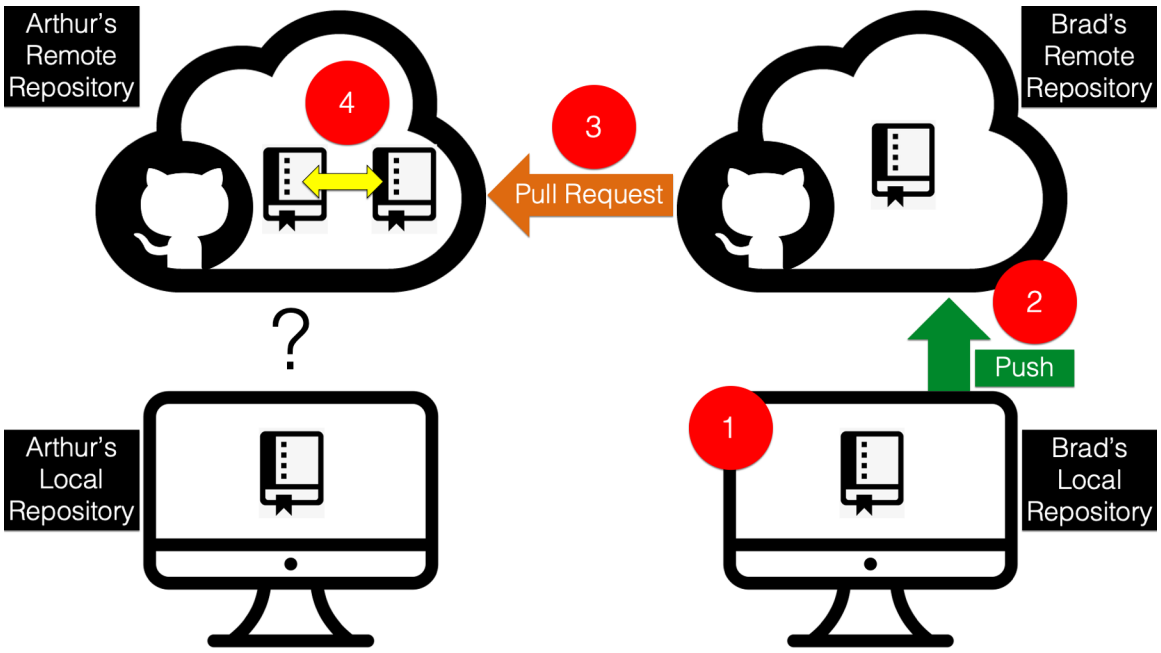


Well, let's open GitKraken on Arthur's computer and see if we can help him figure it out. In the repository graph, Arthur's local repository (i.e., the little laptop icon) and the remote repository (i.e., the little gray and white icon) are on different rows. Additionally, there is a little 2 next to 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.



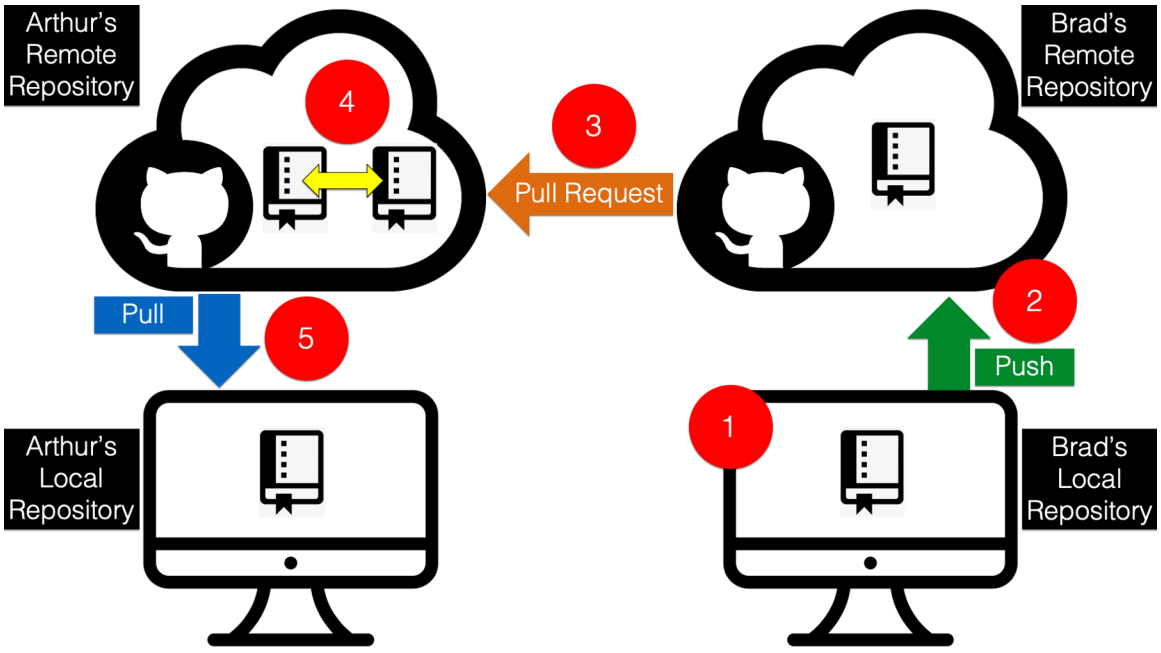
So, let's pause here for a second and review what we've done so far. As shown in the figure below:

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.



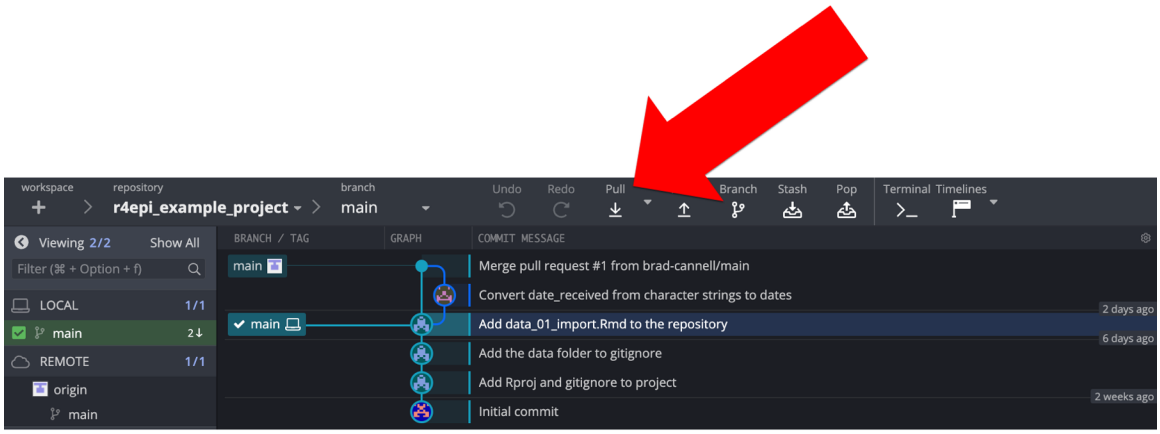
So, how does Arthur get his local repository in sync with his remote repository?

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).

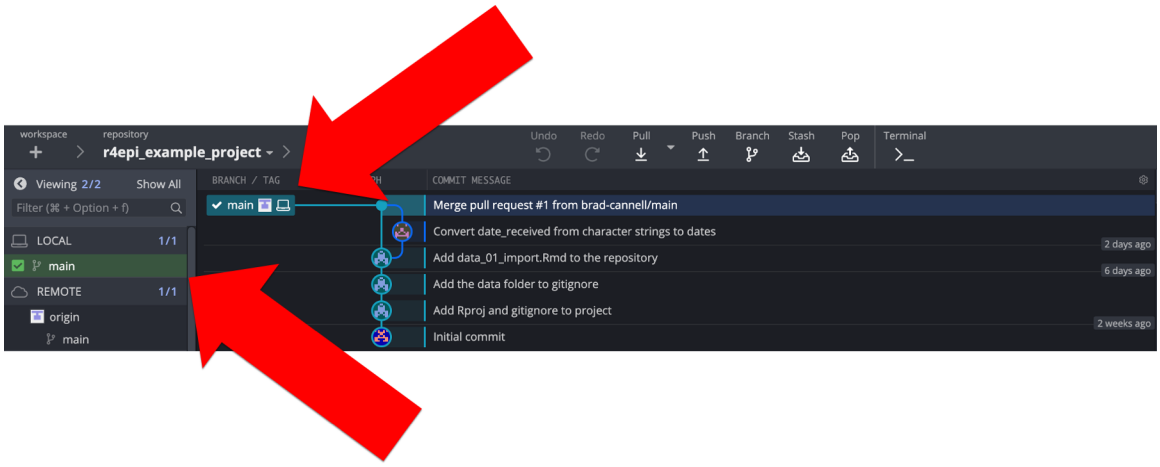


And GitKraken makes pulling the files from his remote repository really easy. All Arthur needs

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.

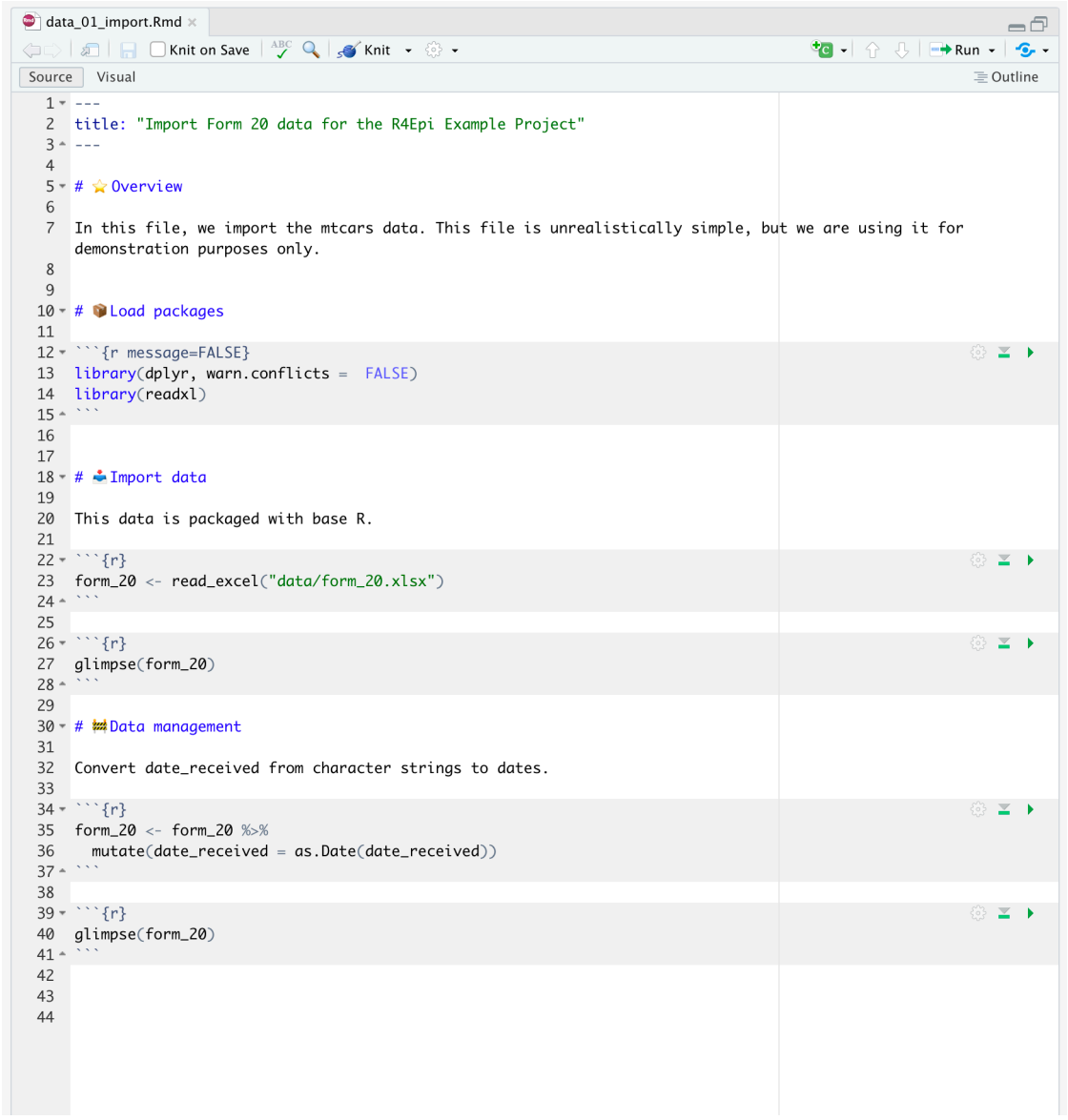


And as shown in the screenshot below, Arthur can now see that his local repository is now in sync with his remote repository once again!



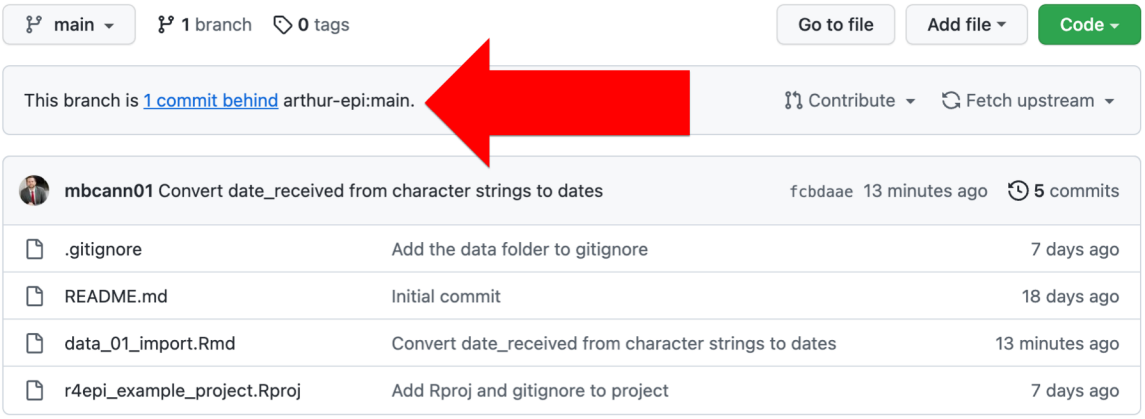
But, what about Brad's repository? Well, as you can see in the screenshot below, Brad's remote repository is now 1 commit *behind* Arthur's. Why?

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!

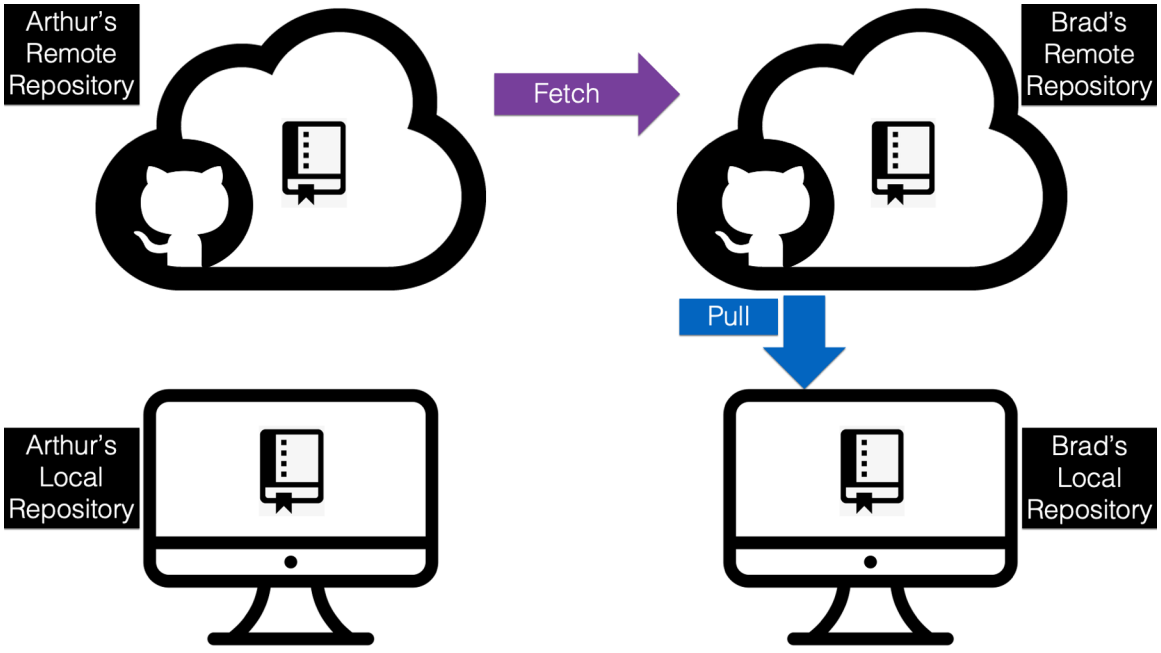


All Brad needs to do is a quick **fetch** from Arthur's remote repository to merge that last

commit into his commit history, and then **pull** it down to his local repository.

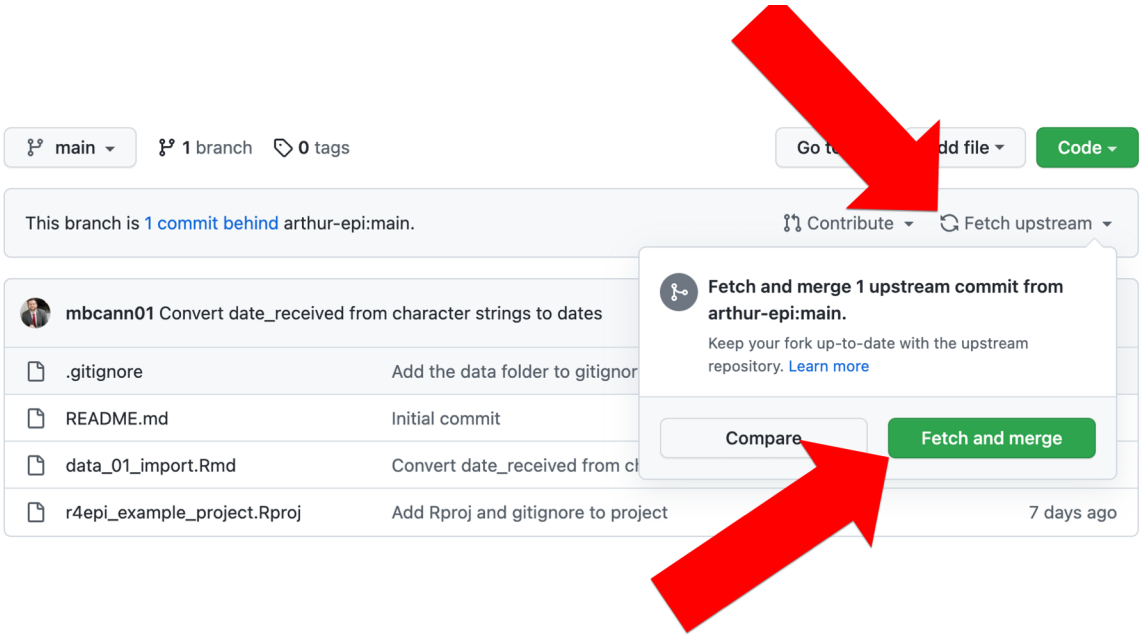


To do so, Brad will first click `Fetch upstream` followed by the green `Fetch and merge` button.



After a few seconds, GitHub will show him that his remote repository is now synced up with

Arthur's remote repository. All he as 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.

39.8 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 VIII

Presenting Results

40 Creating Tables with R and Microsoft Word

At this point, you should all know that it is generally a bad idea to submit raw R output as part of a report, presentation, or publication. You should also understand when it is most appropriate to use tables, as opposed to charts and graphs, to present your results. If not, please stop here and read Chapter 7 of *Successful Scientific Writing*, which discusses the "why" behind much of what we will show you "how" to do in this chapter.[15]

R for Epidemiology is predominantly a book about using R to manage, visualize, and analyze data in ways that are common in the field of epidemiology. However, in most modern work/research environments it is difficult to escape the requirement to share your results in a Microsoft Word document. And often, because we are dealing with data, those results include tables of some sort. However, not all tables communicate your results equally well. In this chapter, we will walk you through the process of starting with some results you calculated in R and ending with a nicely formatted table in Microsoft Word. Specifically, we are going to create a Table 1.

40.1 Table 1

In epidemiology, medicine, and other disciplines, "Table 1" has a special meaning. Yes, it's the first table shown to the reader of your article, report, or presentation, but the special meaning goes beyond that. In many disciplines, including epidemiology, when you speak to a colleague about their "Table 1" it is understood that you are speaking about a table that describes (statistically) the relevant characteristics of the sample being studied. Often, but not always, the sample being studied is made up of people, and the relevant descriptive characteristics about those people include sociodemographic and/or general health information. Therefore, it is important that you don't label any of your tables as "Table 1" arbitrarily. Unless you have a *really* good reason to do otherwise, your Table 1 should always be a descriptive overview of your sample.

Here is a list of other traits that should consider when creating your Table 1:

- All other formatting best practices that apply to scientific tables in general. This includes formatting requirements specific to wherever you are submitting your table (e.g., formatting requirements in the American Journal of Public Health).

- Table 1 is often, but not always, stratified into subgroups (i.e., descriptive results are presented separately for each subgroup of the study sample in a way that lends itself to between-group comparisons).
- When Table 1 is stratified into subgroups, the variable that contains the subgroups is typically the primary exposure/predictor of interest in your study.

40.2 Opioid drug use

As a motivating example, let's say that we are working at the North Texas Regional Health Department and have been asked to create a report about drug use in our region. Our stakeholders are particularly interested in opioid drug use. To create this report, we will analyze data from a sample of 9,985 adults who were asked about their use of drugs. One of the first analyses that we did was a descriptive comparison of the sociodemographic characteristics of 3 subgroups of people in our data. We will use these analyses to create our Table 1.

You can view/download the data by clicking here

```
Rows: 9985 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (4): age, edu, female, use

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.
```

```
    var              cat    n n_total   percent        se   t_crit      lcl
1 use_f        Non-users 8315    9985 83.274912 0.3734986 1.960202 82.52992
2 use_f  Use other drugs 1532    9985 15.343015 0.3606903 1.960202 14.64925
3 use_f Use opioid drugs  138    9985  1.382073 0.1168399 1.960202  1.17080
        ucl
1 83.994296
2 16.063453
3  1.630841
```

```
  var use    n     mean       sd   t_crit        sem      lcl      ucl
1 age   0 8315 36.80173 9.997545 1.960249 0.10963828 36.58681 37.01665
2 age   1 1532 21.98362 2.979511 1.961515 0.07612296 21.83431 22.13294
3 age   2  138 17.34740 3.081049 1.977431 0.26227634 16.82877 17.86603
```

```
  row_var row_cat  col_var col_cat    n n_row n_total percent_total   se_total
1     use       0 female_f      No 3077  8315    9985    30.8162243 0.46210382
2     use       0 female_f     Yes 5238  8315    9985    52.4586880 0.49979512
3     use       1 female_f      No  796  1532    9985     7.9719579 0.27107552
4     use       1 female_f     Yes  736  1532    9985     7.3710566 0.26150858
5     use       2 female_f      No   91   138    9985     0.9113671 0.09510564
6     use       2 female_f     Yes   47   138    9985     0.4707061 0.06850118
  t_crit_total  lcl_total  ucl_total percent_row    se_row t_crit_row  lcl_row
1     1.960202 29.9178650 31.7293461    37.00541 0.5295162   1.960202 35.97359
2     1.960202 51.4781679 53.4373162    62.99459 0.5295162   1.960202 61.95076
3     1.960202  7.4565108  8.5197562    51.95822 1.2768770   1.960202 49.45247
4     1.960202  6.8745702  7.9003577    48.04178 1.2768770   1.960202 45.54583
5     1.960202  0.7426382  1.1179996    65.94203 4.0488366   1.960202 57.62323
6     1.960202  0.3538217  0.6259604    34.05797 4.0488366   1.960202 26.61786
   ucl_row
1 38.04924
2 64.02641
3 54.45417
4 50.54753
5 73.38214
6 42.37677
```

```
   row_var row_cat col_var              col_cat    n n_row n_total
1      use       0   edu_f Less than high school 3908  8315    9985
2      use       0   edu_f           High school 2494  8315    9985
3      use       0   edu_f          Some college  915  8315    9985
4      use       0   edu_f      College graduate  998  8315    9985
5      use       1   edu_f Less than high school  322  1532    9985
6      use       1   edu_f           High school  567  1532    9985
7      use       1   edu_f          Some college  321  1532    9985
8      use       1   edu_f      College graduate  322  1532    9985
9      use       2   edu_f Less than high school   36   138    9985
10     use       2   edu_f           High school   36   138    9985
11     use       2   edu_f          Some college   40   138    9985
12     use       2   edu_f      College graduate   26   138    9985
   percent_total   se_total t_crit_total  lcl_total  ucl_total percent_row
1     39.1387081 0.48845160     1.960202 38.1855341 40.1002400    46.99940
2     24.9774662 0.43322925     1.960202 24.1379137 25.8362747    29.99399
3      9.1637456 0.28874458     1.960202  8.6132458  9.7456775    11.00421
4      9.9949925 0.30017346     1.960202  9.4217947 10.5989817    12.00241
5      3.2248373 0.17680053     1.960202  2.8957068  3.5899942    21.01828
6      5.6785178 0.23161705     1.960202  5.2411937  6.1499638    37.01044
```

```
7      3.2148222 0.17653492     1.960202  2.8862151  3.5794637    20.95300
8      3.2248373 0.17680053     1.960202  2.8957068  3.5899942    21.01828
9      0.3605408 0.05998472     1.960202  0.2601621  0.4994549    26.08696
10     0.3605408 0.05998472     1.960202  0.2601621  0.4994549    26.08696
11     0.4006009 0.06321673     1.960202  0.2939655  0.5457064    28.98551
12     0.2603906 0.05100282     1.960202  0.1773397  0.3821865    18.84058
      se_row t_crit_row  lcl_row  ucl_row
1  0.5473707   1.960202 45.92799 48.07358
2  0.5025506   1.960202 29.01822 30.98824
3  0.3432094   1.960202 10.34925 11.69521
4  0.3564220   1.960202 11.32112 12.71881
5  1.0412961   1.960202 19.04975 23.13209
6  1.2339820   1.960202 34.62587 39.46012
7  1.0401075   1.960202 18.98696 23.06466
8  1.0412961   1.960202 19.04975 23.13209
9  3.7515606   1.960202 19.42163 34.07250
10 3.7515606   1.960202 19.42163 34.07250
11 3.8761776   1.960202 22.00774 37.12261
12 3.3408449   1.960202 13.13952 26.26721
```

Above, we have the results of several different descriptive analyses we did in R. Remember that we never want to present raw R output. Perhaps you've already thought to yourselves, "wow, these results are really overwhelming. We're not sure what we're even looking at." Well, that's exactly how many of the people in your audience will feel as well. In its current form, this information is really hard for us to process. We want to take some of the information from the output above and use it to create a Table 1 in Word that is much easier to read.

Specifically, we want our final Table 1 to look like this:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

You may also click here to view/download the Word file that contains Table 1.

Now that you've seen the end result, let's learn how to make this Table 1 together, step-by-step. Go ahead and open Microsoft Word now if you want to follow along.

40.3 Table columns

The first thing we typically do is figure out how many columns and rows our table will need. This is generally pretty straightforward; although, there are exceptions. For a basic Table 1 like the one we are creating above we need the following columns:

One column for our row headers (i.e., the names and categories of the variables we are presenting in our analysis).

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

One column for each subgroup that we will be describing in our table. In this case, there are 3 subgroups so we will need 3 additional columns.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

So, we will need 4 total columns.

> **Note**
>
> If you are going to describe the entire sample overall without stratifying it into subgroups then you would simply have 2 columns. One for the row headers and one for the values.

40.4 Table rows

Next, we need to figure out how many rows our table will need. This is also pretty straightforward. Generally, we will need the following rows:

One row for the title. Some people write their table titles outside (above or below) the actual table. We like to include the title directly in the top row of the table. That way, it moves with the table if the table gets moved around.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

One row for the column headers. The column headers generally include a label like "Characteristic" for the row headers column and a descriptive label for each subgroup we are describing in our table.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

One row for each variable we will analyze in our analysis. In this example, we have three – age, sex, and education. **NOTE** that we do *NOT* need a separate row for each category of each variable.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

One row for the footer.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

So, we will need 6 total rows.

40.5 Make the table skeleton

Now that we know we need to create a table with 4 columns and 6 rows, let's go ahead and do that in Microsoft Word. We do so by clicking the `Insert` tab in the ribbon above our document. Then, we click the `Table` button and select the number of columns and rows we want.

40.6 Fill in column headers

Now we have our table skeleton. The next thing we would typically do is fill in the column headers. Remember that our column headers look like this:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

Here are a couple of suggestions for filling in your column headers:

- Put your column headers in the second row of the empty table shell. The title will eventually go into the first row. We don't add the title right away because it is typically long and will distort the table's dimensions. Later, we will see how to horizontally merge table cells to remove this distortion, but we don't want to do that now. Right now, we want to leave all the cells unmerged so that we can easily resize our columns.
- The first column header is generally a label for our row headers. Because the rows are typically characteristics of our sample, we almost always use the word "characteristic" here. If you come up with a better word, please feel free to use it.
- The rest of the column headers are generally devoted to the subgroups we are describing.
  - The subgroups should be ordered in a way that is meaningful. For example, by level of severity or chronological order. Typically, ordering in alphabetical order isn't that meaningful.
  - The subgroup labels should be informative and meaningful, but also succinct. This can sometimes be a challenge.
  - we have seen terms like "Value", "All", and "Full Sample" used when Table 1 was describing the entire sample overall rather than describing the sample by subgroups.

40.6.1 Group sample sizes

You should always include the group sample size in the column header. They should typically be in the format "(n = sample size)" and typed in the same cell as the label, but below the label (i.e., hit the return key). The group sample sizes can often provide important context to the statistics listed below in the table, and clue the reader into missing data issues.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

40.6.2 Formatting column headers

we generally bold our column headers, horizontally center them, and vertically align them to the bottom of the row.

At this point, your table should look like this in Microsoft Word:

Characteristic	**Non-user (n = 8,315)**	**Use Other Drugs (n = 1,532)**	**Use Opioid Drugs (n = 138)**

40.7 Fill in row headers

The next thing we would typically do is fill in the row headers. Remember, that our row headers look like this:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

Here are a couple of suggestions for filling in your row headers:

- The variables should be organized in a way that is meaningful. In our example, we have only 3 sociodemographic variables. However, if we also had some variables about health status and some variables related to criminal history, then we would almost certainly want the variables that fit into each of these categories to be vertically arranged next to each other.
- Like the column headers, the row headers should be informative and meaningful, but also succinct. Again, this can sometimes be a challenge. In our example, we use "Age", "Sex", and "Education". Something like "Highest level of formal education completed" would have also been informative and meaningful, but not succinct. Something like "Question 6" is succinct, but isn't informative or meaningful at all.

40.7.1 Label statistics

You should always tell the reader what kind of statistics they are looking at – don't assume that they know. For example, the highlighted numbers in figure @ref(fig:what-stats) are 36.8 and 10. What is 36.8? The mean, the median? The percentage of people who had a non-missing value for age? What is 10? The sample size? The standard error of the mean? An odds ratio? You know that 36.8 is a mean and 10 is the standard deviation because we identified what they were in the row header. @ref(fig:identify-stats) When you label the statistics in the row headers as we've done in our example, they should take the format you see in figure @ref(fig:identify-stats). That is, the variable name, followed by a comma, followed by the statistics used in that row. Also notice the use of parentheses. We used parentheses around the letters "sd" (for standard deviation) because the numbers inside the parentheses in that row are standard deviations. So, the label used to identify the statistics should give the reader a blueprint for interpreting the statistics that matches the format of the statistics themselves.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

Figure 40.1: What are these numbers?

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age, mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

Figure 40.2: Identifying statistics in the row header.

The statistics can, and sometimes are, labeled in the column header instead of the row header. This can sometimes be a great idea. However, it can also be a source of confusion. For example, in the figure below, the column headers include labels (i.e., n (%)) for the statistics below. However, *not all the statistics below are counts (n) and percentages*!

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315) n (%)	Use Other Drugs (n = 1,532) n (%)	Use Opioid Drugs (n = 138) n (%)
Sex			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education			
Less than hig	3908 (47.0)	322 (21.0)	36 (26.1)
High school g	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

All n (%)

Even though the Age variable has its own separate statistics label in the row header, this is still generally a really bad idea! Therefore, we highly recommend only labeling your statistics in the column header when those labels are accurate for *every* value in the column. For example:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315) n (%)	Use Other Drugs (n = 1,532) n (%)	Use Opioid Drugs (n = 138) n (%)
Sex			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education			
Less than hig	3908 (47.0)	322 (21.0)	36 (26.1)
High school g	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

All n (%)

1. Age is in years.

40.7.2 Formatting row headers

- Whenever possible, make sure that variable name and statistic identifier fit on one line (i.e., they don't carry over into the line below).
- Always type the category labels for categorical variables in the same cell as the variable name. However, each category should have its own line (i.e., hit the return key).
- Whenever possible, make sure that each category label fits on one line (i.e., it doesn't carry over into the line below).
- Indent each category label two spaces to the left of the variable name.
- Hit the return key once after the last category for each categorical variable. This creates a blank line that adds vertical separation between row headers and makes them easier to read.

At this point, your table should look like this in Microsoft Word:

Characteristic	**Non-user (n = 8,315)**	**Use Other Drugs (n = 1,532)**	**Use Opioid Drugs (n = 138)**
Age, mean (sd)			
Sex, n (%) Male Female			
Education, n (%) Less than high school High school graduate Some college College graduate			

40.8 Fill in data values

So, we have some statistics visible to us on the screen in RStudio. Somehow, we have to get those numbers over to our table in Microsoft Word. There are many different ways we can do this. We're going to compare a few of those ways here.

40.8.1 Manually type values

One option is to manually type the numbers into your word document.

If you are in a hurry, or if you just need to update a small handful of statistics in your table, then this option is super straightforward. However, there are at least two big problems with this method.

First, it is *extremely* error prone. Most people are very likely to type a wrong number or misplace a decimal here and there when they manually type statistics into their Word tables.

Second, it isn't very scalable. What if you need to make very large tables with lots and lots of numbers? What if you update your data set and need to change every number in your Word table? This is not fun to do manually.

40.8.2 Copy and paste values

Another option is to copy and paste values from RStudio into Word. This option is similar to above, but instead of *typing* each value into your Word table, you highlight and copy the value in RStudio and *paste* it into Word.

If you are in a hurry, or if you just need to update a small handful of statistics in your table, then this option is also pretty straightforward. However, there are still issues associated with this method.

First, it is still somewhat error prone. It's true that the numbers and decimal placements should always be correct when you copy and paste; however, you may be surprised by how often many people accidentally paste the values into the wrong place or in the wrong order.

Second, I've noticed that there are often weird formatting things that happen when we copy from RStudio and paste into Word. They are usually pretty easy to fix, but this is still a small bit of extra hassle.

Third, it isn't very scalable. Again, if we need to make very large tables with lots and lots of numbers or update our data set and need to change every number in your Word table, this method is time-consuming and tedious.

40.8.3 Knit a Word document

So far, we have only used the HTML Notebook output type for our R markdown files. However, it's actually very easy have RStudio create a Word document from you R markdown files. We don't have all the R knowledge we need to fully implement this method yet, so we don't want to confuse you by going into the details here. But, we do want to mention that it is possible.

The main advantages of this method are that it is much less error prone and much more scalable than manually typing or copying and pasting values.

The main disadvantages are that it requires more work on the front end and still requires you to open Microsoft Word a do a good deal of formatting of the table(s).

40.8.4 flextable and officer

A final option we'll mention is to create your table with the flextable and officer packages. This is our favorite option, but it is also definitely the most complicated. Again, we're not going to go into the details here because they would likely just be confusing for most readers.

This method essentially overcomes all of the previous methods' limitations. It is the least error prone, it is extremely scalable, and it allows us to do basically all the formatting in R. With a push of a button we have a complete, perfectly formatted table output to a Word

document. If we update our data, we just push the button again and we have a new perfectly formatted table.

The primary downside is that this method requires you to invest some time in learning these packages, and requires the greatest amount of writing code up front. If you just need to create a single small table that you will never update, this method is probably not worth the effort. However, if you absolutely need to make sure that your table has no errors, or if you will need to update your table on a regular basis, then this method is definitely worth learning.

40.8.5 Significant digits

No matter which of the methods above you choose, you will almost never want to give your reader the level of precision that R will give you. For example, the first row of the R results below indicates that 83.274912% of our sample reported that they don't use drugs.

```
    var              cat    n n_total   percent        se   t_crit      lcl
1 use_f        Non-users 8315    9985 83.274912 0.3734986 1.960202 82.52992
2 use_f  Use other drugs 1532    9985 15.343015 0.3606903 1.960202 14.64925
3 use_f Use opioid drugs  138    9985  1.382073 0.1168399 1.960202  1.17080
        ucl
1 83.994296
2 16.063453
3  1.630841
```

Notice the level of precision there. R gives us the percentage out to 6 decimal places. If you fill your table with numbers like this, it will be much harder for your readers to digest your table and make comparisons between groups. It's just the way our brains work. So, the logical next question is, "how many decimal places should we report?" Unfortunately, this is another one of those times that we have to give you an answer that may be a little unsatisfying. It is true that there are rules for significant figures (significant digits); however, the rules are not always helpful to students in my experience. Therefore, we're going to share with you a few things we try to consider when deciding how many digits to present.

- we don't recall *ever* presenting a level of precision greater than 3 decimal places in the research we've been involved with. If you are working in physics or genetics and measuring really tiny things it may be totally valid to report 6, 8, or 10 digits to the right of the decimal. But, in epidemiology – a population science – this is rarely, if ever, useful.
- What is the overall message we are trying to communicate? That is the point of the table, right? We're trying to clearly and honestly communicate information to our readers. In general, the simpler the numbers are to read and compare, the clearer the communication. So, we tend to err on the side of simplifying as much as possible. For example, in the

R results below, we could say that 83.274912% of our sample reported that they don't use drugs, 15.343015% reported that they use drugs other than opioids, and 1.382073% reported that they use opioid drugs. Is saying it that way really more useful than saying that "83% of our sample reported that they don't use drugs, 15% reported that they use drugs other than opioids, and 1% reported that they use opioid drugs"? Are we missing any actionable information by rounding our percentages to the nearest integer here? Are our overall conclusions about drug use any different? No, probably not. And, the rounded percentages are much easier to read, compare, and remember.

- Be consistent – especially within a single table. We have experienced some rare occasions where it made sense to round one variable to 1 decimal place and another variable to 3 decimal places in the same table. But, circumstances like this are definitely the exception. Generally speaking, if you round one variable to 1 decimal place then you want to round them all to one decimal place.

Like all other calculations we've done in this book, we suggest you let R do the heavy lifting when it comes to rounding. In other words, have R round the values for you *before* you move them to Word. R is much less likely to make a rounding error than you are! You may recall that we learned how to round in the chapter on numerical descriptions of categorical variables.

40.8.6 Formatting data values

Now that we have our appropriately rounded values in our table, we just need to do a little formatting before we move on.

First, make sure to fix any fonts, font sizes, and/or background colors that may have been changed if you copied and pasted the values from RStudio into Word.

Second, make sure the values line up horizontally with the correct variable names and category labels.

Third, we tend to horizontally center all our values in their columns.

At this point, your table should look like this in Microsoft Word:

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age, mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

40.9 Fill in title

At this point in the process, we will typically go ahead and add the title to the first cell of our Word table. The title should always start with "Table #." In our case, it will start with "Table 1." In general, we use bold text for this part of the title. What comes next will change a little bit from table to table but is extremely important and worth putting some thought into.

Remember, all tables and figures need to be able to *stand on their own*. What does that mean? It means that if we pick up your report and flip straight to the table, we should be able to understand what it's about and how to read it without reading any of the other text in your report. The title is a critical part of making a table stand on its own. In general, your title should tell the reader what the table contains (e.g., sociodemographic characteristics) and who the table is about (e.g., results of the Texas Opioid Study). We will usually also add the size of the sample of people included in the table (e.g., n = 9985) and the year the data was collected (e.g., 2020).

In different circumstances, more or less information may be needed. However, always ask yourself, "can this table stand on its own? Can most readers understand what's going on in this table even if they didn't read the full report?"

At this point, your table should look like this in Microsoft Word:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.			
Characteristic	**Non-user (n = 8,315)**	**Use Other Drugs (n = 1,532)**	**Use Opioid Drugs (n = 138)**
Age, mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

Don't worry about your title being all bunched up in the corner. We will fix it soon.

40.10 Fill in footnotes

Footnotes are another tool we can use to help our table stand on its own. The footnotes give readers additional information that they may need to read and understand our table. Again, there are few hard and fast rules regarding what footnotes you should include, but we can give you some general categories of things to think about.

First, use footnotes to explain any abbreviations in your table that aren't standard and broadly understood. These abbreviations are typically related to statistics used in the table (e.g., RR = risk ratio) and/or units of measure (e.g., mmHg = millimeters of mercury). Admittedly, there is some subjectivity associated with "standard and broadly understood." In our example, we did not provide a footnote for "n", "sd", or "%" because most researchers would agree that these abbreviations are standard and broadly understood, but we typically do provide footnotes for statistics like "OR" (odds ratio) and "RR" (relative risk or risk ratio).

Additionally, we mentioned above that it is desirable, but sometimes challenging, to get your variable names and category labels to fit on a single line. Footnotes can sometimes help with this. In our example, instead of writing "Age in years, mean (sd)" as a row header we wrote "Age, mean (sd)" and added a footnote that tells the reader that age is measured in years. This may not be the best possible example, but hopefully you get the idea.

40.10.1 Formatting footnotes

When using footnotes, you need to somehow let the reader know which element in the table each footnote goes with. Sometimes, there will be guidelines that require you to use certain symbols (e.g., *, †, and ‡), but we typically use numbers to match table elements to footnotes when we can. In the example below, there is a superscript "1" immediately after the word "Age" that lets the reader know that footnote number 1 is adding additional information to this part of the table.

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

If you do use numbers to match table elements to footnotes, make sure you do so in the order people read [English], which is left to right and top to bottom. For example, the following would be inappropriate because the number 2 comes before the number 1 when reading from top to bottom:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[2], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education[1], n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Proportion of people who reported having completed each category of formal education.
2. Age is in years.

As another example, the following would be inappropriate because the number 2 comes before the number 1 when reading from left to right:

Table 1. Sociodemographic characteristics of study participants[2] by drug use status[1], results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age, mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Participants were asked about their use of opioids and other drugs in the past year.
2. This sample include 9,985 people age 18 and older.

Additionally, when using numbers to match table elements to footnotes, it's a good idea to superscript the numbers in the table. This makes it clear that the number is being used to identify a footnote rather than being part of the text or abbreviation. Formatting a number

as a superscript is easy in Microsoft Word. Just highlight the number you want to format and click the superscript button.

At this point, your table should look like this in Microsoft Word:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.			
Characteristic	**Non-user (n = 8,315)**	**Use Other Drugs (n = 1,532)**	**Use Opioid Drugs (n = 138)**
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)
1. Age is in years.			

40.11 Final formatting

We have all of our data and explanatory text in our table. The last few remaining steps are just about formatting our table to make it as easy to read and digest as possible.

40.11.1 Adjust column widths

As I've already mentioned more than once, we don't want our text carryover onto multiple lines whenever we can help it. In my experience, this occurs most often in the row headings. Therefore, we will often need to adjust (widen) the first column of our table. You can do that by clicking on the black border that separates the columns and moving your mouse left or right.

After you adjust the width of your first column, the widths of the remaining columns will likely be uneven. To distribute the remaining space in the table evenly among the remaining columns, first select the columns by clicking immediately above the first column you want to select and dragging your cursor across the remaining columns. Then, click the `layout` tab in the ribbon above your document and the `Distribute Columns` button.

In our particular example, there was no need to adjust column widths because all of our text fit into the default widths.

40.11.2 Merge cells

Now, we can finally merge some cells so that our title and footnote spread the entire width of the table. We waited until now to merge cells because if we had done so earlier it would have made the previous step (i.e., adjusting column widths) more difficult.

To spread our title out across the entire width of the table, we just need to select all the cells in the first row, then right click and select `merge cells`.

After merging the footnote cells in exactly the same way, your table should look like this:

Table 1. Sociodemographic characteristics of study participants (n = 9985) by drug use status, results of the Texas Opioid Study, 2020.

Characteristic	Non-user (n = 8,315)	Use Other Drugs (n = 1,532)	Use Opioid Drugs (n = 138)
Age[1], mean (sd)	36.8 (10.0)	22.0 (3.0)	17.3 (3.1)
Sex, n (%)			
Male	3077 (37.0)	796 (52.0)	91 (65.9)
Female	5238 (63.0)	736 (48.0)	47 (34.1)
Education, n (%)			
Less than high school	3908 (47.0)	322 (21.0)	36 (26.1)
High school graduate	2494 (30.0)	567 (37.0)	36 (26.1)
Some college	915 (11.0)	321 (21.0)	40 (29.0)
College graduate	998 (12.0)	322 (21.0)	26 (18.8)

1. Age is in years.

40.11.3 Remove cell borders

The final step is to clean up our borders. In my experience, students like to do all kinds of creative things with cell borders. However, when it comes to borders, keeping it simple is usually the best approach. Therefore, we will start by removing *all* borders in the table. We do so by clicking the little cross with arrowheads that pops up diagonal to the top-left corner of the table when you move your mouse over it. Clicking this button will highlight your entire table. Then, we will click the downward facing arrow next to the `borders` button in the ribbon above your document. Then, we will click the `No Border` option.

Our final step will be to add a single horizontal border under the title, a single horizontal border under the column header row, and a single horizontal border above the footnotes. We will add the borders by highlighting the correct rows and selecting the correct options for the same `borders` dropdown menu we used above.

Notice that there are *no vertical lines (borders) anywhere on our table*. That should almost always be the case for your tables too.

40.12 Summary

Just like with guidelines we've discussed about R coding style; you don't have to create tables in exactly the same way that we do. But, you should have a good reason for all the decisions you make leading up

Part IX

References

41 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. Field A, Miles J, Field Z. *Discovering Statistics Using R.* Sage; 2013.

8. Wickham H. Tidy data. *Journal of Statistical Software, Articles.* 2014;59(10):1-23.

9. Grolemund G, Wickham H. *R for Data Science.*; 2017.

10. Wickham H, François R, Henry L, Müller K, RStudio. *Programming with Dplyr.*; 2020.

11. Peng RD, Hicks SC. Reproducible research: A retrospective. *Annu Rev Public Health.* 2021;42:79-93.

12. Peng RD. Reproducible research in computational science. *Science.* 2011;334(6060):1226-1227.

13. GitHub. Licensing a repository. Published online May 2022.

14. Bryan J. *Happy Git and GitHub for the useR.*; 2016.

15. Matthews JR, Matthews RW. *Successful Scientific Writing: A Step-by-Step Guide for the Biological and Medical Sciences.* Cambridge University Press; 2014.

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

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

18. RStudio. RStudio. Published online 2020.

A Glossary

Anchor A regular expression (regex) metacharacter that anchors a match to a position in a string. The caret (`^`) anchors to the start of the string, and the dollar sign (`$`) anchors to the end of the string.

Arguments Values provided inside the parentheses of a function to specify what the function should act on or how it should behave.

Bivariate Describes analyses or relationships involving exactly two variables.

Collapse To summarize a data set by grouping and reducing multiple observations into a single summary value per group, often using functions like `summarise()` in `dplyr`.

Complete case analysis An analysis that includes only observations with complete data for all variables in the model, excluding any rows with missing values.

Console The interactive window in RStudio (usually bottom-left) where R commands can be typed and executed immediately. 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.

Conditional Operations Conditional operations control the flow of a program by executing different blocks of code depending on whether specified conditions are TRUE or FALSE. In R this includes the `if` / `else` family, vectorised helpers such as `ifelse()`, and higher-level wrappers like `case_when()`.

Data Checks Processes that verify the accuracy, completeness, or validity of data *before* analysis. Examples include type checks (e.g., numeric vs. character), range checks (e.g., no ages below 0), completeness checks (e.g., missing-value rates), and cross-field consistency checks (e.g., start end dates).

Data frame R's primary data structure for storing tabular data. Each column is a vector, and all columns must have the same number of rows.

For loop A control structure that repeats a block of code once for each element in a sequence or vector.

Frequency count The number of times a value or category appears in a dataset. Also called a **frequency**, **count**, or **n**.

Functions A reusable block of code that performs a specific task when called. Functions take inputs (arguments) and return outputs. Functions promote modularity, abstraction, and reproducibility.

Global environment The main workspace in an R session where user-defined variables and functions are stored unless otherwise specified.

Issue (GitHub) A GitHub feature used to track tasks, bugs, enhancements, or other requests within a project.

Iteratively A method of solving a problem by repeatedly executing a set of instructions in a step-by-step manner, often using loops. This approach can improve efficiency and help prevent errors.

Join An operation that merges two tables based on shared key columns. In dplyr, functions like `inner_join()`, `left_join()`, etc., determine how unmatched rows are handled.

Key A column or set of columns that uniquely identifies each row in a data set and enables precise merging with other tables.

Lexical scoping rules A set of rules that determine which variables are accessible in a function based on where they were defined in the code hierarchy.

List-wise Deletion A method of handling missing data by excluding any row that has at least one missing value in variables of interest, leaving only complete cases.

Long A tidy data format where each row represents one measurement at one time point for one unit, and columns contain values and their corresponding identifiers (e.g., variable name or time).

MDL The Minimum Description Length (MDL) principle is a model selection principle stating that the best model is the one that minimizes the combined complexity of the model and the data encoded using that model.

Marginal totals Row and column totals in a contingency table, used to summarize the distribution of each variable and to calculate the overall total.

Mean The arithmetic mean—often denoted $\bar{x}$—is calculated by summing all values in a numeric variable and dividing by the total number of values.

Median The middle number in an ordered list of values. When the list has an even number of elements, the median is the average of the two middle numbers. Compared with the mean, the median is relatively resistant to extreme values.

Merge A base-R term (function `merge()`) for combining two data frames by matching rows on one or more *key* variables. Rows that do not match can be kept, dropped, or produce missing values depending on the arguments.

Mode The value that occurs most often in a set of data. A data set may be unimodal (one mode), multimodal (many modes), or have no mode (all values are equally frequent).

Non-standard Evaluation A programming behavior in which functions capture or modify expressions instead of immediately evaluating their values, commonly used in `tidyverse` packages.

Objects Named containers for storing data or functions in R. Common object types include vectors, lists, data frames, and functions.

Outcome variable The variable being predicted or explained in an analysis; also called the dependent variable.

Pass To supply a value or object to a function's argument when calling that function. For example, passing `2` to the `from` argument in `seq(from = 2, to = 100, by = 2)`.

Percentage A value representing a part per hundred. Calculated by dividing the number of occurrences by the total number of observations and multiplying by 100. For example, 25% means 25 out of 100.

Person-level Describes data organized at the level of the individual, where each row corresponds to one person.

Person-period Describes data structured with multiple rows per person, usually representing repeated measurements across time or conditions.

Predictor variable A variable used to explain or predict the value of another variable; also called an independent variable or explanatory variable.

Proportion A number between 0 and 1 that represents the fraction of a total. Calculated by dividing the number of occurrences by the total number of observations.

Quantifier A regular expression metacharacter that defines how many times a pattern must repeat. Common quantifiers include `*` $(0+)$, `+` $(1+)$, `?` $(0-1)$, and `{m,n}` (between m and n times).

R "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."[16] R is open source, and you can download it for free from The Comprehensive R Archive Network (CRAN) at https://cran.r-project.org/.

Range The difference between the maximum and minimum values in a data set.

Regular Expressions Compact strings that describe search patterns for text. Regular expressions (regexes) are used for tasks such as finding, extracting, replacing, or validating character data (`stringr`, `grepl()`, `gsub()`, etc.).

Repository "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."[17] A repository can exist *locally* on your computer or *remotely* on a server such as GitHub.

Return A command in a function that specifies what value to output when the function finishes running. Instead of saying, "the `seq()` function *gives us* a sequence of numbers…," we could say, "the `seq()` function *returns* a sequence of numbers."

RStudio An integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor with direct code execution, and tools for plotting, debugging, and work space management. RStudio is available as open-source desktop software and as server versions.[18]

Split - Apply - Combine A data-analysis strategy used by `dplyr::group_by()`, for example, that involves splitting data into smaller components, applying a calculation separately to each component, and then combining the individual results into a single output.

Standard deviation A measure of spread equal to the square root of the variance—the average of the squared differences between each value and the mean.

Token A basic unit in text processing, typically referring to individual pieces of data like words, numbers, or punctuation marks. Tokens include literal characters (`a`), metacharacters (`\d`), or entire character classes (`[A-Z]`).

Two-way frequency table A table that displays the joint distribution of two categorical variables, showing the frequency of each combination of categories. Also called a **crosstab** or **contingency table**.

Univariate Pertaining to a single variable. Univariate analysis describes or summarizes one variable at a time.

Variance A measure of spread calculated as the average of the squared differences between each value and the mean.

Vectorization A programming technique in which operations are applied to entire vectors (or matrices/data frames) in a single step rather than iterating element-by-element. Vectorized code in R (`x * 2`, `mean(x)`) is clear and fast because the heavy computation occurs in compiled code under the hood.

Wide A data format where repeated measures or variables are spread across multiple columns (e.g., `score_T1`, `score_T2`, `score_T3` for test scores across three terms), with one row per subject or unit.