Homage to R
Learn the Language of Data Science

Robert L. Bray

2023-02-28
To Sarah and Frida.
Preface

Large Language Models (LLMs) such as ChatGPT are ridiculously powerful. To maximize your productivity—and stay relevant—you should aim to offload as much of your workflow to these language engines as possible. In practice, this means replacing mouse clicks with keyboard strokes, as chatbots interact with your text, not your cursor. In particular, this means becoming comfortable processing and analyzing data with a computer language, such as R, which LLMs excel at reading and writing. Excel’s once-dominant point-and-click user interface is now obsolete (although most people haven’t realized this yet). For example, ChatGPT can solve nearly every question in this book in a matter of seconds. And it’s extremely good at explaining its solutions. Case in point, when I ask it to “run a regression in R and use the broom package to get its coefficient estimates and fitted values,” I get the following response:

Here is an example of how to run a linear regression in R using the \texttt{lm()} function and the broom package to get the coefficient estimates and fitted values:

```r
# Import the necessary packages
library(broom)

# Create some example data
x <- rnorm(100)
y <- x + rnorm(100)

# Fit the linear regression model
model <- lm(y ~ x)

# Use the broom package to get the coefficient estimates
coefs <- tidy(model)
print(coefs)

# Use the broom package to get the fitted values
fitted_values <- augment(model)
print(fitted_values)
```

In this example, the \texttt{lm()} function is used to fit a linear regression model with \( y \) as the response variable and \( x \) as the predictor variable. The \texttt{tidy()} function
from the broom package is then used to extract the coefficient estimates, and the `augment()` function is used to extract the fitted values. The `print()` function is used to display the results.

Note that in this example, data is generated randomly, you should replace it with your own data.

This is an incredibly good answer—I couldn't have said it better myself. So, as you see, every R programmer now has a personal genie in a bottle that knows how to implement almost every data transformation. Switching to R makes this AI genie your servant, but sticking with Excel or Google Sheets is committing to make it your competitor.

The chatbot assistant is a great equalizer: as an MBA armed with an LLM can now do almost anything that an experienced data engineer can do. The value of your MBA should thus rise, as you can now execute complex data analytics on your own, without relying on a costly layer of data scientists.

Unfortunately, due to the vagaries of the English language, we can’t outsource all of the programming to the chatbot. Just as you would never send an LLM-generated email without first checking it, you should never trust an LLM-generated program without checking it. And while debugging and validating code is easier than writing it from scratch, these final touches will still require an intimate knowledge of the computer language. Hence, we must learn R ourselves (or some analogous language). Learning R is difficult, but the returns to doing so are now orders of magnitude higher than they were in 2021, as the LLMs serve as an unprecedented force multiplier.

***

Chatbots aside, spreadsheets are simply not well-suited for serious empirical inquiry. Indeed, the difference between manipulating data with R code and manipulating it with mouse clicks is like the difference between typing and writing longhand. A pen is ideal for signing checks and writing notes to loved ones but is wholly inappropriate for writing contracts or novels. Likewise, Google Sheets is great for organizing the little scraps of data that garnish everyday life but is wholly inappropriate for industrial-scale empiricism.

First, spreadsheets cannot handle the “big data” that companies increasingly rely on. For example, doing section refwrangleLab’s lab with a spreadsheet would be impossible, as its data set comprises 16.5 million rows, whereas Excel workbooks permit only a little over a million rows, and Google Sheets permit only 40,000. In contrast, R can accommodate tables of arbitrary size: e.g., I can easily manipulate tables with 200 million observations with my modest laptop, and I can stretch that to tens of billions of observations if I migrate to an AWS server.

Second, spreadsheet analyses are difficult to automate. Whereas a well-coded statistical program can be seamlessly executed by a different person with a different input dataset, a spreadsheet analysis usually entails some manual labor, such as copying and pasting data from one sheet to another. These manual steps are time-consuming, error-prone, and not
easily transferable. For example, when I worked at Intel, nearly every industrial engineer I encountered spent hours a week manually updating statistical reports. It was a point-and-click dystopia: scores of high-level engineers mindlessly dragging data around spreadsheets, like temps hired for data entry. And before an engineer could leave their position, they had to spend weeks teaching their replacement their copy-and-paste regimen.

Third, spreadsheets cannot be read linearly. Instead, an Excel model tends to grow like crabgrass—extending helter-skelter every which way until you’re left with an ensnared knot of cells so thoroughly linked and cross-referenced that you can’t easily tease out a loose end that you can call the “beginning.” These overwrought spreadsheets resemble the wall of photos connected by a patchwork of strings that you see in police procedurals. In contrast, code always flows linearly, laying bare your analysis in the sequence in which it was performed. Hence, no matter how complex your statistical analysis is, you can always read it from start to finish. And R programs are especially easy to read—you’ll barely need any comments, as the code generally speaks for itself.

For example, most of this book’s labs are academic studies I conducted in R several years ago. And since R programs are so easy to follow, I could easily interpret what I did, despite the multi-year time gap. Furthermore, since my analysis was automated, I could replicate all of my old results simply by entering Cmd+Shift+S. Contrast this with the complex Excel model I created in my early twenties to simulate the operations of the port of LA: A year after I finished it, my boss asked me to update the simulation—but I couldn’t. After not thinking about it for a year, the sheets of my Excel workbook were as inscrutable as the sheets of a sudoku workbook. Moreover, running the simulation required some ad-hoc mousing, and I couldn’t remember what data to copy where.

Granted, some Excel gurus can produce organized, automated reports with spreadsheets. But mastering Excel is as misguided as mastering Windows 95. Indeed, most of the major design choices underlying Excel were entrenched by 1995, so a commitment to Microsoft Office is a commitment to a 1990s design sensibility. And this is true of most statistical software packages: they get stuck in the past as their users grow accustomed to a particular way of doing things. Thus, Excel, VBA, SQL, STATA, SAS, SPSS, and MATLAB are really just ossified ’90s technologies gussied up with modern veneers.

In contrast, today’s R is completely different from the R of the ’90s. In fact, it’s completely different from the R of the early 2010s. The language didn’t get locked into an antiquated scheme because it’s free, decentralized, and open-source, which means that no customers or centralized authority could veto potentially disruptive changes. Anyone with a better solution could upload it. For example, the most important symbol in R—the piping operator, %>%—which enables data to pass from function to function like a baton in a relay race—wasn’t introduced until 2014. And as soon as I finished the first draft of this book, there was a new piping operator, |> (This new pipe isn’t as convenient, so I haven’t adopted it.) For a second example, there have been three fundamental paradigm shifts in data pivoting since 2007: first with melt() and cast(), then with gather() and spread(), and now with pivot_longer() and pivot_wider(). And for a third example, I wrote and formatted this entire book in R. Since the whole point of data science is to communicate statistical findings, R enables you to report your results in every possible medium, including books.
Why have other statistical packages not figured this out? Because most of them stopped innovating decades ago.

Being free has helped R remain dynamic: no one can hold the language back because it’s not indebted to anyone. And R being free also means that there’s no red tape involved in using it. For example, I can set up an R server on AWS in minutes without asking anyone’s permission. But doing the same for MATLAB would be a headache: First, I’d have to secure funding from Northwestern University (my employer). Then I’d have to figure out how to register the server with MATLAB so they can bill me by the hour. (Yes, they charge by the hour.) And then I’d have to submit monthly expense reports from MATLAB to Northwestern for reimbursement. Using for-profit software thus adds a layer of bureaucracy to your life—it means extra paperwork every time you change computers, move jobs, or hire someone. And these statistical licenses are expensive. For example, Alteryx costs $5,195 per user per year. Hence, adopting Alteryx is equivalent to giving every data scientist on your team a $5,195 salary cut. And even if your current employer is willing to buy your team licenses, your next employer probably won’t be (especially when most serious data scientists use a free option). So, a job change would likely make your Alteryx expertise worthless. (Of course, this threat doesn’t exist for R, because no employer would begrudge you a free product.)

***

In theory, baking cupcakes should be a wholesome and enriching activity for a toddler. In practice, it’s a harrowing enterprise, comprised of flour stains, egg-white puddles, spilled sugar, spilled milk, a potty accident, an outfit change, multiple bouts of crying, at least one full-blown tantrum, an unexpected FedEx delivery, a corresponding set of barking dogs, a second potty accident (this time requiring a rinse-off in the shower), screaming when the toddler is accidentally sprayed with cold water, screaming when the preferred towel is discovered to be in the washer, and, finally, over-done cupcakes.

Data science is like this: in theory, it’s a graceful exploration, but in practice, it’s a torturous slog. To avoid explicitly writing a cliche, I’ll put it this way: the devil of data science is not in the high-level concepts. Unfortunately, most data science classes operate at the high-level plane, where all samples are orderly and all variables are accessible. In other words, they use fake data. But the difference between a real dataset and a fake dataset is like the difference between a tiger born in the jungle and one born in the zoo. In contrast, I will have you do real data science with real data. You will replicate published empirical studies, from initial data cleaning to final results. And you will soon learn that real datasets bristle with inconvenient nuances, the management of which comprises the lion’s share of the work. Indeed, every empirical project comprises two parts: (i) transform a disorderly and buggy collection of raw input tables into a clean sample and (ii) apply a statistical tool to the clean sample to translate it into a “significant” result. Most data science classes focus on the second step because it is more fun, but I will focus on the first step because it is more important. This first step is called data wrangling, and it comprises the overwhelming majority of the actual work. Since it will invariably be the bottleneck, data wrangling is the step in your data-processing workflow that you should optimize. Fast data wranglers
test more hypotheses in less time, and they spend less time cleaning data and more time analyzing it.

***

I have divided this book into lectures, commentaries, and labs. The lectures provide the information you need to do the labs. The commentaries provide miscellaneous tips, tricks, gripes, and rants. And the labs are where you do the real work. Most labs have both a guided part, which provides a clear recipe for how to perform the analysis, and an unguided part, in which you are left to your own devices. The guided sections will help you practice the low-level mechanics of writing code, and the unguided sections will help you practice the high-level task of organizing and structuring the analysis. Whereas you must execute the recipe in the guided sections, you must design the recipe in the unguided sections. The recipes I provide are basically chatbot prompt sequences. In principle, you could copy these prompts into ChatGPT to get the answers right away, but I expect you to first attempt the coding on your own so that you make all the errors necessary to learn. For the unguided lab sections, you must create your chatbot prompts. To learn how to do this, you can study how I wrote and sequenced the prompts for the unguided sections. In this sense, the guided sections will serve as “training wheels” for the unguided sections.

The labs study

- mishaps at nuclear power plants,
- case scheduling at the labor appellate court of Rome,
- JD.com, the “Amazon of China,”
- Alibaba’s package-delivery scores,
- Eversight, a company that uses AI to conduct pricing experiments,
- grocery-store inventory runs (the supply chain analog of bank runs), and
- a company that bills itself as the “Uber of house cleaning,”
- auto part defects reported to the NHTSA,
- illegal fishing on the high seas.

And in addition to these formal labs, we will casually analyze data from

- the Ultimate Fighting Championship (UFC),
- the City of Evanston Police Department,
- Netflix,
- lichess.org,
• *Wine Spectator,*

• the Citi Bike bicycle sharing program,

• Formula 1, and

• Jane Austen.

Most of the data science in this book relates to operations management. This is the discipline I know best, and it pairs well with data analytics—an explosion of new logistical datasets gives us unprecedented operational visibility. Accordingly, we can now study the intricate workings of business processes like we never could in the past.

***

I wrote this book because I believe teaching R is the most effective way for me to “level up” the MBA students at my business school (the Kellogg School of Management). Statistical programming is a concrete, marketable skill that can burnish any resume. More importantly, this skill can be put to immediate use in most walks of business. It enables you to process data more quickly, consistently, and reliably. And as capturing, storing, crunching, and transmitting information becomes ever easier, the management of businesses inexorably progresses into the management of data.

Many of my business students think that analyzing data is for underlings. This is nonsense. Saying “analyzing data is for underlings” is like saying “using the internet” is for underlings. Imagine how frustrated—how *impotent*—you would be if you couldn’t go online. Suppose that every time you wanted something looked up, you had to send a memo to a subordinate and wait an indeterminate amount of time for an answer. You would be in the futile position of relying on your underlings to explain the internet to you.

And yet many business students plan on having their underlings explain the data to them. Delegating the empirical analysis in this fashion puts a level of remove between you and your company’s data. And as being one level removed from the internet makes you essentially blind to the workings of the web, being one level removed from your company’s data makes you essentially blind to the workings of the business. So long as you avoid the data, the operation of your business will remain a vague abstraction that’s in equal parts daunting and boring. Yes *boring,* because it’s impossible to satisfyingly engage with something intangible, described to you second-hand, in jargon you barely understand.
# Contents

1 Data Structures  
1.1 Introduction .................................................. 1  
1.2 Lecture ......................................................... 2  
   1.2.1 RStudio ................................................ 2  
   1.2.2 Variables .............................................. 5  
   1.2.3 Vectors ................................................ 9  
   1.2.4 Logicals, Doubles, and Integers ..................... 14  
   1.2.5 Character Strings, Dates, and Factors ............... 19  
   1.2.6 Tibbles ................................................. 30  
   1.2.7 Saving and Loading Data ............................. 41  
1.3 Lab: Workspaces .............................................. 45  
1.4 Commentary .................................................. 59  
   1.4.1 <- vs. = ............................................... 59  
   1.4.2 Automating Code ...................................... 60  
   1.4.3 Glossary of stringr Functions ....................... 60  
1.5 Solutions .................................................... 65  

2 Functions ..................................................... 71  
2.1 Introduction ................................................ 71  
2.2 Lecture ....................................................... 72  
   2.2.1 Calling Functions .................................... 72  
   2.2.2 Conditional Statements .............................. 77  
   2.2.3 Functional Assembly Lines ......................... 81  
   2.2.4 Formatting Code ..................................... 89
2.2.5 Defining Functions ........................................... 91
2.2.6 Bracketed Expressions ...................................... 93
2.2.7 \texttt{mutate()} .............................................. 101
2.2.8 \texttt{summarise()} .......................................... 106
2.2.9 \texttt{group_by()} ........................................... 112
2.2.10 \texttt{filter()} ............................................... 117

2.3 Lab: Italian Judiciary ........................................... 121
2.3.1 Overview ...................................................... 121
2.3.2 Data .......................................................... 122
2.3.3 Clean ......................................................... 123
2.3.4 Analyze ....................................................... 125
2.3.5 Explore ......................................................... 128

2.4 Commentary ..................................................... 129
2.4.1 Data Factory Layout ........................................ 129
2.4.2 Nested Functions ............................................ 131
2.4.3 Specialized Functions ....................................... 132
2.4.4 Define Multivariate Functions ............................. 134
2.4.5 Download Packages from CRAN ............................ 136
2.4.6 Glossary offorcats Functions ............................. 139
2.4.7 Standardize Objects to Standardize Code ............... 142

2.5 Solutions .......................................................... 144

3 Pivot and Plots .................................................... 159
3.1 Introduction ..................................................... 159
3.2 Lecture ........................................................ 161
3.2.1 Data ........................................................ 161
3.2.2 \texttt{ggplot()} .............................................. 163
3.2.3 Case Study: Best Places to Gamble ....................... 173
3.2.4 Case Study: Knockouts Occur Earlier than Submissions 175
3.2.5 Updating \texttt{ggplot()} Data .............................. 176
3.2.6 Case Study: Formula 1 Lap Times ......................... 179
3.2.7 \texttt{pivot_longer()} and \texttt{pivot_wider()} ............... 180
Chapter 1

Data Structures

1.1 Introduction

The R language comprises nouns, which are called objects, and verbs, which are called functions. We will cover objects in this chapter and functions in the next chapter. Objects are “things” that we would like the computer to represent. For example, the number 2 and the text string "There's time for everything except the things worth doing" are objects. And functions are “actions” that we would like to perform on objects. For example, the log() function takes the logarithm of a number and the str_length() function counts the number of characters in a text string, so that log(2) = 0.6931472 and str_length("There's time for everything except the things worth doing") = 57.

R objects can have names. For example, the object 3.14159265358979 has the name pi and the object c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec") has the name month.abb. We’ll cover these named objects, which are known as variables, in section 1.2.2. And in section 1.2.3 we’ll cover list objects called vectors. For example, month.abb is a vector that comprises twelve character strings. A character string is a basic data type—an intrinsic sort of data. Examples of other data types are integers, dates, and logicals (i.e., TRUE and FALSE). In sections 1.2.4 and 1.2.5 we’ll cover R’s primary data types: logicals, doubles, integers, character strings, factors, and Dates (which we always write with a capital D). These are the basic building blocks of data in R—if there were a period table of data elements it would comprise these six types. Finally, we’ll end the lecture, in section 1.2.6, with a discussion of tibbles, which are tables of data (like Excel worksheets). Tibbles are the primary data structures of R.

After the lecture, we have a lab in section 1.3. This lab will set up the work environments we’ll use for the rest of the book. And finally I’ll conclude, in section 1.4, with a few miscellaneous points. In section 1.4.1 I’ll discuss R’s two different assignment operators. In section 1.4.2 I’ll cover the right way to load and save tibbles. In section 1.4.2 I’ll discuss the importance of writing self-sufficient code—code that can be run in full without human intervention. And in section 1.4.3 I’ll provide a glossary of string-manipulation functions.

To prepare for the lecture in the next section, download and install R and RStudio on your
computer. Here’s¹ a good resource that will walk you through this. I will begin the lecture with a brief primer on RStudio. But for more on this, see the “RStudio IDE Cheatsheet,” which you can find on this page², and see the additional links provided here³.

1.2 Lecture

1.2.1 RStudio

RStudio is different from R. The latter is the programming language and the former is the graphical user interface that we use to program in this language. You could write and run R code without RStudio, but that would be eccentric—the interface is so well-designed that almost all R programmers use it.

The RStudio interface is composed of four boxes: the console pane, the files pane, the environment pane, and the source pane (if any of these panes are collapsed use your mouse to expand them out).⁴ The console pane should have a “Console” tab written somewhere along the top. You can submit little snatches of code here and R will execute them immediately. This pane isn’t for proper programming; rather it’s for ephemera—quick and cheap queries that you don’t want saving. For example, run the following in your console (while connected to the internet):

```r
install.packages("tidyverse")
```

Now follow any prompts that come up and R will download the tidyverse to your computer. As the name suggests, the tidyverse is a universe of tidy functions that streamline and simplify the mundane aspects of data science. It is the heart of R. However, the code above simply downloaded these tidy functions to your computer; it didn’t load them into your current workspace. The following code adds these tidy functions to your workspace, so that you can use them:

```r
library(tidyverse)
library(lubridate)
```

Confusingly, all of the tidyverse functions pertaining to dates are cordoned off in the separate lubridate package (which got downloaded when you executed `install.packages("tidyverse")`). So we load this auxiliary package, in addition to the

---
¹https://learnr-examples.shinyapps.io/ex-setup-r/#section-welcome
²https://rstudio.com/resources/cheatsheets/
³https://education.rstudio.com/learn/beginner/
⁴I recommend moving your console from below the source pane to the right of the source pane, because these panes both require much more vertical space than the environment and files panes. You can move the console pane by going to RStudio/Preferences/Pane Layout.
core tidyverse package, whenever we plan to work with dates. Henceforth, I will assume that you have the tidyverse and lubridate packages loaded in your workspace. (And to ensure our workspaces look the same, please do not load any other packages with `library()` until I ask you to do so.)

For an example of a lubridate function enter `now()` into your console. Or for another example enter `DOB <- "11/06/1983"; today() - mdy(DOB)`, except replace my birth date with your date of birth. Or for a non-lubridate tidyverse example, enter `str_which(letters, "[aeiou]")` to get the position of the vowels in the English alphabet.

Now press the up arrow a few times (while your cursor is in the console pane). Doing so should summon the commands that you just executed. (You’ll soon find this up-arrow trick is indispensable.) Now press Ctrl+Up, if you’re using Windows or Linux, or Cmd+Up, if you’re using a Mac. Doing so should bring up your entire command history.

The files pane should have the word “Files” written somewhere above it. This is where you can view and access the files on your computer. This file navigation window is useful because most R projects span multiple files. For example, I wrote this book in R and saved every section in a different file. The files pane also has a “Plots” tab, which is where any graphs you make will appear. Enter `ggplot() + aes(x = rnorm(10^5)) + geom_density()` into your console and you’ll see what I mean. There is also a “Help” tab, which is where R will fulfill your documentation requests. For example, enter `vignette("paper")` in the console to get the tidyverse manual, and now enter `?str_count` to see the `str_count()` documentation (you can write `?` in front of any function to see its documentation).

The environment pane has an “Environment” tab above it. This tab provides a brief snapshot of all the R objects that are defined in your workspace. For example, one object in my environment is `DOB = "11/06/1983"`. Or for another example, enter `x <- 1` into your console and you’ll see this variable appear in the environment list. Now enter `rm(x)` to remove this variable from your workspace.

Finally the source pane is where you’ll write your serious source code—the code you want to save as a formal program. Code written in the source pane is easier to save, modify, and rerun than code written in the console. Indeed, the difference between the source pane and the console pane is like the difference between Microsoft Word and Twitter: you could execute your entire empirical analysis line by line in the console, but that would be like writing a novel one tweet at a time.

Copy the following into your source pane:

---

5More technically, I wrote this book in R Markdown, which is a slick way to make reports that incorporate R code and outputs. Even more technically, I used the bookdown package, which is a way to write books with R Markdown.

6By default the source pane sits on top of the console, but I recommend going to RStudio/Preferences/Pane Layout and putting it to the side of the console, so that they don’t compete for vertical space. The source pane and console both need plenty of vertical space, but the environment and files panes do not.
```r
library(tidyverse)
library(lubridate)

#I'll next create some raw data
ussr <-
  tribble(
    ~socialist_republic, ~pop_1989, ~area_1991,
    "Armenian Soviet S.R.", 3, 30,
    "Azerbaijan Soviet S.R.", 7, 87,
    "Byelorussian Soviet S.R.", 10, 208,
    "Estonian Soviet S.R.", 2, 45,
    "Georgian Soviet S.R.", 5, 70,
    "Kazakh Soviet S.R.", 17, 2717,
    "Kirghiz Soviet S.R.", 4, 199,
    "Latvian Soviet S.R.", 3, 65,
    "Lithuanian Soviet S.R.", 4, 65,
    "Moldavian Soviet S.R.", 4, 34,
    "Russian Soviet Federative S.R.", 147, 17075,
    "Tajik Soviet S.R.", 5, 143,
    "Turkmen Soviet S.R.", 4, 488,
    "Ukrainian Soviet S.R.", 52, 604,
    "Uzbek Soviet S.R.", 20, 447
  )

ussr %>%
  mutate(
    pop_density = pop_1989/area_1991,
    socialist_republic =
      str_remove(socialist_republic, " Soviet")
  ) %>%
  ggplot(
    aes(
      x = percent_rank(area_1991),
      y = percent_rank(pop_density),
      label = socialist_republic
    )
  ) +
  geom_point() +
  geom_text(hjust=-.1)
```

We usually start a program with `library(tidyverse)` and `library(lubridate)` so it'll work in workspaces that don’t have the tidyverse and lubridate functions loaded. Below these two lines of code is the comment `#I'll next create some raw data`. This line is not code—R ignores all text that follows a hashtag. Rather, it and the second comment—`#Remove`
"Soviet" to simplify plot—are just little notes I’ve added to help the (human) reader understand what the code does.

The block of code that follows the first comment defines a table of data and names it `ussr`. To execute this code chunk, click anywhere on it (e.g., on the word "Lithuanian") and then press either Ctrl+Enter or Cmd+Enter, depending on whether you’re using a Mac. After this, your “Environments” tab should list `ussr` as a table comprising “15 obs. of 3 variables.” You can view this table by entering `ussr` in your console.

Now execute the second block of code by again clicking somewhere on it and pressing Ctrl+Enter or Cmd+Enter. The result should be a weird Soviet plot. Now delete this plot by clicking on the broom icon just above it. Now recreate this plot by typing Ctrl+Shift+P (Windows and Linux) or Cmd+Shift+P (Mac). This keyboard shortcut tells R to rerun the bit of code you most recently executed. It’s an essential hotkey—memorize it now or give up on learning R altogether.

Now let’s run our Soviet plot code from scratch. Delete the plot with the broom icon, click on Session/Clear Workspace, and then click on Session/Restart R. Now click anywhere on the source pane and press Ctrl+Shift+Enter (Windows and Linux) or Cmd+Shift+Enter (Mac) to run the entire R script. This is how you properly run an R program.

Finally, clear out your workspace and run the code again by copying it from the source pane to the console pane. This is not how you should run your R code! However, you will get the same result because R treats code executed from the console the same as code executed from the source pane. In fact, the only real difference is that it’s harder to save what you’ve run in the console, whereas you can save a program that you’ve written in the source panel with File/Save.

### 1.2.2 Variables

We can use R as a scientific calculator:

```r
sqrt(100)
```

> [1] 10

```r
cos(pi)
```

> [1] -1
And we can add comments, which the computer ignores, with the pound sign:

```
sqrt(100) #everything following a hashtag is a comment
```

```
> [1] 10
```

```
#the cosine of 3.141593 is:
cos(pi)
```

```
> [1] -1
```

```
2^8 #this equals 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
```

```
> [1] 256
```

In general, R executes one intelligible bit of code at a time. For example, the first code chunk, below, yields an error, but the second and third do not:

```
#code chunk 1
1 + 2

* 3
```

```
#code chunk 2
1 + 2 *

3
```

```
#code chunk 3
(1 + 2

* 3)
```
The first code chunk yields an error because $1 + 2$ is a comprehensible line of code; accordingly, when it receives this code chunk, R first prints $1 + 2 = 3$ to the screen, and then it attempts to execute $\ast 3$, which is nonsense, since the multiplication sign needs a number on both the left and the right. In contrast, the second code chunk does not yield an error because $1 + 2 \ast$ is not a comprehensible line of code, due to the dangling $\ast$; accordingly, R interprets the second code chunk as $1 + 2 \ast 3$, which induces it to print 7 to the screen. Likewise, $(1 + 2$ isn’t a comprehensible line of code, due to the unclosed parentheses; accordingly, R interprets the third code chunk as $(1 + 2 \ast 3)$, which induces it to print 7 to the screen.

We can give numbers names so we can refer to them in shorthand. For example, the US national debt on 09/30/2012 was $16,066,241,407,385. The following uses assignment operator <- to give this number the name debt_2012:¹

```
debt_2012 <- 16066.241407385  #measured in billions
```

Using R parlance, we say that the code above creates a variable called debt_2012 that equals the number 16066241407385. The following code analogously defines variables debt_2016 and debt_2020 as the US national debt on 09/30/2016 and 09/30/2020:

```
debt_2016 <- 19573.444713936
deb2020 <- 26945.391194615
```

Henceforth, we can use variables debt_2012, debt_2016, and debt_2020 and numbers 16066241407385, 19573444713936, and 26945391194615 interchangeably. For example, the following two expressions are equivalent:

```
(debt_2020 - debt_2016) / (debt_2016 - debt_2012)
```

```
> [1] 2.101944
```

```
(26945.391194615-19573.444713936) / (19573.444713936-16066.241407385)
```

```
> [1] 2.101944
```

¹We could have named it debt.2012, but you should never include a . in a variable name. There’s a technical reason for this—R programmers usually reserve the . symbol for object methods. But there’s also a more practical reason: if you double-click on debt_2012 the whole word always gets highlighted, but if you double-click on debt.2012 then sometimes only debt or only 2012 gets highlighted.
However, the former expression is easier to interpret, because descriptive labels such as `debt_2016` are more telling than long strings of digits such as 19573444713936.

The following defines `pop_2012`, `pop_2016`, and `pop_2020` as the US population in 2012, 2016, and 2020:

```r
pop_2012 <- .3138  #measured in billions
pop_2016 <- .3229
pop_2020 <- .3307
```

And the following redefines variables `debt_2012`, `debt_2016`, and `debt_2020` to equal the per capita debt:

```r
debt_2012 <- debt_2012 / pop_2012
debt_2016 <- debt_2016 / pop_2016
debt_2020 <- debt_2020 / pop_2020
```

For example, we now have `debt_2020 = 81489.72`. The code above illustrates two points. First, we can redefine variables—their values are not permanent. And second, we can define a variable's new values in terms of its old values. For example, the third line of the code above has `debt_2020` to both the left and right of `<-`. And that's allowed! Indeed, you should read this expression as “new debt_2020 equals old debt_2020 divided by pop_2020.”

The following exercise gives you some practice working with variables.

**Exercise 1.1.** Anticipate what the following three chunks of code will output and then execute the code to confirm your answer.

```r
#chunk 1:
x <- 1
x <- x + x + x
x

#chunk 2:
x <- 10
y <- x^2
z <- sqrt(y) #i.e., z equals the square root of y
z

#chunk 3:
x <- 2
x <- x^x
x <- x^x
x
```
Now instead of 2012, 2016, and 2020, suppose we wanted to record the national debt and population every year going back to 1950. We could give each debt and population figure its own variable, defining

```
debt_1950 <- 257            #measured in billions of dollars
pop_1950 <- .1523          #measured in billions of people

debt_1951 <- 255
pop_1951 <- .1549

debt_1952 <- 259
pop_1952 <- .0792

# ...

debt_2020 <- 26950
pop_2020 <- .3299
```

But scattering the information across so many distinct variables makes it difficult to work with. For example, reexpressing `debt` in per capita terms would require the following 71 transformations:

```
debt_1950 <- debt_1950 / pop_1950

debt_1951 <- debt_1951 / pop_1951

# ...

debt_2020 <- debt_2020 / pop_2020
```

A more practical solution is to store the debt and population data in vectors. A vector is an ordered list. We can create such vectors with the concatenation function `c()`:

```
debt <-
c(
    257, 255, 259, 266, 271, 274, 273, 271,
    276, 285, 286, 289, 298, 306, 312, 317,
    320, 326, 348, 354, 371, 398, 427, 458,
    475, 533, 620, 699, 772, 827, 908, 998,
    1142, 1377, 1572, 1823, 2125, 2350, 2602, 2857,
)
The code above defines `debt` as a vector comprising our 71 debt figures and defines `pop` as a vector comprising our 71 population figures. Note that in addition to being vectors, `debt` and `pop` are variables. A variable is any object we’ve assigned a name to with `<-`. Storing all the debt values in the same place enables us to transform them in one fell swoop. For example, we can now redefine all the `debt` values in per-capita terms with a single `<-`:

```r
debt <- debt / pop  #divide nth value of debt by nth value of pop
```

And we can inspect the result in several ways:

```
head(debt)
#first six values

> [1] 1687.459 1646.223 1643.401 1660.424 1662.577 1651.597

tail(debt, 20)
#last 20 values

> [1] 20375.44 21655.08 23381.59 25201.50 26846.82 28508.71 29907.04 32982.57
> [9] 38828.08 43840.93 47464.70 51210.96 52974.68 55984.92 56612.60 60607.00
> [17] 62307.69 65870.83 69226.08 81691.42
```

*The keyboard shortcut for writing the `<-` symbol is “Alt -” (Windows and Linux) or “Option -” (Mac).*
In addition to using concatenation function `c()`, there are several other ways to create vectors:
-7:4  #integers between -7 and 4

> [1] -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4

rep(pi, 5)  #repeat pi 5 times


seq(1, 10)  #sequence of numbers between 1 and 10

> [1] 1 2 3 4 5 6 7 8 9 10

seq(5, 100, by = 8)  #numbers between 5 and 100, counting by 8

> [1] 5 13 21 29 37 45 53 61 69 77 85 93

seq(-.1, .3, length.out = 5)  #5 evenly spaced numbers between -.1 and 3

> [1] -0.1 0.0 0.1 0.2 0.3

rep(seq(-.1, .3, length.out = 5), 3)  #3 copies of previous vector

> [1] -0.1 0.0 0.1 0.2 0.3 -0.1 0.0 0.1 0.2 0.3 -0.1 0.0 0.1 0.2 0.3

**Exercise 1.2.** Anticipate what the following three chunks of code will output and then execute the code to confirm your answer.
Exercise 1.3. Define `fine_grid` as a vector of a million equally spaced points between -1 and 1.

Exercise 1.4. Calculate the `mean()` of the `log()` of the integers between 1 and 100.

Exercise 1.5. Use `prod()` to determine which is larger: the product of all numbers between 1 and 100 (inclusive) or the product of all the numbers between 100 and 170 (inclusive).

Exercise 1.6. The `lag()` function shifts all the elements of a vector over by one position. For example, `lag(c(1, 2, 3, 4)) = c(NA, 1, 2, 3)`, where `NA` is a “not applicable” symbol that indicates an empty slot. Use this function to define `debt_change` as the vector of year-over-year debt changes.

- The first three values of `debt_change` should be `c(1687.459, 1646.223, 1643.401) - c(NA, 1687.459, 1646.223) = c(NA, -41.236, -2.822)`.

Exercise 1.7. Use `lag()` to define `debt_growth` as the vector of year-over-year fractional debt changes.

- If `x` and `y` are vectors of equal length, then `x / y` is a corresponding vector of ratios, calculated element by element. For example: `(1:3) / (11:13) = c(1/11, 2/12, 3/13).

- The first three values of `debt_growth` should be `(c(1687.459, 1646.223, 1643.401) - c(NA, 1687.459, 1646.223)) / c(NA, 1687.459, 1646.223).

Exercise 1.8. A Cauchy random variable is one standard normal random variable divided by another standard normal random variable. We will use this fact to generate and plot 30 Cauchy random variables.

- Use `rnorm(30)` to define `x` and `y` as vectors of 30 standard normal random variables.
- Define `z` as a vector whose nth element equals the nth element of `x` divided by the nth element of `y`.
- Use `hist()` to plot the distribution of `z`.
  - The plot may look weird because the Cauchy distribution has very fat tails.
1.2.4 Logicals, Doubles, and Integers

Here’s a weird fact about R: the language has different types of numbers. Specifically, it has three basic number types: integers, doubles, and logicals. An integer is a whole number like 0, 5, 3,000, or -132,432,345. A double is a decimal number like 0.5, -10000.00001, or 3.141593. And a logical is either TRUE or FALSE.

Now, in the normal universe TRUE and FALSE are not numbers, but in the R universe they are. Indeed, you can plug values TRUE or FALSE into any mathematical expression and R will construe the TRUE terms as ones and the FALSE terms as zeros. For example, TRUE + TRUE + FALSE evaluates to 1 + 1 + 0 = 2 and 7 * FALSE^2 + ((TRUE + FALSE) / (TRUE + TRUE))^(2 +TRUE) evaluates to 7 * 0^2 + ((1 + 0) / (1 + 1))^2 + 1 = 0.125. The TRUE = 1 and FALSE = 0 convention is useful because it enables us to apply sum() and mean() to vectors of logicals to get the number of elements that are TRUE and the fraction of elements that are TRUE.

Let me illustrate with a concrete application: If debt_change <- debt - lag(debt) is a vector of year-over-year debt changes (see exercise 1.6), then debt_increase <- debt_change > 0 is a vector of logicals that indicates whether the debt increased in a given year (i.e., whether the debt change was positive). For example, compare the first five values of debt_change with the first five values of debt_increase:

```
head(debt_change, 5)
```

> [1] NA -41.235593 -2.822355 17.023454 2.152218

```
head(debt_increase, 5)
```

> [1] NA FALSE FALSE TRUE TRUE

The NA terms indicate missing data: since we don’t observe what the debt was in 1949, we don’t know how it changed in 1950. Now from debt_increase we can derive the number of years between 1951 and 2020 in which the per-capita debt increased and the proportion of years in which the debt increased:

```
#Number of values in debt_increase that are TRUE:
sum(debt_increase, na.rm = TRUE)
```

> [1] 61
#Fraction of values in debt_increase that are TRUE:
mean(debt_increase, na.rm = TRUE)

> [1] 0.8714286

In the expressions above, the na.rm = TRUE option tells R to ignore the NA value at the beginning of debt_increase.\(^9\) The preceding example used the greater-than symbol > to convert numeric vector debt_change to logical vector debt_increase. We can also create logicals with the greater-than-or-equal-to symbol \(\geq\), exactly-equal-to symbol ==, less-than-or-equal-to symbol \(\leq\), less-than symbol <, and not-equal-to symbol !=. For example, run the following in your console:

```r
x <- 1:5
x > 3
x >= 3
x == 3
x <= 3
x < 3
x != 3
```

Note that the exactly-equal-to symbol comprises two equals signs. Indeed, there’s a big difference between = and ==: the former is used for assignments, much like <-, and the latter is used to test whether two things are the same.

We can also determine whether all() or any() of a collection of logicals is TRUE:

```r
#Are ANY of the values TRUE?
any(TRUE, FALSE, FALSE, FALSE)
```

> [1] TRUE

---

\(^9\)The na.rm = TRUE option tells R to remove the NA terms; it does not tell R to replace the NA terms with TRUE values. For example, sum(c(TRUE, TRUE, NA), na.rm = TRUE) is equivalent to sum(c(TRUE, TRUE)), not to sum(c(TRUE, TRUE, TRUE)). Hence, you should consider this TRUE as your answer to the question, “True or false: you would like me to remove the missing terms before computing the sum?”, and not as the answer to the question, “What would you like me to replace the NA terms with?” Use the replace_na() function if you’d like to replace NA terms with another value.
Alternatively, we can combine logicals with the and, or, and not operators, & , | , and ! :

- $x \land y = \text{TRUE}$ if both $x = \text{TRUE}$ and $y = \text{TRUE}$, and otherwise equals FALSE.
- $x \lor y = \text{TRUE}$ if $x = \text{TRUE}$ or $y = \text{TRUE}$, and otherwise equals FALSE.
- $! x = \text{TRUE}$ if $x = \text{FALSE}$, and otherwise equals FALSE.

Note that $\text{all}(x, y, z) = x \land y \land z$ and $\text{any}(x, y, z) = x \lor y \lor z$.

Another way to generate logicals is to test whether the elements of one vector are %in% another vector:

#Which values of c(-1, 0, 2) are in 1:30?
c(-1, 0, 2) %in% 1:30

> [1] FALSE FALSE TRUE
Exercise 1.9. Which of the following expressions are TRUE?

\[
\begin{align*}
5 \text{ %in% } 0:10 \\
(5 \text{ %in% } 0:10) & \text{ & all}(3 < 0:9) \\
(2+2 != 5) & \text{ | } ((5 \text{ %in% } 0:10) & \text{ all}(3 < 0:9)) \\
!(2+2 != 5) & \text{ | } ((5 \text{ %in% } 0:10) & \text{ all}(3 < 0:9))
\end{align*}
\]

Exercise 1.10. Let \( x \) and \( y \) be numbers. The following code returns \( x \) if \( x \) equals \( y \) and returns 0 otherwise:

\[
(x==y) \times x
\]

Create an expression that return \( x \) if \( x \) equals \( y \) and returns \( y \) otherwise.

Exercise 1.11. Let \( x \), \( y \), and \( z \) be numbers. Write an expression that returns TRUE if any() of these numbers equals zero and returns FALSE otherwise.

The other numerical types are doubles and integers. The primary difference between them is that a double can have a fractional part, but an integer cannot. For example:

```r
as.integer(2.6)
```

\[
> [1] 2
\]

By default, R sets all numbers to doubles:
To define a number as an integer, use \texttt{as.integer()} or write an “L” after it:

\begin{verbatim}
> is.double(2L)
[1] FALSE

> is.integer(2L)
[1] TRUE
\end{verbatim}

Admittedly, this “L” convention is a bit awkward: e.g., \texttt{year <- 1914} appears more natural than \texttt{year <- 1914L}. But expressing integers as doubles can expose you to rounding errors. For example, if you tell the computer to save the number 10 as a double, then you’re really telling it to save something like 10.0000000000000, and this long string of trailing zeros will get corrupted unless all mathematical operations are performed to within a 0.00000000000001 tolerance (which isn’t always the case). For example, consider the following:

\begin{verbatim}
> number_killed <- 10 + 10 + .1 - 10 - .1

> number_killed
[1] 10
\end{verbatim}

Variable \texttt{number_killed} should be exactly ten. And it looks like it is:

\begin{verbatim}
> number_killed
[1] 10
\end{verbatim}

But it’s not:
The problem is that adding and subtracting by 10 + .1 introduces just enough of a rounding error to make `number_killed` distinguishable from ten. Specifically, `number_killed` exceeds ten by 0.000000000000001776:

```r
number_killed > 10
```

> [1] TRUE

And this small error can lead to big problems. For example, suppose a vaccine is “in compliance” if it kills ten or fewer people. In this case, our vaccine should be in compliance—having killed exactly ten people—but will be flagged as out of compliance since `number_killed <= 10 = FALSE`.

Expressing `number_killed` as an integer solves the problem:

```r
number_killed <- as.integer(10 + 10 + .1 - 10 - .1)

number_killed > 10
```

> [1] FALSE

To avoid this rounding issue, you should save all whole numbers as integers, using doubles only when you need the decimal expansion.10

1.2.5 Character Strings, Dates, and Factors

So now we’re set for numbers, but not all data are numeric. For example, we may want the computer to store the names of the first dozen Civil War battles recognized by the Civil War Sites Advisory Commission (CWSAC):

---

10This is a do-as-I-say-not-as-I-do recommendation, as I violate this rule quite often.
This code sets variable `battle_name` equal to a vector of twelve `character strings`. For example, the third element of `battle_name` is character string "Battle of Aquia Creek". As you see, a character string is just a bit of text. R has some useful built-in character-string vectors:

`letters`

```r
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

`LETTERS`

```r
> LETTERS
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

`month.abb`

```r
> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```
month.name

> [1] "January"  "February"  "March"    "April"    "May"       "June"
> [7] "July"      "August"   "September" "October"  "November" "December"

But really, a character string could be anything. For example, "Break any of these rules sooner than say anything barbarous", "Population of China: 1.393 billion (2018), "*(&#$*#^@*#()@)!", and "123" are all character strings. The last expression looks like a number, but the quotation marks tell the computer to treat it as non-numeric text. For example, the following yields an error:

"123" + 1

> Error in "123" + 1: non-numeric argument to binary operator

We can’t add one to "123" for the same reason that we can’t add one to "Battle of Carthage": character strings are not numbers and are thus not compatible with algebraic operations. However, we could perform this addition if we first turned the character string to an integer:

as.integer("123") + 1

> [1] 124

The %in% operator also applies to character strings:

c("Battle of Hoke's Run", "Battle of Carthage", "SHILOH") %in% battle_name

> [1] TRUE  TRUE FALSE

battle_name %in% c("Battle of Hoke's Run", "Battle of Carthage", "SHILOH")

> [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE  TRUE FALSE FALSE FALSE FALSE
But most string manipulation functions come from the stringr package (which you passively loaded when you called `library(tidyverse)`). We will use the following stringr functions:\(^{11}\)

- `str_c()`,
- `str_count()`,
- `str_detect()`,
- `str_extract()`, `str_extract_all()`,
- `str_flatten()`,
- `str_length()`,
- `str_locate()`,
- `str_remove()`, `str_remove_all()`,
- `str_replace()`, `str_replace_all()`,
- `str_to_lower()`, `str_to_upper()`, `str_to_title()`,
- `str_split()`,
- `str_squish()`,
- `str_sub()`, and
- `str_which()`.

Starting each function with “str” was a great design choice, as it unifies what would otherwise appear a disjoint collection of functions. This unification is useful, because the notation is standardized across stringr functions. Hence, the “str” prefix serves as a visual flag that indicates that all the usual stringr conventions apply.

I’ll introduce these functions to you in a gradual, organic fashion over the course of the book. But I also list their definitions in section 1.4.3, for quick reference.

I’ll now give you some quick examples of stringr functions. First, the following replaces the NA terms of a character-string vector with "", the character string that comprises zero characters:

```r
define function: str_replace_na
define arguments: c("a", "b", NA, "d", NA), ""
define call: str_replace_na(c("a", "b", NA, "d", NA), "")
define output: > [1] "a" "b" "" "d" ""
```

Second, the following outputs a logical vector that identifies the elements of `battle_name` that contain the substring " or " (i.e., the battles with multiple names):

```r
define function: str_detect
define arguments: battle_name, " or "
define call: str_detect(battle_name, " or ")
define output:...
```

\(^{11}\)As you see, all the stringr functions begin with prefix `str`.
Third, the following removes all the instances of "Battle of " from `battle_name`:

```r
battle_name <- str_remove_all(battle_name, "Battle of ")
head(battle_name)
```

> [1] "Fort Sumter"      "Sewell's Point"
> [3] "Aquia Creek"      "Philippi (West Virginia)"
> [5] "Big Bethel"        "Boonville (Missouri)"

And fourth, the following concatenates character string "Battle ", numeric vector 1:12, character string ": ", and character-string vector `battle_name`:

```r
battle_name <- str_c("Battle ", 1:12, ": ", battle_name)
head(battle_name, 3)
```

> [1] "Battle 1: Fort Sumter" "Battle 2: Sewell's Point"
> [3] "Battle 3: Aquia Creek"

Moving on, we could save the battle start and end dates as character strings:

```r
battle_start <- c("April 12, 1861", "May 18, 1861", "May 29, 1861",
                   "June 3, 1861", "June 10, 1861", "June 17, 1861",
                   "July 2, 1861", "July 5, 1861", "July 11, 1861",
```
"July 18, 1861", "July 21, 1861", "August 10, 1861"
)
battle_end <-
c(
  "April 13, 1861", "May 19, 1861", "June 1, 1861",
  "June 3, 1861", "June 10, 1861", "June 17, 1861",
  "July 2, 1861", "July 5, 1861", "July 11, 1861",
  "July 18, 1861", "July 21, 1861", "August 10, 1861"
)

But it would be better to save them as Date variables, to make them compatible with R’s calendar functions.\textsuperscript{12} We can convert these calendar string vectors to Date vectors with mdy():

\begin{verbatim}
battle_start <-
  mdy(battle_start)

battle_end <-
  mdy(battle_end)
\end{verbatim}

Note, we use \texttt{mdy()} because our initial character strings expressed dates with a month/day/year format; in contrast, we would have used \texttt{dmy()} had they followed a day/month/year format, \texttt{ymd()} had they followed a year/month/day format, etc.

And now that they represent Dates, we can perform all sorts of temporal operations on \texttt{battle_start} and \texttt{battle_end}. For example, we can difference these vectors to calculate the battle durations:

\begin{verbatim}
battle_end + days(1) - battle_start
\end{verbatim}

\begin{verbatim}
> [1] 2 2 4 1 1 1 1 1 1
\end{verbatim}

We add \texttt{days(1)} to \texttt{battle_end} so that, e.g., a battle that starts and ends on August 10, 1861 is recorded as lasting one day rather than zero days. Note we couldn’t perform the computation above if \texttt{battle_start} and \texttt{battle_end} were character-string vectors—subtracting character string "May 29, 1861" from character string "June 1, 1861" makes as much sense as subtracting "Donald Duck" from "Bugs Bunny".\textsuperscript{12}

\textsuperscript{12}We refer to Date variables with a capital “D” so that we can use \texttt{"D"} as a shorthand for Dates and \texttt{"d"} as a shorthand for doubles.
Functions `mdy()` and `days()` are from the lubridate package, which provides a function for almost anything you would want to do with a Date. For example, we can get the number of days since these battles ended with

```r
today() - battle_end
```

> Time differences in days
> [1] 59125 59089 59076 59074 59067 59060 59045 59042 59036 59029 59026 59006

We can get the date of the first day of the week in which a battle took place with

```r
floor_date(battle_start, "week")
```

> [1] "1861-04-07" "1861-05-12" "1861-05-26" "1861-06-02" "1861-06-09"
> [6] "1861-06-16" "1861-06-30" "1861-07-07" "1861-07-14"
> [11] "1861-07-21" "1861-08-04"

And we can get the day of the week in which the battles ended with

```r
day_of_week <-
  wday(battle_end, label = TRUE)
day_of_week
```

> [1] Sat Sun Sat Mon Mon Mon Tue Fri Thu Thu Sun Sat
> Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat

All these functions are from lubridate.

In the last example above, `day_of_week` looks like a vector of character strings, but it’s actually a vector of `factors`. A factor variable is a character variable that’s restricted to a specific set of values, or “levels.” For example, `day_of_week` has seven valid levels:

```r
levels(day_of_week)
```

> [1] "Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"
Constraining the elements of `day_of_week` to these seven values imposes a useful safeguard on our data. For example, we could change the first element of `day_of_week` to "Wed", which is a valid level, but not to "Wwed", which isn’t a valid level. Hence, the factor structure makes such spelling mistakes impossible.

The other benefit of factor variables is that they can have non-alphabetical orderings. For example, the expression `Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat` in the output above indicates that "Sun" is the first level, "Mon" is the second level, etc. And any plot we make with `day_of_week` will honor this ordering:

```r
ggplot() +
aes(x = day_of_week) +
geom_bar()
```

![Bar chart with correct order]

In contrast, if we treat the elements of `day_of_week` as character strings, then R arranges the bars *alphabetically*, which makes for a pretty lousy plot:

```r
ggplot() +
aes(x = as.character(day_of_week)) +
geom_bar()
```

![Bar chart with alphabetical order]
While the United States, Canada, and Japan treat Sunday as the first day of the week, the international standard ISO 8601 specifies Monday as the first day of the week. To make our plot ISO compliant, we’ll use `fct_relevel()` to reorder its levels from "Sun", ..., "Sat" to "Mon", ..., "Sun".\[13\]

```r
day_of_week <-
fct_relevel(
  day_of_week,
  "Sun",
  after = Inf
)

day_of_week

> [1] Sat Sun Sat Mon Mon Tue Fri Thu Thu Sun Sat
> Levels: Mon < Tue < Wed < Thu < Fri < Sat < Sun
```

```
ggplot() +
  aes(x = day_of_week) +
  geom_bar()
```

Now the level ordering reads "Levels: Mon < Tue < Wed < Thu < Fri < Sat < Sun", and the "Mon" bar is correspondingly presented first.

The `fct_relevel()` function comes from the forcats package (which got loaded when we loaded the tidyverse). The forcats package provides functions for manipulating factors. The forcats functions we will use in this book are:\[14\]

---

13 In the code above `Inf` stands for infinity, which is a valid concept in R. For example, \( \frac{1}{0} = \text{Inf} \) and \( \frac{1}{\text{Inf}} = 0 \). In this context, `Inf` is shorthand for the “last place in the line.” In general, writing `fct_relevel(day_of_week, "Sun", after = n)` tells R to move "Sun" to after the \( n \)th position—i.e., to the \( n+1 \)th position. And if we set \( n = \text{Inf} \), we’re telling R to make "Sun" follow the infiniteith level, so that "Sun" is always at the end, no matter how many levels `day_of_week` has.

14 As you see, all the forcats functions start with prefix `fct_`. 27
• `fct_recode()`,
• `fct_relabel()`,
• `fct_relevel()`, and
• `fct_reorder()`.

I’ll define these in section 2.4.6, after I’ve more thoroughly discussed R functions. But here’s a quick example:

```r
day_of_week <-
fct_recode(
    day_of_week, 
    Weekend = "Sun", 
    Weekend = "Sat" 
)

day_of_week
```

> [1] Weekend Weekend Weekend Mon Mon Mon Tue Fri Thu
> [10] Thu Weekend Weekend
> Levels: Mon < Tue < Wed < Thu < Fri < Weekend

```r
ggplot() +
aes(x = day_of_week) +
geom_bar()
```

We’ve now covered most of the important data types. Indeed, we’ll store nearly all our data in vectors that comprise logicals, doubles, integers, character strings, Dates, and factors. But to be clear: a given vector may store only one of these types of data. For example, a vector can comprise numbers or character strings, but not both numbers and character strings. Let’s see what happens when we try to stuff both types of data into one vector:
c(1, 2, 3, "a", "b", "c")

> [1] "1" "2" "3" "a" "b" "c"

In this case R downgrades numbers 1, 2, 3 into character strings "1", "2", "3" to make them compatible with the rest of the vector. Hence, even if you try, R will not allow you to create vectors with multiple data types.\(^{15}\)

**Exercise 1.12.** The following maps a vector with numbers to a vector without numbers:

str_remove_all(
  c("CuT", "cApiTAL", "lEtTErS", "FROm", "thIs", "VECTor"),
  "[A-Z]"
)

> [1] "u" "cpi" "lttr" "m" "ths" "or"

Use str_remove_all() and "[aeiou]" to remove the vowels from character string vector c("remove", "vowels", "from", "this", "vector").

**Exercise 1.13.** Use today() and mdy() to calculate the number of days old you are.

**Exercise 1.14.** What’s the date a million days() from today()?

**Exercise 1.15.** What’s the time a million seconds() from now()?

**Exercise 1.16.** How many weeks() are in a billion seconds()?

**Exercise 1.17.** What fraction of the next 100,000 days in are in November?

- Create a vector called vector_of_dates that includes the 100,000 days() that follow today().
  - Input the vector 1:10^5 into days() and add that to today().
- Apply month(label = TRUE) to vector_of_dates to create a vector_of_months that comprises 100,000 character strings.
- Calculate the mean() fraction of elements in vector_of_months that equal "Nov".

\(^{15}\)While we can’t store multiple data types in vectors, we can store multiple data types in *lists*, which we’ll cover in chapter 5.
1.2.6 Tibbles

In section 1.2.3 we saw the benefit of organizing a scattered collection of numbers in an orderly vector. In this section we will see the benefit of organizing a scattered collection of vectors in an orderly table of data. For example, suppose that in addition to battle_name, battle_start, and battle_end, we also had the following:

```r
battle_casualties <-
  c(
    0, 10, 10, 30, 86, 87, 114,
    244, 346, 151, 4690, 2549
  )

battle_victor <-
  c(
    "Confederate", "Inconclusive", "Inconclusive", "Union",
    "Confederate", "Union", "Union", "Confederate",
    "Union", "Confederate", "Confederate", "Confederate"
  )

battle_description <-
  c(
    "Beauregard takes Charleston Federal fort",
    "Union gunboats fight Confederate artillery",
    "Confederate artillery hit by naval bombardment",
    "Small Confederate detachment rooted out of WV",
    "Union attack near a church repelled",
    "Union forces defeat Missouri State Guard",
    "Patterson defeats Jackson, but squanders victory",
    "Confederate victory in Missouri",
    "Union forces divide Confederate forces; half surrender",
    "Irvin McDowell's recon-in-force defeated at Manassas",
    "Thomas Jackson earned the moniker 'Stonewall'",
    "First major battle west of the Mississippi"
  )
```

Since they're not officially related to one another, it's difficult to cross-reference the information stored in our battle vectors. For example, we can't tell, simply by looking at the data, in which battle "Thomas Jackson earned the moniker 'Stonewall'", or which battle had the most casualties, or when the "First major battle west of the Mississippi" ended. However, we can easily answer these questions if we align our vectors as columns of a table:
The first line of this output describes `battle_data` as a *tibble*—i.e., a table of data (like an Excel spreadsheet)—with 12 rows and 5 columns.16 A tibble's rows are also called “observations” and its columns are also called “variables.” Thus, the word “variable” has two different meanings: (i) an object that’s been assigned a name with `<-` and (ii) a column of a tibble.

We can inspect our tibble in multiple ways. We can view it in all it's glory with `View(battle_data)`. We can get a summary of its variables with:

```r
glimpse(battle_data)
```

---

16The predecessor of the tibble is the *data frame*. Data frames are like tibbles, except they're clunkier and messier. So we'll use tibbles throughout this book. (Although nothing substantive would change if we switched to data frames.)
We can get its top three rows with:

```
head(battle_data, 3)
```

```
# A tibble: 3 x 6
  start          end  casualty victor name                           description
  <date>         <date>    <dbl> <chr> <chr>                              <chr>
1 1861-04-12    1861-04-13  0 Confederate Battle 1: Fort Sumter Beauregard-
2 1861-05-18    1861-05-19 10 Inconclusive Battle 2: Sewell's Po- Union gunb-
3 1861-05-29    1861-06-01 10 Inconclusive Battle 3: Aquia Creek Confederat-
```

We can get its bottom four rows with:

```
```
tail(battle_data, 4)

> # A tibble: 4 x 6
> start   end   casualty victor name description
> <date>  <date>  <dbl> <chr> <chr> <chr>
> 1 1861-07-11 1861-07-11 346 Union Battle 9: Rich Mountain Union forc-
> 2 1861-07-18 1861-07-18 151 Confederate Battle 10: Blackburn's- Irvin McDo-
> 3 1861-07-21 1861-07-21 4690 Confederate Battle 11: First Bull ~ Thomas Jac-
> 4 1861-08-10 1861-08-10 2549 Confederate Battle 12: Wilson's Cr~ First majo-

We can get its fourth, eighth, ninth, and tenth rows with:

slice(battle_data, 4, 8:10)

> # A tibble: 4 x 6
> start   end   casualty victor name description
> <date>  <date>  <dbl> <chr> <chr> <chr>
> 1 1861-06-03 1861-06-03 30 Union Battle 4: Philippi (We~ Small Conf-
> 2 1861-07-05 1861-07-05 244 Confederate Battle 8: Carthage Confederat-
> 3 1861-07-11 1861-07-11 346 Union Battle 9: Rich Mountain Union forc-
> 4 1861-07-18 1861-07-18 151 Confederate Battle 10: Blackburn's- Irvin McDo-

We can get its first two rows when it's ordered alphabetically by description with:

slice_min(
  battle_data,
  order_by = description,
  n = 2
)

> # A tibble: 2 x 6
> start   end   casualty victor name description
> <date>  <date>  <dbl> <chr> <chr> <chr>
> 1 1861-04-12 1861-04-13 0 Confederate Battle 1: Fort Sumter Beauregard ~
> 2 1861-05-29 1861-06-01 10 Inconclusive Battle 3: Aquia Creek Confederate-

We can get get its last three rows when it’s ordered by end with:

33
slice_max(
  battle_data,
  order_by = end,
  n = 3
)

> # A tibble: 3 x 6
>   start    end casualty victor name description
>   <date> <date>    <dbl> <chr> <chr>               
> 1 1861-08-10 1861-08-10 2549 Confederate Battle 12: Wilson's Cr~ First majo~
> 2 1861-07-21 1861-07-21 4690 Confederate Battle 11: First Bull ~ Thomas Jac~
> 3 1861-07-18 1861-07-18 151 Confederate Battle 10: Blackburn's~ Irvin McDo~

We can get a random two rows with:

slice_sample(
  battle_data,
  n = 2
)

> # A tibble: 2 x 6
>   start    end casualty victor name description
>   <date> <date>    <dbl> <chr> <chr>               
> 1 1861-06-03 1861-06-03 30 Union Battle 4: Philippi (W~ Small Conf~
> 2 1861-05-18 1861-05-19 10 Inconclusive Battle 2: Sewell's Po~ Union gunb~

We can get its name and casualty columns with:

select(
  battle_data,
  name, casualty
)

> # A tibble: 12 x 2
>   name        casualty
>   <chr>       <dbl>
> 1 Battle 1: Fort Sumter      0
> 2 Battle 2: Sewell's Point   10
> 3 Battle 3: Aquia Creek      10
We can get everything but its name and casualty columns with:

```r
select(
  battle_data,
  -name, -casualty
)
```

> # A tibble: 12 x 4
>     start   end victor description
>     <dbl> <dbl> <chr>                     
> 1       0     0 Fort Sumter               
> 2       10    10 Sewell's Point           
> 3       10    10 Aquia Creek              
> 4       30    30 Philippi (West Virginia)
> 5       86    86 Big Bethel               
> 6       87    87 Boonville (Missouri)     
> 7      114   114 Hoke's Run              
> 8      244   244 Carthage                
> 9      346   346 Rich Mountain           
>10      151   151 Blackburn's Ford         
>11     4690  4690 First Bull Run or First Manassas
>12     2549  2549 Wilson's Creek or Oak Hills
```
Confederate Beauregard takes Charleston Federal fort
Inconclusive Union gunboats fight Confederate artillery
Inconclusive Confederate artillery hit by naval bomber
Small Confederate detachment rooted out of
Union attack near a church repelled
Union forces defeat Missouri State Guard
Patterson defeats Jackson, but squanders
Confederate victory in Missouri
Union forces divide Confederate forces; h-
Irvin McDowell's recon-in-force defeated
Thomas Jackson earned the moniker 'Stonew-
First major battle west of the Mississippi

We can get its victor column as a single-column tibble with:

select( battle_data, victor )
> # A tibble: 12 x 1
>   victor
>   <chr>
>  1 Confederate
>  2 Inconclusive
>  3 Inconclusive
>  4 Union
>  5 Confederate
>  6 Union
>  7 Union
>  8 Confederate
>  9 Union
> 10 Confederate
> 11 Confederate
> 12 Confederate

Or we can get its `victor` column as a vector with:

```r
pull(
    battle_data,
    victor
  )
```

```
[1] "Confederate" "Inconclusive" "Inconclusive" "Union"  "Confederate"
[6] "Union"    "Union"    "Confederate" "Union"  "Confederate"
```

**Exercise 1.18.** `pull()` the casualty vector and calculate its `median()` value.

**Exercise 1.19.** `pull()` the battle description vector and `pluck()` its third value.

**Exercise 1.20.** `slice()` out the third battle’s data, and `select()` its `description`.

**Exercise 1.21.** Use `slice_max()` to `select()` the `description` of the bloodiest battle.

We can `rename()` the columns of a tibble. For example, the following changes the names `start` and `end` to `First_Day` and `Last_Day`:

```r
rename(
    battle_data,
    First_Day = start,
    Last_Day = end
  )
```
We can also relocate() the columns of a tibble. For example, the following makes name the left-most column:

```r
relocate(
  battle_data,
  name,
  .before = 1  #move before first column
)
```

The following moves start and end to the right of victor:
relocate(
  battle_data,
  start, end,
  .after = victor
)

And the following makes casualty the right-most column:

relocate(
  battle_data,
  casualty,
  .after = last_col()
)

**Exercise 1.22.** **rename() victor to winner** and made it the penultimate column (i.e., the second from the right).

- Refer to the last_col().

There are two basic ways to define a tibble: column-wise with `tibble()` or row-wise with `tribble()` (you can remember that the latter corresponds to rows because it contains the letter *r*). For example, the following are two equivalent ways to define the same table:

```r
char <-
c(
  "Menelaus", "Iphigenia", "Electra", "Orestes"
)
dad <-
c(
  "Zeus", "Tantalus", "Pelops", "Atreus",
  "Atreus", "Agamemnon", "Agamemnon", "Agamemnon"
)
mom <-
c(
  "Clytemnestra", "Clytemnestra", "Clytemnestra"
)
```
tibble(
  character = char,
  father = dad,
  mother = mom
)

> # A tibble: 8 x 3
> character father mother
> <chr> <chr> <chr>
> 1 Tantalus Zeus Plouto
> 2 Pelops Tantalus Dione
> 3 Atreus Pelops Hippodamia
> 4 Agamemnon Atreus Aerope
> 5 Menelaus Atreus Aerope
> 6 Iphigenia Agamemnon Clytemnestra
> 7 Electra Agamemnon Clytemnestra
> 8 Orestes Agamemnon Clytemnestra

tribble(
  ~ character, ~ father, ~ mother,
  "Tantalus", "Zeus", "Plouto",
  "Pelops", "Tantalus", "Dione",
  "Atreus", "Pelops", "Hippodamia",
  "Agamemnon", "Atreus", "Aerope",
  "Menelaus", "Atreus", "Aerope",
  "Iphigenia", "Agamemnon", "Clytemnestra",
  "Electra", "Agamemnon", "Clytemnestra",
  "Orestes", "Agamemnon", "Clytemnestra"
)

> # A tibble: 8 x 3
> character father mother
> <chr> <chr> <chr>
> 1 Tantalus Zeus Plouto
> 2 Pelops Tantalus Dione
> 3 Atreus Pelops Hippodamia
> 4 Agamemnon Atreus Aerope
> 5 Menelaus Atreus Aerope
> 6 Iphigenia Agamemnon Clytemnestra
> 7 Electra Agamemnon Clytemnestra
> 8 Orestes Agamemnon Clytemnestra
Of the two, we will more frequently use `tibble()`, since it constructs tibbles from vectors (e.g., `char`, `dad`, and `mom`) rather from individual elements (e.g., "Tantalus", "Zeus", and "Pluto"), and almost all the data we work with will be prepackaged in a vector.

**Exercise 1.23.** Use `tibble()` to organize the following data in a tibble:

- "ATLiens", 1996L, "2 x Platinum",
- "Aquemini", 1998L, "2 x Platinum",
- "Stankonia", 2000L, "4 x Platinum",
- "Speakerboxxx/The Love Below", 2003L, "Diamond",
- "Idlewild", 2006L, "Platinum".

**Exercise 1.24.** Use `tribble()` to organize the following data in a tibble:

- "This Is a Long Drive for Someone with Nothing to Think About", 1996L, "",
- "The Lonesome Crowded West", 1997L, "",
- "We Were Dead Before the Ship Even Sank", 2007L, "Gold",
- "Strangers to Ourselves", 2015L, "".

### 1.2.7 Saving and Loading Data

Rather than create our own tables with `tibble()` or `tribble()`, we’ll usually start with some data that’s already provided. The input sample could come in several forms: e.g., a .txt file, a .csv file, an Excel file, a SAS file, a STATA file, or an SPSS file. Fortunately, R can read all of these formats. To load a sample of data in RStudio, simply go to File/Import Dataset and choose the option that suits your file format (e.g., choose File/Import Dataset/From Text (readr) if your input file is comma separated or tab delimited). A window will open where you can specify the file to open and some additional options. Once you’ve finished with that, click “Import” and R will load the data into your workspace and illustrate it in a new window. Moreover, R will print to your console the code that you could have run to load and view these data.

For example, download the Civil War data here and load it into your workspace with File/Import Dataset/From Text (readr). Doing so should print something like this to your workspace:

```r
library(readr)
CWSAC_civil_war_data <- read_csv("~/Downloads/CWSAC_civil_war_data.csv")
View(civil_war)
```

[17]https://www.dropbox.com/s/56007hal7fpfps5/CWSAC_civil_war_data.csv?dl=1
The first line is redundant because loading the tidyverse package implicitly loaded the readr package. The second line reads the data from the file saved at "~/Downloads/CWSAC_civil_war_data.csv"—your file path name will probably differ—and loads it into a tibble called CWSAC_civil_war_data. And the third line tells R to show us this tibble.

At this point, I’d usually fine-tune the code above and rerun it. For example, I’d cut the needless `library(readr)` and `View(civil_war)` expressions, shorten the name of the tibble created to `civil_war`, and add a `col_types` option to specify the data type of each column:

```r
civil_war <-
  read_csv(
    "~/Downloads/CWSAC_civil_war_data.csv",
    col_types = "cDDcfcl"
  )
```

In the code above, "cDDcfcl" is shorthand for “character, Date, Date, character, factor, character, and logical.” Without the `col_types = "cDDcfcl"` option, `read_csv()` would make `Significance` a character string (rather than a factor) and `Union_Victory` a double (rather than a logical).

The solution above required us to manually download the data to our computer. But we can skip this step by passing the URL into our `read_csv()` call:

```r
civil_war <-
  read_csv(
    "https://www.dropbox.com/s/56007hal7fpfps5/CWSAC_civil_war_data.csv?dl=1",
    col_types = "cDDcfcl"
  )
```

Now `read_csv()` downloads the data for us!

I could save a tibble as a `.csv` file with `write_csv()`. But exporting data as a `.csv` is slow. And, more troubleshooting, `.csv` files don’t save our column data types. For example, the `Significance` column of `civil_war` is a factor:

```r
glimpse(civil_war)
```

But this column gets downgraded to a character string if I save the tibble as a `.csv` file and then reload it:
To save and load data quickly without sacrificing any metadata, we use `write_rds()` and `read_rds()`. For example, using these functions doesn’t change `Significance` from a factor to a character string:

```r
write_rds(civil_war, "~/Downloads/my_cw_data.rds")
civil_war_2 <- read_rds("~/Downloads/my_cw_data.rds")
glimpse(civil_war_2)
```

For another example, the following saves the tibble `save_me` in the file "saved_tibble.rds":

```r
save_me <-
  tibble(
    x = 1:3,
    y = c("a", "b", "c")
  )
write_rds(
  save_me,
  "saved_tibble.rds"
)
```

And the following loads this tibble with `read_rds()`:

```r
loaded_tibble <- read_rds("saved_tibble.rds")
loaded_tibble
```
Note that I use <- to name the loaded data loaded_tibble. I must rename this tibble because its old name, save_me, didn’t get saved in "saved_tibble.rds".

Now let me show you how not to save and load tibbles:

```r
save(
save_me,
  file = "saved_tib.rds"
)

#Delete tibble after saving it
rm(save_me)

load("saved_tib.rds")

#Now tibble is back, with original name
save_me
```

So, as you see, `save()` and `load()` can also save and load the tibble. But you should never use these functions. The problem is that `load()` not only loaded the tibble, it also loaded the old tibble name, save_me. And this convention makes your code nearly impossible to follow. For example, if you write `load("saved_tib.rds")` and `load("other_tib.rds")` at the top of your file and then reference `save_me` 100 lines later, the reader will have no clue where this tibble came from. To them, it’ll seem to have appeared out of thin air. They’ll look for a definition by searching the document for "save_me <-", but nothing will come up (assuming the tibble was defined and saved in a different script file); then they’ll google “R save_me” thinking that it must be some built-in R thing, and nothing will come up;
and then they’ll finally realize that it stems from the evil `load()` function, but even then they won’t know whether it came from "saved_tib.rds" or "other_tib.rds". In contrast, the reader knows exactly where `loaded_tibble` comes from when you write `loaded_tibble <- read_rds("saved_tibble.rds").`

**Exercise 1.25.** We will now load and save a data set.

- Download the arrest data found at this website\(^{18}\) as a .csv file.
- Use “File/Import Dataset/From Text (readr)” to load the data set. This should execute `read_csv()`.
- Modify the `read_csv()` calls so that columns 1-3, 7-9, 11, and 14 are factor, columns 5, 6, and 12 are integers, and the rest are character strings.
  - Specify `col_types = c("fffciifffcficfc")`.
- Call your tibble `arrests`.
- Write `glimpse(arrests)` to confirm that the variables have reasonable data types.
  - Unfortunately, we can’t easily convert *Arrest Date* into a Date variable because its dates have an unconventional format. (We’ll sort this out in the next chapter.)
- Use `write_rds()` to save this tibble as an .rds file.
- Use `read_rds()` to load file you just created into a tibble called `evanston`.
- `glimpse()` the result to confirm that `evanston` has the same column data types as `arrest`.

### 1.3 Lab: Workspaces

In this lab, you’ll download and load all the data you’ll use in this book. It’ll be good to get this out of the way, so that there won’t be any data issues going forward.

First, we will download all the packages that we will use. Packages are collections of functions that other R users have written and uploaded to a central repository for others to use. We already installed the tidyverse and lubridate packages in section 1.2.1, but we will use a few more packages in addition to these two. You can download all the required packages by running the code below in the console—while connected to the internet—and following the given prompts:

```r
install.packages("tidygraph")
install.packages("ggraph")
install.packages("tidymodels")
install.packages("janeaustenr")
install.packages("tidytext")
install.packages("geosphere")
```

\(^{18}\)https://data.cityofevanston.org/Police/Evanston-Arrests/25em-v4fn
install.packages("sf")
install.packages("spData")
install.packages("tictoc")

You will save each batch of data in a separate R project folder. To create a new R project folder, go to File/New Project/New Directory/New Project and then specify a name and location for the folder. Doing so will create a new folder on your hard drive with a single .Rproj file inside it. Click on this .Rproj file to open the R project with RStudio. Now press Ctrl+Shift+N (Windows or Linux) or Cmd+Shift+N (Mac) to create a new R script file. Write `library(tidyverse)` and `library(lubridate)` at the top of this file. And press Ctrl+S or Cmd+S to save this R script file as code.R, in the project folder. You will do this entire process for each exercise below.\footnote{When closing a project, R will ask you whether you would like to “Save workspace image.” I recommend always clicking “No” here, so that whenever you open the project (by clicking on the .Rproj file) you start with a clean slate. Otherwise, clutter tends to accumulate in your workspace.}

**Exercise 1.26.** Download the `R_projects` folder here.\footnote{https://www.dropbox.com/sh/q8zkf4b4dkgb484/AABqYO0E8EieXX1zq_QkT52Ua?dl=0} This folder contains all the project folders you that will need for this book. If you manage to download `R_projects` then you can skip the remaining exercises, but otherwise following the remaining exercises to create the R projects for yourself.

**Exercise 1.27.** Create an R project called “quartet” for the lecture in section 2.2. Run the following from the project’s code.R file:

```r
netflix <-
  "https://www.dropbox.com/s/brcz45ouuvgd3fc/netflix.csv?dl=1" %>%
  read_csv

evanston <-
  "https://www.dropbox.com/s/cfhitem7f4fnb7c/Evanston_Arrests.csv?dl=1" %>%
  read_csv(col_types = c("fffciifffcficfc"))
```

Confirm that these tibbles look like this:

```r
netflix %>% glimpse
```

> Rows: 6,223
> Columns: 12
> $ show_id <dbl> 81145628, 80117401, 70234439, 80058654, 80125979, 8016389~
> $ type <chr> "Movie", "Movie", "TV Show", "TV Show", "Movie", "TV Show~
Recall that you downloaded `evanston` from the city of Evanston’s website in exercise 1.25.

**Exercise 1.28.** Create an R project called “judges” for the lab in section 2.3. Run the following from the project’s code.R file:

```r
court_data <-
"https://www.dropbox.com/s/h51bns987xqviic/court_data.csv?dl=1" %>%
read_csv(col_types = "icccclllccccc")
```

Confirm that `court_data` looks like this:
court_data %>% glimpse

> Rows: 206,477
> Columns: 12
> cid <int> 42468, 42468, 42468, 42468, 42468, 42469, 42469, 42469, 42-
> jid <chr> "0e0e7", "0e0e7", "0e0e7", "0e0e7", "0e0e7", "04edc", "04e-
> fdate <chr> "03jan2005", "03jan2005", "03jan2005", "03jan2005", "03jan-
> hdate <chr> "11jan2006", "28jun2006", "21feb2007", "07nov2007", "14may-
> treated <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FA-
> closed <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, T-
> settled <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FA-
> party.count <int> 5, 5, 5, 5, 3, 3, 2, 2, 2, 2, 2, 3, 3, 3, 2-
> ptype <chr> "Persona fisica", "Persona fisica", "Persona fisica", "Per-
> dtype <chr> "ENP", "ENP", "ENP", "ENP", "ENP", NA, NA, NA, NA, NA,
> ctype <chr> "Indennita' accompagnamento", "Indennita' accompagnamento"-
> cassa <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, NA, NA, NA, FALSE, FAL-

Exercise 1.29. Create an R project called “ggplot” for the lecture in section 3.2. Run the following from the project’s code.R file:

```r
ufc <- "https://www.dropbox.com/s/le5qsv3ty5vk5hs/ufc.csv?dl=1" %>%
  read_csv(col_types = "iDcllffifciddlcdfiiddiiiiiiiiiiiiicdfiiddiiiiiiiiiiiii"
formula_1 <- "https://www.dropbox.com/s/dn00cwolr9l04vw/f_1.csv?dl=1" %>%
  read_csv

Confirm that ufc looks like this:

ufc %>% head

> # A tibble: 6 x 54
>   fight_id date     location red_win title    weight_class gender rounds result
>   <int> <date>     <chr>    <lgl>  <fct>     <fct>         <fct>     <int> <fct>
> 1    1 2020-08-29 Las Vegas~ FALSE FALSE Light Heavy~ MALE 3 U-DEC
> 2    2 2020-08-29 Las Vegas~ FALSE FALSE Welterweight MALE 3 U-DEC
> 3    3 2020-08-29 Las Vegas~ FALSE FALSE Women's Fly~ FEMALE 3 U-DEC
> 4    4 2020-08-29 Las Vegas~ TRUE FALSE Featherweight~ FEMALE 3 U-DEC
> 5    5 2020-08-29 Las Vegas~ FALSE FALSE Middleweight MALE 3 U-DEC
```

48
Exercise 1.30. Create an R project called “alibaba” for the lab in section 3.3. Run the following from the project’s code.R file:

```r
alibaba_wide <-
  "https://www.dropbox.com/s/fh25g7fulmjmei/alibaba.csv?dl=1" %>%
  read_csv(
    col_types =
    cols(
      order = col_factor(),
      score = col_factor(),
      item = col_factor(),
      brand = col_factor(),
      category = col_factor(),
      sub_category = col_factor(),
      merchant = col_factor(),
      quantity = col_integer()
    )
  )
```

Confirm that `alibaba_wide` looks like this:

```r
alibaba_wide %>% head

> # A tibble: 6 x 68
> order score item brand category sub_category merchant quantity action.1
> <fct> <fct> <fct> <fct> <fct> <fct> <fct> <int> <chr>
> 1 31 5 95537 457 11 299 5476 1 ORDER
> 2 212 3 33546 890 53 229 7804 1 ORDER
> 3 1151 5 52101 829 54 7 6731 1 ORDER
> 4 1352 4 15450 889 4 254 7042 1 ORDER
> 5 1522 5 24751 889 1 254 7042 1 ORDER
> 6 1648 5 47333 713 8 362 9 1 ORDER
> # ... with 59 more variables: action.2 <chr>, action.3 <chr>, action.4 <chr>,
```
Exercise 1.31. Create an R project called “wrangle” for the lecture in section 4.2. Run the following from the project’s code.R file:

```r
chess_panel <-
  "https://www.dropbox.com/s/8dsrtmf0l9kkjep/chess_games.csv?dl=1" %>%
  read_csv

eversight_quantity <-
  "https://www.dropbox.com/s/4uf614qkuqtendedr/eversight_quantity.rds?dl=1" %>%
  read_rds

eversight_price <-
  "https://www.dropbox.com/s/i45zedjyjnc3e41/eversight_price.rds?dl=1" %>%
  read_rds
```

Confirm that `chess_panel` looks like this:

```
chess_panel %>% glimpse
```

```
Rows: 14,322
Columns: 14
$ game <chr> "l1NXvwaE", "mIICvQHh", "kwKvqrYI", "9tXo1AUZ", "qwU9ra~
$ rated <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE,~
$ moves <chr> "d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 fxe5 Nxe5 Qd4 Nc6 Qe5+ Nx~
$ turns <dbl> 16, 61, 61, 95, 33, 66, 119, 39, 31, 31, 43, 52, 66, 10~
$ victory_status <chr> "resign", "mate", "mate", "mate", "resign", "resign", ~
$ winner <chr> "black", "white", "white", "white", "black", "black", "~
$ increment_code <chr> "5+10", "5+10", "20+0", "30+3", "10+0", "15+0", "10+0",~
$ white_id <chr> "a-00", "ischia", "daniamurashov", "capa_j~
$ white_rating <dbl> 1322, 1496, 1439, 1523, 1039, 1381, 1381, 1~
$ black_id <chr> "skinnerua", "a-00", "adivanov2009", "adivanov2009", "d~
$ black_rating <dbl> 1261, 1500, 1454, 1469, 1423, 1392, 1209, 1272, 1607, 1~
$ opening_eco <chr> "B00", "C20", "D02", "C41", "D00", "C50", "B01", "A00",~
$ opening_name <chr> "Nimzowitsch Defense: Kennedy Variation", "King's Pawn ~
$ opening_ply <dbl> 4, 3, 3, 5, 10, 6, 4, 1, 2, 8, 7, 8, 8, 5, 4, 4, 4, ~
```
Exercise 1.32. Create an R project called “rationing” for the lab in section 4.4. Run the following from the project’s code.R file:

```r
category <-
  "https://www.dropbox.com/s/p2hkgz28w2hdevc/category.csv?dl=1" %>%
  read_csv(col_types = "fc")

order_dates <-
  "https://www.dropbox.com/s/vprgd8ddo3e4bnf/order_dates.csv?dl=1" %>%
  read_csv(col_types = "Dl")

inv_DC <-
  "https://www.dropbox.com/s/uowsp5v7pqd66ng/inv_DC.csv?dl=1" %>%
  read_csv(col_types = "fDi")

inv_store <-
  "https://www.dropbox.com/s/zncvif0y38wtetx/inv_store.csv?dl=1" %>%
  read_csv(col_types = "ffDi")

order_store <-
  "https://www.dropbox.com/s/crlqxr2y4nqm7pa/order_store.csv?dl=1" %>%
  read_csv(col_types = "ffDcic")

ship_store <-
  "https://www.dropbox.com/s/5l2rjmj5jlgfb8/ship_store.csv?dl=1" %>%
  read_csv(col_types = "ffDci")
```

Confirm that your tibbles look the following:

```r
category %>% glimpse

> Rows: 308
> Columns: 2
> $ item <fct> 10210, 10219, 11004, 11006, 11036, 11041, 11042, 11079, 11085-
> $ category <chr> "Drinks", "Drinks", "Oil/Vinegar", "Tissues", "Drinks", "Drin-

order_dates %>% glimpse
```
> Rows: 1,371  
> Columns: 2  
> $ order_observed <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, T-  

inv_DC %>% glimpse

> Rows: 414,419  
> Columns: 3  
> $ item <fct> 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10210, 102-  
> $ date <date> 2011-04-01, 2011-04-02, 2011-04-03, 2011-04-04, 2011-04-05, 201-  

inv_store %>% glimpse

> Rows: 16,490,877  
> Columns: 4  
> $ store <fct> 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003, 100-  
> $ item <fct> 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10210, 102-  
> $ date <date> 2011-04-01, 2011-04-02, 2011-04-03, 2011-04-04, 2011-04-05, 201-  
> $ inv <int> 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 9, 9, 9, 9, 9, 9-  

order_store %>% glimpse

> Rows: 413,020  
> Columns: 6  
> $ store <fct> 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003, 100-  
> $ item <fct> 11041, 11041, 11041, 11041, 11041, 11041, 11041, 11041, 110-  
> $ order_code <chr> "391270378435", "42007920296", "608091274043", "49221730205-  
> $ order <int> 15, 30, 45, 30, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15-  
> $ order_type <chr> "DC", "DC", "DC", "DC", "DC", "DC", "DC", "DC", "DC", "D-  

ship_store %>% glimpse
Exercise 1.33. Create an R project called “disintermediation” for the lab in section 4.5. Run the following from the project’s code.R file:

```r
Jobs <-
"https://www.dropbox.com/s/7z5zgjm8cau2v5j/disintermediation_orders.csv?dl=1" %>%
read_csv

Shortest_Distances <-
"https://www.dropbox.com/s/dgb6go372ah34oc/disintermediation_distances.csv?dl=1" %>%
read_csv

Confirm that these tibbles looks like this:

Jobs %>% glimpse

> Rows: 320,383
> Columns: 35
> $ cleaner_id <dbl> 698661, 698661, 698661, 698661, 698661, 698661, 698661,
> $ address_id <chr> "354849c1", "395c6f4e", "27347342", "111b2-"
> $ rooms <dbl> 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, ~
> $ bathrooms <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
> $ pets <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
> $ refrigerator <lgl> FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE,~
> $ oven <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
> $ windows <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
> $ ironing <dbl> 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, ~
> $ kitchen_cabinets <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
> $ tableware <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE,~
> $ balconies <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
> $ need_vacuum_cleaner <lgl> TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
> $ microwave <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
> $ balcony_windows <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
> $ winter_windows <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
```
Exercise 1.34. Create an R project called “lm” for the lecture in section 5.2. Run the following from the project’s code.R file:

```r
wine_list <-
  "https://www.dropbox.com/s/u87sd1bucws6ea2/winemag-data.rds?dl=1" %>%
  read_rds

Confirm that the first element of `wine_list` looks like this:

```
Exercise 1.35. Create an R project called “cars” for the lab in section 5.3. Run the following from the project’s code.R file:
cars_raw <-
  "https://www.dropbox.com/s/6jvg7nb41lgxtf5/car_data.csv?dl=1" %>%
  read_csv(col_types = "fffiffddddd")

Confirm that `cars_raw` looks like this:

```r
cars_raw %>% glimpse
```

Exercise 1.36. Create an R project called “glm” for the lecture in section 6.2. Run the following from the project’s code.R file:

```r
bike <-
  "https://www.dropbox.com/s/d91dnhph6zfqp6j/citi_bike.rds?dl=1" %>%
  read_rds

Confirm that `bike` looks like this:

```r
bike %>% glimpse
```

```r
cars_raw <-
  "https://www.dropbox.com/s/6jvg7nb41lgxtf5/car_data.csv?dl=1" %>%
  read_csv(col_types = "fffiffddddd")
```

Exercise 1.36. Create an R project called “glm” for the lecture in section 6.2. Run the following from the project’s code.R file:

```r
bike <-
  "https://www.dropbox.com/s/d91dnhph6zfqp6j/citi_bike.rds?dl=1" %>%
  read_rds
```

Confirm that `bike` looks like this:

```r
bike %>% glimpse
```
Exercise 1.37. Create an R project called “radiation” for the lab in section 6.3. Run the following from the project’s code.R file:

```r
facility <-
  "https://www.dropbox.com/s/2fw57ayxtgrja20/facility.csv?dl=1" %>
  read_csv(col_types = "ffffffD")

ler <-
  "https://www.dropbox.com/s/f07kbtpu4gk2cdc/ler.csv?dl=1" %>
  read_csv(col_types = "fDiiiiii")

power <-
  "https://www.dropbox.com/s/zgsyt7zgl0dpkic/power.csv?dl=1" %>
  read_csv(col_types = "fDdd")

radiation <-
  "https://www.dropbox.com/s/4s8shd41b4e220c/radiation.csv?dl=1" %>
  read_csv(col_types = "fidd")

shutdown <-
  "https://www.dropbox.com/s/7j5yu6h3tehjm7z/shutdown.csv?dl=1" %>
  read_csv(col_types = "fDDf")
```

Confirm that your tibbles look the following:
facility %>% glimpse

> Rows: 103
> Columns: 7
> $ facility_name <fct> Arkansas Nuclear, Arkansas Nuclear, Beaver Valley, Beave-
> $ reactor <fct> Arkansas Nuclear 1, Arkansas Nuclear 2, Beaver Valley 1,-
> $ manufacturer <fct> B&W, CE, WEST, WEST, WEST, GE, GE, GE, GE, WEST, W-
> $ pwr_bwr <fct> PWR, PWR, PWR, PWR, PWR, BWR, BWR, BWR, PWR, P-
> $ nrc_region <fct> 4, 4, 1, 3, 3, 2, 2, 2, 3, 4, 1, 1, 2, 2, 3, 4,-
> $ state <fct> ARKANSAS, ARKANSAS, PENNSYLVANIA, PENNSYLVANIA, ILLINOIS-
> $ pra_month <date> 1993-04-01, 1992-08-01, 1992-10-01, 1992-03-01, 1994-06-

ler %>% glimpse

> Rows: 36,667
> Columns: 8
> $ reactor <fct> Arkansas Nuclear 1, Arkansas Nuclear 1, Arkansas Nucl-
> $ month <date> 1974-12-01, 1975-01-01, 1975-02-01, 1975-03-01, 1975-
> $ ler <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ actuation_ler <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ techspec_ler <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ degrade_ler <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ inoperable_ler <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ safetyEquip_ler <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-

power %>% glimpse

> Rows: 36,667
> Columns: 4
> $ reactor <fct> Arkansas Nuclear 1, Arkansas Nuclear 1, Arkansas Nucle-
> $ month <date> 1974-12-01, 1975-01-01, 1975-02-01, 1975-03-01, 1975--
> $ generation <dbl> 360558, 507424, 480300, 208023, -3328, 338523, 494173,~
> $ capacity_factor <dbl> 0.570142315, 0.802378242, 0.840861345, 0.328942125, -0-

radiation %>% glimpse

58
shutdow %% glimpse

> Rows: 1,749
> Columns: 4
> $ facility_name <fct> Arkansas Nuclear, Arkansas Nuclear, Arkansas Nuclear,~
> $ exposure_average <dbl> 0.14, 0.61, 0.43, 0.26, 0.28, 0.28, 0.50, 0.50, 0.66,~
> $ exposure_sum <dbl> 147, 476, 601, 722, 1321, 1233, 2225, 1608, 2109, 174~

Exercise 1.38. Create an R project called “fishing” for the lab in section 7.3. Save the illegal fishing data found here21 to this project’s folder. However, do not attempt to load these data into your workspace, as doing so will require some special spatial functions.

1.4 Commentary

1.4.1 <- vs. =

The <- and = operators are different, but the difference between them is highly technical (e.g., see this thread22 and section 8.2.26 of the R Inferno book23). And while we often can’t replace = with <-—e.g., mean(c(1, 2, NA), na.rm <- TRUE) yields an error—we can almost always replace <- with =. For example, we could have written debt_2012 = 16066241407385 instead of debt_2012 <- 16066241407385 in section 1.2.2. Therefore, you may be tempted to forego the <- symbol altogether, and use = for everything. Unfortunately, that would be deemed poor coding etiquette. Instead, the standard practice is to use = if and only if your code is inside a function’s parentheses, such as mean(c(1, 2, NA), na.rm = TRUE) and to use <- otherwise. Admittedly, this is a rather ad hoc rule—in many circumstances, either <- or = would do—but it’s now a deeply entrenched convention: e.g., it’s formally mandated by Google’s R style guide.

Finally, to reiterate, the == symbol is completely different from = and <-. Whereas we use = and <- to define new R objects or parameters, we use == to create logicals. That is, we use == to create TRUE or FALSE values (see section 1.2.4).

21https://www.dropbox.com/sh/hh4e1c0ty12nyqf/AAAvjR0tc8Wd_oVaUnOUZH7na?dl=0  
### 1.4.2 Automating Code

I cleaned the code that got generated from the File/Import Dataset/From Text (readr) call at the end of section 1.2.6 for two reasons. First, I wanted to incorporate the `col_types = "cddcfcl"` option. Second, and more importantly, I always want my entire empirical methodology—from loading the raw data to saving the final plots—to be documented in code so that I have a full transcript of the data science process. Since it’s so easy to load data with a mouse, you may be tempted to omit these data-loading steps in your code. But you’ll be angry with yourself if you come back to your code in a year’s time and find a gaping hole where you imported all your data, the details of which you may have forgotten. To avoid having to remember idiosyncratic details of the data-cleaning process, it’s best to make your code fully automated, with no manual overrides. Indeed, you should aim to write all your programs in such a way that they can be executed in their entirety from a clean slate with a simple Ctrl+Shift+Enter (or Cmd+Shift+Enter). The difference between fully automated code and nearly fully automated code is like the difference between a job on your to-do list that’s completely finished—you can cross it off and forget about it—and a job that’s finished except for one pesky detail that you’ve got to keep lingering tabs on.

Initially it’ll be difficult to avoid manual overrides. But whenever you’re tempted to switch from the keyboard to the mouse, take five minutes to Google the proper coding solution. R has a programming answer for basically any data problem, and the fastest code-based solution is almost always faster to implement than the fastest point-and-click alternative. Thus, you’ll ultimately become a faster data scientist if you commit to doing everything with code. And, what’s more, updating a report that’s based on a point-and-click process requires redoing all the pointing and clicking, whereas updating a report that’s based on a computer program requires simply pressing Ctrl+Shift+Enter. In short, if you leave your code in a set-it-and-forget-it state, you can costlessly rerun it any number of times, after any stretch of time.

### 1.4.3 Glossary of stringr Functions

The following examples illustrate the most common stringr functions. This list doesn’t include every stringr function, but it includes all the ones we’ll use in this book.

**str_c()** concatenates character strings:

```r
str_c("a", c("b", "c"), sep = "_____")
```

> [1] "a_____b" "a_____c"

**str_count()** counts character string elements:

---

24My friend Juan Serpa tells me that that I’m too doctrinaire about this point. He reminds me that a paper we wrote together relied on quite a bit of manual data cleaning. So, I’m probably being a bit of a hypocrite here.
In the code above, "\\bw\\w*" is shorthand for “a word that starts with w” and "\\bW\\w*" is shorthand for “a word that starts with W.” Hence, the second call returns 3—for "words", "with", and "w"—the third call returns 0 because none of its words begin with a W.

`str_detect()` identifies whether a character string contains a specific substring:
str_extract() and str_extract_all() pull out substrings from a character string:

```r
str_extract(
  "Get the first word starting with a w",
  "\bw\w*"
)
```

> [1] "word"

```r
str_extract_all(
  "Get all the words starting with a w",
  "\bw\w*"
)
```

> [[1]]
> [1] "words" "with" "w"

str_flatten() compresses a character string vector into a single character string:

```r
c(c("paste", "these", "words", "together") %>%
  str_flatten(collapse = "___"))
```

> [1] "paste___these___words___together"

str_length() counts the number of characters in a character string:

```r
str_length("How many characters are here?")
```

> [1] 29

str_locate() and str_locate_all() report the location of substrings in a character string:
str_locate(
  "Find the first word starting with a w",
  "\\bw\\w*"
)

> start end
> [1,] 16 19

str_locate_all(
  "Find all the words starting with a w",
  "\\bw\\w*"
)

> [[1]]
> start end
> [1,] 14 18
> [2,] 29 32
> [3,] 36 36

str_remove() and str_remove_all() delete substrings in a character string:

str_remove(
  "Remove the first word starting with a w",
  "\\bw\\w*"
)

> [1] "Remove the first starting with a w"

str_remove_all(
  "Remove all the words starting with a w",
  "\\bw\\w*"
)

> [1] "Remove all the starting a "

str_replace() and str_replace_all() delete substrings in a character string:
str_replace(
    "Replace the first word starting with a w",
    "\bw\w*",
    "***"
)

> [1] "Replace the first *** starting with a w"

str_replace_all(
    "Replace all the words starting with a w",
    "\bw\w*",
    "***"
)

> [1] "Replace all the *** starting *** a ***"

str_to_lower(), str_to_upper(), and str_to_title() re-express a character string in lowercase, uppercase, and title case:

str_to_lower("lower UPPER Title")

> [1] "lower upper title"

str_to_upper("lower UPPER Title")

> [1] "LOWER UPPER TITLE"

str_to_title("lower UPPER Title")

> [1] "Lower Upper Title"

str_squish() removes redundant white space in a character string:
str_squish(" remove extra spaces ")

> [1] "remove extra spaces"

str_sub() returns a specified substring:

str_sub(  
  "Get 10th through 14th characters",  
  start = 10,  
  end = 14  
)

> [1] "throu"

str_subset() returns the strings of a character string vector that contain a specific substring:

str_subset(  
  c("which", "words", "contain", "a", "w?"),  
  "w"
)

> [1] "which" "words" "w?"

str_which() is like str_subset(), except it returns the location of the strings rather than the strings themselves:

str_which(  
  c("which", "words", "contain", "a", "w?"),  
  "w"
)

> [1] 1 2 5

1.5 Solutions

1.3
fine_grid <- seq(-1, 1, length.out = 10^6)

1.4

x <- log(1:100)
mean(x)

1.5

prod(1:100) / prod(100:170)  #this is greater than one

1.6

debt_change <- debt - lag(debt)

1.7

debt_growth <- (debt - lag(debt)) / lag(debt)

1.8

x <- rnorm(n = 30)
y <- rnorm(n = 30)
z <- x/y
hist(z)

1.10

(x==y) * x + (x!=y) * y

1.11
(x==0) | (y==0) | (z==0)
#or
any(x==0, y==0, z==0)

1.12

str_remove_all(
c("remove", "vowels", "from", "this", "vector"),
"[aeiou]"
)

1.13

today() - mdy("11/06/1983")

1.14

today() + days(10^6)

1.15

now() + seconds(10^6)

1.16

seconds(10^9)/weeks(1)

1.17

vector_of_dates <- today() + days(1:10^5)
vector_of_months <- month(vector_of_dates, label = TRUE)
mean(vector_of_months == "Nov")

1.18
casualty_vec <-
pull(
  battle_data, casualty
)

median(casualty_vec)

1.19

desc_vector <-
pull(
  battle_data, description
)

pluck(desc_vector, 3)

1.20

third_battle <-
slice(
  battle_data, 3
)

select(third_battle, description)

1.21

bloodiest <-
slice_max(
  battle_data, casualty
)

select(bloodiest, description)

1.22
temp_tib <-
  rename(
    battle_data,
    winner = "victor"
  )

relocate(
  temp_tib,
  winner,
  .before = last_col()
)

1.23

tibble(
  album =
    c(
      "Southernplayalisticadillacmuzik",
      "ATLiens", "Aquemini", "Stankonia",
      "Speakerboxxx/The Love Below",
      "Idlewild"
    ),
  year =
  RIAA =
    c(
      "Platinum", "2 × Platinum", "2 × Platinum",
      "4 × Platinum", "Diamond", "Platinum"
    )
)

1.24

tribble(
  ~ album, ~ year, ~ RIAA,
  "This Is a Long Drive for Someone with Nothing to Think About", 1996L, "",
  "The Lonesome Crowded West", 1997L, "",
  "The Moon & Antarctica", 2000L, "Gold",
  "We Were Dead Before the Ship Even Sank", 2007L, "Gold",
  "Strangers to Ourselves", 2015L, ""
)
arrests <- read_csv("~/Downloads/Evanston_Arrests.csv", col_types = c("fffciifffcficfc")
)
glimpse(arrests)
write_rds(arrests, "~/Downloads/evanston_arrest_data.rds")
evanston <- read_rds("~/Downloads/evanston_arrest_data.rds")
glimpse(evanston)
Chapter 2

Functions

2.1 Introduction

In this chapter, I cover the verbs of the R programming language: the functions that act upon data. I begin, in section 2.2.1, by explaining how to apply functions (although you’ve probably already figured most of this out). In section 2.2.2 I cover the special case of if-else statements, which, annoyingly, follow a different notation. I then supercharge this discussion in section 2.2.3 by showing you how to apply an entire sequence of functions. You do so by using conveyor belt symbol %>% to pass the data through a functional assembly line. This technique will level up your data science game from craft manufacturing to mass production.

In sections 2.2.4 and 2.2.5, I will explain how to format functional assembly lines, and how to turn them into new, stand-alone functions. However, we won’t need to define many functions because R already provides us with so many.

In section 2.2.6 I define the bracketed expression as a “universal connector” that allows us to attach any block of R code to a functional assembly line. In short the bracketed expression turns a given chunk of R code into a black box, which takes inputs from one conveyor belt and passes outputs to another conveyor belt. For example, we’ll use bracketed expressions to add to our assembly lines old-fashioned functions, which wouldn’t otherwise be compatible with the %>% symbol due to their outdated input-output conventions.

In sections 2.2.7–2.2.10 I cover the four most important functions in R: mutate(), filter(), summarise(), and group_by(). Three quarters of R programming boils down to this functional quartet. However, rather than functions, it’s better to think of the first three elements of this quartet as platforms: mutate() being the platform for modifying tibble columns, filter() the platform for selecting tibble rows, and summarise() the platform for deriving summary statistics. These platforms serve as vehicles for applying other functions: e.g., will embed functions inside of mutate() to define new variables and embed functions inside of filter() to remove observations. The last element of the quartet, group_by(), modifies the other three. It partitions the sample into discrete chunks, which subsequent mutate(), filter(), and summarise() calls get implemented by.
After this chapter’s lecture, you will be able to perform most common data manipulations. The lab in section 2.3 will demonstrate this fact. In this lab you will do real data science, replicating the core analysis of an empirical project published in an academic journal.

I conclude the chapter with some afterthoughts. In section 2.4.1 I discuss the standard layout of modern R code: a sequence of tibble-input-tibble-output functions connected with %>% conveyor belts into a functional assembly line. However, the functions comprising this tibble-level assembly line will usually contain assembly lines of their own, nested within their parentheses. These nested assembly lines comprise a sequence of vector-input-vector-output functions that modify the tibble’s vector columns. Hence, our code will contain one macro-level assembly line, which passes our tibble through a sequence of tibble transforming functions, and a set of micro-level assembly lines, which pass the tibble columns through a sequence of vector transforming functions. In section 2.4.2 I explain why it’s better to arrange functions into an assembly line than it is to nest them inside one another (which is the old-school approach). In section 2.4.3 I mention some specialized functions that we could have used to streamline our analysis. In section 2.4.4 I explain how to define functions of multiple variables. In section 2.4.5 I provide a brief primer on downloading packages from the Comprehensive R Archive Network (CRAN). In section 2.4.6 I provide examples of the four forcats functions I introduced in section 1.2.5. And I conclude, in section 2.4.7, with a piece of high-level coding advice: try to make different things the same to avoid writing multiple distinct sets of instructions.

For the lecture, you can work in the code.R file of the quartet project you created in exercise 1.27, and for the lab you can work in the code.R file of the judges project you created in exercise 1.28.

### 2.2 Lecture

#### 2.2.1 Calling Functions

Functions transform input objects into output objects. To indicate something is a function we write a set of parentheses after it. For example, we refer to the sum function as `sum()` and the mean function as `mean()`. To apply a function to an object, we place the object inside the function’s parentheses: e.g., `sum(1:10) = 55` and `mean(1:5) = 5.5`. But not all functions require inputs: e.g., the time this sentence was written is

```
now()
```

> [1] "2023-02-28 12:00:29 CST"

What is `now()` you may ask? Well, it’s a function that returns the current time (try it: it’s pretty fun). But if you ever want to know what a function does—as you will often when reading this book—just write ? and then the function’s name in your console. For example,
running `str_count` in your R console yields the documentation for function `str_count()`. This documentation can be a bit daunting, but here’s a trick: start at the Examples section, which is at the bottom of almost every help page. For example, we find the following at the bottom of the `str_count()` documentation:

```r
fruit <- c("apple", "banana", "pear", "pineapple")
str_count(fruit, "a")

> [1] 1 3 1 1

str_count(fruit, "p")

> [1] 2 0 1 3

str_count(fruit, "e")

> [1] 1 0 1 2

str_count(fruit, c("a", "b", "p", "p"))

> [1] 1 1 1 3
```

These four `str_count()` calls clearly illustrate what this function does. Indeed, the fastest way to learn how a function works is to execute the examples provided in its help page.

For another case, the bottom of the `str_replace_all` documentation provides the following:

```r
fruits <- c("one apple", "two pears", "three bananas")
str_replace_all(fruits, "[aeiou]", toupper)

> [1] "OnE ApPlE" "twO pEArS" "thrEE bAnAnAs"
```
str_replace_all(fruits, "[aeiou]", ")

> [1] "-n- -pple-" "tw- p--rs" "thr-- b-n-n-s"

str_replace(fruits, c("a", "e", "i"), ")

> [1] "one -pple" "two p-ars" "three bananas"

These examples gives you a pretty good sense for how this function works.

When you’re finished with the Examples section, you can then read the Usage and Arguments sections, which explain what the function expects as inputs. For the case of str_replace_all(), the documentation reports that the function requires three inputs, or “arguments”. As always, these arguments are ordered and named, with the first input called string, the second called pattern, and the third called replacement. Hence, we can invoke this function with:

```
str_replace_all(
  string = "Four legs good, two legs bad",
  pattern = "bad",
  replacement = "better"
)
```

> [1] "Four legs good, two legs better"

Or we can invoke it with

```
str_replace_all(
  "Four legs good, two legs bad",
  "bad",
  "better"
)
```

> [1] "Four legs good, two legs better"
In the latter case, R uses the first string as the first argument, the second string as the second argument, and the third string as the third argument.

Note that we give each input its own line. We do so for aesthetic purposes, as how we format the white space between the parentheses doesn’t matter. So, for example, the following are equivalent (albeit uglier) ways of calling this function:

```r
str_replace("Four legs good, two legs bad", "bad", "better")
```

> [1] "Four legs good, two legs better"

```r
str_replace("Four legs good, two legs bad", "bad", "better")
```

> [1] "Four legs good, two legs better"

```r
str_replace("Four legs good, two legs bad", "bad", "better")
```

> [1] "Four legs good, two legs better"

Some functions have default inputs. For example, the `str_flatten` documentation reports that this function has arguments `string` and `collapse = ""`. The `= ""` part after `collapse` indicates that R will treat this input as "" unless we otherwise specify it. Thus, ignoring this input is equivalent to setting it to an empty character string:
\begin{verbatim}
str_flatten(string = LETTERS)

> [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

str_flatten(
    string = LETTERS,
    collapse = ""
)

> [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

But, of course, we are free to choose a different \texttt{collapse} value:

str_flatten(
    string = LETTERS,
    collapse = "&&"
)

\end{verbatim}

\textbf{Exercise 2.1.} Use \texttt{str_detect()} to create a vector of logicals that identify the elements of the \texttt{im_burning} vector, defined below, that contain an exclamation mark.

- Consult the examples provided in \texttt{?str_detect}.
- Ignore the \texttt{negate} argument, and its value will default to \texttt{FALSE}.
- Your \texttt{str_detect()} call should output the vector \texttt{c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE)}.

\begin{verbatim}
im_burning <-
c("Oh, I'm burning!",
  "I wish I were out of doors!",
  "I wish I were a girl again,",
  "half savage and hardy, and free;",
  "and laughing at injuries,",
  "not maddening under them!",
  "Why am I so changed?"
)
\end{verbatim}
Exercise 2.2. Use `str_detect()` to identify the elements of `im_burning` that do not contain an exclamation mark.

- Use the `negate` argument.
- Your `str_detect()` call should output the vector `c(FALSE, FALSE, TRUE, TRUE, TRUE, FALSE, TRUE)`.

Exercise 2.3. We will now pick a random 10 days over the next year.

- Create a vector that includes the 365 `days()` that follow `today()`.
- `sample()` 10 random elements from this vector.
  - Use the `replace = FALSE` option to ensure that you don’t pick the same day twice.

Exercise 2.4. We will now pick a random 10 weekend days over the next year.

- Create a vector called `vector_of_dates` that includes the 365 `days()` that follow `today()`.
- Create a second vector of logicals called `vector_of_weights` that specifies whether the `wday(label = TRUE)` of each element in `vector_of_dates` is `%in% c("Sat", "Sun")`. The `nth` element of `vector_of_weights` should be `FALSE` if the `nth` date in `vector_of_dates` is a weekday.
- `sample()` 10 random elements from `vector_of_dates` with `replace = FALSE` and `prob = vector_of_weights`. This last option makes the probability of selecting a weekday zero.

2.2.2 Conditional Statements

An important function that does not follow the standard notation is the if-else statement. The notation for these expressions looks like this:

```
if(logical_expression) output_if_TRUE else output_if_FALSE
```

Or we could add brackets to accommodate multiple lines of code:

```
if(logical_expression) {
    output_if_TRUE
    #This could be a multi-line code chunk
} else {
    output_if_FALSE
    #This could be a multi-line code chunk
}
```
The expressions above yield output_if_TRUE if logical_expression = TRUE, yield output_if_FALSE if logical_expression = FALSE, and yield an error if logical_expression is neither TRUE nor FALSE. For example, the following outputs a frowny face if it’s Friday the 13th and outputs a smiley face otherwise:

```r
if(day(today()) == 13 & wday(today(), label=TRUE) == "Fri") "(:(" else ":")"
```

The “else” portion of the conditional statement is optional. For example, the following returns "happy hour" if it’s between 5:00 and 6:00PM, and otherwise it does nothing.

```r
if(hour(now()) == 17) "happy hour"
```

**Exercise 2.5.** We will now count the days until the next Halloween (like in The Nightmare Before Christmas).

- Define this_years_halloween <- mdy(str_c("10/31/", year(today()))).
- Define next_years_halloween <- mdy(str_c("10/31/", year(today()) + 1)).
- Use an if-else statement to return this_years_halloween - today() if this year’s Halloween has yet to pass, and return next_years_halloween - today() otherwise.

The `ifelse()` function is a vectorized version of the `if() { } else { }` protocol. The notation looks like this:

```r
ifelse(
  logical_vector,
  output_vector_if_TRUE,
  output_vector_if_FALSE
)
```

The function creates a new vector by collating the elements of the `output_vector_if_TRUE` and `output_vector_if_FALSE` vectors, based on the values of the `logical_vector` vector. Specifically, the nth element of the output vector equals the nth element of `output_vector_if_TRUE` if the nth element of `logical_vector` is TRUE and equals the nth element of `output_vector_if_FALSE` otherwise.

For example, setting `logical_vector` to a vector of alternating logicals, setting `output_vector_if_TRUE` to the lower-cased alphabet, and setting `output_vector_if_FALSE` to the upper-cased alphabet yields the following:
ifelse(
    rep(c(FALSE, TRUE), 13),
    letters,
    LETTERS
)

> [1] "A" "b" "C" "d" "E" "f" "G" "h" "I" "j" "K" "l" "M" "n" "O" "p" "Q" "r" "S"
> [20] "t" "U" "v" "W" "x" "Y" "z"

Here’s another example:

ifelse(-2:2 == 0, "can't divide by zero", 1/(-2:2))

> [1] "-0.5" "-1" "can't divide by zero"
> [4] "1" "0.5"

In this case output_vector_if_TRUE is not a vector but the fixed value "can't divide by zero", which is used whenever the logical is TRUE.

For a more elaborate example, define director_vector, cast_vector, and type_vector as the first eight values of the director, cast, and type columns of the netflix tibble (which we loaded in exercise 1.27, section 1.3):

director_vector <- pull(netflix, director)
director_vector <- head(director_vector, 8)
director_vector

> [1] "Richard Finn, Tim Maltby" NA
> [3] NA
> [5] "Fernando Lebrija" NA
> [7] "Gabe Ibáñez" "Rodrigo Toro, Francisco Schultz"

cast_vector <- pull(netflix, cast)
cast_vector <- head(cast_vector, 8)
cast_vector

79
Now the following defines `notable_people` as the Netflix program’s first two cast members, if the program is a TV show, and defines it as the program’s director otherwise:

```r
notable_people <- ifelse(type_vector == "TV Show", str_extract(cast_vector, "\[^,]*,?[\^,]*"), director_vector )
```

In the code above "\[^,]*,?[\^,]*" is a *regular expression* that is R shorthand for “everything before the second comma” (write `vignette("regular-expressions")` in your console for more on regular expressions).

**Exercise 2.6.** Use `ifelse()` and `is.na()` to change the NA values in `director_vector` to "*missing data*".

- In practice, we would solve this problem with the specialized `replace_na()` function. (See section 2.4.3.)
Exercise 2.7. Use `ifelse()` and `%in%` to create a vector comprising the letters of the alphabet, with every vowel capitalized.

Exercise 2.8. Use `ifelse()`, `str_detect()`, and `"[aeiou]"` to create a vector comprising the letters of the alphabet, with every vowel capitalized.

Exercise 2.9. Use `str_length()`, `month.abb`, and `month.name` to create the following vector:

```r
> [1] "Jan"  "Feb"  "March"  "April"  "May"  "June"  "July"  "August"
> [9] "Sep"  "Oct"  "Nov"  "Dec"
```

Note, this vector uses the full month names if these names are no more than six letters long, and otherwise uses the month abbreviations.

2.2.3 Functional Assembly Lines

So far we’ve applied all our functions in one-off faction. But to do real data science we will need to run functions at an *industrial scale*. Indeed, a proper empirical analysis is like a factory that turns raw data into finished plots and tables. And translating the inputs into outputs in this case can require hundreds of transformations performed by hundreds of functions.

To shepherd our data through all these functions we will align them in an assembly line and use a conveyor belt to pass the data from function to function. For our conveyor belt we will use the symbol `%>%`. The literature calls this symbol as the “pipe operator,” but I will refer to it as a “conveyor belt,” because an assembly line is a better metaphor than a pipe. Conveyor belt `%>%`, is the most important symbol in R; in fact, you will use it so frequently that you should learn its hot key now: the keyboard shortcut for `%>%` is Ctrl+Shift+M with Windows and Linux and Cmd+Shift+M with a Mac.

Let’s start with a simple example. Suppose we want to evaluate \(\cos(\exp(\sin(\sqrt{\log(3)})))\).

We could do so with

```r
cos(exp(sin(sqrt(log(3)))))
```

```r
> [1] -0.7227508
```

However, nesting functions in this manner is generally impractical (see section 2.4.2), so we’d like another approach. Here's an alternative method:
log_3 <- log(x = 3)
sqrt_log_3 <- sqrt(x = log_3)
sin_sqrt_log_3 <- sin(x = sqrt_log_3)
exp_sin_sqrt_log_3 <- exp(x = sin_sqrt_log_3)
cos_exp_sin_sqrt_log_3 <- cos(x = exp_sin_sqrt_log_3)
cos_exp_sin_sqrt_log_3

> [1] -0.7227508

But this approach has a fundamental problem: it creates a new R object each time it invokes a function, which creates a lot of clutter. Indeed, variables log_3, sqrt_log_3, sin_sqrt_log_3, exp_sin_sqrt_log_3, and cos_exp_sin_sqrt_log_3 will now junk up our working environment until we explicitly delete them. And this problem is not only aesthetic: e.g., instead of 3 we could have started with a 100Gb vector of numbers, in which case this code would commandeer 500Gb of RAM.

We can avoid this problem by recycling the same variable. For example, the following performs the same calculation, but with only one variable.

current_val <- log(x = 3)
current_val <- sqrt(x = current_val)
current_val <- sin(x = current_val)
current_val <- exp(x = current_val)
current_val <- cos(x = current_val)
current_val

> [1] -0.7227508

In this case, current_val—which gets redefined from \( \log(3) = 1.098612 \) to \( \sqrt{1.098612} \) = 1.048147 to \( \sin(1.048147) = 0.8664998 \) to \( \exp(1.147203) = 2.378571 \) to \( \cos(2.378571) = -0.7227508 \)—acts like a wheelbarrow, ferrying data from step to step. Unfortunately, to pass data from one operation to the next we must first load the wheelbarrow with a current_val <- step and then unload it with x = current_val step. However, we can circumvent these pesky loading and unloading operations if we replace our wheelbarrow with conveyor belts:

3 %>% log %>% sqrt %>% sin %>% exp %>% cos

> [1] -0.7227508
The %>% conveyor belts integrate the five operations into a *functional assembly line*, where the output of one function serves as the input to the next. In this case, 3 enters the line as raw materials and is transformed by five work stations—\( \log() \), \( \sqrt{} \), \( \sin() \), \( \exp() \), and \( \cos() \)—into the finished product of \(-0.7227508\). To accentuate the assembly line metaphor I have positioned the functions side by side, but we usually arrange them vertically:

```r
3 %>%
  log %>%
  sqrt %>%
  sin %>%
  exp %>%
  cos

> [1] -0.7227508
```

Here's another example:

```r
c(
  " veni ",
  " vidi ",
  " vici 
)
%>%
  str_flatten %>%
  str_squish %>%
  str_to_upper

> [1] "VENI VIDI VICI"
```

This assembly line comprises three transformations: the first compresses the input vector into a single character string, the second removes the excess white space from this string, and the third capitalizes it. The following code yields the same result:

```r
my_value <-
c(
  " veni ",
  " vidi ",
  " vici 
)
my_value <- str_flatten(my_value)
my_value <- str_squish(my_value)
my_value <- str_to_upper(my_value)
my_value
```

83
> [1] "VENI VIDI VICI"

But this solution requires wheelbarrow variable `my_value`, and it’s less aesthetically pleasing, since `my_value <-` is harder on the eye than `%>%`.

**Exercise 2.10.** Create a functional assembly line that evaluates $\sqrt{\sqrt{\sqrt{\sqrt{2}}}}$. Confirm that your solution yields the same output as `sqrt(sqrt(sqrt(sqrt(2))))`.

**Exercise 2.11.** Create a functional assembly line that produces this output:

> [1] "cccccciiciiicivcixclclclicliliiclivclixclvclviiclvclvclxiiclvclxixcliiclxvclxvc..."

The assembly line should first express the first 200 integers as `as.roman()` numerals. It should then `str_to_lower()` case the character strings and `str_sort()` them alphabetically. And finally it should `str_flatten()` the result into a single character string.

In general, the following two code chunks yield the same output:

```r
#Chunk 1
x %>% fun_1 %>% fun_2 %>% # ...
%>% fun_n

#Chunk 2
y <- fun_1(x)
y <- fun_2(y)
# ...
y <- fun_n(y)
y
```

A functional assembly line could comprise any number of functions. If it comprises just one function then `x %>% f_1` is equivalent to `f_1(x)`. For example, `9 %>% sqrt = 3`. Going forward, we’ll implement most functions this way. And in English we will refer to the expression `x %>% f_1` as “x piped into f_1.” For example, if I ask you to “pipe 1:100 into sum()” then you should write `1:100 %>% sum`.

Note that we can drop the parentheses after our functions when we string them into an assembly line. For example, `x %>% f_1 %>% ... %>% f_n` and `x %>% f_1() %>% ... %>%`
f_n() are equivalent. However, we could still use these parentheses to pass in additional variables. For example, the following are equivalent:

```
"Is it alright to boil a lobster alive" %>%
  str_replace(
    "lobster",
    "sentient creature"
  )
```

> [1] "Is it alright to boil a sentient creature alive"

```
str_replace(
  "Is it alright to boil a lobster alive?",
  "lobster",
  "sentient creature"
)
```

> [1] "Is it alright to boil a sentient creature alive?"

**Exercise 2.12.** What do the following three functional assembly lines output?

```
#Line 1
1:3 %>%
  lag %>%
  rep(2) %>%
  sum(na.rm = TRUE)

#Line 2
1:100 %>%
  head(3) %>%
  rep(3) %>%
  pluck(9)

#Line 3
c("1", "10", "100", NA) %>%
  replace_na("1000") %>% #replace NA terms with 1000
  as.integer %>%
  is.integer %>%
  all
```

---

1This isn’t true for the new pipe operator, |>, which is one of the reasons why I don’t like it as much.
By default, conveyor belts feed into the slot reserved for a function’s first argument. For example R interprets "x" %>% str_c("y", "z") as str_c("x", "y", "z"). However, you can redirect the conveyor belt with the . symbol. For example, R interprets "x" %>% str_c("y", ., "z") as str_c("y", "x", "z") and interprets "x" %>% str_c("y", "z", .) as str_c("y", "z", "x"). For another example, the following code chunks all output "There's time for everything except the things worth doing"

```
"There's time for nothing besides the things worth doing" %>%
  str_replace(
    "nothing besides",
    "everything except"
  )

"nothing besides" %>%
  str_replace(
    "There's time for nothing besides the things worth doing",
    ".",
    "everything except"
  )

"everything except" %>%
  str_replace(
    string = "There's time for nothing besides the things worth doing",
    pattern = "nothing besides",
    replacement = .
  )
```

The following exercises provide additional examples. These exercises use functions that you will not be familiar with, but isn’t a problem because you don’t need to know how functions work to arrange them into an assembly line.

**Exercise 2.13.** What do the following three expressions output?

```
"Hadley" %>% str_c("--the best human ever--", "is a kiwi")

"Hadley" %>% str_c("The best human ever--", ., "--is a kiwi")

"Hadley" %>% str_c("The best human ever", " is a kiwi--", .)
```
Exercise 2.14. Create assembly-line versions of the following code. In other words, use conveyor belts to replace wheelbarrow variable my_string:

```r
my_string <- "comrade Napoleon is always right"
my_string <- str_to_title(my_string)
my_string <- str_c(my_string, "!")
my_string <- str_replace(my_string, "Comrade", "Master")
my_string <- rep(my_string, 2)
my_string <- str_flatten(my_string, collapse = " ")
my_string
```

> [1] "Master Napoleon Is Always Right! Master Napoleon Is Always Right!"

- Use the pattern = . option when you call str_count() to align the conveyor belt to the correct input.

Exercise 2.15. The following code calculates the average number of times each character digit, from "0" to "9", appears in the past ten thousand dates:

```r
x <- tibble(date = today() + days(-10^4:-1))
x <- mutate(x, date_char = str_remove_all(date, "-"))
x <- group_by(x, date, date_char)
x <- summarise(x, digits = as.character(0:9))
x <- mutate(x, digit_count = str_count(date_char, digits))
x <- group_by(x, digits)
x <- summarise(x, m = mean(digit_count))
x <- pivot_wider(x, names_from = digits, values_from = m)
x
```

> # A tibble: 1 x 10
>   `0`  `1`  `2`  `3`  `4`  `5`  `6`  `7`  `8`  `9`
> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1  2.54  1.50  1.66  0.310 0.253 0.263 0.289 0.292 0.292 0.595

For example, the date "12-21-1999" has three "1" characters, two "2" characters, and three "9" characters. Create an assembly-line version of this code, so that it has the following form:
Exercise 2.16. The following code selects the longest name (in terms of the number of characters) listed in either the director or cast columns of netflix.

```r
tibble(date = today() + days(-10^4:-1)) %>%
  # transformation_1 %>%
  # transformation_2 %>
  # transformation_3 %>
  # transformation_4 %>
  # transformation_5 %>
  # transformation_6 %>
  # transformation_7

netflix %>%
  replace_na(list(director = "", cast = "")) %>%
  mutate(
    director = str_split(director, ", "),
    cast = str_split(cast, ", ")
  ) %>%
  unnest(director) %>%
  unnest(cast) %>%
  summarise(
    director =
      last(
        director,
        order_by = str_length(director)
      ),
    cast =
      last(
        cast,
        order_by = str_length(cast)
      )
  ) %>%
  summarise(
    longest_name =
      ifelse(
        str_length(director) > str_length(cast),
        director,
        cast
      )
  )

> # A tibble: 1 x 1

88
Perform this calculation without a functional assembly line. In other words, use a wheelbarrow variable to remove the conveyor belts.

### 2.2.4 Formatting Code

Look at the code provided for exercise 2.16. Notice how every line starts at one of five columns: either 0, 1, 2, 3 or 4 tabs from the left, depending on how deeply it is nested. This formatting makes it easy for our eye to scan down a column, to quickly see what’s going on. For example, by examining the code that begins one tab in, it’s immediately apparent that it will apply six functions to our data: `replace_na()`, `mutate()`, `unnest()`, `unnest()`, `summarise()`, and `summarise()`. And by examining the code that begins two tabs in, it’s immediately apparent that that the first `summarise()` has two arguments. Let me contrast the code above with how you may be tempted to write it:

```r
netflix %>%
  replace_na(list(director = "", cast = "")) %>%
  mutate(director = str_split(director, ", "),
         cast = str_split(cast, ", ")
) %>%
  unnest(director) %>%
  unnest(cast) %>%
  summarise(director =
             last(
                 director,
                 order_by = str_length(director)
             ),
             cast = last(cast,
                          order_by = str_length(cast)
             )
) %>%
  summarise(longest_name = ifelse(str_length(director) > str_length(cast), director, cast))
```

There are several things wrong with this formatting. First, it makes `netflix` looks like a function, rather than an input tibble, because it has the same alignment as `replace_na()`, `mutate()`, etc. To distinguish data from functions, you should always incorporate a tab after a `%>%`. Second, the code inside the `mutate()` and the first `summarise()` do not start an integer number of tabs from the left. For example, the two `director =` expressions are not aligned, because mutate is a shorter word than summarise. There are two problems with this formatting: first, it requires the eye to jump a variable distance to the right each time, which is tiring, and second, distance to the left margin no longer determines how deeply code is nested. For example, that `order_by = str_length(cast)` is nested twice is immediately
obvious under the previous formatting, but not under the new formatting. The solution
to this problem is to go to a new line after opening the parentheses and then going to a
new line before closing the parentheses. This is a universal rule: whenever you put multiple
lines of code inside a set of parentheses, always put a line break after the ( and a line break
before the ). The third problem is the two unnest() operations sharing the same line: the
reader will almost certainly miss the second one. Accordingly, you should generally go to a
new line after each %>% The only exception to this is if the entire functional assembly line
is short enough to fit in a single line. For example, to get the first letter of the alphabet,
I would probably write letters %>% first without a linebreak after a pipe. The fourth
problem is that the second argument of the second summarise() is likely to get overlooked,
as it’s not given its own line. Accordingly, we usually go to a new line after a comma,
unless the entire expression fits in one line, like to the input to replace_na(), under the
old formatting. The final problem is the last line, which is simply too long. In general,
you should try to avoid exceeding the faint vertical line that RStudio provides in its coding
pane.

Fortunately, RStudio makes correct formatting simple, because if you provide the correct
line breaks then it will provide the correct indenting. Also, if the indenting is incorrect
then you can fix it by highlighting the text and pressing Ctrl+I, for Windows or Linux,
or Cmd+I, for Mac. So, for example, to fix the formatting above, I would add line breaks
where they need to be, and then highlight the entire text and press Cmd+I (since I use a
Mac).

**Exercise 2.17.** Use Ctrl+I or Cmd+I to fix the formatting in the example above.

**Exercise 2.18.** Use Ctrl+I, or Cmd+I to fix the formatting of the following code. (Don’t
worry about what it does.)

```r
"output/combined_experimental_sample.rds" %>% read_rds %>%
filter(!is.na(UNITCOST)) %>%
group_by(product) %>%
summarise(cv = sd(UNITCOST, na.rm = TRUE)/mean(UNITCOST, na.rm = TRUE),
.groups = "drop") %>%
na.omit %>%
summarise(probs = c(.25, .5, .75), quantile =
  cv %>% quantile(probs = probs) %>%
  format_fraction_same_length(3))
) %>%
pivot_wider(names_from = probs, values_from = quantile) %>%
iwalk(~.x %>%
  as_tibble %>%
  write_delim(str_c(stat_folder, "wholesale_cv_quartile_", .y, ".txt"),
  col_names = FALSE)
```
2.2.5 Defining Functions

So far we’ve constructed each of our functional assembly lines for a single use. For example, when we write `1:10 %>% log %>% sum %>% exp`, we’re really telling R to do the following:

1. create an assembly line comprising `log()`, `sum()`, and `exp()` operations,
2. pass the vector `1:10` through this assembly line, and
3. destroy all traces of the assembly line.

This is analogous to creating an auto factory to produce a single car. And yet this is what we’ll usually do as we’ll most often have just one sample of data to analyze. However, sometimes we’ll want to pass multiple data sets through a functional assembly line. And in this case, it’s easier to save the assembly line as a stand-alone function, which we can apply any number of times, than to reconstruct it for each data set. For example, the following encapsulates the assembly line comprising four `sqrt()` operations from exercise 2.10 in the new function `sqrt_four_times()`:

```r
sqrt_four_times <-
  . %>%
  sqrt %>%
  sqrt %>%
  sqrt %>%
  sqrt
```

Now `sqrt_four_times(x)`, `x %>% sqrt_four_times`, and `x %>% sqrt %>% sqrt %>% sqrt %>% sqrt` are all equivalent. For example, `3 %>% sqrt_four_times = 1.071075` and `3 %>% sqrt %>% sqrt %>% sqrt %>% sqrt = 1.071075`. Or for another example, define

```r
weird_function <-
  . %>%
  log %>%
  sqrt_four_times %>%
  sum
```

Now `weird_function(x)`, `x %>% weird_function`, `x %>% log %>% sqrt_four_times %>% sum`, and `x %>% s log %>% sqrt %>% sqrt %>% sqrt %>% sqrt %>% sqrt %>% sum` all yield the same output.
For example, `weird_function(1:100) = 107.1425` and `1:100 %>% log %>% sqrt %>% sqrt %>% sqrt %>% sqrt %>% sqrt %>% sqrt %>% sum = 107.1425`.

In general, we can compress an assembly line comprising functions `f_1()`, `f_2()`, ..., and `f_n()` into function `f()` with the following:

```r
f <- #go to a new line before putting the . %>%
  . %>%
  f_1 %>%
  f_2 %>%
  #...
  f_n
```

In other words, running this code defines a function `f()` that satisfies `f(x) = x %>% f_1 %>% f_2 ... %>% f_n`. Note that we add a line break after the <-; the code looks nicer this way.

**Exercise 2.19.** Use the <- . %>% notation to create an R function called `soft_max()` that takes a vector `x` and applies the `exp()`, `sum()`, and `log()` functions to it, in that order. Apply `soft_max` to vectors `-4:4` and `-64:64`.

**Exercise 2.20.** Use the <- . %>% notation to create a function called `clean_text()`, where `clean_text(x)` equals the output of the following code:

```r
y <- str_replace_na(x, replacement = "")
y <- str_flatten(y, collapse = " ")
y <- str_remove_all(y, "\[[:punct:]]")
y <- str_squish(y)
y <- str_to_lower(y)
y
```

Apply `clean_text()` to the following two character string vectors:

```r
#Vector 1
tasty_breakfast <-
c("If it's your job to eat a frog,",
"it's best to do it first thing in the morning.",
"And if it's your job to eat two frogs",
"it's best to eat the biggest one first."
)
```

#Vector 2
brotherly_love <-
c(NA,
   "When Ajax lifted his shield
   Teucer would peer round, ",
   "and when he had hit any one in the throng,
   the man would fall dead; ",
   NA,
   "then Teucer would hie back to Ajax as a
   child to its mother, and again
   duck down under his shield. "
)

**Exercise 2.21.** Define `double_string <- . %>% rep(2)` as a function that doubles a vector. For example, `c("a", "b") %>% double_string = c("a", "b", "a", "b")`. Given that, what would the following code output?

```r
f_1 <-
  . %>%
  double_string %>%
  double_string
def_2 <-
  . %>%
def_1 %>%
def_1
1:3 %>%
def_2 %>%
sum
```

### 2.2.6 Bracketed Expressions

We began section 2.2.3 with the problem of evaluating `\cos(\exp(\sin(\sqrt{\log(3)})))`. But now suppose we wanted to calculate `\cos(\exp(\sin((\log(3))^2)))`. This would be easy enough if we allowed ourselves a wheelbarrow variable:

```r
current_val <- log(3)
current_val <- current_val^2
current_val <- sin(current_val)
```
current_val <- exp(current_val)
current_val <- cos(current_val)
current_val

> [1] -0.827832

But the assembly line analog, `3 %>% log %>% ^2 %>% sin %>% exp %>% cos`, yields an error because `^2` is not a standard function. Here’s the fix:

```r
3 %>%
  log %>%
  {.^2} %>%
  sin %>%
  exp %>%
  cos
```

> [1] -0.827832

This code uses a *bracketed expression* to create an ad hoc squaring function. When the . symbol is positioned inside of brackets, `{ }`, it inherits the value from inflowing conveyor belt, which in this case is \( \log(3) = 1.098612 \). Hence, this bracketed expression turns 1.098612 into \( 1.098612^2 = 1.206948 \).

Here are two more examples:

```r
#R treats this:
c(2, 5, 8) %>%
  {max(.) - min(.)}
```

> [1] 6

```r
#as this:
max(c(2, 5, 8)) - min(c(2, 5, 8))
```

> [1] 6
#R treats this:

```r
library(dplyr)

c(c("how", "many", "times", "does", "the", "last", "word", "in", "this", "character", "string", "vector", "appear", "in", "this", "vector")
  %>%
  {last_word <- last(.)
   vector_of_logicals <- last_word == .
   sum(vector_of_logicals)}

> [1] 2
```

#as this

```r
last_word <-
  last(c("how", "many", "times", "does", "the", "last", "word", "in", "this", "character", "string", "vector", "appear", "in", "this", "vector")
  }

vector_of_logicals <-
  last_word ==
  c("how", "many", "times", "does", "the", "last", "word", "in", "this", "character", "string", "vector", "appear", "in", "this", "vector")
  }

sum(vector_of_logicals)

> [1] 2
```

Note that we add vertical space before and after the brackets when the bracketed expression spans multiple lines.

We can remove the brackets when the . terms are nested in a common function. For example, the following four chunks of code are equivalent:
In the latter two cases, the `str_c()` parentheses serve as the brackets. The fourth chunk uses the fact that the conveyor belt feeds into the function’s first slot, unless explicitly redirected with a . (the . inside of the `str_length()` doesn’t redirect the conveyor belt, because it’s nested within a function).
Exercise 2.22. Use bracketed expressions to rewrite the code below without wheelbarrow variable \( x \). This code tests whether the number \( y \) is prime. Specifically, it returns `TRUE` if \( y \) is not divisible by \( \text{all()} \) the numbers between 2 (the smallest possible divisor of \( y \)) and \( \text{sqrt}(y) \) (the largest possible possible divisor of \( y \)).

\[
\begin{align*}
x & \leftarrow y \\
x & \leftarrow x / (2: \text{sqrt}(x)) \\
x & \leftarrow x \neq \text{floor}(x) \\
x & \leftarrow \text{all}(x) \\
x
\end{align*}
\]

Exercise 2.23. Simplify the following code’s bracketed expression so that it fits in a single line and doesn’t actually include brackets.

- Nest `word()` within `str_replace_all()`.

"When a man is tired of London, he is tired of life" %>% {
  input_string <- .
  word_to_remove <- word(input_string, 5) # cut fifth word

  str_replace_all(
    string = input_string,
    pattern = word_to_remove,
    replacement = "understanding"
  )
}

> [1] "When a man is understanding of London, he is understanding of life"

The bracketed expression notation, `%>% { .}`, complements the function definition notation, `<- . %>%`. (Note that the . symbol means different things under these different conventions.) For example, we can use a bracketed expression to raise a number to its own power:

\[
3 \ %>\% \ {^\cdot \cdot} \quad \text{#calculate } 3^3
\]

> [1] 27
And we can combine this bracketed expression with <- . %>% to create a function that raises a number to its own power:

```r
self_exponentiation <-
  . %>%
  {.^.}
3 %>% self_exponentiation

> [1] 27
```

For another example, the following creates a function that maps vector `c(x_1, x_2)` to vector `c(x_2, x_1 + x_2)`:  

```r
fib <-
  . %>
  { c(pluck(.), 2), sum(.) } 

c(1, 2) %>% fib

> [1] 2 3

c(1, 2) %>% fib %>% fib

> [1] 3 5

c(1, 2) %>% fib %>% fib %>% fib

> [1] 5 8
```

The second letter of each output enumerates the Fibonacci sequence\(^2\).

For a third example, the following defines `na_detector()` as a function that returns "NA found" if the input contains an NA term and returns the input otherwise:

```r
na_detector()
```

\(^2\)https://en.wikipedia.org/wiki/Fibonacci_number
The if-else statement makes the function return "NA found" if the logical `any(is.na(.))` is TRUE and otherwise makes it return . (i.e., the input to the function).³

The last example below defines `santas_calendar()` as a function that takes a date and returns the word "Christmas" if it’s December 25th, and otherwise appends the word "Eve" to the output of the function evaluated under the following day.⁴

³See section 2.2.2 for more on if-else statements.
⁴Defining `santas_calendar()` in terms of `santas_calendar()` is an example of recursion⁵.
Exercise 2.24. Use a bracketed expression to define `euclid_length()` as a function that transforms a numeric vector into the square root of the sum of its squared elements. For example, `1:4 %>% euclid_length` should return $\sqrt{1^2 + 2^2 + 3^2 + 4^2} = 5.48$.

Exercise 2.25. Use a bracketed expression to define `logical_not()` as a pipe-compatible function that applies the `!` operator to its input. For example, `c(TRUE, FALSE) %>% logical_not` should yield `c(FALSE, TRUE)`.

Exercise 2.26. Use a bracketed expression to define `reverse_symmetric()` as a function that indicates whether a vector equals the reverse of itself. Specifically, `reverse_symmetric(x)` should return `TRUE` if vector `x` equals vector `rev(x)` and should return `FALSE` otherwise. For example, `c("back", "to", "back") %>% reverse_symmetric` should be `TRUE` whereas `c("back", "to", "front") %>% reverse_symmetric` should be `FALSE`. 

100
**Exercise 2.27.** Use a bracketed expression to define `smart_viewer()` as a function that receives a tibble and returns its `summary()` of the tibble’s `ncol()` is less than five and otherwise returns its `glimpse()`. Pipe `netflix` into `smart_viewer()` and then pipe a tibble comprising the first three columns of `netflix` into `smart_viewer()`.

### 2.2.7 `mutate()`

So we now know how to create our own functions. But, honestly, we won’t have much need to do so because R already has a built-in function for almost everything. And despite the thousands of R functions at our disposal, just four functions—`mutate()`, `summarise()`, `group_by()`, and `filter()`—will undergird most of our analysis. I will now cover this quartet one function at a time, starting with `mutate()`.

The `mutate()` function enables us to “get under the hood” of a tibble and muck about with its columns. For example, we can use it to clean `netflix`, which we defined in section 1.3. This tibble looks like:

```r
netflix %>% glimpse
```

```
> Rows: 6,223  
> Columns: 12  
> $ show_id    <dbl> 81145628, 80117401, 70234439, 80058654, 80125979, 8016389~  
> $ type       <chr> "Movie", "Movie", "TV Show", "TV Show", "Movie", "TV Show-  
> $ title      <chr> "Norm of the North: King Sized Adventure", "Jandino: What-  
> $ director   <chr> "Richard Finn, Tim Maltby", NA, NA, NA, "Fernando Lebrija-  
> $ cast       <chr> "Alan Marriott, Andrew Toth, Brian Dobson, Cole Howard, J-  
> $ country    <chr> "United States, India, South Korea, China", "United Kingd-  
> $ date_added <chr> "September 9, 2019", "September 9, 2016", "September 8, 2-  
> $ rating     <chr> "TV-PG", "TV-MA", "TV-Y7-FV", "TV-Y7", "TV-14", "TV-MA", ~  
> $ duration   <chr> "90 min", "94 min", "1 Season", "1 Season", "99 min", "1 ~  
> $ listed_in  <chr> "Children & Family Movies, Comedies", "Stand-Up Comedy", ~  
> $ description <chr> "Before planning an awesome wedding for his grandfather, ~
```

As you see, the `show_id` and `release_year` variables are tagged with `<dbl>`, which indicates that they’re doubles. But these variables are whole numbers and should thus be saved as integers (see section 1.2.4). We’ll redefine them as such with `mutate()`:

```r
netflix <-
netflix %>%
mutate(
  show_id = as.integer(show_id),
```
release_year = as.integer(release_year)

netflix %>%
glimpse  # now show_id and release_year are tagged with <int>

> Rows: 6,223
> Columns: 12
> $ show_id      <int> 81145628, 80117401, 70234439, 80058654, 80125979, 8016389-
> $ type         <chr> "Movie", "Movie", "TV Show", "TV Show", "Movie", "TV Show-
> $ title        <chr> "Norm of the North: King Sized Adventure", "Jandino: What-
> $ director     <chr> "Richard Finn, Tim Maltby", NA, NA, NA, "Fernando Lebrija-
> $ cast         <chr> "Alan Marriott, Andrew Toth, Brian Dobson, Cole Howard, J-
> $ country      <chr> "United States, India, South Korea, China", "United Kingd-
> $ date_added   <chr> "September 9, 2019", "September 9, 2016", "September 8, 2-
> $ rating       <chr> "TV-PG", "TV-MA", "TV-Y7-FV", "TV-Y7", "TV-14", "TV-MA", ~
> $ duration     <chr> "90 min", "94 min", "1 Season", "1 Season", "99 min", "1 ~
> $ listed_in    <chr> "Children & Family Movies, Comedies", "Stand-Up Comedy", ~
> $ description  <chr> "Before planning an awesome wedding for his grandfather, ~

This example illustrates three points. First, piping a tibble into `mutate()` enables us to reference its columns: if we refer to `show_id` outside of `mutate()` then R will return "Error: object 'show_id' not found", since `show_id` is not a variable in our environment. But if we pass `netflix` into `mutate()` and refer to `show_id` within this function's parentheses then R will know that we're referencing a column of this tibble.

Second, `mutate()` takes a tibble as an input and returns a modified tibble as an output. It does not change the variable storing the input tibble unless we explicitly overwrite this variable with `<-`. For example, without the `netflix <-` line, `mutate()` would simply print the modified tibble to the screen, leaving the value of `netflix` unchanged. And in this fashion `mutate()` behaves like any other tibble function: e.g., writing `netflix %>% head(3)` does not modify `netflix`, but writing `netflix <- netflix %>% head(3)` does.

Third, we use `=` expressions to articulate how our `mutate()` call should transform the input tibble to the output tibble. These `=` expressions incorporate other functions, such as `as.integer()`, which do the real work. Thus, rather than directly modify our columns, `mutate()` serves as a platform for other functions to do so. Each `=` expression modifies or defines one column of the tibble. We can include any number of `=` expressions (separated by commas) to modify any number of columns. And we can apply multiple `=` expressions to one column: e.g., `mutate(type = str_to_upper(type), type = str_c(type, "!"))` would first capitalize the type column and then add an exclamation mark to it.

---

6We use `=` rather than `<-` within `mutate()` calls, since we never put the latter symbol inside a function's parentheses (see section 1.4.1).
Whereas the previous example used `mutate()` to modify old variables, the following example uses `mutate()` to construct new variables.

```r
nenflix <- #use <- to "save" change
  netflix %>%
  mutate(
    will_smith = str_detect(cast, "Will Smith"),  #is Will Smith in movie?
    cast_size = str_count(cast, ",") + 1,          #num people = num commas +1
    country_primary = str_remove(country, ",.*")  #delete text after first comma
  )

netflix %>%
  select(country, cast, will_smith, cast_size, country_primary) %>%
  slice(623:626) %>%
  head
```

Now `will_smith` indicates whether this particular actor is listed in the `cast`; `cast_size` reports one more than the number of commas in `cast`; and `country_primary` lists the text in `country` that comes before the first comma. To define this last variable, we used the regular expressions ",.*", which is shorthand for “the first comma everything after it.”

**Exercise 2.28.** Use `%in%` and `mutate()` to add a logical variable to `netflix` called `restricted` that is `TRUE` if and only if the show’s rating is "R", "TV-MA", or "NC-17".

**Exercise 2.29.** Use `ifelse()`, `%in%`, and `mutate()` to add a variable to `netflix` called `kid_rating` that equals `rating` if this value is "G", "PG", "TV-G", or "TV-PG", and otherwise equals "not child friendly".

**Exercise 2.30.** Combine `mutate()` with ...

- `mdy()` to change the `date_added` column of `netflix` from a character string to a Date,
- `month(label = TRUE)` to add a new variable that reports the month the show was added, and
- `wday(label = TRUE)` to add a new variable that reports the day of the week the show was added.

Perform these three operations in one `mutate()` call.
We can embed functional assembly lines inside of `mutate()`. For example, the following defines `last_cast_member_is_director` as a logical that is `TRUE` if the last person listed in the `cast` is also listed as a `director`:

```
netflix %>%
  mutate(
    last_cast_member_is_director =
    cast %>%
    str_extract("\b[^,]*\$") %>%
    str_detect(director)
  ) %>%
  select(cast, director, last_cast_member_is_director)
```  

In the code above, "\b[^,]*\$" is shorthand for “everything after the last comma (besides any white space that immediately follows it).”

**Exercise 2.31.** The `duration` values of `netflix` are text strings that comprise both a measurement, which is a number, and a metric, which is either "min" or "Season". For example, the first six `duration` values are:

```
netflix %>%
  select(duration) %>%
  head
```

```
> # A tibble: 6 x 1
>   duration
>      <chr>
> 1 90 min
> 2 94 min
> 3 1 Season
> 4 1 Season
> 5 99 min
> 6 1 Season
```

However, storing two values in one variable makes accessing either value difficult. For example, there’s currently no easy way to calculate the median show duration. Accordingly, we will now separate `duration` into two variables, one that captures the numeric value and the other that specifies the unit of measure.
• Pipe `netflix` into `mutate()`.
• Within the `mutate()` call, use `word()` to define `duration_metric` as the second word of `duration`, which is always either "min" or "Season".
  
  - `word("fee fi fo fum", 3) = "fo".
• Within the same `mutate()` call redefine `duration` as the “first word” of `duration` expressed as an integer.
  
  - Use a functional assembly line that comprises `word()` and `as.integer()`.
  - The first observation should have `duration = 90` and `duration_metric = "min"`.
• In practice, we would solve this problem with the specialized `separate()` function. (See section 2.4.3.)

**Exercise 2.32.** The `evanston` tibble stores the `Arrest Date` values as character strings and the `Arrest Time` values as integers. We will now convert the former into Dates and the latter into times. We will do so by embedding the following two functional assembly lines inside a `mutate()` step.

```r
#assembly line 1:
'Arrest Date' =
'Arrest Date' %>%
  mdy_hm %>%
  as_date

#assembly line 2:
'Arrest Time' =
'Arrest Time' %>%
  as.character %>%
  str_pad(4, "left", pad = 0) %>%
  str_replace("(..)\", "\\1:") %>%
  hm
```

• Pipe `evanston` into `mutate()`.
• Copy the first functional assembly line within the `mutate()` parentheses.
  
  - Since it comprises two words, we must refer to the date variable as `Arrest Date` rather than as `Arrest Date`. To avoid these “grave accents,” we usually make all variable names one word long. For example, a better name for this variable would be `Arrest_Date`.
  
  - The `mdy_hm()` function tells R to read each value in `Arrest Date` as a date and time. For example, it tells R to interpret "10/12/21 0:00" as “October 12th, 2021 at midnight.” The time values are junk, however, because they’re always midnight. So we pass the output of `mdy_hm()` into `as_date()`, which tells R to keep only the date portion of these values.
– Note that we use `=` rather than `<-` when inside a `mutate()`.

- Add a comma to the end of the first assembly line and then copy in the second assembly line, starting from a new line within the `mutate()` call.

- The first two steps turn the time integers into four-digit character strings. For example, `713 %>% as.character %>% str_pad(4, "left", pad = 0) = "0713"`.
- The `str_replace()` step uses a regular expression\(^7\) to insert a ":" after the first two characters. For example, "))", \"1\":") = "07:13"`.
- The `hm()` step tells R to read the resulting character strings as hours and minutes. For example, ")") returns 7:13 AM.

- Name the resulting tibble `evanston`, overwriting the previous version of this tibble.

**Exercise 2.33.** If you write `evanston %>% distinct(City) %>% print(n = nrow(.))`, you will see that the `City` column has seven different spellings of Evanston: "EVANSTON", "EVNSTN", "EANSTON", "EVANTSTON", "N EVANSTO", "EV/ANSTON", "EVSTON". We will now fix these spelling mistakes.

- Pipe `evanston` into `mutate()`.
- Within the `mutate()` call, write `City = City %>% fct_recode()`, and then write some additional code inside the `fct_recode()` parentheses to tell this function to change all the misspelled versions of Evanston to "EVANSTON".
  - The `City` variable is a factor, not a character string, so we need the fancy `fct_recode()` function to change its values.
  - Follow the examples provided in the `fct_recode` documentation.

- Save the resulting tibble as `evanston`.

### 2.2.8 `summarise()`

The second quartet function is `summarise()`. As its name suggests, it computes summary statistics. For example, we can use it to calculate the fraction of shows released on a Friday or Saturday (i.e., the mean() of the `wday_added %in% c("Fri", "Sat")` logical vector):\(^8\)

```r
netflix %>%
  summarise(weekend_release = mean(wday_added %in% c("Fri", "Sat")))
```

```
> # A tibble: 1 x 1
> weekend_release
> <dbl>
>   1 0.403
```

\(^7\)https://en.wikipedia.org/wiki/Regular_expression

\(^8\)We added `wday_added` to `netflix` in exercise 2.30.
Unfortunately, it's easy to mix up `summarise()` with `summary()`, which we saw in section 1.2.6. But the former is far more powerful: whereas `summary()` always returns the same canned report, `summarise()` can yield almost any function of the data.

For example, the following calculates the mean and median length of time since the show was added to the platform:

```r
netflix %>%
  summarise(
    mean_age = mean(today() - date_added),
    median_age = median(today() - date_added)
  )
```

> # A tibble: 1 x 2
>   mean_age median_age
>      <dtrn>    <dtrn>
>  1  1702.35 days 1612 days

This example uses two `=` expressions to define two summary statistics. But in general a `summarise()` call can incorporate any number of `=` expressions (separated by commas) to derive any number of summary statistics.

For another example, the following returns the `title` of the 1000th show when the sample is ordered by `release_year`:

```r
netflix %>%
  summarise(show_1000 = nth(title, n = 1000, order_by = release_year))
```

> # A tibble: 1 x 1
>   show_1000
>       <chr>
> 1 Soul Eater

**Exercise 2.34.** We will now get the age of the youngest and oldest arrestees in `evanston`.

- Pipe `evanston` into `summarise()`.
- Within the `summarise()` parentheses, use `min()` and `max()` to extract the extreme values of `Age`.
  - Use the `na.rm=TRUE` option to tell R to disregard the `Age = NA` observations.

---

9The `nth()` function returns the nth observation of one vector when the data are sorted by another vector: e.g., `nth(c("a", "b", "c"), n = 2, order_by = c(3, 1, 2)) = "c"`. 

Exercise 2.35. We will now determine whether male arrestees in evanston are more likely to carry a weapon.

- Pipe evanston into summarise().
- Within the summarise() parentheses, use cor() to calculate the correlation between the Sex == "Male" vector of logicals and the `Weapon Code` != "" vector of logicals.
  - Recall that R will treat the TRUE elements as ones and the FALSE elements as zeros. For example, if `Weapon Code` = "Club/Blackjack/Brass Knuckles" then `Weapon Code` != "" is TRUE, which is evaluated as 1.

Exercise 2.36. We will now compare the average age of unarmed arrestees with the average age of arrestees armed with a handgun.

- Pipe evanston into summarise().
- Within the summarise() call, use weighted.mean() to define age_gun as the average Age when we weight by w = `Weapon Code` == "Handgun".
  - This weighting scheme gives each handgun observation a weight of TRUE = 1 and gives each non-handgun observation a weight of FALSE = 0.
  - Use the na.rm = TRUE option to disregard the Age = NA observations.
- Within the same summarise() call, use weighted.mean() to define age_no_gun as the average Age when we weight by w = `Weapon Code` == "".
- Within the same summarise() call, define age_difference = age_no_gun - age_gun.

Exercise 2.37. We will now calculate the fraction of days spanned by our netflix sample in which a new show was released.

- Pipe netflix into summarise().
- Within this summarise() call, do the following:
  - Use max() and min() to define day_count as the number of days between the first and last date_added.
  - Use as.integer() to convert day_count to an integer data type. (By default, R saves the difference between two dates as “difftime” data type.)
  - Use n_distinct() to define show_day_count as the number of days in our sample in which a new show was added (i.e., as the number of distinct date_added values).
  - Define new_show_fraction = show_day_count / day_count as the fraction of days with a new show.

Exercise 2.38. We will now calculate the difference between the average age of netflix movies and TV shows.

- Pipe netflix into summarise().
- Within the summarise() call use weighted.mean() to define avg_year_movie as the average release_year of programs with type = "Movie".
Weight by the `type == "Movie"` logical vector. Doing so tells R to give movies a weight of `TRUE = 1` and give TV shows a weight of `FALSE = 0`. In other words, it calculates the mean `release_year` after disregarding the TV shows. For example, `weighted.mean(c(1, 2, 100, 200), c("a", "a", "b", "b") == "a") = 1.5`.

- Within the same `summarise()` call use `weighted.mean()` to define `avg_year_tv` as the average `release_year` of programs with `type = "TV Show"`.
- Within the same `summarise()` call define `age_difference = avg_year_tv - avg_year_movie`.

`summarise()`, like `mutate()`, can incorporate functional assembly lines. For example, the following `summarise()` call contains an assembly line that calculates the fraction of shows filmed in South Korea:

```r
netflix %>%
  summarise(
    filmed_in_SK =
      country %>%
        str_detect("South Korea") %>%
        mean(na.rm = TRUE)
  )
```

> # A tibble: 1 x 1
> filmed_in_SK
> <dbl>
> 1 0.0282

The `str_detect()` function translates the character string vector `country` into a logical vector that indicates whether the given `country` value contains the phrase "South Korea". For example, the first element of the logical vector is `TRUE` because the first element of `country` is "United States, India, South Korea, China". This logical vector then gets piped into `mean(na.rm = TRUE)`, which computes the fraction of elements that are `TRUE` (after the `NA` terms are dropped).

**Exercise 2.39.** We will now calculate the fraction of `director` values that are `NA`.

- Pipe `netflix` into `summarise()`.
- Within the `summarise()` call, use `is.na()` to create a vector of logicals that indicates which `director` values are `NA`.
- Pipe this logical vector into `mean()` (while still within the `summarise()` call).
So far all of our `summarise()` statistics have been individual elements. But `summarise()` can also provide vector outputs. For example, the following calculates the deciles (i.e., 10th, 20th, ..., 80th, and 90th percentiles) of `release_year`:

```r
netflix %>%
  summarise(
    decile_val = seq(.1, .9, by = .1),
    quant_val = quantile(release_year, probs = decile_val)
  )
```

> # A tibble: 9 x 2
>   decile_val quant_val
>   <dbl>     <dbl>
> 1 0.1     2005
> 2 0.2     2011
> 3 0.3     2014
> 4 0.4     2015
> 5 0.5     2016
> 6 0.6     2017
> 7 0.7     2018
> 8 0.8     2018
> 9 0.9     2019

And the following provides the first four values of `title` and `listed_in`:

```r
netflix %>%
  summarise(
    head(title, 4),
    head(listed_in, 4)
  )
```

> # A tibble: 4 x 2
> `head(title, 4)` `head(listed_in, 4)`
> <chr> <chr>
> 1 Norm of the North: King Sized Adventure Children & Family Movies, Comedies
> 2 Jandino: Whatever it Takes Stand-Up Comedy
> 3 Transformers Prime Kids' TV
> 4 Transformers: Robots in Disguise Kids' TV

And the following lists all the `country` values that contain the word "Uruguay":

110
Netflix %>%
    summarise(country %>% keep(str_detect(. , "Uruguay")))

> # A tibble: 9 x 1
> `country %>% keep(str_detect(. , "Uruguay"))`
> <chr>
> 1 United States, Uruguay
> 2 Argentina, Uruguay, Spain, France
> 3 Uruguay
> 4 Spain, France, Uruguay
> 5 Uruguay, Spain, Mexico
> 6 Uruguay, Argentina, Spain
> 7 Argentina, Uruguay, Serbia
> 8 Uruguay
> 9 Australia, Armenia, Japan, Jordan, Mexico, Mongolia, New Zealand, Philippines

In the example above, keep() is a function that selects the elements of a vector that correspond to the TRUE values of another logical vector (in this case str_detect(country, "Uruguay").

**Exercise 2.40.** We will now get all the people that are listed as the first director of a Netflix show.

- Pipe `netflix` into `summarise()`.
- Within the `summarise()` call pipe `director` into `str_remove(",.\"\")` to remove all the text after the first comma (and thus extract the first person listed).
- Pipe the output of `str_remove()` into `unique %>% na.omit` to get all the unique non-NA terms.
- In practice, we would solve this problem with the specialized `distinct()` function. (See section 2.4.3.)

**Exercise 2.41.** We will now calculate the quartiles (i.e., the 25th, 50th, and 75th percentiles) of the number of people listed in the cast.

- Pipe `netflix` into `summarise()`.
- Use `str_count()` to count the number of commas listed in each of the cast entries.
  - "no, no, Hodge shall not be shot." %>% str_count("\,") = 2.
- The number of people in the cast is always one more than the number of commas, so add one to each element of the vector outputted by `str_count()`.
  - Piping a vector into {. + 1} increases it by one. For example, 1:5 %>% {. + 1} = 2:6. (See section 2.2.6.)
- Pipe the result into `quantile(probs = c(0.25, .5, .75), na.rm = TRUE).`
2.2.9 group_by()

Suppose we wanted to calculate the average release_year for each value of country_primary. We could do so with the technique developed in exercise 2.38:

```
netflix %>%
  summarise(
    average_Argentina = 
      weighted.mean(
        release_year, 
        country_primary == "Argentina"
      ),
    average_Australia = 
      weighted.mean(
        release_year, 
        country_primary == "Australia"
      ),
    # ...
    average_WestGermany = 
      weighted.mean(
        release_year, 
        country_primary == "West Germany"
      )
  )
```

But this solution is ungainly, comprising a `weighted.mean()` for each of the 72 values of country_primary. A better approach is to first `group_by()` country_primary and then `summarise()`:

```
netflix %>%
  group_by(country_primary) %>%
  summarise(mean_release_year = mean(release_year)) %>%
  head(4)
```

> # A tibble: 4 x 2
>  country_primary mean_release_year
>   <chr>                  <dbl>
> 1 Argentina             2015.
> 2 Australia             2015.
> 3 Austria               2017.
> 4 Bangladesh            2018
group_by() is the third member of the functional quartet. It groups the rows of a tibble into chunks. For example, consider the following:

```
netflix %>%
  group_by(type, rating) %>%
  head(3)
```

> # A tibble: 3 x 17
> # Groups: type, rating [3]
>   show_id type  title director cast country date_added release_year rating
>  <int> <chr> <chr> <chr> <chr> <chr> <date> <int> <chr>
> 1 81145628 Movie Norm o~ Richard~ Alan~ United~ 2019-09-09 2019 TV-PG
> 2 80117401 Movie Jandin~ <NA> Jand~ United~ 2016-09-09 2016 TV-MA
> 3 70234439 TV Show Transf~ <NA> Pete~ United~ 2018-09-08 2013 TV-Y7~
> # ... with 8 more variables: duration <chr>, listed_in <chr>,
> # description <chr>, will_smith <lgl>, cast_size <dbl>,
> # country_primary <chr>, wday_added <ord>, month_added <ord>

At the top of the output you’ll find Groups: type, rating [27], which indicates that this tibble is now divided into 27 clusters that correspond to the 27 distinct (type, rating) pairs. Most subsequent operations will then be performed group-wise. For example, the following calculates the mean and standard deviation of the number of characters in the description text string, by type and rating:

```
netflix %>%
  group_by(type, rating) %>%
  summarise(
    length_mean = description %>% str_length %>% mean,
    length_sd = description %>% str_length %>% sd
  ) %>%
  head(3)
```

> # A tibble: 3 x 4
> # Groups: type [1]
>   type rating length_mean length_sd
>   <chr> <chr>     <dbl>     <dbl>
> 1 Movie G    <NA>     144.       18.9
> 2 Movie NC-17 <NA>     142.       9.19
> 3 Movie NR   <NA>     145.       9.72

Usually we group a tibble by the distinct values of some collection of its variables. For example,
• `group_by(wday_added)` divides the tibble into the 7 combinations of `wday_added`,
• `group_by(wday_added, type)` divides the tibble into the 14 combinations of `(wday_added, type)`,
• `group_by(wday_added, type, rating)` divides the tibble into the 154 combinations of `(wday_added, type, rating)`, and
• `group_by(wday_added, type, rating, release_year)` divides the tibble into the 1,555 combinations of `(wday_added, type, rating, release_year)`. But we can also define more exotic groupings. For example, the following defines grouping scheme `year_added = year(date_added)` within the `group_by()` operation, and uses `summarise()` to calculate the mean `release_year` for each corresponding chunk:

```r
netflix %>%
  group_by(year_added = year(date_added)) %>%
  summarise(mean_released = mean(release_year)) %>%
tail
```

> # A tibble: 6 x 2
>   year_added mean_released
>  <dbl>       <dbl>
> 1  2015       2014.
> 2  2016       2013.
> 3  2017       2013.
> 4  2018       2014.
> 5  2019       2014.

Netflix must have added a lot of old shows 2020.

**Exercise 2.42.** We will now count the number of shows added each month.

• Start with `netflix` and `group_by()` `month_added` (which we defined in exercise 2.30).
• Use `summarise()` and `n()` to define `num_shows` as the number of observations in each group.
  – The function `n()` returns the number of observations in a given `group_by()` group.
  – Your "Jan" group should have `num_shows = 610`.
• In practice, we would solve this problem with the specialized `count()` function. (See section 2.4.3.)

**Exercise 2.43.** We will now calculate the *fraction* of shows added in each month.

• Use `summarise()` to define `num_shows`, in the fashion of exercise 2.42.
• ungroup() the tibble produced by summarise().
• Use mutate() to define frac = num_shows / sum(num_shows).
  – The value of frac should sum to 1 across all groups.
  – The ungroup() step makes the sum(num_shows) step sum across the entire tibble,
    rather than across the current month_added group.

**Exercise 2.44.** We found in section 2.2.8 that Netflix adds a disproportionately high number of shows on Fridays and Saturdays. We will now demonstrate a related fact: a high proportion of the programs added on Fridays and Saturdays are TV shows, as opposed to movies.

• Use group_by() to divide netflix into two groups: (i) the programs added on Fridays and Saturdays and (ii) all other programs.
  – Use logical vector wday_added %in% c("Fri", "Sat").
• Use summarise() to calculate for the two groups the fraction of observations in which type = "TV Show".
  – Calculate the mean of the type == "TV Show" logical vector.

**Exercise 2.45.** We will now show how the propensity for evanston arrestee to be armed decreases with age.

• Use group_by() and ntile(n = 10) to divide evanston by the deciles\(^{10}\) of Age.
• Use summarise() to define weapon_frac as the mean() of `Weapon Code` != "".
• plot() the result to show how the fraction of armed suspects varies with Age decile.

**Exercise 2.46.** An "On View" arrest is one where the police officer established probable cause to make an arrest by directly "viewing" a criminal offense. We will now plot how the proportion of evanston arrests made "On View" changes over the course of the day.

• Start with evanston and group_by() hour(Arrest Time).
• Use summarise() and mean(na.rm = TRUE) to calculate the fraction of observations with ‘Arrest Type’ == "On View".
• plot() the result to show how the fraction of "On View" arrests changes by hour.

---

\(^{10}\)https://en.wikipedia.org/wiki/Quantile
```r
netflix %>%
  group_by(rating) %>%
  mutate(avg_release_year_by_rating = mean(release_year))
```

> # A tibble: 6,223 x 18
> # Groups: rating [15]
> show_id type title director cast country date_added release_year rating
> <int> <chr> <chr> <chr> <chr> <chr> <date> <int> <chr>
> 1 81145628 Movie Norm ~ Richard~ Alan~ United~ 2019-09-09 2019 TV-PG
> 2 80117401 Movie Jandi~ <NA> Jand~ United~ 2016-09-09 2016 TV-MA
> 3 70234439 TV Show Trans~ <NA> Pete~ United~ 2018-09-08 2013 TV-Y7~
> 4 80058654 TV Show Trans~ <NA> Will~ United~ 2018-09-08 2016 TV-Y7
> 5 80125979 Movie #real~ Fernand~ Nest~ United~ 2017-09-08 2017 TV-14
> 6 80163890 Movie Apach~ <NA> Albe~ Spain 2017-09-08 2016 TV-MA
> 7 70304989 Movie Autom~ Gabe Ib~ Anto~ Bulgar~ 2017-09-08 2014 R
> 8 80164077 Movie Fabri~ Rodrigo~ Fabr~ Chile 2017-09-08 2017 TV-MA
> 9 80117902 TV Show Fire ~ <NA> <NA> United~ 2017-09-08 2017 TV-MA
> 10 70304990 Movie Good ~ Henrik ~ Jame~ United~ 2017-09-08 2014 R
> # ... with 6,213 more rows, and 9 more variables: duration <chr>,
> # listed_in <chr>, description <chr>, will_smith <lgl>, cast_size <dbl>,
> # country_primary <chr>, wday_added <ord>, month_added <ord>,
> # avg_release_year_by_rating <dbl>

And the following `cp_directors` as the number of distinct `director` values that correspond to the given `country_primary`:

```r
netflix %>%
  group_by(country_primary) %>%
  mutate(cp_directors = n_distinct(director, na.rm = FALSE))
```

For example, there are 30 distinct `director` values across the 48 rows with `country_primary` = "Indonesia", and each of these rows is assigned `cp_directors = 30`.

**Exercise 2.47.** We will now add a variable to `netflix` called `director_count` that specifies the number of times that a given director appears in the sample.

- Start with `netflix` and `group_by() director`.
- Pipe the result into `mutate()`.
- Use `n()` to define `director_count`.
- Exit the `mutate()` step and `ungroup()` the tibble.
  - You should always remove the grouping structure as soon as you’ve finished using it to ensure that you don’t mistakenly apply the groups where they don’t belong (which is a common error).
In practice, we would solve this problem with the specialized `add_count()` function. (See section 2.4.3.)

**Exercise 2.48.** The observations with `director = NA` now have `director_count = 1958`, since there are 1958 programs with missing director values. But we would prefer for these observations to have `director_count = NA`. Use `mutate()`, `ifelse()`, and `is.na()` to set `director_count = NA` if `director = NA`.

**Exercise 2.49.** We will now illustrate a third fact about the programs added on Fridays and Saturdays: a higher proportion of them are made by inexperienced directors.

- Use `group_by()` to divide `netflix` into four groups: (i) TV shows added on Fridays and Saturdays, (ii) TV shows added on Sundays–Thursdays, (iii) movies added on Fridays and Saturdays, and (iv) movies added on Sundays–Thursdays.
- Pipe the result into `summarise()`.
- For each group calculate the fraction of programs that have a unique director value (i.e., a director that isn’t associated with any other show in the sample).
  - Use the `director_count == 1` logical.
  - Use the `mean()` function with `na.rm = TRUE` option.

**Exercise 2.50.** We will now add a variable to `evanston` called `Daily_Arrest_Number`, which is 1 for the first arrest of the day, 2 for the second arrest, and so forth.

- Start with `evanston` and `group_by()` `Arrest Date`.
- Use `mutate()` and `row_number()` to define `Daily_Arrest_Number` as one more than the number of prior arrests made that day. In other words, the nth arrest of the day should have `Daily_Arrest_Number = n`.
  - Do not reorder the rows of the tibble.
  - Consider the output of the following: `c(30, -2, 0, 0) %>% row_number`.

### 2.2.10 filter()

Exercises 2.44 and 2.49 illustrate that the `netflix` programs added on Fridays and Saturdays differ systematically from those added on other days of the week. Suppose we wanted to create a tibble comprising only the Friday and Saturday shows. We could create this tibble with `filter()`, the fourth member of the functional quartet:

```r
fri_sat_shows <- netflix %>%
  filter(wday_added %in% c("Fri", "Sat"))

# Confirm that all shows are Fri or Sat
fri_sat_shows %>%
  distinct(wday_added)
```
> # A tibble: 2 x 1
>   wday_added
>       <ord>
> 1 Fri
> 2 Sat

As the name suggests, the `filter()` function filters (i.e., removes rows from) our sample. It takes one or more conditions (separated by commas) and drops the observations that violate any of them. For example, if you wanted to watch an R-rated, new-release, thriller or horror movie, filmed outside of the United States, then you’d be limited to this set:

```r
netflix %>%
  filter(
    type == "Movie",
    rating == "R",
    str_detect(listed_in, "Horror Movies|Thrillers"),
    !str_detect(country, "United States"),
    release_year >= 2018
  ) %>%
  select(title, type, rating, listed_in, country, release_year)
```

> # A tibble: 4 x 6
> title type rating listed_in country release_year
> <chr> <chr> <chr> <chr> <chr> <int>
> 1 Domino Movie R International Movies, Denmark 2019
> 2 Everybody Knows Movie R Dramas, Internation Movies, Spain 2019
> 3 Between Worlds Movie R Thrillers Spain 2018
> 4 Day of the Dead: Bloodline Movie R Horror Movies, Bulgaria 2018

**Exercise 2.51.** Use `filter()` and `str_detect()` to list the programs directed by "Riri Riza".

**Exercise 2.52.** Use `filter()` and `str_detect()` to list the programs directed by "Riri Riza" and at least one other person.

**Exercise 2.53.** Use `filter()` and `str_detect()` to select all the programs with just one person listed in the cast.

- Remove the programs with a comma listed in the cast character string.
- Use the `negate = TRUE` option of `str_detect()` to identify the rows without a comma.
We can combine `group_by()` with `filter()` to create group-level filters. For example, the following filters out all the programs in `netflix` that do not have a unique `country_primary` value:

```r
netflix %>%
  group_by(country_primary) %>%
  filter(n() == 1)
```

More specifically, the `filter()` step throws out all shows that belong to a `country_primary` group with more than `n() == 1` observations.

The following `filter()` limits the sample to 21st century directors (i.e., those with an average `release_year` larger than 2000):

```r
netflix %>%
  group_by(director) %>%
  filter(
    !is.na(director),
    mean(release_year) >= 2000
  )
```

For example, Robert Altman directed two programs in our sample: Popeye, which was released in 1980, and Gosford Park, which was released in 2001. Hence, the `director == "Robert Altman"` cluster has `mean(release_year) = (1980 + 2001)/2 = 1990.5 < 2000`, and is thus eliminated. Thus, Gosford Park is dropped even though it was filmed after 2000 because it belongs to the defunct `director == "Robert Altman"` group.

For a more complex example, the following lists all shows that were not filmed in Mexico but whose directors have filmed in Mexico:

```r
netflix %>%
  group_by(director) %>%
  filter(
    str_detect(country, "Mexico", negate = TRUE),
    country %>% str_detect("Mexico") %>% any,
    !is.na(director)
  ) %>%
  select(title, director, country) %>%
  tail
```

> # A tibble: 6 x 3
> # Groups: director [4]
The first condition, `str_detect(country, "Mexico", negate = TRUE)`, is a row-level restriction. It returns a vector of logicals, the FALSE values of which condemn select rows. Specifically, it boots the shows with the word "Mexico" appearing in their country text strings (note the negate = TRUE option). The second condition, `country %>% str_detect("Mexico") %>% any`, is a group-level restriction. It returns a single logical, which gets applied to the entire group: if this logical is FALSE then every row of this group is damned. This condition stipulates that the word "Mexico" must appear in at least one of this group's country text strings. For example, the movie Hellboy satisfies the first condition because it was not filmed in Mexico, and it satisfies the second condition because its director, Guillermo del Toro, filmed Pan’s Labyrinth in Mexico.

**Exercise 2.54.** Select the programs filmed by the directors who at some point directed the actor Forest Whitaker. For example, all of David Fincher’s films should be included since he directed Forest Whitaker in Panic Room.

- Start with `netflix` and `group_by()` `director`.
- Use `filter()` to select the groups in which you can `str_detect()` `any()` programs with "Forest Whitaker" in the cast.

**Exercise 2.55.** We will now select the programs made by directors that are `listed_in` at least six distinct genres.

- Start with `netflix` and `group_by()` `director`.
- Use `filter()` and `is.na()` to remove the observations with `director = NA`.
- Within the same `filter()` call use `n_distinct()` to drop the `director` values that correspond to fewer than six different `listed_in` values.

**Exercise 2.56.** We will now select the shows whose cast members have all previously worked together.

- Start with `netflix` and `group_by()` `cast`.
- Use `filter()` to remove the shows with `NA` `cast` values.
- Within the same `filter()` call remove all shows whose `release_year` is not the minimum value for the given `cast` value. That is, keep a show only if there was another show with the same `cast` released in an earlier year.
Exercise 2.57. We will now filter evanston down to the days in which at least two handguns were confiscated.

- Start with evanston and group_by() `Arrest Date`.
- filter() away the date groups with fewer than two `Weapon Code` == "Handgun" observations.
- ungroup() the data

2.3 Lab: Italian Judiciary

2.3.1 Overview

The Roman Labor Court of Appeals has been struggling to keep up with demand: e.g., the median judge had a backlog of over 500 open cases waiting for a verdict in 2010. In an attempt to reduce case completion times, Bray et al. [2016] experimented with a new court scheduling scheme in 2011. Previously, the judges arranged their dockets according to a first-in-first-out (FIFO) basis, giving precedence to whichever case has been waiting longest for their attention. However, there are two ways to implement a FIFO scheduling policy in this context: at the hearing level (which is what the judges were doing previously) and at the case level (which is what the researchers proposed they switch to). Whereas the hearing-level FIFO policy prioritizes cases by the length of the time since their last hearing, the case-level FIFO policy prioritizes cases by when they were filed. Specifically, under hearing-level FIFO, a judge picks the case that’s had the longest wait since the last hearing, and holds the next hearing for that case. And under case-level FIFO, a judge picks the case that’s had the longest wait overall, and works on it until completion.

For example, suppose a judge has cases A, B, and C, all of which require hearings 1, 2, and 3. Under case-level FIFO, the judge will first order by case and then by hearing, observing the following hearing schedule: A1, A2, A3, B1, B2, B3, C1, C2, C3. But under hearing-level FIFO the judge will first order by hearing and then by case, observing the following hearing schedule: A1, B1, C1, A2, B2, C2, A3, B3, C3. Now if the judge holds one hearing per day then he would finish case A on day three, case B on day six, and case C on day nine under case-level FIFO, and would finish case A on day seven, case B on day eight, and case C on day nine under hearing-level FIFO. Hence, case-level FIFO would have an average completion time of \((3 + 6 + 9) / 3 = 6\) days whereas hearing-level FIFO would have an average completion time of \((7 + 8 + 9) / 3 = 8\) days.

Bray et al. [2016] used this example to convince six judges in the Roman labor court to switch from hearing-level to case-level FIFO on January 1, 2011. Bray et al. then tracked how the scheduling efficiency of these “treated” judges changed after this intervention date, relative to the “control” judges who did not adopted case-level FIFO. Their experimental results suggest that the scheduling change reduced case durations by 12%. We will replicate the core of Bray et al.’s analysis, to create figure 4.2, which depicts the time series of their primary efficiency metric: the case-completion hazard rate (i.e., the fraction of hearings that complete a case). The plot illustrates that the completion hazard rates of the treated
judges matched those of the control judges before the January 1, 2011 intervention date, and exceeded them afterwards. Accordingly, this exhibit suggests that the intervention increased the treated judges’ efficiency, relative to the control judges.

Figure 2.1: Case-Completion Hazard Rate Estimates

This lab is real data science: every exercise will execute a data transformation that Bray et al. actually implemented. So none of the problems in this lab are make-work—each step was important enough for Bray et al. to incorporate in their methodology. Before proceeding, you may want to skim Bray et al.’s article, to get a better sense for what we’re about to do.

2.3.2 Data

We will use the court_data tibble that you defined in section 1.3, exercise 1.28. It should have 12 variables:

- **cid** is a case ID number. Each row in the tibble corresponds to a hearing, and cid indicates which case the given hearing belongs to. For example, there are four observations with \( \text{cid} = 152339 \), and thus this case had four hearings.
- **jid** is a judge ID number. That is, the 71 jid values represent the court’s 71 judges.
- **fdate** records the date the case was filed, and **hdate** records the date the hearing was held.
- **treated** is a logical that indicates whether the judge received treatment (i.e., switched to case-level FIFO). This value is constant for 68 judges. But three judges in our

---

11 Cramming all this disparate data into one large tibble is a poor way to store it. For example, the tibble has a lot of redundancy, as it needlessly repeats case-level attributes ptype, dtype, and ctype for each hearing. It would be more efficient to store the case-level variables in one tibble and the hearing-level variables in another tibble. However, we won’t cover joining tibbles until chapter 4 so we’ll stick to one tibble for this lab.

12 More technically, jid identifies the judge who serves as the rapporteur for the case. As Bray et al. [2016] explain, “Each case is assigned to a [three-judge] panel for adjudication, and to a rapporteur, a judge on the panel, for supervision. The rapporteur analyzes the testimony, oversees the ruling, writes the opinion, and schedules the hearings. Since panels are stable ... we treat a case’s rapporteur as its sole judge.”
sample switched from the control group to the treated group.\textsuperscript{13} So there are three jid values that have both treated = TRUE and treated = FALSE observations. (We will remove these jid values from our sample in exercise 2.58.)

- **closed** is a logical that indicates whether the case had concluded by the time that the sample was collected.
- **settled** is a logical that indicates whether the case ended in a settlement. Note, settled = TRUE implies closed = TRUE.
- **party.count** reports the number of interested parties.
- **ptype** classifies the plaintiff, **dtype** classifies the defendant, and **ctype** classifies the case (see Table 1 of Bray et al.’s article).
- **cassa** is a logical that indicates whether the verdict was appealed to the Supreme Court of Cassation.

### 2.3.3 Clean

**Exercise 2.58.** We will now remove the judges that switched from the control group to the treated group midway through the experiment. We can identify these judges because they are the only ones with both treated = TRUE and treated = FALSE observations.

- **Start with court_data and group_by() jid.**
- **Use filter() and n_distinct() to remove jid chunks with multiple distinct treated values.**
- **ungroup()** the tibble, and save the result as **court_data** (overwriting the previous object)
  - court_data should have 193452 rows after this step.
  - You can get the number of rows in a table with dim() or nrow().

**Exercise 2.59.** Rather than keep track of two types of dates, Bray et al. treated the filing of the case as “hearing zero.” We will now make this change, which will enable us to combine fdate and hdate into a single date column. We will implement this transformation in three steps. First we will capture the hearing dates in a tibble called hearing_dates:

- **Start with court_data and select() away the fdate column.**
- **rename()** hdate to date.
- **Use filter() and is.na() to remove values in which date is NA.**
- **Call the resulting tibble hearing_dates.**
  - This tibble should have 192949 rows and 11 columns.

Now we will capture the filing dates in a tibble called file_dates:

\textsuperscript{13}More specifically, treated indicates whether the judge belongs to the colegio that received treatment. As Bray et al. explain, the court is divided into five colegios, only one of which implemented our scheduling policy. Three judges transferred into the treated colegio after the experiment was underway.
Start with `court_data` and `select()` away `hdate`.
- `rename()` `fdate` to `date`.
- Save the `distinct()` observations of this tibble as `file_dates`.
  - There should be as many rows in `file_dates` as there are cases in `court_data`.
  - Unlike `hdate`, the `fdate` variable has no `NA` values.

Now we will combine `hearing_dates` and `file_dates` into a new `court_data` tibble:

- Use `bind_rows()` to stack `file_dates` on top of `hearing_dates`.
  - This tibble should have 279542 rows.
- Use `mutate()` and `dmy()` to convert `date` from a character variable to a Date variable.
- `group_by()` `cid`.
- Use `mutate()` to define `hid = row_number(date) - 1`. This hearing ID number is 0 for the earliest `date`, 1 for the second-earliest `date`, and so on. Accordingly, the the `hid = 0` observations correspond to case filings and the `hid > 0` observations to hearings.
- `ungroup()` the resulting tibble and use it to overwrite `court_data`.

**Exercise 2.60.** Since (i) the court sometimes fails to mark completed cases as closed and (ii) it’s extremely unlikely for an active case to go two years without having a hearing, we will set `closed = TRUE` for all cases that didn’t have a hearing in the last two years of our sample.

- Define `cutoff_date` as the day two years before the last `date` in our sample.
  - `cutoff_date` should be a standalone Date variable, not a tibble or an element of a tibble.
  - Do not hard-code the value of `cutoff_date`. Instead, derive it from `court_data` so that when the tibble changes, the variable updates accordingly.
  - See what `today() - years(2)` yields.
  - `summarise()` always outputs a tibble, but you can `pull()` out statistics from it. For example, `court_data %>% summarise(first_date = first(date)) %>% pull(first_date)` outputs a Date.
- `group court_data by cid`.
- Use `ifelse()` to set `closed = TRUE` if the given `cid` hasn’t had a hearing since `cutoff_date`.
  - Inequalities hold for Date variables. For example, `today() < today() + days(1) = TRUE`.
- `ungroup()` the tibble and save it as the new `court_data`.
  - 75.3% of the `closed` values in `court_data` should now be `TRUE`.

**Exercise 2.61.** We will now add a variable to our tibble that indicates whether the given hearing concludes a case.
• Start with `court_data` and `group_by()` `cid`.
• Use `mutate()` to define `last_hearing` as a logical that is `TRUE` if and only if (i) the case is completed and (ii) there are no later hearings. For example, the case with `cid = 109211` has `closed = FALSE` and so all its hearings have `last_hearing = FALSE`, whereas the case with `cid = 78087` has `closed = TRUE` and so its last hearing, held on `2010-07-22`, has `last_hearing = TRUE`.
• `ungroup()` the tibble and use it to overwrite `court_data`.
  – The `last_hearing` column of `court_data` should sum to 62823.

Exercise 2.62. Use `filter()` to remove all hearings that were held more than 2,000 days before the January 1, 2011 intervention date.

• See what `today() - days(2000)` yields.
• `court_data` should have 274066 rows after this step.

Exercise 2.63. Use `filter()` and `n_distinct()` to remove all judges that saw fewer than 100 cases.

We have now finished cleaning our sample. We have six treated judges and 51 control judges. Bray et al. had only 44 control judges, because they imposed a few more data filters than we have. (We’ll skip these data filters because they have little effect on the overall results.)

2.3.4 Analyze

We will now show that the efficiency of the treated judges increased relative to that of the control judges after the intervention date. Since the goal is to reduce case adjudication times it seems the most sensible test would be to compare the change in the treated and control judges’ case durations, before and after January 1, 2011. Unfortunately, this comparison is impractical because cases last so long. For example, 10% of cases last over four years, so directly measuring the new average case durations would require around a decade of observation: say, three years to build up a cohort of new cases and seven years to wait for the last of them to finish.

To get a more timely estimate, Bray et al. [2016] conducted a more subtle empirical test. They noted that the average case duration is a function of two factors: the arrival rate of new cases and the number of open cases.\footnote{In fact, a classic result from operations management establishes that the average case duration exactly equals the average arrival rate of new cases times the average number of open cases. This result is known as (Little’s law)\[https://en.wikipedia.org/wiki/Little%27s_law].} And since their intervention can’t influence the arrival rate of new cases—which the court has no control over—it can only reduce case durations by reducing the number of open cases. And there are three ways to reduce the number of open cases: (i) decrease the rate at which new cases arrive, (ii) increase the rate at which judges hold hearings, and (iii) increase the fraction of hearings that conclude a case, otherwise known as the case-completion hazard rate. We’ve already established that the intervention
cannot influence the first mechanism, and it cannot influence the second mechanism because re-prioritizing the cases has no bearing on the number of hearings a judge holds per day. Hence, the only way the new scheduling policy can reduce case durations is by temporarily increasing the case-completion hazard rate—i.e., by moving nearly completed cases to the front of the queue. Thus, we will test whether switching to case-level FIFO decreased case durations by measuring whether it increased case-completion hazard rates.

In the following exercise we will create a plot similar to figure 5 of Bray et al. [2016] that demonstrates the relative increase in the treated judges’ hazard rates after the intervention.

Exercise 2.64. We will now plot the treated and control judges’ hazard rates, month by month. To begin, we will calculate the hazard rates by month and treated:

- Use `filter()` to remove the hid = 0 observations from `court_data`.
  - We want to focus on hearings, which involve judges, and not on case filings, which do not involve judges.
- `group_by()` treated, `year(date)`, and `month(date)`.
  - There should be 221 groups.
  - If we grouped by `month(date)` but not `year(date)` then the November 2005 hearings would be lumped in with the November 2006.
- Use `summarise()` to define `hazard_rate` as the fraction of hearings in the given group that completed a case.
  - The fraction of hearings that completed a case equals the fraction of hearings with `last_hearing = TRUE`, which you can calculate with `mean()`.
- Within the same `summarise()` call, use `first()` and `floor_date()` to define `date_start` as the first day of the given month.
  - `floor_date(ymd("1865-June-19"), "month") = "1865-06-01"`.
  - `floor_date()` maps vectors to vectors. But we want to define `date_start` as a single date, not a vector of dates. So apply `first()` before applying `floor_date()`, to translate the list of dates to a single date. For example, `c("1865-06-19", "1865-06-20", "1865-06-21") %>% first = "1865-06-19"`.
  - There’s nothing special about `first()` here. Indeed, basically any function that maps a vector to one of its individual elements—e.g., `last()`, `min()`, or `median()`—would do in this case.
  - Your tibble should have 221 rows after this step.
- Within the same `summarise()` call, use `n()` to define `num_obs` as the number of observations in the current group and define `sigma = sd(last_hearing)`. (We won’t use `sigma` until the next exercise.)
- `ungroup()` the tibble.
- Remove the rows with `num_obs < 50`, since we need at least 50 observations to get a reasonable estimate of the case-completion hazard rate.
- Call the resulting tibble `data_to_plot`.
• data_to_plot should have 201 rows.

• data_to_plot tibble comprises two time series: one with treated = TRUE and one with treated = FALSE. We will now plot these time series with the ggplot() package, which we will discuss at length in the following chapter. This package enables you to define plots without data. For example, the following object defines the parameters of the plot I would like you to make:

```r
hazard_rate_plot <-
ggplot() +
aes(
    x = date_start,
    y = hazard_rate,
    color = treated
) +
geom_point() +
geom_vline(xintercept = dmy("1/1/2011")) +
theme_bw()
```

To turn hazard_rate_plot into a full-fledge plot, we must fill it with data. To do so, enter hazard_rate_plot %+% data_to_plot into your console. You should receive a scatter plot in return.

The plot you created illustrates how the treated and control judges’ hazard rates vary over time. The vertical black line denotes the intervention date. As you see, the treated = TRUE hazard rates roughly track the treated = FALSE hazard rates before this date, and noticeably exceeded them after. To further highlight the statistical difference between the treated = TRUE and treated = FALSE hazard rate estimates, we will add 95% confidence intervals to each point estimate in our plot.

**Exercise 2.65.** Each point in the plot we made in the previous exercise corresponds to mean value of last_hearing across a collection of observations. Hence, each point in this plot reports a sample average. And we can calculate the 95% confidence interval of any sample average. Specifically, the 95% confidence interval of $X = \frac{x_1 + \ldots + x_N}{N}$ is lower_limit to upper_limit, where

- lower_limit = $X - 1.96 \cdot \frac{\sigma}{\sqrt{N}}$,
- upper_limit = $X + 1.96 \cdot \frac{\sigma}{\sqrt{N}}$, and
- $\sigma = \sqrt{\frac{(x_1 - X)^2 + \ldots + (x_N - X)^2}{N}}$. 

127
We will add these 95% confidence intervals to our plot by adding a `geom_errorbar()` layer to our plot blueprints:

```

hazard_rate_plot <-
  hazard_rate_plot +
  geom_errorbar(
    aes(       
      ymin = lower_limit,
      ymax = upper_limit
    )
  )

```

Now each dot in `hazard_rate_plot` has a corresponding vertical line, which runs from `lower_limit` to `upper_limit`.

- Use `mutate()` to add these limit variables to `data_to_plot`, setting `lower_limit = hazard_rate - qnorm(.975) * sigma / sqrt(num_obs)` and `upper_limit = hazard_rate + qnorm(.975) * sigma / sqrt(num_obs)`.
  - `hazard_rate`, `qnorm(.975)`, `sigma` and `num_obs` correspond to $X$, 1.96, $\sigma$, and $N$ in the mathematical expressions above.
- Run `hazard_rate_plot %+% data_to_plot` to update your plot.

Now the figure should depict 201 point estimates (dots) embedded within 201 confidence intervals (vertical bars). Roughly speaking, we can consider the difference between the `treated = TRUE` and `treated = FALSE` hazard rates “statistically significant” if the corresponding error bars don’t overlap. Hence, most of the `treated = TRUE` hazard rates are significantly higher than their `treated = FALSE` counterparts after the intervention.

### 2.3.5 Explore

At the end of their study, Bray et al. [2016] explained that

About a year after our intervention, the treated judges reported a serendipitous side effect: they forgot fewer case facts under case-level FIFO because of the reduced time between hearings. They speculated that better remembering the cases led to fairer rulings.
To test whether the intervention lead to fairer rulings, the authors measured whether the rate at which cases were appealed to the Supreme Court of Cassation decreased after the intervention. More specifically, they measured a “difference in differences,” comparing the change in the appeals rate of the treated judges, before and after the intervention, to the change in the appeals rate of the control judges, before and after the intervention. They found that treated judges’ appeals rate dropped more than the control judges’ appeals rate, which suggests that switching to case-level FIFO improved the quality of the rulings, because, as the authors explain, “Unjust rulings should be more frequently appealed.”

Exercise 2.66. Confirm the claim of Bray et al. [2016] that the rate of verdict appeals (i.e., fraction of cases with `cassa = TRUE`) decreased more for the treated judges than for the control judges after the intervention date. Perform a separate analysis for each `ctype`.

2.4 Commentary

2.4.1 Data Factory Layout

We’ve now seen several examples of functional assembly lines. I’ll conclude this lecture by explaining how arrange these assembly lines into a “data factory” that can process a statistical sample.

The vast majority of R functions either transform tibbles into other tibbles or vectors into other vectors. Of course, there are tibble-to-vector functions, such as `pull()`, and vector-to-tibble functions, such as `as_tibble()`, but they are rare. Hence there are are two basic flavors of functional assembly line: tibblewide lines, which exclusively comprise tibble-to-tibble functions, and vectorwide lines, which exclusively comprise vector-to-vector functions. Since we’ll store nearly all our data in tibbles we’ll perform nearly all our data analysis with tibblewide assembly lines. Thus, our data factories will generally have the following form:

```r
output_tibble <-
  input_tibble %>%
  tibble_fun_1 %>%
  tibble_fun_2 %>%
  tibble_fun_3
```

However, on closer inspection you’ll notice that many of these tibble-to-tibble functions contain vectorwide assembly lines in their parentheses. Thus, our data factory will actually look more like the following:

```r
output_tibble <-
  input_tibble %>%
  129
```
tibble_fun_1(
  tibble_column %>%
  vector_fun_1A %>%
  vector_fun_1B %>%
  vector_fun_1C
) %>%
tibble_fun_2(
  tibble_column %>%
  vector_fun_2A %>%
  vector_fun_2B %>%
  vector_fun_2C
) %>%
tibble_fun_3(
  tibble_column %>%
  vector_fun_3A %>%
  vector_fun_3B %>%
  vector_fun_3C
)

For example, in the previous sections we embedded vectorwide assembly lines inside of
tibblewide assembly line functions whenever we nested the %>% symbol inside of `summarise()`,
`mutate()`, and `filter()` steps. For additional examples skim the solutions of the subsiquent
chapters and you’ll find most of the answers have this form: a master tibblewide assembly
line connecting a set of nested vectorwide assembly lines.

The data processing factories we will construct resemble the Intel computer chip factory I
used to work at. We make computer chips on round plates of silicon, 300-millimeters in
diameter, called “wafers.” A given wafer could stores 148 computer chips. Since a speck
of dust could destroy a chip, we keep these wafers in air-tight boxes called Front-Opening,
Unified Pods (FOUPs). A given FOUP stores 25 wafers, stacked vertically side by side.
These FOUPs move down an assembly line from from station to station via an Automated
Material Handling System (AMHS) (which is basically a high-tech conveyor belt). When a
FOUP arrives at a work station—say a photolithography machine for etching, a diffusion
furnace for doping, or a wet bench for cleaning—it docks and creates an air lock with the
tool. The FOUP then passes its wafers into the air-tight machine, which processes them
with a sequence of operations. When the machine is done it passes the wafers back to the
FOUP, which closes its “front-opening” door and undocks. Finally, the FOUP hops back
on the AMHS conveyor belt, headed for the next work station.

In this analogy the computer chips represent individual units of data—e.g., single numbers
or character strings. Comprising 148 computer chips, the wafers thus correspond to vectors
of length 148. And comprising 25 wafers, the FOUPs correspond to tibbles with 25 columns
and 148 rows. Finally, the various work stations—the photolithography machines, diffusion
furnaces, wet benches, etc.—represent tibble-to-tibble functions, as they receive FOUP
inputs and return FOUP outputs. However, although they receive and return FOUPs,
these machines work with the silicone wafers—like how tibble-to-tibble functions work with the vector columns of a tibble. Hence, the processing these machines do to the wafers is analogous to the processing the nested vectorwide assembly lines do to the vector columns. And these vector columns are always safely in a tibble or a function’s parentheses like how the semiconductor wafers are always safely in a FOUP or air-tight toolset.

### 2.4.2 Nested Functions

Instead of arranging functions in an assembly line we could nest them one inside of one another, like Matryoshka dolls. And from a *computational perspective* the nesting approach is equivalent to the piping approach: $f_n( \ldots f_3(f_2(f_1(x))) \ldots )$ equals $x \%\% f_1 \%\% f_2 \%\% f_3 \%\% \ldots \%\% f_n$. But from a *design* perspective, the piping approach is superior.

For example, compare the following two solutions to exercise 2.15 (you won’t need to understand the code to follow my argument):

```r
#Assembly-line solution
# tibble(date = today() + days(-10^6:-1)) \%\% mutate(date_char = str_replace_all(date, "-", "")) \%\% group_by(date, date_char) \%\% summarise(digits = as.character(0:9)) \%\% mutate(digit_count = str_count(date_char, digits)) \%\% group_by(digits) \%\% summarise(m = mean(digit_count)) \%\% pivot_wider(names_from = digits, values_from = m)

#Nested solution
# pivot_wider(
# summarise(
# group_by(
# mutate(
# summarise(
# group_by(
# mutate(
# tibble(date = today() + days(-10^6:-1)),
# date_char = str_replace_all(date, "-", "")),
# date, date_char ),
# digits = as.character(0:9),
# ),
# digit_count = str_count(date_char, digits)
# ),
```

131
The nested solution is worse in three ways. First, it presents the functions in reverse-
chronological order: whereas the assembly-line solution arranges the functions by their
order of operation, from `tibble()` to `pivot_wider()`, the nested solution arranges them in
the opposite order, from `pivot_wider()` to `tibble()`. Hence, the nesting protocol forces us
to read the code backwards (i.e., to move our eyes from bottom-right to top-left). The
reason is that R runs nested code inside out, starting with the innermost operation and
working its way backward to the outermost operation. Hence, the first function portrayed
is the last function executed, and vice versa.

Second, it’s harder to match functions with their inputs under the nested format:
whereas the assembly-line solution writes a function and its inputs in the same line, the
nested solution puts space between most function calls and their inputs. For example,
16 lines of code separate the outermost `summarise()` call and its second input, \( m = \text{mean(digit\_count)} \). For another example, try to quickly determine which function the
`digit\_count = \text{str\_count(date\_char, digits)}` input belongs to.

To compensate for the vertical distance between functions and their inputs, we’re compelled
to use indenting under the nested scheme, without which it would be nearly impossible
to pair functions with their inputs. For example, there are two indentations before \( m = \text{mean(digit\_count)} \), which indicates that this input corresponds to the second-outermost
function, which is `summarise()`. However this indenting technique leads to the third problem
of nested code: the loss of horizontal space. Specifically, since each nest introduces another
indent, we can only nest so deeply before we “tab” away the entire line. In contrast, the
unnested code is fully left-justified, and thus has no reserved dead white space.

For these reasons, it’s better to arrange a long sequence of functions into an assembly line
rather than a nested stack. However, there is a time and a place for nesting. For example,
we’ll use nesting in chapter 4 to streamline our joins.

### 2.4.3 Specialized Functions

`mutate()` and `summarise()` allow us to transform variables and derive summary statistics in a
general fashion. But some variable transformations and summary statistics are so common
that R has tailor-made functions for them. We’ll now use these specialized functions to
streamline some of our exercise solutions.

First, in exercise 2.31 we used `mutate()` and `word()` to separate the two “words” of `duration`
into two variables. But in practice we would do this with the built-in `separate()` function:
The opposite of `separate()` is `unite()`, which pastes two columns together. For example, we can undo the `separate()` operation above, making `duration` revert back to its old definition, with:

```r
netflix <-
  netflix %>%
  unite(
    duration,
    c("duration", "duration_metric"),
    sep = " ",
  )
```

Second, in exercise 2.40 we used `summarise()` and `unique()` to get the distinct values of the first directors. But in practice we would do this with the built-in `distinct()` function:

```r
netflix %>%
  distinct(director %>% str_remove(".*")) %>%
  na.omit
```

> # A tibble: 3,209 x 1
> `director %>% str_remove(".*")` <chr>  
> 1 Richard Finn  
> 2 Fernando Lebrija  
> 3 Gabe Ibáñez  
> 4 Rodrigo Toro  
> 5 Henrik Ruben Genz  
> 6 José Miguel Contreras  
> 7 Daniel Alfredson  
> 8 Munjal Shroff  
> 9 Tilak Shetty  
> 10 Gaspar Noé  
> # ... with 3,199 more rows
Third, in exercise 2.43 we used `summarise()` and `n()` to count the number of shows in each `month_added` group. But in practice we would do this with the built-in `count()` function:

```r
netflix %>%
  count(month_added)
```

Note, this function enables us to count the number of elements in rather general groupings. For example, the last line in the output below indicates that there are 12 TV shows in `netflix` whose primary country is either "Russia" or the "Soviet Union".

```r
netflix %>%
  count(
    type,
    USSR = country_primary %in% c("Russia", "Soviet Union")
  )
```

Fourth, in exercise 2.47 we used `mutate()` and `n()` to add the number of programs the director has directed as a new variable. But in practice we would do this with the built-in `add_count()` function:

```r
netflix %>%
  add_count(
    director,
    name = "director_count"
  )
```

And fifth, in exercise 2.6 we used `mutate()`, `ifelse()`, and `is.na()` to turn the `NA` values in the `director` column to "*missing data*". But in practice we would do this with the built-in `replace_na()` function:

```r
netflix %>%
  replace_na(list(director = "*missing data*"))
```

### 2.4.4 Define Multivariate Functions

In section 2.2.3 we learned how to define functions with the `<- . %>%` notation. For example, the following defines `appender()` as a function that appends the suffix "*ing" to its input:
appender <-
  . %>% str_c("ing")

"I love R code" %>% appender

> [1] "I love R codeing"

However, the <- . %>% approach only allows us to define functions of one variable. To define functions of two or more variables we need to use the more cumbersome <- function(){} approach. For example, the following defines appender() as a function of two variables, string and suffix:

appender <- function(string, suffix = "ing") {
  string %>% str_c(suffix)
}

"I love R code" %>% appender

> [1] "I love R codeing"

"I love R code" %>% appender("rs")

> [1] "I love R coders"

Note, when defining our function, we give suffix argument the default value of "ing". Thus, if we don’t specify another suffix—i.e., if we only pass one input into the function—it’ll use "ing".

Fortunately, you don’t have to remember the complex <- function(){} notation, because you can have RStudio add it for you. For example, in an R coding window write:

string %>% str_c(suffix)

Now highlight this text and click on Code/Extract Function. A window should pop up asking you for a “Function Name.” Input the word “appender” and RStudio should write the following in your coding window:
appender <- function(string, suffix) {
  string %>% str_c(suffix)
}

As you see, RStudio is smart enough to identify `string` and `suffix` as the arguments for this function.

Let’s do one more example. Write the following in a coding window:

```r
all(vec_1 >= vec_2) | all(vec_2 >= vec_1)
```

Now highlight it, click Code/Extract Function, and input the name “is_one_vector_dominant.” Now you should have the following in your coding window:

```r
is_one_vector_dominant <- function(vec_1, vec_2) {
  all(vec_1 >= vec_2) | all(vec_2 >= vec_1)
}
```

This is a function that takes two vectors and returns `TRUE` if all the elements of one vector are greater or equal to all the elements of the other vector and returns `FALSE` otherwise. For example:

```r
is_one_vector_dominant(1:3, c(-1, 0, 1))
```

> [1] TRUE

```r
is_one_vector_dominant(1:3, c(-1, 0, 10))
```

> [1] FALSE

### 2.4.5 Download Packages from CRAN

In section 2.2.7 I alluded to the fact that we have “thousands of R functions at our disposal.” And this is indeed the case: because the language is open-source—anyone can contribute packages to the Comprehensive R Archive Network (CRAN)—almost any R function we could want has already been written. We just have to find and download it.
For example, suppose we sought a function that computes the moving average of the past three values, transforming vector \( c(x_1, x_2, x_3, \ldots x_{20}) \) to vector \( c(\text{NA}, \text{NA}, (x_1 + x_2 + 3)/3, (x_2 + x_3 + x_4)/3, \ldots, (x_{18} + x_{19} + x_{20})/3) \). To find such a function, google “Moving average tidyverse”. (Searching by “tidyverse” tends to yield more up-to-date solutions than searching by “R”.) Your google results will list several packages that include a moving average function: e.g., forecast, smooth, zoo, and ReppRoll. And this is usually the case, as most functions have several competing versions online. Now you’ve got to pick one of these packages. There are several factors to consider when deciding which package to choose:

- Age: The R language has matured greatly in the past decade, so it’s crucial to choose a package with a modern design sensibility (e.g., one that’s compatible with the tidyverse). Moreover, the latest version is usually the best version.
- Documentation. The R community has a tradition of excellent documentation. For example, it’s not uncommon for a programmer to describe a package with a comprehensive Journal of Statistical Software article or an elaborate vignette(). If you find a package with extremely thorough documentation, you should probably go with it. Of course, a well-written manual is useful, but more importantly the meticulous documentation signals the programmer’s competence and effort.
- Size: R packages generally comprise an entire suite of functions. And when we load a package with library() we add all of them to our workspace. Hence, the more functions a package comprises the more clutter it adds to our workspace. Thus, you should favor smaller packages.
- Relevance: Try to avoid packages that focus on something other than what you’re looking for. For example, the abjutils package has an escape_unicode() function that “removes all accented characters from the current file, replacing them by their equivalent Unicode-escaped values.” But this package doesn’t specialize in text manipulation, as it provides general “Useful Tools for Jurimetical Analysis Used by the Brazilian Jurimetrics Association.” So escape_unicode() is almost certainly not the best accent-removal function we can find.
- Popularity: Give preference to more widely adopted packages. These packages are better vetted and have more support—e.g., more blog posts and more stackoverflow\(^\text{15}\) answers.

For example, when deciding between the various moving average options, I found an article on tidyverse.org\(^\text{16}\) that recommends the RcppRoll package. And the imprimatur of the tidyverse is the highest possible endorsement, so I choose this option. To install this package, I simply enter install.packages("RcppRoll") into the console, which downloads the package from CRAN to my computer.

Once you’ve installed this package, you can load it into your workspace with library() to access its functions. For example, the following calculates the moving average of 1:20:

\(^{15}\text{https://stackoverflow.com/}\)
\(^{16}\text{https://dplyr.tidyverse.org/articles/window-functions.html}\)
library(RcppRoll)

roll_mean(
  1:20,
  n = 3,
  fill = NA,
  align = "right"
)

> [1] NA NA 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Or you can execute this function without loading the RcppRoll package into your workspace with:

RcppRoll::roll_mean(
  1:10,
  n = 5,
  fill = NA,
  align = "right"
)

> [1] NA NA NA NA 3 4 5 6 7 8

This `package::function()` notation enables you to run any function from any package installed on your computer but not loaded in your R workspace.

You should use this technique to keep your workspace streamlined. In fact, you should load a package into your R workspace only if you intend to use it extensively. Otherwise, it’s better to keep it unloaded and reference its functions with the `package::function()` notation. As I mentioned above, loading packages clutters your workspace. But loading many packages can cause more serious problems than an untidy working environment: as taking multiple medications can lead to undesirable drug interactions loading multiple packages can lead to undesirable function interactions. The root of the problem is that functions from different packages can share the same name. For example, both the plyr and the dplyr packages have a function called `summarise()`. So if we run `library(dplyr)` and then `library(plyr)`, the `summarise()` function of the latter package “masks” that of the former package. Now, of course, you could still implement the masked version by calling `dplyr::summarise()`, but chances are you wouldn’t even be aware that this function is masked. Instead, you would more likely just write `summarise()` as usual and mistakenly apply the wrong version of the function.

This problem—having multiple variants of a given function—is quite common when you have, say, a dozen different packages loaded. What’s more, these bugs are especially difficult
to root out because the code is technically correct, so there’s no blatant “error” that you can spot. And once you’ve gotten burned by this a couple times you start to get paranoid, losing faith in the code you’re writing as you incessantly ask yourself “Is this function masked?” Hence, it’s best to avoid this whole masking issue by loading only the essential packages.

2.4.6 Glossary of forcats Functions

Having defined mutate() and functional assembly lines, I can now properly illustrate forcats’ factor-manipulating functions. We will use four such functions: fct_recode(), fct_relabel(), fct_reorder(), and fct_relevel(). I will use these functions to manipulate the following tibble:

```r
mini_tib <-
  tibble(
    X = c(1, 5, -1, 0, -3, -2),
    Y =
      c("a", "b", "c", "a", "b", "d") %>%
      as.factor
  )
```

> # A tibble: 6 x 2
>   X  Y
>  <dbl> <fct>
> 1   1   a
> 2   2   b
> 3   3   c
> 4   4   a
> 5   5   b
> 6   6   d

The fct_recode() and fct_relabel() functions change the factor labels. fct_recode() allows us to change the labels by hand:

```r
mini_tib %>%
  mutate(
    Y_2 =
      fct_recode(
        Y,
        A = "a",
```

139
C_or_D = "c",  
C_or_D = "d"
)
)

> # A tibble: 6 x 3
>    X  Y Y_2
>  <dbl> <fct> <fct>
> 1    1  a    A
> 2    5  b    b
> 3   -1  c    C_or_D
> 4    0  a    A
> 5   -3  b    b
> 6   -2  d    C_or_D

mini_tib

> # A tibble: 6 x 2
>    X  Y
>  <dbl> <fct>
> 1    1  a
> 2    5  b
> 3   -1  c
> 4    0  a
> 5   -3  b
> 6   -2  d

And `fct_relabel()` allows us to change the labels by function (or functional assembly line):

```
mini_tib %>%
mutate(
    Y_2 =
        fct_relabel(
            Y,
            ~ .x %>%
                str_to_upper %>%
                str_c("!")
        )
    )
```
The `fct_reorder()` and `fct_relevel()` functions change the factor orderings. `fct_reorder()` allows us to change the labels by hand:

```
mini_tib %>%
  mutate(
    Y_2 = fct_relevel(Y, "b", after = 3),
    Y_2 = fct_relevel(Y, "a", after = Inf)
  ) %>%
  summarise(
    levels(Y),
    levels(Y_2)
  )
```

And `fct_reorder()` allows us to change the labels by function:

```
mini_tib %>%
  mutate(
    Y_2 = fct_reorder(Y, X, min),  # order by min(X)
    Y_3 = fct_reorder(Y, X, median),  # order by mean(X)
  ) %>%
  summarise(
    levels(Y),
    levels(Y_2),
    levels(Y_3)
  )
```
In general, \texttt{fct_reorder()} receives (i) a factor variable \( y \), (ii) another variable \( x \), and (iii) a function \( f \). It then evaluates \( f(x) \) across each \( y \) grouping, and arranges the levels of \( y \) so that the \( f(x) \) values are in ascending order.

### 2.4.7 Standardize Objects to Standardize Code

Suppose you want to produce hot dogs, some of which must be kosher. You could operate kosher and non-kosher assembly lines, with different lines following different rules, or you could make your entire facility kosher, so that all lines follow the same rules. The latter solution benefits from standardization, as one process is generally easier to manage than two.

Standardization is analogously useful when coding. Indeed, a general principle of programming is to make similar things the same, so that we can govern them with the same code. For example, the initial \texttt{court_data} sample we loaded in section 2.3.2 had two similar variables: \texttt{fdate}, which denoted when a case was filed, and \texttt{hdate}, which denoted when a hearing was held. However, rather than discriminate between these two types of dates, which necessitated us maintain two distinct variables, it’s easier to think of them as two special cases of the same thing, so we can distill \texttt{fdate} and \texttt{hdate} down to one universal date variable. To do so, we recast the filing of a case as “hearing zero” to make the one exceptional case filing date the same as the rest.

For another example, consider the following code, from exercise 2.16:

```r
netflix %>%
  replace_na(list(director = '', cast = '')) %>%
  mutate(
    director = str_split(director, ',', ),
    cast = str_split(cast, ',', )
  ) %>%
  unnest(director) %>%
  unnest(cast) %>%
  summarise(
    director =
    last(director,
  
```
order_by = str_length(director)
),
cast =
last(
  cast,
  order_by = str_length(cast)
)
)
) %>%
summarise(
  longest_name =
    ifelse(
      str_length(director) > str_length(cast),
      director,
      cast
    )
)

> # A tibble: 1 x 1
> longest_name
> <chr>
> 1 Gabrielle Susanne Solheim Leithaug

This code violates the “make similar things the same” principle. The objective is to find the longest name listed in either the director or cast columns of netflix. But for this query there’s no material difference between directors and cast members, so there’s no reason to distinguish between them. Yet the code does distinguish between them, applying each operation to the director and cast variables separately, which leads to a lot of redundancy.

A more efficient solution starts by lumping directors and cast members into a common variable called person, so that, e.g., a show with director = "Nia Dinata" and cast = "Surya Saputra, Cut Mini Theo, Tora Sudiro" would have person = "Nia Dinata, Surya Saputra, Cut Mini Theo, Tora Sudiro". By turning similar things (i.e., directors and cast members) into the same thing (i.e., people), this change enables us to apply each operation only once, which makes for a simpler solution:

```r
netflix %>%
  unite( # Create person variable from director and cast
    person,
    c(director, cast),
    sep = ",",
    na.rm = TRUE
  ) %>%
```
```r
mutate(person = str_split(person, ", ")) %>%
unnest(person) %>%
summarise(
  person = last(person, order_by=str_length(person))
)
```

> # A tibble: 1 x 1
> person
> <chr>
> 1 Gabrielle Susanne Solheim Leithaug

### 2.5 Solutions

#### 2.1

```r
str_detect(im_burning, ":") # ignoring negate input implicitly makes it FALSE
```

#### 2.2

```r
str_detect(im_burning, ":", negate = TRUE) # explicitly set negate to TRUE
```

#### 2.3

```r
vector_of_dates <- today() + days(1:365)
sample(vector_of_dates, 10)
```

#### 2.4

```r
vector_of_dates <- today() + days(1:365)

vector_of_weights <-
  wday(vector_of_dates, label = TRUE) %in% c("Sat", "Sun")

sample(
  vector_of_dates,
  size = 10,
  replace = FALSE,
  prob = vector_of_weights
)
```
2.5

```r
this_years_halloween <- mdy(str_c("10/31/", year(today())))
next_years_halloween <- mdy(str_c("10/31/", year(today()) + 1))

if(this_years_halloween >= today()) {
  this_years_halloween - today()
} else {
  next_years_halloween - today()
}
```

2.6

```r
ifelse(
  is.na(director_vector),
  "*missing data*",
  director_vector
)
```

2.7

```r
ifelse(
  letters %in% c("a", "e", "i", "o", "u"),
  LETTERS,
  letters
)
```

2.8

```r
ifelse(
  str_detect(letters, "[aeiou]"),
  LETTERS,
  letters
)
```

2.9
ifelse(
    str_length(month.name) > 6,
    month.abb,
    month.name
)

2.10

2 %>%
sqrt %>%
sqrt %>%
sqrt %>%
sqrt

2.11

1:200 %>%
as.roman %>%
str_to_lower %>%
str_sort %>%
str_flatten

2.14

"comrade Napoleon is always right" %>%
str_to_title %>%
str_c(., ",!" ) %>%
str_replace("Comrade", "Master") %>%
rep(2) %>%
str_flatten(collapse = " ")

2.15

tibble(date = today() + days(-10^4:-1)) %>%
mutate(date_char = str_remove_all(date, "-")) %>%
group_by(date, date_char) %>%
summarise(digits = as.character(0:9)) %>%
mutate(digit_count = str_count(date_char, digits)) %>%
```r
group_by(digits) %>%
summarise(m = mean(digit_count)) %>%
pivot_wider(names_from = digits, values_from = m)

2.16

d <-
  replace_na(
    netflix,
    list(director = "", cast = "")
  )
d <-
  mutate(
    d,
    director = str_split(director, \"\,"),
    cast = str_split(cast, \"\,")
  )
d <- unnest(d, director)
d <- unnest(d, cast)
d <-
  summarise(
    d,
    director = 
      last(
        director,
        order_by = str_length(director)
      ),
    cast = 
      last(
        cast,
        order_by = str_length(cast)
      )
  )
d <-
  summarise(
    d,
    longest_name = 
      ifelse(
        str_length(director) > str_length(cast),
        director,
        cast
      )
  )
```
"output/combined_experimental_sample.rds" %>%
  read_rds %>%
  filter(!is.na(UNITCOST)) %>%
  group_by(product) %>%
  summarise(
    cv = sd(UNITCOST, na.rm = TRUE)/mean(UNITCOST, na.rm = TRUE),
    .groups = "drop"
  ) %>%
  na.omit %>%
  summarise(
    probs = c(.25, .5, .75),
    quantile =
      cv %>%
      quantile(probs = probs) %>%
      format_fraction_same_length(3)
  ) %>%
  pivot_wider(names_from = probs, values_from = quantile) %>%
  iwalk(
    ~.x %>%
    as_tibble %>%
    write_delim(
      str_c(stat_folder, "wholesale_cv_quartile_", .y, ".txt"),
      col_names = FALSE
    )
  )

2.19

soft_max <-
  . %>%
  exp %>%
  sum %>%
  log

-4:4 %>% soft_max
-64:64 %>% soft_max
2.20

clean_text <-
  . %>%
  str_replace_na(replacement = "") %>%
  str_flatten(collapse = " ") %>%
  str_remove_all("[:punct:]") %>%
  str_squish %>%
  str_to_lower

tasty_breakfast %>% clean_text

brotherly_love %>% clean_text

2.22

y %>%{
  . / (2:sqrt(.))
} %>% {
  . != floor(.)
} %>%
  all

2.23

"When a man is tired of London, he is tired of life" %>%
  str_replace_all(word(., 5), "understanding")

2.24

euclid_length <-
  . %>%
  {.^2} %>%
  sum %>%
  sqrt

2.25
logical_not <-
  . %>% {!}

2.25

self_exponentiation <-
  . %>%
  {. ^ .}

2.26

reverse_symmetric <-
  . %>%
  {. == rev(.)} %>%
  all

2.27

smart_viewer <-
  . %>%
  { if(ncol(.)<5) summary(.) else glimpse(.) }

netflix %>% smart_viewer

netflix %>%
  select(1:3) %>%
  smart_viewer

2.28

netflix <-
  netflix %>%
  mutate(restricted = rating %in% c("R", "TV-MA", "NC-17"))
```r
netflix <- netflix %>%
  mutate(
    kid_rating = 
      ifelse(
        rating %in% c("G", "PG", "TV-G", "TV-PG"),
        rating,
        "not child friendly"
      )
  )

2.30

netflix <- netflix %>%
  mutate( 
    date_added = mdy(date_added),
    month_added = month(date_added, label = TRUE),
    wday_added = wday(date_added, label = TRUE)
  )

2.31

netflix %>%
  mutate( 
    duration_metric = word(duration, 2),
    duration = 
      duration %>%
        word(1) %>%
        as.integer
  ) %>%
  select(title, duration_metric, duration) %>%
  head(4)

2.32

evanston <- evanston %>%
  mutate(
```
`Arrest Date` =
  mdy_hm(`Arrest Date`) %>%
  as_date,

`Arrest Time` =
  `Arrest Time` %>%
  as.character %>%
  str_pad(4, "left", pad = 0) %>%
  str_replace("(\.)", "\\1:") %>%
  hm

2.33

evanston <-
  evanston %>%
  mutate(
    City =
    City %>%
    fct_recode(
      EVANSTON = "EVNSTN",
      EVANSTON = "EANSTON",
      EVANSTON = "EVANTSTON",
      EVANSTON = "N EVANSTO",
      EVANSTON = "EV/ANSTON",
      EVANSTON = "EVSTON"
    )
  )

2.34

evanston %>%
  summarise(
    youngest = min(Age, na.rm=TRUE),
    oldest = max(Age, na.rm=TRUE)
  )

2.35
evanston %>%
  summarise(cor(Sex == "Male", `Weapon Code` != ""))

2.36

evanston %>%
  summarise(
    age_gun = weighted.mean(Age, w = `Weapon Code` == "Handgun", na.rm = TRUE),
    age_no_gun = weighted.mean(Age, w = `Weapon Code` == "", na.rm = TRUE),
    age_difference = age_no_gun - age_gun
  )

2.37

netflix %>%
  summarise(
    day_count = max(date_added) - min(date_added),
    day_count = as.integer(day_count),
    show_day_count = n_distinct(date_added),
    new_show_fraction = show_day_count / day_count
  )

2.38

netflix %>%
  summarise(
    avg_year_movie = weighted.mean(release_year, type == "Movie"),
    avg_year_tv = weighted.mean(release_year, type == "TV Show"),
    age_difference = avg_year_tv - avg_year_movie
  )

2.39

netflix %>%
  summarise(
    fraction_na =
      director %>%
        is.na %>%
        mean
  )
2.40

```
netflix %>%
summarise(
  director %>%
    str_remove("", ".") %>%
    unique %>%
    na.omit
)
```

2.41

```
netflix %>%
summarise(
  quantiles =
    cast %>%
      str_count("", ") %>%
    {. + 1} %>%
    quantile(
      probs = c(0.25, .5, .75),
      na.rm = TRUE
    )
)
```

2.42

```
netflix %>%
  group_by(month_added) %>%
  summarise(num_shows = n())
```

2.43

```
netflix %>
  group_by(month_added) %>%
  summarise(num_shows = n()) %>%
  ungroup %>%
  mutate(frac = num_shows / sum(num_shows)) %>%
  head(4)
```
```r
netflix %>%
  group_by(Fri_Sat = wday_added %in% c("Fri", "Sat")) %>%
  summarise(fraction_tv = mean(type == "TV Show"))
```

```r
evanston %>%
  group_by(ntile(Age, n = 10)) %>%
  summarise(weapon_frac = mean('Weapon Code' != "")) %>%
  plot
```

```r
evanston %>%
  group_by(hour('Arrest Time')) %>%
  summarise(
    on_view_frac = mean('Arrest Type' == "On View", na.rm = TRUE)
  ) %>%
  plot
```

```r
netflix <-
  netflix %>%
  group_by(director) %>%
  mutate(director_count = n()) %>%
  ungroup
```

```r
netflix <-
  netflix %>%
  mutate(
    director_count =
      ifelse(is.na(director),
```
NA,  
director_count  
)  
)  

2.49  

netflix %>%  
group_by(  
  type,  
  Fri_Sat = wday_added %in% c("Fri", "Sat")  
) %>%  
summarise(  
inexperienced_director =  
  mean(director_count == 1, na.rm = TRUE)  
)  

2.50  

evanston %>%  
group_by(`Arrest Date`) %>%  
mutate(Daily_Arrest_Number = row_number(`Arrest Time`)) %>%  
ungroup  

2.51  

netflix %>%  
filter(str_detect(director, "Riri Riza")) %>%  
select(title, description)  

2.52  

netflix %>%  
filter(  
  str_detect(director, "Riri Riza"),  
  director != "Riri Riza"  
) %>%  
select(title, director)
2.53

```r
netflix %>%
  filter(str_detect(cast, ",", negate = TRUE)) %>%
  select(title, cast) %>%
  head
```

2.54

```r
netflix %>%
  group_by(director) %>%
  filter(
    cast %>%
      str_detect("Forest Whitaker") %>%
      any
  ) %>%
  head(5)
```

2.55

```r
netflix %>%
  group_by(director) %>%
  filter(
    !is.na(director),
    n_distinct(listed_in) >= 6
  ) %>%
  select(director, title, listed_in) %>%
  head(5)
```

2.56

```r
netflix %>%
  group_by(cast) %>%
  filter(
    release_year != min(release_year),
    !is.na(cast)
  ) %>%
  select(title, cast, release_year) %>%
  head(5)
```
evanston %>%
  group_by('Arrest Date') %>%
  filter(sum('Weapon Code' == "Handgun") >= 2) %>%
  ungroup
Chapter 3

Pivots and Plots

3.1 Introduction

We’ll now learn how to make plots and pivot data. These topics are related because making just the right plot usually requires some data pivoting. Pivoting a tibble means moving its data around. There are two primary places to store information in a table: in a column (i.e., “inside” the tibble) and in a column name (i.e., “above” the tibble). And thus there are two basic pivot functions: `pivot_longer()`, which converts metadata stored above the tibble into formal data stored inside the tibble, and `pivot_wider()`, which does the opposite. For example, compare the following three tibbles:

```r
planet_1 <-
  tribble(
    ~planet, ~radius, ~dist, ~year,
    "Mercury", 1516, 193, 88,
    "Venus", 3760, 361, 225,
    "Earth", 3958, 499, 365,
    "Mars", 2106, 760, 687
  )

planet_2 <-
  tribble(
    ~planet, ~stat, ~value,
    "Mercury", "radius", 1516,
    "Mercury", "dist", 193,
    "Mercury", "year", 88,
    "Venus", "radius", 3760,
    "Venus", "dist", 361,
    "Venus", "year", 225,
    "Earth", "radius", 3958,
  )
```
These tibbles store the same data in different ways. None of these configurations is superior, as different analyses call for different data arrangements. Accordingly, we would like to be able to convert one of these tibbles into another without redefining the sample from scratch. Data pivoting will enable us to do so.

In addition to pivot functions `pivot_longer()` and `pivot_wider()` we will study plotting function `ggplot()`. The plotting capability of R is stunning—it is one of the languages’ core competencies. In fact, `ggplot()` is reason enough to use R. Learning `ggplot()` is like joining an exclusive visualization fraternity—you will get excited when you see another `ggplot()` “in the wild”, like an expat who recognizes fellow countrymen. But let me warn you: making plots in `ggplot()` will turn you into a graph snob. Once you’ve grown accustomed to its beautiful graphs, you’ll never again look upon an Excel figure without contempt. Once you’ve learned how to make graphs with computer code, you’ll view point-and-click plotting approaches as indecent, if not outright uncivilized.

After you’ve honed your plotting and pivoting skills, you’ll apply them to a real-world problem in section 3.3. In this lab, you will follow the methodology of Bray (2020) to study how the track-package activities that Alibaba reports to its customers affects how these customers view the company’s logistics performance.

Finally, I conclude, in section 3.4, with some ad hoc remarks: In section 3.4.1 I explain how to save a ggplot (long story short: use `ggsave()`). In section 3.4.2 I explain how you’ll probably use `pivot_longer()` more than `pivot_wider()`, as it’s generally easier to manipulate information stored in columns than information stored in column names. In section 3.4.3 I explain how to collapse vectors into summary statistics with `pivot_wider()`. And in section 3.4.5 I explain that the graphs are clearer if we plot the distribution of the independent variable conditional on the dependent variable—i.e., if we illustrate the dependence in reverse.

For the lecture, you can work in the code.R file of the ggplot project you created in exercise 1.29, and for the lab you can work in the code.R file of the alibaba project you created in
We will primarily study $\text{ufc}$, a sample of 4111 Ultimate Fighting Championship\(^1\) (UFC) bouts that you loaded in exercise 1.29 of section 1.3. Each bout has two combatants, and each combatant is assigned to a corner of the octagonal ring. The favorite is usually assigned to the red corner and the underdog to the blue corner. The variables corresponding to the red-corner fighter begin with $R_-$ and the variables corresponding to the blue-corner fighter begin with $B_-$. I will use $X_-$ to refer to the generic case: e.g., $X_{\text{fighter}}$ means $R_{\text{fighter}}$ or $B_{\text{fighter}}$. Here are the variables:

- $\text{fight_id}$: Distinct bout identification number.
- $\text{date}, \text{location}$: Date and location of the bout.
- $\text{red_win}$: A logical indicated whether the the boxer in the red corner wins.
- $\text{title}$: A logical indicating whether this was a championship title bout.
- $\text{weight_class}$: The weight class of the fighters.
- $\text{gender}$: Fighter gender, either "MALE" or "FEMALE".
- $\text{rounds}$: Maximum number rounds until decision, either three, four, or five.
- $\text{result}$: Result of the bout.
  - "U-DEC" denotes a unanimous decision.
  - "M-DEC" denotes a majority decision, where two judges consider one fighter the winner and the third judge considers the fight a tie.
  - "S-DEC" denotes a split decision, where two judges considers one fighter the winner and the third judge considers the other fighter the winner.
  - "SUB" denotes a submission, where the losing fighter “taps out” of the bout.
  - "KO/TKO" denotes a knockout or technical knockout.
  - "DQ" denotes that a fighter was disqualified (e.g., for biting or pulling hair).
- $\text{finish_details}$: The move that finished the fight, if it ended in a knockout or submission.
- $\text{finish_round}$: The round the fight ended, a number between one and five.
- $\text{finish_round_time}$: Duration of the final round, in seconds.
- $\text{fight_time}$: Duration of the fight, in seconds.
- $\text{audience}$: A logical indicating whether there was an audience in the arena.
- $X_{\text{fighter}}$: Fighter name.
- $X_{\text{odds}}$: The decimal odds that that this fighter will win, from bestfightodds.com. For example, if $R_{\text{odds}} = 1.5$ and you place a $100$ bet that the boxer in the red corner will win then you will receive back $150$ if this boxer does win (reaping a $50$ profit) and you will receive nothing back if this boxer loses (incurring a $100$ loss).

\(^1\)https://en.wikipedia.org/wiki/Ultimate_Fighting_Championship
• X_stance: Fighting stance.
– "Orthodox" denotes a left-shoulder-forward stance.
– "Southpaw" denotes a right-shoulder-forward stance.
– "Switch" denotes a combination of left-shoulder-forward and right-shoulderforward stances.
• X_age, X_weight, X_height, X_reach: Fighter age (in years), weight (in pounds), and
height and arm span (in centimeters).
• X_wins, X_losses, X_draw, X_rounds_fought: Total number of UFC wins, losses, draws,
and rounds fought prior to the current fight.
• X_kd: Number of times this fighter knocked down the other fighter in this bout.
• X_strong_attempt, X_strong_achieve, X_weak_attempt, X_weak_achieve: Number of
strong strikes attempted, strong strikes landed, weak strikes attempted, and weak
strikes landed by this fighter in this bout.
• X_td_attempt, X_td_achieve: Number of take-downs attempted and achieve by this
fighter in this bout.
• X_sub_attempt, X_sub_achieve: Number of submissions attempted and achieve by this
fighter in this bout.
We will also study formula_1, which reports the lap times of Formula One2 car races, going
back to 1996. The tibble looks like this:

formula_1 %>% glimpse

> Rows: 517,573
> Columns: 10
> $ date

<date> 2008-03-16, 2008-03-16, 2008-03-16, 2008-03-16, 2008-03-16, 2~

> $ driver

<chr> "Lewis Hamilton", "Lewis Hamilton", "Lewis Hamilton", "Lewis H~

> $ team


> $ flag_d


> $ flag_t


> $ start

<dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~

> $ end

<dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~

> $ lap

<dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,~

> $ seconds <dbl> 121.465, 152.471, 89.257, 88.614, 88.556, 88.513, 88.442, 88.6~

Its variables are:
• date reports when the race took place. (There’s only one race on any given day.)
• circuit specifies the race, or “Grand Prix.”
2

https://en.wikipedia.org/wiki/Formula_One

162


• **driver** reports the driver’s name.
• **team** reports the name of the Formula 1 team, or “constructor.”
• **flag_d** and **flag_t** report the nationality of the driver and team.
• **start** reports the order on the starting grid. Since the track isn’t wide enough to accommodate the cars side-by-side, they begin the race aligned one after another, with the car in “pole position” at the front of the pack.
• **end** reports the sequence in which the cars completed the race.
• **lap** specifies the lap number. A car will drive around 50–70 laps around the course in a typical race.
• **seconds** specifies the length of time it took the car to complete the given lap.

### 3.2.2 ggplot()

The key output of data science is the plot. And the key plotting function is *ggplot()*, from the ggplot2 package, a member of the tidyverse.\(^3\) Hence, most of our analyses will terminate in a `ggplot()` step.

Here is a basic `ggplot()` example:

```r
ufc %>%
ggplot +
aes(
  x = R_reach,
  y = B_reach,
  color = weight_class
) +
geom_point(size = 0.1) +
facet_grid(
  rows = vars(wday(date, lab = TRUE)),
  cols = vars(gender, str_detect(location, "USA"))
)
```

\(^3\)The “gg” in *ggplot()* stands for “grammar of graphics.”
The code above illustrates the four basic components of a ggplot. The first component is the data, which we pass in with `ufc %>%`. We will usually input our data with a conveyor belt because most of our `ggplot()` calls will be positioned at the end of longer functional assembly lines. That is, we'll usually want to process our tibble a bit before plotting it.

The second component is the set of “facets,” which we specify with the `facet_grid()` expression. A facet is a mini-plot that corresponds to a specific group of data. We define our groups with the `vars()` function, which can parse general expressions. For example, the `rows = vars(wday(date, lab = TRUE))` option calls for a new row of facets for each distinct value of `wday(date, lab = TRUE)` and the `cols = vars(gender, str_detect(location, "USA"))` option calls for a new column of facets for each distinct `(gender, str_detect(location, "USA"))` pair. For example, the top-left facet corresponds to male fights held on Sunday in a country other than the United States (i.e., `gender = "MALE"`, `wday(date, lab = TRUE) = "Sun"`, and `str_detect(location, "USA") = FALSE`), and the bottom-right facet to female fights held on Saturday in the United States (i.e., `gender = "FEMALE"`, `wday(date, lab = TRUE) = "Sat"`, and `str_detect(location, "USA") = TRUE`). If we do not specify a faceting scheme, R squeezes all the data into a single facet (i.e., a single plot).

The third component is the cast of characters, which we specify with `aes(x = R_reach, y = B_reach, color = weight_class)`. The cast of characters specifies the variables that comprise a facet’s plot and the roles they play. For example, our cast of characters stipulates that `R_reach` will play the role of the x-coordinate, `B_reach` will play the role of the y-coordinate, and `weight_class` will play the role of color. We always define our cast of characters with `aes()`, which stands for “aesthetic.”

The final component is the “layer,” which we specify with `geom_point()`. There are many varieties of graph we can make with our cast of characters—line plots, bar charts, histograms,
boxplots, scatter plots, etc. The `geom_point()` layer specifies that we want a scatter plot (and the `size = 0.1` option tells R to make the dots one-tenth their normal size). In other words, if the `aes()` call serves as the cast of characters, specifying who acts in each role, then layer serves as the script, specifying what each role actually does.

So putting this together, the code above tells R to scatter plot the `R_reach` and `B_reach` variables of `ufc`, while coloring each point according to the `weight_class` variable, and to create separates mini-plots for each gender, day of the week, and USA/non-USA groupings.

Here’s a second example:

```r
stance_plot <-
ufc %>%
  mutate(
    spread = B_odds - R_odds,
    R_stance = str_c("Red: ", R_stance),
    B_stance = str_c("Blue: ", B_stance)
  ) %>%
  ggplot +
  aes(
    x = spread,
    color = R_stance,
    fill = R_stance
  ) +
  geom_histogram(
    bins = 40,
    alpha = .4
  ) +
  facet_wrap(vars(B_stance))

stance_plot
```

![Graph](image_url)
Let’s walk through the four elements of this plot:

- **Data**: The tibble inputted to `ggplot()` is the tibble outputted by `mutate()`. Hence, the sample that `ggplot()` receives has a variable called `spread` that equals the difference between `B_odds` and `R_odds`. The more positive this `spread` is, the more favored the red-cornered boxer is to win.

- **Facets**: We facet by `B_stance` by writing `facet_wrap(vars(B_stance))`. Note that we’ve switched from `facet_grid()` to `facet_wrap()`. The only difference between these functions is how R lays out the facets: under `facet_grid()` the computer arranges the facets as a grid and under `facet_wrap()` the computer arranges them as a list. We usually use `facet_wrap()` when faceting by one variable and `facet_grid()` when faceting by two variables.

- **Cast of characters**: The `aes()` call specifies that the x-coordinate role is played by `spread` and the color and fill roles are played by `R_stance`. There is no y-coordinate role.

- **Layers**: The `geom_histogram()` call tells R to make the plot a histogram. And this is why didn’t need to specify a variable for the y-coordinate: R knows to set the height of the bar to the number of observations in the given bin. The `bins = 40` option tells R that we want this histogram to comprise 40 vertical bars. And the `alpha = .4` option tells R to make the bars 60% transparent (so they have a pastel look to them).

The code above saves our ggplot as a variable called `stance_plot`. We can add extra options to this variable with the `+` symbol to further customize the figure:

```r
stance_plot +
  facet_wrap(
    vars(B_stance),
    ncol = 1,
    scales = "free"
  ) +
  labs(
    x = "Spread",
    y = "Number of Fights"
  ) +
  theme_bw() +
  theme(
    legend.position = "top",
    legend.title = element_blank(),
    strip.background = element_rect(color = "white", fill = "white")
  )
```
We’ve now modified our plot in four ways:

- The `ncol = 1` and `scales = "free"` options we added to `facet_wrap()` tell R to stack the facets in a single column and give each facet its own scaling, so that, e.g., one y-axis goes up to 400 whereas another maxes out at 25.
- The `labs()` call modifies the axis labels.
- The `theme_bw()` call changes the background from gray to white.
- The `theme()` call moves the legend from the side to the top, removes the legend title, and turns the facet labels from gray to white.

The preceding example illustrate the key feature of ggplots: their modularity. Rather than recreate our histogram from scratch, we simply appended the additional options to the `stance_plot` variable. In fact, a ggplot is really just a loose collection of features pasted together with `+` signs. Hence, a ggplot is like a Christmas tree that we can attach any number of ornaments to.

**Exercise 3.1.** Define the following layerless ggplot:

```r
layerless_height_plot <-
  ufc %>%
```
ggplot +
aes(x = R_height, y = B_height) +
labs(
  x = "Red Height",
  y = "Blue Height"
) +
theme(panel.background = element_rect(fill = "white"))

layerless_height_plot

We must give layerless_height_plot a layer for it to actually plot anything:

* Use the + symbol to add a geom_point(size = .01) layer to layerless_height_plot and call the result point_height_plot.
* Use the + symbol to add a geom_jitter(size = .01, height = 1, width = 1) layer to layerless_height_plot and call the result jitter_height_plot.

Note that jitter_height_plot is the same as point_height_plot, except its point positions are slightly randomized to minimize overlap. As you see, introducing a bit of noise can yield a much more informative graphic.

**Exercise 3.2.** Use the + symbol to derive from jitter_height_plot an analogous plot called jitter_reach_plot that illustrates the distribution of R_reach and B_reach. Label the axes "Red Reach" and "Blue Reach".

The ggplot paradigm is so flexible that it can produce almost any graph imaginable. You can find many beautiful examples here. And you can find an ugly example below:

---

4https://plotly.com/r/
ufc %>%
  mutate(
    R_weak_achieve = ntile(R_weak_achieve, 3),
    R_weak_achieve =
      ifelse(
        R_weak_achieve == 1,
        "Few Weak Red Punches",
        "Many Weak Red Punches"
      ),
    B_weak_achieve = ntile(B_weak_achieve, 2),
    B_weak_achieve =
      ifelse(
        B_weak_achieve == 1,
        "Few Weak Blue Punches",
        "Many Weak Blue Punches"
      )
  ) %>%
  group_by(
    R_weak_achieve,
    B_weak_achieve
  ) %>%
  mutate(
    mean_R_strong = mean(R_strong_achieve),
    mean_B_strong = mean(B_strong_achieve)
  ) %>%
  na.omit %>%
  ggplot +
  aes(
    x = R_strong_achieve,
    y = B_strong_achieve
  ) +
  geom_vline(
    aes(xintercept = mean_R_strong),
    color = "purple"
  ) +
  geom_hline(
    aes(yintercept = mean_R_strong),
    color = "purple"
  ) +
  geom_point(size = 0.1) +
  geom_smooth(aes(color = gender)) +
  geom_quantile(
    quantiles = c(0.1, 0.9),
    color = "pink"
This plot has an absurd number of features. For example, it has five distinct layers: the purple lines from geom_vline() and geom_hline(), the dots from geom_point(), the green and orange lines from geom_smooth(), and the pink lines from geom_quantile(). Also, there are four character definitions made by four aes() calls (but the latter three apply only within their respective geom_vline(), geom_hline(), and geom_smooth() layers).

The last example illustrates the typical ggplot syntax, which in general looks like this:

```r
initial_data %>%
  transformation_fun_1 %>%
```
transformation_fun_2 %>%
# ...
transformation_fun_n %>%
ggplot +
plot_fun_1() +
plot_fun_2() +
# ...
plot_fun_n()

The `transformation_fun_x()` expressions represent functions like `mutate()` and `filter()`, which modify the data, and the `plot_fun_x()` expressions represent functions like `geom_vline()` and `theme_bw()`, which modify the plot. Note that we switch from `%%` to `+` after the `ggplot()` line. Also, the function parentheses are optional before the `ggplot()` line but mandatory after. For example, in the code for the plot above we exclude the parentheses from `na.omit()` but not from `geom_smooth()`.

**Exercise 3.3.** We will now create a heat map that depicts the joint distribution of `R_weight` and `B_weight`.

- Pipe `ufc` into `ggplot()`.
- Use `aes()` to set `R_weight` to the x-axis and `B_weight` to the y-axis.
- Add a `geom_density2d_filled()` layer.
- Add the option `labs(x = "Red Weight", y = "Blue Weight")`.
- Save the plot as `heat_map_plot`, so we can modify it later.

**Exercise 3.4.** The following plot depicts the number of bouts that took place over time, by weight class. We will now recreate this plot.

- Use `mutate()` to redefine `weight_class = fct_reorder(weight_class, R_weight, mean)`. This changes `weight_class` from a character string to a factor whose levels are ordered according to mean(`R_weight`). For example, the "Women's Strawweight" level would be first, since it comprises the lightest fighters, and the "Heavyweight" level would be last, since it comprises the heaviest fighters.
- Pipe the result into `ggplot()`.
- Specify the cast of characters with `aes(x = date, color = weight_class, linetype = weight_class)`.
- Add a `geom_freqpoly(bins = 20)` layer.
- `facet_wrap()` by gender with `ncol = 1`.
- Add options `scale_color_grey()` , `theme_bw()`, and `theme(strip.background = element_rect(color = "white", fill = "white"))` to the `ggplot`.
Exercise 3.5. The following plot—which depicts the `fight_time` as a function of `abs(B_odds - R_odds)`—confirms that more mismatched fights end sooner. We will now recreate this plot.

- Use `filter()` to remove the `weight_class = "Catch Weight"` observations, since catch weight isn't actually a weight class.
- Pipe the result into `mutate()`. Within this `mutate()` call `str_replace()` the word "Women's" in `weight_class` with the initial "W." and `str_remove()` the word "weight". For example, these operations should translate "Women's Bantamweight" into "W. Bantam".
- Pipe the result into `ggplot()`.
- Specify the cast of characters with `aes(x = abs(B_odds - R_odds), y = fight_time)`.
- Add layers `geom_point(size = .2)` and `geom_smooth(method = "lm")`.
- `facet_wrap()` by `weight_class`. Set `scales = "free"` and `ncol = 4`. 

<table>
<thead>
<tr>
<th>date</th>
<th>count</th>
<th>weight_class</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>0</td>
<td>Women's Strawweight</td>
</tr>
<tr>
<td>2015</td>
<td>20</td>
<td>Women's Flyweight</td>
</tr>
<tr>
<td>2020</td>
<td>40</td>
<td>Flyweight</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>Women's Bantamweight</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Bantamweight</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Women's Featherweight</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>Featherweight</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>Catch Weight</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Lightweight</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Welterweight</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>Middleweight</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>Light Heavyweight</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Heavyweight</td>
</tr>
</tbody>
</table>
3.2.3 Case Study: Best Places to Gamble

We will now create the following plot, which depicts the distribution of the casino's commission by country.
Exercise 3.6. If you bet $1/R_{odds}$ dollars on the red fighter then you will receive $R_{odds} \times (1/R_{odds}) = 1$ dollar from the bookie if this fighter wins. Similarly, if you bet $1/B_{odds}$ dollars on the blue fighter then you will receive $1$ if this fighter wins. Hence, if you simultaneously bet $1/R_{odds}$ dollars on the red fighter and $1/B_{odds}$ dollars on the blue fighter then you’ll be guaranteed to receive $1$ from the bookie after the fight. Now, if the market were perfectly efficient then the cost of placing these bets, $1/R_{odds} + 1/B_{odds}$, would equal the reward from placing these bets, $1$. But the casino always takes a cut, so $1/R_{odds} + 1/B_{odds}$ will always exceed $1$ by some margin, which we will call the house take.

- Use `mutate()` to add a `house_take` column to `ufc`.
- Within the same `mutate()` call use `str_extract()` to define `country` as the text that follows the last comma in `location`, which is always the country of the bout. For example, `location = "Abu Dhabi, Abu Dhabi, United Arab Emirates"` should yield `country = "United Arab Emirates"`. And after use `fct_relevel(rev)` to arrange the levels in reverse alphabetical order. (The plot will look better with this ordering.)
  - The regular expression "\b[^,]+$" is R shorthand for “the text after the last comma.”

Exercise 3.7. `ggplot()` the result of the previous exercise.

- Use the `+` symbol to attach an `aes()` call that defines characters `x = house_take` and `y = country`.
- Use `+` to add a `geom_violin()` layer and look at the resulting plot.
- Confirm that the plot looks better with the `country = fct_relevel(country, rev)` step.
3.2.4 Case Study: Knockouts Occur Earlier than Submissions

Since it generally takes longer to ensnare someone in a choke hold than it does to bash them in the head, submissions usually occur later in the round than do knockouts. For example, the median submission happens 170 seconds into the final round whereas the median knockout happens 152 seconds into the final round. To emphasize this point, we will create the following plot, which illustrates the distribution of `finish_round_time` as a function of `finish_details` and `result`.

![Plot illustrating the distribution of finish_round_time as a function of finish_details and result.](image)

Exercise 3.8. We will begin with a few data tidying steps.

- Use `filter()` and `is.na()` to remove the observations in `ufc` that have `result = "DQ"`, `finish_details = NA`, or `finish_round_time = NA`.
- `group_by()` `finish_details`.
- Use `mutate()`, `ifelse()`, and `n()` to change the `finish_details` values that correspond to fewer than 18 bouts to `str_c(result, "Other", sep = " ")`. For example, the four `finish_details = "North-South Choke"` values should change to `finish_details = "SUB Other"`, the six `finish_details = "Spinning Back Kick"` values should change to `"KO/TKO Other"`, but the but the 46 `finish_details = "Triangle Choke"` values should not change.
- `ungroup()` the tibble.

Exercise 3.9. `mutate()` the result of the previous exercise to `fct_reorder()` `finish_details` so that its levels are arranged according to the `median()` `finish_round_time`.

175
Exercise 3.10. \texttt{ggplot()} the result of exercise 3.9.

- Let \texttt{finish\_round\_time} play the x-coordinate role, \texttt{finish\_details} play the y-coordinate role, and \texttt{result} playing the fill role.
- Add a \texttt{geom\_boxplot()} layer.
- Tidy the plot as you see fit and save it as \texttt{finish\_details\_plot} (so we can modify it in a later exercise).
- Confirm that the plot looks better with the \texttt{fct\_reorder(finish\_details)} step.

3.2.5 Updating \texttt{ggplot()} Data

We usually modify \texttt{ggplots} with the \texttt{+} symbol. For example, the following creates a black and white version of the plot we made in exercise 3.3:

\begin{verbatim}
heat_map_plot + scale_fill_grey()
\end{verbatim}

But to add new data rather than new options we use \texttt{ %+%} rather than \texttt{+}. For example, the following removes the \texttt{gender = "MALE"} observations from \texttt{heat_map_plot} and changes the graphs’ axes from measuring pounds to measuring body mass index (BMI):
female_bmi_sample <-
  ufc %>%
  filter(gender == "FEMALE") %>%
  mutate(
    R_weight = 4535.92 * R_weight / R_height^2, #Redefine weight vars as BMI
    B_weight = 4535.92 * B_weight / B_height^2
  )

female_bmi_heatmat_plot <-
  heat_map_plot %+% female_bmi_sample

female_bmi_heatmat_plot

Note, we must always position the new data to the right of the `%>%` sign, so that the code looks like `plot %+% data` rather than `data %+% plot`.

We can extract a ggplot’s data by piping it into `pluck("data")`. This technique enables us to modify a ggplot’s data without reconstructing it from the raw sample. For example, we can apply an additional `filter()` to the sample that underlies `female_bmi_heatmat_plot` without starting over from `ufc`:

```
```
Exercise 3.11. We will now facet finish_details_plot, from exercise 3.10, by whether or not there was an upset.

- `pluck()` the "data" from finish_details_plot.
- Pipe the output of `pluck()` into `mutate()` to define `winner` as a variable that equals "Underdog Won" or "Favored Fighter Won", depending on whether or not there was an upset.
  - An upset happens either when the red fighter wins and $R_{odds} > B_{odds}$ or when the blue fighter wins and $R_{odds} \leq B_{odds}$.
  - The `red_win` variable indicates who won.
- Use `<-` to save the output of `mutate()` as `new_plot_data`.
- Add `new_plot_data` to `finish_details_plot` with `%+%`.
- Use `+` to `facet_wrap()` by `winner`. 

```r
new_plot_data <-
female_bmi_heatmat_plot %>%
  pluck("data") %>%
  filter(location == "Las Vegas, Nevada, USA")

female_bmi_heatmat_plot %+
new_plot_data
```
Exercise 3.12. Use the %>% symbol and a bracketed expression to recreate stance_plot, from section 3.2.2, with the tibble generated by the following assembly line:

```r
ufc %>%
  group_by(location, weight_class, B_stance, R_stance) %>%
  summarise(spread = min(B_rounds_fought - R_rounds_fought))
```

Exercise 3.13. Define `make_stance_plot()` as a function that receives a tibble, applies `mutate(spread = B_age - R_age)` to it, and then recreates `stance_plot` with the result. For example, `make_stance_plot(x)` should be equivalent to `stance_plot %>% mutate(x, spread = B_age - R_age)`. Apply `make_stance_plot` to the outputs of `ufc %>% filter(R_stance != B_stance)` and `ufc %>% filter(R_age < 28)`.

Exercise 3.14. We will now add dots to `heat_map_plot` that depict the `median()` weight of each weight class.

- `pluck()` the "data" from `finish_details_plot`.
- `group_by()` `weight_class`.
- `mutate()` the result to add columns `R_aggregate_weight = median(R_weight)` and `B_aggregate_weight = median(B_weight)`.
- Use `<-` to save the result as `new_plot_data`.
- Use `%>%` to add `new_plot_data` to `finish_details_plot`.
- Use `+` to add a `geom_point()` layer. Within this function call embed `aes(x = R_aggregate_weight, y = B_aggregate_weight)` to place the points at the median weights.

3.2.6 Case Study: Formula 1 Lap Times

Formula 1 cars generally speed up over the course of the race. There are five reasons for this. First, the cars start from a resting position, and it takes a few seconds to get up to speed. Second, it can take some time for drivers to “get in the zone.” Third, cold tires are more slippery, and hence more dangerous to drive at high speeds. Fourth, since the cars start in one big pack, the race usually begins with a traffic jam, which gradually eases as the cars space out over the course of the race. And fifth, the cars get lighter, and hence faster, as they as they burn their fuel.

Exercise 3.15. Derive a plot from `formula_1` that most convincingly demonstrates that the race cars get faster and faster as the race progresses.

- Note, if you don’t account for the number of laps in a race you could get spurious results, because races with shorter courses tend to have more laps. For example, only the very shortest courses have more than 75 laps, so lap 80 is driven on a shorter course, on average, than lap 70.

Exercise 3.16. Use %>% to create a second version of your plot, with a modified set of data. This plot should should make the same point in a different way.
3.2.7 `pivot_longer()` and `pivot_wider()`

Suppose we wanted to combine `jitter_height_plot` and `jitter_reach_plot`, from exercise 3.2, into a single ggplot, with one facet scatter plotting `R_height` and `B_height` and another facet scatter plotting `R_reach` and `B_reach`, like this:

![Plot](image)

Since different facets always plot the same tibble columns, it seems impossible to plot heights in one facet and reaches in another: setting `aes(x = R_height, y = B_height)` breaks the right-hand facet and setting `aes(x = R_reach, y = B_reach)` breaks the left-hand facet. However, we can create this ggplot() if we rearrange the data in our tibble. How would the tibble have to be structured to create the ggplot() above? This is an important puzzle—think it over before proceeding.

OK, here’s the solution: pool the `R_height` and `R_reach` data in one column called `R`, pool the `B_height` and `B_reach` data in another column called `B`, and introduce another column called `stat` that specifies whether a given row reports a height or a reach. With this, we can use `R` for the x-axis, `B` for the y-axis, and `stat` for the facet. More specifically, here’s the reshaped tibble:

```r
data_to_plot <-
  ufc %>%
  select(fight_id, R_height, B_height, R_reach, B_reach) %>%
  pivot_longer(
    cols = -fight_id,
    names_to = c("corner_color", "stat"),
    names_sep = ".",
    values_to = "val"
  ) %>%
  pivot_wider(
    names_from = corner_color,
    values_from = val
  )
```

180
mutate(stat = str_to_title(stat))

data_to_plot

> # A tibble: 8,222 x 4
>   fight_id stat    R   B
>   <int> <chr>   <dbl> <dbl>
>  1     1  Height 193. 193.
>  2     1  Reach 193. 198.
>  3     2  Height 180. 190.
>  4     2  Reach 188. 203.
>  5     3  Height 170. 165.
>  6     3  Reach 183. 168.
>  7     4  Height 173. 183.
>  8     4  Reach 180. 185.
>  9     5  Height 178. 180.
> 10     5  Reach 190. 190.
> # ... with 8,212 more rows

And here's the corresponding plot:

data_to_plot %>%
ggplot +
aes(x = R, y = B) +
geom_jitter(  
  size = .01,
  height = 1,
  width = 1
) +
facet_wrap(  
  vars(stat),
  ncol = 2,
  scales = "free"
) +
labs(x = "Red Fighter", y = "Blue Fighter") +
theme(  
  strip.background = element_rect(color = "white", fill = "white"),
  panel.background = element_rect(fill = "white"),
  strip.text.x = element_text(size = 12)
)
Reconfiguring the tibble in this fashion is called **pivoting**. There are two basic pivot functions: `pivot_wider()` and `pivot_longer()`. The former adds columns and the latter removes them. Data pivots neither create nor destroy data, so a data pivot that creates columns must destroy rows, and vice versa. To create a column, we need two elements: the new column’s name and the new column’s values. Hence, `pivot_wider()` has two key arguments:

- where to get the new columns’ names_from, and
- where to get the new columns’ values_from.

In contrast, when we kill a column, we’re left with two orphans: the killed column’s names and the killed column’s values. Hence, `pivot_longer()` has three key arguments:

- which cols to remove,
- where to move the removed columns’ names_to, and
- where to move the removed columns’ values_to.

For example, define

```r
long_tib <-
ufc %>%
group_by(R_fighter) %>%
summarise(
  metric =
    c(
      "strong_att", "strong_ach",
      "weak_att", "weak_ach"
    ),

5Well, technically, `pivot_wider()` can destroy data, with the `values_fn` option, but we won’t consider these summarizing pivots until section 3.4.3.
measurement =
c(
  mean(R_strong_attempt, na.rm = TRUE),
  mean(R_strong_achieve, na.rm = TRUE),
  mean(R_weak_attempt, na.rm = TRUE),
  mean(R_weak_achieve, na.rm = TRUE)
)

long_tib

```
> # A tibble: 4,004 x 3
> # Groups: R_fighter [1,001]
> R_fighter metric measurement
> <chr> <chr> <dbl>
> 1 Aalon Cruz strong_att 12
> 2 Aalon Cruz strong_ach 2
> 3 Aalon Cruz weak_att 0
> 4 Aalon Cruz weak_ach 0
> 5 Aaron Phillips strong_att 54
> 6 Aaron Phillips strong_ach 38
> 7 Aaron Phillips weak_att 225
> 8 Aaron Phillips weak_ach 192
> 9 Aaron Riley strong_att 138
>10 Aaron Riley strong_ach 36
> # ... with 3,994 more rows
```

We can widen this tibble from 4004 x 3 to 1001 x 5 with:

```
wide_tib <-
long_tib %>%
pivot_wider(
  names_from = metric,
  values_from = measurement
)

wide_tib

```
```
> # A tibble: 1,001 x 5
> # Groups: R_fighter [1,001]
> R_fighter strong_att strong_ach weak_att weak_ach
> <chr> <dbl> <dbl> <dbl> <dbl>
>1 Aaron Phillips 54 38 225 192
>2 Aaron Phillips 138 36 192 138
>3 Aaron Phillips 225 38 138 225
>4 Aaron Phillips 192 36 138 192
>5 Aaron Phillips 138 36 138 138
> # ... with 995 more rows
```

183
This function’s first option tells R to get the new column names from the data in the metric column and the second option tells R to get the new column values from the data in the measurement column. Hence, this operation divides the old measurement column into the new strong_att, strong_ach, weak_att, and weak_ach columns.

We can undo the pivot_wider() step with pivot_longer() to recover long_tib from wide_tib:

```r
wide_tib %>%
pivot_longer(
  cols = c(strong_att, strong_ach, weak_att, weak_ach),
  names_to = "metric",
  values_to = "measurement"
)
```

> # A tibble: 4,004 x 3
> # Groups:   R_fighter [1,001]
> R_fighter metric   measurement
> <chr>     <chr>     <dbl>
> 1 Aalon Cruz strong_att   12
> 2 Aalon Cruz strong_ach   2
> 3 Aalon Cruz weak_att    0
> 4 Aalon Cruz weak_ach   0
> 5 Aaron Phillips strong_att 54
> 6 Aaron Phillips strong_ach 38
> 7 Aaron Phillips weak_att 225
> 8 Aaron Phillips weak_ach 192
> 9 Aaron Riley  strong_att 138
>10 Aaron Riley  strong_ach 36
> # ... with 3,994 more rows
This function’s first option tells R that the cols to eliminate are strong_att, strong_ach, weak_att, and weak_ach; its second option tells R to move these columns’ names_to the new metric column; and its third option tells R to move these columns’ values_to the new measurement column. Hence, this operation combines the old strong_att, strong_ach, weak_att, and weak_ach columns into the new measurement column and stores the old column names in the new metric column.

Note that the variables passed in as names_to and values_to have quotation marks around them, but the variables passed in as cols do not. Unfortunately, pivot_longer() and pivot_wider() can be a bit persnickety about whether its input arguments are quoted or not, so if you get an error try adding or removing some quotation marks.

Exercise 3.17. Use pivot_longer() to convert the planet_1 tibble, defined in section 3.1, into planet_2.

Exercise 3.18. Use pivot_wider() to convert the planet_2 into planet_3.

Exercise 3.19. Use pivot_wider() to convert planet_2 into a new tibble called planet_4, which has one row and 12 columns. The first column should be Mercury_radius and the last column should be Mars_year.

Here’s a more slick pivot_longer() call:

```r
long_tib_2 <-
  wide_tib %>%
  pivot_longer(
    cols = -R_fighter,
    names_to = c("strength", "outcome"),
    names_sep = "_",
    values_to = "measurement"
  )

long_tib_2
```

> # A tibble: 4,004 x 4
> # Groups: R_fighter [1,001]
> R_fighter strength outcome measurement
> <chr> <chr> <chr> <dbl>
> 1 Aalon Cruz strong att 12
> 2 Aalon Cruz strong ach 2
> 3 Aalon Cruz weak att 0
> 4 Aalon Cruz weak ach 0
> 5 Aaron Phillips strong att 54
This function’s first option tells R to eliminate all cols *besides* `R_fighter` (the `-` symbol is shorthand for “everything but”); its second option tells R to move these columns’ names_to the new strength and outcome columns; its third option tells R to split the names at the underscore, allocating the text before the "_" to the strength column and the text after the "_" to the outcome column; and the fourth option tells R to move the columns’ values_to the new measurement column.

We can derive `wide_tib` from `long_tib_2`:

```r
long_tib_2 %>%
  pivot_wider(
    names_from = c(strength, outcome),
    values_from = measurement
  )
```

But we can also `pivot_wider()` `long_tib_2` into other shapes:

**Example 1:**
```r
medium_tib_1 <-
```

186
long_tib_2 %>%
pivot_wider(
  names_from = outcome,
  values_from = measurement
)

medium_tib_1 %>%
head(3)

> # A tibble: 3 x 4
> # Groups: R_fighter [2]
> R_fighter strength att ach
> <chr> <chr> <dbl> <dbl>
> 1 Aalon Cruz strong 12 2
> 2 Aalon Cruz weak 0 0
> 3 Aaron Phillips strong 54 38

#Example 2:
medium_tib_2 <-
long_tib_2 %>%
pivot_wider(
  names_from = strength,
  values_from = measurement
)

medium_tib_2 %>%
head(3)

> # A tibble: 3 x 4
> # Groups: R_fighter [2]
> R_fighter outcome strong weak
> <chr> <chr> <dbl> <dbl>
> 1 Aalon Cruz att 12 0
> 2 Aalon Cruz ach 2 0
> 3 Aaron Phillips att 54 225

The first example gives every (R_fighter, strength) pair its own row and every outcome value its own column and the second example gives every (R_fighter, outcome) pair its own row and every strength value its own column.

Exercise 3.20. Use pivot_longer() to derive the following tibble from planet_1:
Exercise 3.21. Use `pivot_longer()` to derive `planet_2` from `planet_4`, which you created in exercise 3.19.

- Use `cols = everything()` to specify that we want to destroy every column.
- Use `names_sep = "_"` to split the planet names from the statistic names.

Our different tibble configurations permit different plots. For example, only `long_tib_2` can yield this plot:

```r
long_tib_2 %>%
ggplot +
aes(x = measurement) +
geom_density() +
facet_grid(
  rows = vars(strength),
  cols = vars(outcome)
)
```
Only `wide_tib` can yield this plot:

```r
code
only wide_tib can yield this plot:

```wide_tib %>%
mutate(weak_ratio = weak_ach / weak_att) %>%
ggplot +
aes(
    x = strong_att,
    y = strong_ach,
    color = weak_ratio
) +
geom_point()
```

Only `medium_tib_1` can yield this plot:

```r
code
only medium_tib_1 can yield this plot:

```medium_tib_1 %>%
ggplot +
aes(
    x = att,
    y = ach
) +
geom_point() +
facet_wrap(vars(strength))
```
And only medium_tib_2 can yield this plot:

```r
medium_tib_2 %>%
ggplot +
aes(
x = strong,
y = weak
) +
geom_point() +
facet_wrap(vars(outcome))
```

**Exercise 3.22.** We will now create a histogram that illustrates the distribution of the number of times that a fighter appears in `ufc`. For example, Ben Nguyen appears in the sample seven times, as there are two bouts with `R_fighter = "Ben Nguyen"` and five bouts with `B_fighter = "Ben Nguyen"`.  

190
• Use `pivot_longer()` to merge the names in `R_fighter` and `B_fighter` into a single column.
• `count()` the number of times each name appears.
  – The `count()` function returns the same thing as a `group_by()` step followed by `summarise(n = n())`. See section 2.4.3 for more on `count()`.
• `ggplot()` the result with a `geom_histogram()` layer.

If all roads lead to Rome then you can travel between any two cities, by way of Rome. Pivoting has a similar hub-and-spoke logic: all wide tibbles can transform into the longest tibble, and the longest tibble can widen into all wide tibbles. Hence, all wide tibbles can transform into all other wide tibbles, by way of the longest tibble. Accordingly, we can reconfigure a tibble from any shape to any other shape with a `pivot_longer()` followed by a `pivot_wider()`. For example, the most straightforward way to derive `medium_tib_1` and `medium_tib_2` from `wide_tib` is to fully lengthen the tibble and then selectively widen it:

```r
#Get medium_tib_1 from wide_tib:
wide_tib %>%
pivot_longer(
  cols = -R_fighter,
  names_to = c("strength", "outcome"),
  names_sep = "_",
  values_to = "measurement"
) %>%
pivot_wider(
  names_from = outcome,
  values_from = measurement
)
```

> # A tibble: 2,002 x 4
> # Groups: R_fighter [1,001]
> R_fighter strength att ach
> <chr> <chr> <dbl> <dbl>
> 1 Aalon Cruz strong 12 2
> 2 Aalon Cruz weak 0 0
> 3 Aaron Phillips strong 54 38
> 4 Aaron Phillips weak 225 192
> 5 Aaron Riley strong 138 36
> 6 Aaron Riley weak 40 39
> 7 Aaron Rosa strong 99 59
> 8 Aaron Rosa weak 212 197
>  9 Aaron Simpson strong  90.7  48.7
> 10 Aaron Simpson weak  37.7  32
> # ... with 1,992 more rows

#Get medium_tib_2 from wide_tib:
wide_tib %>%
  pivot_longer(
    cols = -R_fighter,
    names_to = c("strength", "outcome"),
    names_sep = "_",
    values_to = "measurement"
  )
  pivot_wider(
    names_from = strength,
    values_from = measurement
  )

> # A tibble: 2,002 x 4
> # Groups: R_fighter [1,001]
> R_fighter outcome strong weak
> <chr> <chr> <dbl> <dbl>
> 1 Aalon Cruz att  12  0
> 2 Aalon Cruz ach  2  0
> 3 Aaron Phillips att  54 225
> 4 Aaron Phillips ach  38 192
> 5 Aaron Riley att 138  40
> 6 Aaron Riley ach  36  39
> 7 Aaron Rosa att  99 212
> 8 Aaron Rosa ach  59 197
> 9 Aaron Simpson att 90.7  37.7
>10 Aaron Simpson ach 48.7  32
> # ... with 1,992 more rows

For another example, the following transposes `wide_tib`, converting its columns to rows and rows to columns:

wide_tib %>%
  pivot_longer(
    cols = -R_fighter,
    names_to = "measure",
    values_to = "val"
Exercise 3.23. Translate `planet_1` into `planet_3`, by way of `planet_2`.

- Use `pivot_longer()` to translate `planet_1` into `planet_2`.
- Use `pivot_wider()` to translate `planet_2` into `planet_3`.

Exercise 3.24. The shortest fight in UFC history is a sight to behold⁶ (if you can stomach the violence). We will now compile the fighter statistics for this bout.

- Use `filter()` to identify the fight with the shortest `fight_time`.
  - Don’t forget the `na.rm = TRUE` option, if you use the `min()` function.
  - You should have `finish_details = "Flying Knee"`.
- Pipe the result into `select(starts_with(c("R_", "B_")) & where(is.numeric))` to drop all but the numeric columns that begin with either "R_" or "B_".
- Use `pivot_longer()` to convert the tibble into one with columns `corner_color`, `metric`, and `stat`, where the values of `corner_color` are "R" or "B" and the values of `metric` are "odds", "age", "weight", "height", "reach", "wins", "losses", "draw", "rounds", or "sub".
  - Use option `cols = everything()` to pull down all columns.
  - Use option `values_drop_na = TRUE` to remove `NA` terms.
  - Use option `names_pattern = "(.).+(.)"` to tell R to split the names at the first "_".⁷ You can’t use `names_sep = "_"` in this case because some names, such as `R_rounds_fought` and `B_sub_achieve`, have multiple underscores.

⁶https://www.youtube.com/watch?v=z8LdH5lgWvg&ab_channel=LuckyBtw
⁷"(.).+(.)" is a regular expression that defines two substrings within a given text string. The first substring corresponds to the first set of parentheses and the second substring to the second set of parentheses. In R, . stands for any single character and .* stands for any string of characters. Hence, the regular expression sets the first substring to the single character immediately preceding the first "_" and sets the second substring to everything after the first "_".
- Use `names_to = c("corner_color", "metric")` to send the first letter of the old column names to a column called `corner_color` and to send the latter part of the old column names to another column called `metric`.

- Use `pivot_wider()` to convert the resulting tibble into one with columns `metric`, `R`, and `B`.

**Exercise 3.25.** Recreate the following plot, which we made at the beginning of this section.

- Start with `ufc` and `select()` `fight_id`, `R_height`, `B_height`, `R_reach`, and `B_reach`.
- Use `pivot_longer()` with `names_sep = "_"` to transform the tibble into one with columns `fight_id`, `corner_color`, `stat`, and `val`, where `corner_color` is either "R" or "B" and `stat` is either "height" or "reach".
- Use `pivot_wider()` to transform the tibble into one with columns `fight_id`, `stat`, `R`, and `B`.
- `ggplot()` the result with a `geom_jitter()` layer and a `stats facet_wrap()`.

![Plot](image)

**Exercise 3.26.** Create the following `geom_jitter()` plot, which depicts `B_wins` and `B_losses` in the left facet and `R_wins` and `R_losses` in the right facet.

![Plot](image)
While most pivots neither create nor destroy data, it is sometimes convenient to use the `id_cols` option of `pivot_wider()` to delete unwanted columns. For example, compare the following two pivots:

```r
#without id_cols option:
formula_1 %>%
pivot_wider(
  names_from = lap,
  values_from = seconds
) %>%
head(3)
```

```r
# A tibble: 3 x 95
> # A tibble: 3 x 95
> date  circuit driver  team  flag_d  flag_t  start  end `1` `2` `3` `4` `5` `6` `7` `8` `9` `10`
> <date> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1 2008-03-16 Australia Lewis  McLaren U.K. U.K. 1 1 121. 152. 89.3 88.6 88.6 88.5 88.4 88.7 88.1 88.4
> 2 2008-03-16 Australia Nick BMW Germany Germany 2 2 126. 150. 92.0 90.3 90.1 89.6 91.2 89.4 89.4 90.2
> 3 2008-03-16 Australia Nico Will Germany U.K. 3 3 125. 151. 91.7 90.2 90.1 89.6 90.0 89.8 89.8 89.8
```

```r
#with id_cols option:
formula_1 %>%
pivot_wider(
  id_cols = c(date, driver),
  names_from = lap,
  values_from = seconds
) %>%
head(3)
```

```r
# A tibble: 3 x 89
> # A tibble: 3 x 89
> date  driver `1` `2` `3` `4` `5` `6` `7` `8` `9` `10`
> <date> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1 2008-03-16 Lewis 121. 152. 89.3 88.6 88.6 88.5 88.4 88.7 88.1 88.4
> 2 2008-03-16 Nick 126. 150. 92.0 90.3 90.1 89.6 91.2 89.4 89.4 90.2
> 3 2008-03-16 Nico 125. 151. 91.7 90.2 90.1 89.6 90.0 89.8 89.8 89.8
```
Both pivots tabulate the number of seconds that a given driver took on a given lap on a given date, but the former junks up the results with extra circuit, team, flag_d, flag_t, start, and end columns. The id_cols = c(date, driver) option tells R to disregard these superfluous columns. Note, the id_cols must uniquely identify the rows of the resulting tibble. For example, writing id_cols = c(circuit, driver) wouldn’t work, because there are 11 observations with driver == "Sebastian Vettel", circuit == "Singapore G.P.", and lap == 8, and we can’t cram 11 different seconds values into a single tibble cell. (Well, we technically can, with the “aggregating pivots” discussed in section 3.4.3.)

If you’re uncomfortable with the id_cols option, then you can always manually select() away the unwanted columns ahead of time:

```r
# ... with 77 more variables: `11` <dbl>, `12` <dbl>, `13` <dbl>, `14` <dbl>,
> # `15` <dbl>, `16` <dbl>, `17` <dbl>, `18` <dbl>, `19` <dbl>, `20` <dbl>,
> # `21` <dbl>, `22` <dbl>, `23` <dbl>, `24` <dbl>, `25` <dbl>, `26` <dbl>,
> # `27` <dbl>, `28` <dbl>, `29` <dbl>, `30` <dbl>, `31` <dbl>, `32` <dbl>,
> # `33` <dbl>, `34` <dbl>, `35` <dbl>, `36` <dbl>, `37` <dbl>, `38` <dbl>,
> # `39` <dbl>, `40` <dbl>, `41` <dbl>, `42` <dbl>, `43` <dbl>, `44` <dbl>,
> # `45` <dbl>, `46` <dbl>, `47` <dbl>, `48` <dbl>, `49` <dbl>, `50` <dbl>, ...

Exercise 3.27. We will now create the tibble below, which records the teams that drove the five fastest laps in the history of each Grand Prix. For example, the second row indicates

```r
# A tibble: 3 x 89
> # A tibble: 3 x 89
> date driver `1` `2` `3` `4` `5` `6` `7` `8` `9` `10`
> <date> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1 2008-03-16 Lewis ~ 121. 152. 89.3 88.6 88.6 88.5 88.4 88.1 88.4
> 2 2008-03-16 Nick H~ 126. 150. 92.0 90.3 90.1 89.6 91.2 89.4 89.0
> 3 2008-03-16 Nico R~ 125. 151. 91.7 90.2 90.1 89.6 90.0 89.8 89.8
> # ... with 77 more variables: `11` <dbl>, `12` <dbl>, `13` <dbl>, `14` <dbl>,
> # `15` <dbl>, `16` <dbl>, `17` <dbl>, `18` <dbl>, `19` <dbl>, `20` <dbl>,
> # `21` <dbl>, `22` <dbl>, `23` <dbl>, `24` <dbl>, `25` <dbl>, `26` <dbl>,
> # `27` <dbl>, `28` <dbl>, `29` <dbl>, `30` <dbl>, `31` <dbl>, `32` <dbl>,
> # `33` <dbl>, `34` <dbl>, `35` <dbl>, `36` <dbl>, `37` <dbl>, `38` <dbl>,
> # `39` <dbl>, `40` <dbl>, `41` <dbl>, `42` <dbl>, `43` <dbl>, `44` <dbl>,
> # `45` <dbl>, `46` <dbl>, `47` <dbl>, `48` <dbl>, `49` <dbl>, `50` <dbl>, ...

```
that Renault drove the fastest lap in Spanish Grand Prix history, and that Ferrari drove the second fastest lap.

- Use `group_by()` and `mutate()` to define `speed_rank = row_number(seconds)` for each circuit. For example, the fastest lap of each circuit should have `speed_rank = 1`.
- `filter()` the sample down to the observations with `speed_rank <= 5`, and `arrange()` the tibble by `speed_rank`. Arranging the tibble in this fashion will ensure that our final tibble columns are ordered from 1 to 5.
- `pivot_wider()` the result with `names_from = speed_rank`. Use the `id_cols` option or a `select()` operation to remove the unwanted columns. (Note that each row in the pre-pivoted tibble has a unique circuit and `speed_rank` pair.)

```r
> # A tibble: 41 x 6
> circuit `1` `2` `3` `4` `5`
> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 Japanese G.P. Ferrari Ferrari Ferrari Ferrari Ferrari
> 2 Spanish G.P. Renault Ferrari Renault McLaren McLaren
> 3 Turkish G.P. McLaren McLaren McLaren McLaren McLaren
> 4 Bahrain G.P. Ferrari Renault Ferrari Ferrari Williams
> 5 San Marino G.P. Ferrari Ferrari Ferrari Ferrari Ferrari
> 6 United States G.P. Ferrari Ferrari Ferrari Ferrari Ferrari
> 7 British G.P. Ferrari Ferrari Ferrari Ferrari Ferrari
> 8 German G.P. McLaren Ferrari Ferrari McLaren Ferrari
> 9 Belgian G.P. McLaren Ferrari Ferrari McLaren Ferrari
> 10 Italian G.P. Ferrari Ferrari Ferrari Ferrari McLaren
> # ... with 31 more rows
```

**Exercise 3.28.** We will now create the tibble below, which reports the `start` and `end` values of the 2022 races.

- `filter()` formula to down to the observations with `year(date) >= 2022`.
- `select()` away the `lap` and `seconds` columns and extract the `distinct()` rows of the resulting tibble. These operations remove the redundant values in `start` and `end`, which makes these columns easier to work with.
- `Pivot` the result to transform columns `start` and `end` into columns `measure` and `position`, the former of which has values "start" and "end" and the latter of which has values 1 to 20.
- `arrange()` the tibble by `position`. Arranging the rows in this fashion will ensure that the columns of our final tibble are properly ordered.
- `pivot_wider()` the result to pull the `position` values into the column names. Use the `id_cols` option or a `select()` operation to remove the unwanted columns. (Note that each row in the pre-pivoted tibble has a unique circuit, `measure`, and `position` combination.)

```r
> # A tibble: 6 x 22
```
3.2.8 Case Study: House Takes Have Increased

We will now create the following plot, which illustrates that the casino takes have generally increased from around 0.02 to around 0.037.

![Graph showing the increase in casino takes over time.]

**Exercise 3.29.** `mutate()` `ufc` as follows:

- Define `house_take = 1/R_odds + 1/B_odds - 1`.
- Define `date_bucket = ntile(date, 200)`. Taking values 1 through 200, this variable divides `ufc` into 200 equally sized subsamples based on the value of `date`. For example, the earliest half a percent of fights have `date_bucket = 1` and the latest half a percent of fights have `date_bucket = 200`.

198
• Use ntile() to create a variable that equals 1 when \(R\_weight + B\_weight\) is less than its median() value and equals 2 otherwise, and then use fct_recode() to turn this into a variable that equals "light_fights" when \(R\_weight + B\_weight\) is less than its median() value and equals "heavy_fights" otherwise. Call this variable weight_bucket.

**Exercise 3.30.** Start with the result of exercise 3.29 and group_by() date_bucket. Use mutate() to redefine date as the median() date value across the given date_bucket grouping. Take the resulting tibble and group_by() date and weight_bucket. Use summarise() to define house_take_mean = mean(house_take).

**Exercise 3.31.** pivot_wider() the result of exercise 3.30 from a tibble with columns date, weight_bucket, and house_take_mean to one with columns date, heavy, and light. ggplot() the result, with aes(x = light, y = heavy, color = date) and geom_path().

**Exercise 3.32.** While it's quite striking, the geom_path() plot you created in the last exercise isn't the clearest way to depict the increase in house takes. Create a new ggplot that more plainly depicts the change in house takes over time.

### 3.2.9 Case Study: Winners Punch More

We will now create the following plot, which shows that the winning fighter usually attempts and lands more punches and takedowns.

![Plot showing the distribution of strong and weak strikes, takedowns, and submissions between red and blue fighters.]

**Exercise 3.33.** We will begin by pivoting our tibble into the correct shape.

• Pipe ufc into select(fight_id, red_win, contains(c("attempt", "achieve"))) to limit the columns to fight_id, red_win, and those that contain the words "attempt" or "achieve" (e.g., B_weak_attempt or R_sub_achieve).
• `pivot_longer()` the tibble into one with columns `fight_id`, `red_win`, `corner_color`, `move`, `outcome`, and `value`, where the values of `corner_color` are "R" and "B", the values of `move` are "strong", "weak", "td", and "sub", and the values of `outcome` are "achieve" and "attempt".
  
  – Use `names_sep = "_"` and `values_drop_na = TRUE`.

• `pivot_wider()` the resulting tibble into one with columns `fight_id`, `red_win`, `move`, `outcome`, `R`, and `B`.
  
  – The `mean()` R value should be 19.88.

**Exercise 3.34.** `mutate()` the result of exercise 3.33 as follows:

• Convert `red_win` into a factor whose levels are "Red Win" and "Blue Win".
• Use `fct_relevel()` and `fct_recode()` to convert `move` into a factor whose levels are "Strong Strike", "Weak Strike", "Takedown", and "Submission", in that order.
• Apply `str_to_title()` to `outcome`. (We don't need to convert this variable to a factor because its alphabetical ordering is sufficient.)

**Exercise 3.35.** `ggplot()` the result of exercise 3.34.

• Set `x = R`, `y = B`, `color = red_win`.
• Add a `geom_point(size = .02)` layer.
• `facet_wrap()` by `vars(outcome, move)`.
• Label the x-axis "Red Fighter" and the y-axis "Blue Fighter".
• Tidy the plot and save it as `red_blue_scatterplot`.

**Exercise 3.36.** We have chosen to `facet_wrap()` `red_blue_scatterplot` by `vars(outcome, move)`. However, we usually use `facet_grid()` when faceting by two variables. Modify `red_blue_scatterplot` so that its panels are arranged according to `facet_grid()` rather than `facet_wrap()`. Can you see why `facet_wrap()` is superior in this case?

• Under `facet_grid()` an entire column of facets shares one horizontal axis and an entire row of facets shares one vertical axis. In contrast, every facet has its own horizontal axis and vertical axis under `facet_wrap()`.

### 3.2.10 Semi-Lengthening Pivots

You can arrange a tibble into any configuration with the all-roads-lead-to-Rome technique: first `pivot_longer()` the tibble to its narrowest form—the universally accessible “Rome” state—and then `pivot_wider()` the tibble into its new desired form. But this lengthen-then-widen approach has a drawback: it can change our columns’ data types. For example, in exercise 3.24 we collated the statistics of the shortest fight with the following:
ufc %>%
  filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
select(starts_with(c("R_", "B_")) & where(is.numeric)) %>%
  pivot_longer(
    col = everything(),
    names_to = c("corner_color", "metric"),
    names_pattern = "(.\_)(.+)",
    values_to = "stat",
    values_drop_na = TRUE
  ) %>%
  pivot_wider(
    names_from = corner_color,
    values_from = stat
  )

> # A tibble: 10 x 3
>   metric    R    B
>   <chr>    <dbl> <dbl>
> 1 odds      2.8  1.45
> 2 age       34   34
> 3 weight    170  170
> 4 height    180. 180.
> 5 reach     188. 185.
> 6 wins      15   1
> 7 losses    7    0
> 8 draw      0    0
> 9 rounds_fought 61   1
>10 sub_achieve 0    0

The where(is.numeric) condition restricts attention to numeric statistics. Removing this condition yields an error:

ufc %>%
  filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
select(starts_with(c("R_", "B_"))) %>%
  pivot_longer(
    col = everything(),
    names_to = c("corner_color", "metric"),
    names_pattern = "(.\_)(.+)",
    values_to = "stat",
    values_drop_na = TRUE
  ) %>%
The problem is that `pivot_longer()` can't store the `R_fighter` character strings and the `R_odds` numbers in the same vector, since they have different data types. We can resolve this issue by adding `values_transform = list(stat = as.character)` to the `pivot_longer()` call, which tells R to convert all numbers into character strings:

```r
ufc %>%
  filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
  select(starts_with(c("R_", "B_")))) %>%
  pivot_longer(
    col = everything(),
    names_to = c("corner_color", "metric"),
    names_pattern = "(.)._(.)",
    values_to = "stat",
    values_drop_na = TRUE,
    values_transform = list(stat = as.character)
  ) %>%
  pivot_wider(
    names_from = corner_color,
    values_from = stat
  )

# A tibble: 12 x 3
  metric   R       B
  <chr>    <chr>   <chr>
1 fighter Jorge Masvidal Ben Askren
2 odds    2.8     1.45454545454545
3 stance  Orthodox Orthodox
4 age     34      34
5 weight  170     170
6 height  180.34  180.34
7 reach   187.96  185.42
8 wins    15      1
9 losses  7       0
10 draw   0       0
```
The R and B columns are now character string vectors. Accordingly, all the numbers in this tibble are saved at text strings, which isn’t ideal.

To save numbers as integers or doubles we must assign the numbers and character strings to different columns. And we can do so by replacing `names_from = corner_color` with `names_from = metric` in our `pivot_wider()` call:

```r
ufc %>%
  filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
  select(starts_with(c("R", "B"))) %>%
  pivot_longer(
    col = everything(),
    names_to = c("corner_color", "metric"),
    names_pattern = "(.\_\.+)",
    values_to = "stat",
    values_drop_na = TRUE,
    values_transform = list(stat = as.character)
  ) %>%
  pivot_wider(
    names_from = metric,
    values_from = stat
  )
```

Unfortunately, the numeric columns—e.g., `odds`, `age`, `weight`, etc.—still have the character string data type. Here’s the problem: although the numeric and text-string data are segregated in both the initial tibble and final tibble, they’re pooled in the intermediary tibble between `pivot_longer()` and `pivot_wider()`. Specifically, `pivot_longer()` stacks all the statistics in a long character string column called `stat`.

This is the fundamental problem of the lengthen-then-widen pivoting strategy: by pooling the data, `pivot_longer()` gives each column the lowest-common-denominator data type. We can obviate this problem with the semi-lengthening pivot, which is like a `pivot_longer()`-cum-`pivot_wider()` packed into a single `pivot_longer()` call:
ufc %>%
  filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
  select(starts_with(c("R_", "B_"))) %>%
  pivot_longer(
    col = everything(),
    names_to = c("corner_color", "value"),
    names_pattern = "(.+)_(.+)"
  )

> # A tibble: 2 x 21
> corner_color fighter odds stance age weight height reach wins losses draw
> <chr>   <chr>  <dbl> <fct> <int> <int> <dbl> <dbl> <int> <int> <int>
> 1 R      Jorge  2.8  Ortho- 34 170 180. 188. 15    7   0
> 2 B      Ben    1.45 Ortho- 34 170 180. 185.  1    0   0
> # ... with 10 more variables: rounds_fought <int>, kd <int>,
> # strong_attempt <int>, strong_achieve <int>, weak_attempt <int>,
> # weak_achieve <int>, td_attempt <int>, td_achieve <int>, sub_attempt <int>,
> # sub_achieve <int>

The ".value" term in the names_to option makes this a semi-lengthening pivot. It is a special flag that tells R not to pull down this portion of the column names. Previously, we moved the words fighter, odds, stance, etc. from the column names to the metric column, only to immediately hoist them back above the tibble to serve as the names for our values_from columns. The semi-lengthening pivot doesn’t bother creating and then destroying this metric column. Instead, it keeps the words fighter, odds, stance, etc. above the tibble, as the names of our values_from columns, and it selectively pulls out of the column names only what it needs: the corner_color values, "R" and "B". And since it never stacks dissimilar variables, this semi-lengthening pivot does not corrupt our tibble’s data types.\(^8\)

For another example, the following semi-lengthening pivot derives medium_tib_1 from wide_tib:

wide_tib %>%
  pivot_longer(
    -R_fighter,
    names_to = c("strength", "value"),
    names_sep = "_
  )

\(^8\)Also, since it doesn’t create the temporary stat column, this semi-lengthening pivot does not need the values_to option.
> # A tibble: 2,002 x 4
> # Groups: R_fighter [1,001]
> R_fighter strength att ach
> <chr> <chr> <dbl> <dbl>
> 1 Aalon Cruz strong 12 2
> 2 Aalon Cruz weak 0 0
> 3 Aaron Phillips strong 54 38
> 4 Aaron Phillips weak 225 192
> 5 Aaron Riley strong 138 36
> 6 Aaron Riley weak 40 39
> 7 Aaron Rosa strong 99 59
> 8 Aaron Rosa weak 212 197
> 9 Aaron Simpson strong 90.7 48.7
> 10 Aaron Simpson weak 37.7 32
> # ... with 1,992 more rows

Previously, we needed both `pivot_longer()` and `pivot_wider()` to perform this transformation. But now we can combine these two functions into one `pivot_longer()` call by using ".value" to specify the component of the column names that should stay put. Specifically, this ".value" tells R to act as if we applied an extra `pivot_wider()` to the data, using the text after the "_" in the old column names as the names for the new `values_from` columns.

And the following semi-lengthening pivot derives `medium_tib_2` from `wide_tib`, with the extra proviso that `outcome` be a factor, `strong` be an integer, and `weak` be a double:

```r
wide_tib %>%
  pivot_longer(
    -R_fighter,
    names_to = c(".value", "outcome"),
    names_sep = "_",
    names_transform = list(outcome = as.factor),
    values_transform =
      list(
        strong = as.integer,
        weak = as.double
      )
  )
```

> # A tibble: 2,002 x 4
> # Groups: R_fighter [1,001]
> R_fighter outcome strong weak
> <chr> <fct> <int> <dbl>
> 1 Aalon Cruz att 12 0
> 2 Aalon Cruz ach 2 0

205
Whereas the previous semi-lengthening pivot lengthens the tibble along the strength axis, this semi-lengthening pivot lengthens it along the outcome axis.

**Exercise 3.37.** Define the following tibble:

```r
wide_punch <-
ufc %>%
  select(
    fight_id,
    contains(c("strong", "weak"))
  )
%>
na.omit

wide_punch %>%
head(4)
```

> # A tibble: 4 x 9
>       fight_id R_strong_attempt R_strong_achieve B_strong_attempt B_strong_achieve
>          <int>          <int>          <int>          <int>          <int>
> 1          75           151           94           125            66
> 2          76           272          137           268          102
> 3          77           309          194           157            83
> 4          78           205           71           235            82
> # ... with 4 more variables: R_weak_attempt <int>, R_weak_achieve <int>,
> # B_weak_attempt <int>, B_weak_achieve <int>

Each variable name of `wide_punch`, besides `fight_id`, encodes three variables: a color, either "R" or "B", a strength level, either "strong" or "weak", and an outcome, either "attempt" or "achieve". The following code pivots the tibble so that the color and outcome components are “lowered” (i.e., expressed inside the table) and the strength component is “raised” (i.e., expressed in the column names):
wide_punch %>%
  pivot_longer(
    -fight_id,
    names_to = c("color", "strength", "outcome"),
    values_to = "val",
    names_sep = "_"
  ) %>%
  pivot_wider(
    names_from = strength,
    values_from = val
  )

> # A tibble: 11,076 x 5
> fight_id color outcome strong weak
> <int> <chr> <chr> <int> <int>
> 1 75 R attempt 151 190
> 2 75 R achieve 94 169
> 3 75 B attempt 125 32
> 4 75 B achieve 66 22
> 5 76 R attempt 272 3
> 6 76 R achieve 137 2
> 7 76 B attempt 268 9
> 8 76 B achieve 102 9
> 9 77 R attempt 309 67
> 10 77 R achieve 194 64
> # ... with 11,066 more rows

Recreate this output without using a pivot_wider() call.

**Exercise 3.38.** Modify your solution to exercise 3.37 to create two additional tibbles. The first should have the color and strength components lowered and the outcome component raised, and the second should have the strength and outcome components lowered and the color component raised.

**Exercise 3.39.** Define the following tibble:

```r
fight_counts <-
  ufc %>%
  count(
    wday = wday(date, label = TRUE),
    month = month(date, label = TRUE),
    year = year(date)
)```

207
Recreate the following semi-lengthening pivots with traditional `pivot_longer()` and `pivot_wider()` calls:

**#Example 1:**
```r
fight_counts %>%
pivot_longer(everything(),
  names_to = c(".value", "month", "year"),
  names_sep = "_")
```

> # A tibble: 125 x 8
>   month year Sun Tue Wed Thu Fri Sat
> 1 Jan  2015  11 NA NA NA  NA  31
> 2 Jan  2016  13 NA NA NA  NA  24
> 3 Jan  2017  12 NA NA NA  NA  11
> 4 Jan  2018  10 NA NA NA  NA  23
> 5 Feb  2015   9 NA NA NA  NA  20
> 6 Feb  2016  12 NA NA NA  NA  23
> 7 Feb  2017  10 NA NA NA  NA  19
> 8 Feb  2018  11 NA NA NA  NA  33
> 9 Feb  2019  11 NA NA NA  NA  35
>10 Mar  2010  11 10 NA NA  NA  10
> # ... with 115 more rows
```

**#Example 2:**
```r
fight_counts %>%
pivot_longer(everything(),
  names_to = c(".value", "year"),
  names_pattern = "(.+)_([^-]+)"
)
```
3.3 Lab: Alibaba

3.3.1 Overview

Bray [2020] found a striking pattern in Alibaba’s track-package records: customers left higher shipping scores when their package’s logistics activities were reported later, closer to the parcel’s delivery. For example, consider the following track-package records that Alibaba shared with its customers:

```
Shipment   Action   Facility   Timestamp    Score
15007307   Order    -         2017-03-08 13:15 -
15007307   Consign   -         2017-03-10 17:14 -
15007307   Receive  49199 2017-03-14 19:27 -
15007307   Depart   49199 2017-03-14 19:51 -
15007307   Arrive   162115 2017-03-15 20:48 -
15007307   Depart   162115 2017-03-15 05:12 -
15007307   Arrive   166957 2017-03-15 06:29 -
15007307   Scan     166957 2017-03-15 07:28 -
15007307   Sign     166957 2017-03-15 10:04 5
```
Shipment 15007307 received a delivery score of 5/5, despite starting with an idle spell (between the 10th and 14th of March); in contrast, shipment 3144672 received a delivery score of only 1/5, likely due to its final spell of idleness (between the 6th and 10th of February). These examples illustrate the general principle: consumers leave higher parcel delivery ratings when the track-package activities scrunch toward the end of the shipping window. When the actions cluster towards the end of the shipping horizon, the shipper’s hard work is top of mind when the customer receives their package and uploads their shipping score. In contrast, when actions cluster towards the beginning of the shipping horizon, the shipper’s work is an old memory by the time the customer receives their package and uploads their shipping score. Moreover, a flurry of activity at the beginning could set artificially high expectations: e.g., if a Beijing resident sees a package wend its way from Handan to Beijing on Monday and Tuesday then they will probably expect it by Wednesday, and will be chagrined to wait until Friday for the local delivery. In contrast, a slow start can temper the customers’ expectations, so that they’re not as dismayed when the package hasn’t arrived by Thursday.

In this lab we will replicate the core of Bray’s analysis. However, we’ll use our `ggplot()` know-how to improve his graphs: our plots will be clearer and more convincing than those Bray originally presented. You can find Bray’s article here\textsuperscript{9}, and you can find summaries of it here\textsuperscript{10} and here\textsuperscript{11}.

### 3.3.2 Data

We will study `alibaba_wide`, which you created in section 1.3, exercise 1.30. This sample is a streamlined version of the panel Bray [2020] studied. (I reduced the sample size to shorten the computation time.) The tibble should have the following variables:

- **order**: Order ID number.
- **score**: Delivery logistics score left by the customer after receiving the package, rated from 1 (the worst) to 5 (the best).
- **item**: ID number for the product purchased. (Orders comprising multiple product types have been removed.)

\textsuperscript{9}https://pubsonline.informs.org/doi/10.1287/msom.2020.0899
\textsuperscript{10}https://insight.kellogg.northwestern.edu/article/improve-delivery-ratings
\textsuperscript{11}https://www.wsj.com/articles/with-online-package-delivery-alerts-later-in-the-process-is-better-11585352291
• brand, category, sub_category: ID number for the brand, category, and sub-category of the product purchased.

• merchant: Seller ID number.

• quantity: Number of products purchased in the given order. Orders may only comprise multiple products if they have the same item number.

• action.1–action.15: The track-package actions reported to the customer. action.1 corresponds to the first posted action, action.2 to the second posted action, etc. NA terms indicate no action: e.g., order = 6350829 has NA values for action.6–action.15 because it had only five posted actions. (Orders with more than 15 posted actions have been removed from the sample.) Besides NA, variables action.1–action.15 can take seven other values:

• "ORDER": The customer places the order.

• "CONSIGN": The warehouse sends the package to the shipper.

• "GOT": The shipper receives the package.

• "DEPARTURE": The package departs a facility.

• "ARRIVAL": The package arrives at a facility.

• "SENT_SCAN": The package is scanned for final delivery.

• "SIGNED": The customer signs for the package.

• "FAILURE": The shipper made a failed attempt to deliver the package.

• time.1–time.15: The times associated with the corresponding action: time.1 pertains to action action.1, time.2 to action action.2, etc. For the "ORDER" and "SIGNED" actions these timestamps record when the actions occurred. However, for the other actions these timestamps record when the actions were reported to the customer (i.e., posted online), which can happen with a bit of a time lag. Hence, all timestamps record when the customer was made aware of the given action (since customers are involved in "ORDER" and "SIGNED" actions, they know about these actions the moment they occur).

• facility.1–facility.15: The ID number of the facility the package was most recently recorded being at when the corresponding track-package action was posted. Unfortunately, facility values are updated only sporadically, so these variables often don’t make sense. For example, an "ARRIVAL" action won’t necessarily change the facility value.

• shipper.1–shipper.15: ID number for the shipper handling the package when the corresponding track-package action was posted. Although it’s rare, a package is sometimes handed off from shipper to another shipper. (For example, see order = 77359.)
3.3.3 Clean

We will now follow the methodology of Bray [2020] to clean our sample.

Exercise 3.40. Run `alibaba_wide %>% summarise(levels(score))` and you will see that the factor levels of `score` are not sequentially ordered. Use `mutate()` and `fct_relevel()` to order the `score` levels from "1" to "5".

- Consult the examples under the “Relevel with a function” heading of the `?fct_relevel` documentation.

Exercise 3.41. Use one of the two methods outlined below to pivot `alibaba_wide`—which has variables `order`, `score`, `item`, `brand`, `category`, `sub_category`, `merchant`, `quantity`, `action.1–action.15`, `time.1–time.15`, `facility.1–facility.15`, and `shipper.1–shipper.15`—into `alibaba_long`—which has variables `order`, `score`, `item`, `brand`, `category`, `sub_category`, `merchant`, `quantity`, `action`, `time`, `facility`, `shipper`, and `action_num`, the last of which takes values in 1 to 15.

Method 1: `pivot_longer()` then `pivot_wider()`

- Pipe `alibaba_wide` into `pivot_longer()`.
- Eliminate the columns whose names contain a "." with `cols = contains(".").
- Use `names_sep = "\\."` to separate the column names at the period (e.g., to split `shipper.12` into "shipper" and "12"). We need the double backslash here because the "." symbol has a special meaning in the language of regular expressions. The double backslash tells R to interpret this "." as a literal period rather than as this special symbol.
- Send the first part of the old column names to a new column called `stat` and send the second part to a new column called `action_num`. Define the latter column as an integer with `names_transform = list(action_num = as.integer)`.
- Send the values to a column called `value`.
- Turn the new value column into a character string vector with `values_transform = list(value = as.character)`. Since R can't store multiple data types in one column, we must convert all values to the lowest common denominator data type—the character string—before we can stack them vertically.
- Disregard superflous NA terms with `values_drop_na = TRUE`.
- `pivot_wider()` the result so that each row corresponds to a different (`order`, `action_num`) pair.
- This step should convert the `stat` and `value` columns into `action`, `time`, `facility`, and `shipper` columns.
- `mutate()` the resulting tibble to apply `ymd_hms()` to the `time` column and apply `as.factor()` to the `action`, `facility`, and `shipper` columns.
- `<-` the result into `alibaba_long`.
- This tibble should have 212305 rows and 13 columns. All but three of its columns should be factors.
Method 2: Semi-lengthening pivot.

- Pipe `alibaba_wide` into `pivot_longer()`.
- Eliminate the columns whose names contain a "." with `cols = contains(".").`
- Use `names_sep = "\\."` to separate the column names at the period (e.g., to split `shipper.12` into "shipper" and "12"). We need the double backslash here because the "." symbol has a special meaning in the language of regular expressions. The double backslash tells R to interpret this "." as a literal period rather than as this special symbol.
- Set the first part of the old column names to the new ".value" columns, and send the second part to a new column called `action_num`. Define the latter column as an integer with `names_transform = list(action_num = as.integer)`.
- Similarly use the `values_transform` option to define the `action`, `facility`, and `shipper` variables as factors.
- Use the `values_drop_na` option to disregard the `NA` values.
- `<-` the result into `alibaba_long`.
- This tibble should have 212305 rows and 13 columns. All but three of its columns should be factors.

**Exercise 3.42.** We will now redefine `time` so that an order’s earliest action occurs at `time = 0` and its latest action occurs at `time = 1`. In other words, we will modify `time` so that it measures the fraction of the total shipping time that has elapsed.

- Take `alibaba_long` and `group_by()` order.
- `mutate()` the result to do the following:
  - Subtract `min(time)` from `time`, so that `time` now measures the time since the first action.
  - Use `as.numeric(units="days")` to convert `time` from a “difftime” object to a standard number, measured in days.
  - Use `ceiling()` to define `day_count` as `max(time)` rounded up to the next integer. For example, if an order’s largest time value is 2.3 then it should have `day_count = 3` for all its observations.
  - Divide `time` by `max(time)`. (The median `time` should be 0.465 after this last step.)
- `ungroup()` the output of `mutate()` and `<-` it into `alibaba_long` (overwriting the old version).

Following Bray [2020], we will now remove the extreme shipments to standardize our sample.

**Exercise 3.43.** We will subject our sample to 10 data filters.

- Start with `alibaba_long` and `group_by()` order.
- `filter()` the grouped tibble accordingly:
• Remove the orders with any() "FAILURE" action. For example, you should remove all the order = 87717 observations since this order's 12th action was a "FAILURE".
• Remove orders without exactly one "ORDER" action, one "SIGNED" action, and one "CONSIGN" action.
• Remove orders that have an action before the "ORDER" action or after the "SIGNED" action. For example, you should remove all the order = 7923 observations since this order had a "DEPARTURE" reported after the package was "SIGNED". It's easier to base this filter on action_num because sometimes two actions happen at the exact same time. For example, your filter should not remove order = 43110, although its "SIGNED" action_num is smaller than this order's max(action_num).
• Remove orders that correspond to multiple shipper values.
• Remove orders with day_count > 8.
• Remove orders with more than 10 or fewer than 5 posted actions.
• Remove observations with "ORDER" and "SIGNED" actions, because their time values are degenerate (mechanically being either 0 or 1).
• ungroup() the filtered tibble and <- it into alibaba_long.
• alibaba_long should have 102331 rows after this step.

3.3.4 Analyze

We are now ready to analyze our sample. Our objective is to demonstrate that later actions yield higher scores—i.e., that time and score are positively correlated. Our empirical strategy is simple: we will plot the distribution of time for each value of score. These plots will demonstrate that actions cluster more toward the time = 0 pole when the score is low and cluster more toward the time = 1 pole when the score is high. We will create six plots of this nature in the next six exercises.

Exercise 3.44. Starting with alibaba_long, use geom_density() to plot the distribution of time conditional on score. Give each of the five distributions a different color. Save the plot as an R object called density_plot, so that we can modify it in later exercises.

• Near the time = 0 end, the score = 1 distribution should be highest, then the score = 2 distribution, then the score = 3 distribution, and then the score >= 4 distributions.
• And near the time = 1 end, the score = 5 distribution should be highest, then the score = 4 distribution, then the score = 3 distribution, then the score = 2 distribution, and then the score = 1 distribution.

Exercise 3.45. Use + to add day_count facets to density_plot.

• Instead of vars(day_count) use vars(str_c("Day Count: ", day_count)).

Exercise 3.46. Facet density_plot by the number of actions in a given order.

• pluck() the "data" from density_plot and pipe it into add_count() to create a new variable called action_count that records the number of actions (i.e., observations) that corresponding to a given order.
The values of `action_count` should be between three and eight. Initially we limited our sample to orders with between five and ten actions, but then we removed the "ORDER" and "SIGNED" actions.

Exercise 3.47. We will now depict the time distributions with box plots.

Exercise 3.48. A convincing way to establish a pattern in your data is to chop the sample into many independent groups and show that the pattern holds across nearly all of them. Adopting this strategy, we will show that `time` and `score` are positively correlated across all large `(shipper, category)` pairs.

Exercise 3.49. So far we’ve illustrated the positive correlation between `score` and `time` by showing how the univariate distribution of `time` varies with `score`. We will now illustrate the effect by showing how the bivariate distribution between the `action = "ARRIVAL"` and `action = "DEPARTURE"` time values vary with `score`.
• Use `summarise()` to calculate the number of observations in each group and the average time value in each group. Call the first statistic `n` and call the second statistic `time`.

• The order = 3686434, score = 5, and action = "GOT" row should have `n = 1` and `time = 0.0388`.

• Use `pivot_wider()` to translate the tibble to one with names `score`, `order`, `n_ARRIVAL`, `n_CONSIGN`, `n_DEPARTURE`, `n_GOT`, `n_SENT_SCAN`, `time_ARRIVAL`, `time_CONSIGN`, `time_DEPARTURE`, `time_GOT`, and `time_SENT_SCAN`.

• Your tibble should have 17895 rows after this step.

• Use `filter()` to impose `n_ARRIVAL < 3` and `n_DEPARTURE < 3`. This step will limit the number of facets in our plot.

• `ggplot()` the tibble with a `geom_density_2d_filled()` layer evaluated under `x = time_ARRIVAL` and `y = time_DEPARTURE`.

• `facet_grid()` by `str_c("Arrivals: ", n_ARRIVAL)` and `str_c("Departures: ", n_DEPARTURE)` in the columns and `str_c("Score: ", score)` in the rows.\(^\text{12}\)

• Your plot should have 20 facets in total.

3.3.5 Explore

Exercise 3.50. Depict Alibaba’s shipping network by applying the tidygraph and ggraph packages, which you can read about here\(^\text{13}\), to the `connection_count` tibble defined below to. Use `facet_edges()`, which is like `facet_wrap()`, to depict the network for various values of `connection_count`.

```r
#count number of deliveries that pass through both facility_a and facility_b
connection_count <-
  alibaba_long %>%
  distinct(order, facility) %>%
  rename(facility_a = facility) %>%
  inner_join(rename(., facility_b = facility_a)) %>%
  count(facility_a, facility_b, name = "num_deliveries") %>%
  filter(facility_a != facility_b)
```

\(^\text{12}\)Faceting is a good way to control for potential confounding variables. For example, faceting by `n_ARRIVAL` and `n_DEPARTURE` controls for these variables: none of the variation we plot in a facet is attributable to `n_ARRIVAL` or `n_DEPARTURE` because every observation in a facet has the same `n_ARRIVAL` and `n_DEPARTURE` values.

\(^\text{13}\)https://ggplot2-book.org/networks.html
3.4 Commentary

3.4.1 Saving Plots

You can save a plot by clicking on the “Export” tab by the plot in RStudio. Or, alternatively, you can use the `ggsave()` function. You don’t pipe into this function—you just call it immediately after creating your plot. For example, the following saves a box plot picture to boxplot.png:

```r
alibaba_long %>%
ggplot +
aes(
  x = as.factor(action_num),
  y = time
) +
geom_boxplot()

ggsave("boxplot.png")
```

We could customize this output by adding options to `ggsave()`. (See `?ggsave`.) And in addition to the plot itself, we also save the code that underlies it. This code is like the DNA of the plot—it specifies everything we need to regenerate it. So long as we have this code, we can update the plot with ease. Indeed, since we don’t manually fine-tune a `ggplot()` we can update a graph every day at essentially no cost, or we can wait a decade to update it and the new version will look as slick as ever, even if we’ve forgotten nearly everything about the data. And even if you have forgotten how a plot works, the code lays out the recipe in a straightforward linear fashion, with each option clearly “plugged in” with a `+` sign.

3.4.2 Don’t Overburden Metadata

Long tibbles are generally more practicable than wide tibbles. There are several reasons for this.

First, lengthening a tibble transforms metadata—the column labels—to formal data—a column in the tibble. And formal data are easier to work with than metadata. For example, suppose we wanted to combine `planet_1`, from section 3.1, with the following tibble:

```r
unit_of_measure <-
tibble(~stat,
    "radius", ~unit,
    "miles",
```

217
"dist", "light-seconds",
"year", "Earth days"
)

unit_of_measure

> # A tibble: 3 x 2
> stat unit
> <chr> <chr>
> 1 radius miles
> 2 dist light-seconds
> 3 year Earth days

Unfortunately, merging `unit_of_measure` and `planet_1` would be difficult, since the latter stores `radius`, `dist`, and `year` in the column names metadata. However, if we convert these column labels to a proper column in our tibble then the merge becomes tractable:

```r
planet_long <-
  planet_1 %>%
  pivot_longer(
    -planet,
    names_to = "stat"
  )

planet_merged <-
  planet_long %>%
  inner_join(unit_of_measure)

planet_merged

> # A tibble: 12 x 4
> planet stat value unit
> <chr> <chr> <dbl> <chr>
> 1 Mercury radius 1516 miles
> 2 Mercury dist 193 light-seconds
> 3 Mercury year 88 Earth days
> 4 Venus radius 3760 miles
> 5 Venus dist 361 light-seconds
> 6 Venus year 225 Earth days
> 7 Earth radius 3958 miles
> 8 Earth dist 499 light-seconds
> 9 Earth year 365 Earth days
```
As you see, moving data from above the tibble, in the column labels, to inside the tibble makes it more accessible. Likewise, moving data from besides a tibble, in its row labels, to inside the tibble makes it more accessible. This is why you should never refer to a tibble’s rownames() metadata; instead, if these row names contain valuable information then you should add them as a column with rownames_to_column() or rowid_to_column().

Second, it’s usually harder to analyze data that are scattered across many variables than data that are concentrated within a few variables. For example, had we not converted alibaba_wide to alibaba_long, our solution to exercise 3.42 would have looked something like this:
pmin( #Note: use pmin() instead of min()
  time.1, time.2, time.3, time.4, time.5,
  time.6, time.7, time.8, time.9, time.10,
  time.11, time.12, time.13, time.14, time.15,
  na.rm = TRUE
),

time.1 = time.1 - time_min,
time.2 = time.2 - time_min,
time.3 = time.3 - time_min,
time.4 = time.4 - time_min,
time.5 = time.5 - time_min,
time.6 = time.6 - time_min,
time.7 = time.7 - time_min,
time.8 = time.8 - time_min,
time.9 = time.9 - time_min,
time.10 = time.10 - time_min,
time.11 = time.11 - time_min,
time.12 = time.12 - time_min,
time.13 = time.13 - time_min,
time.14 = time.14 - time_min,
time.15 = time.15 - time_min,

time.1 = as.numeric(time.1, units="days"),
time.2 = as.numeric(time.2, units="days"),
time.3 = as.numeric(time.3, units="days"),
time.4 = as.numeric(time.4, units="days"),
time.5 = as.numeric(time.5, units="days"),
time.6 = as.numeric(time.6, units="days"),
time.7 = as.numeric(time.7, units="days"),
time.8 = as.numeric(time.8, units="days"),
time.9 = as.numeric(time.9, units="days"),
time.10 = as.numeric(time.10, units="days"),
time.11 = as.numeric(time.11, units="days"),
time.12 = as.numeric(time.12, units="days"),
time.13 = as.numeric(time.13, units="days"),
time.14 = as.numeric(time.14, units="days"),
time.15 = as.numeric(time.15, units="days"),

time_max =
  pmax(
    time.1, time.2, time.3, time.4, time.5,
    time.6, time.7, time.8, time.9, time.10,
    time.11, time.12, time.13, time.14, time.15,
na.rm = TRUE
)

day_count = ceiling(time_max),

time.1 = time.1/time_max,
time.2 = time.2/time_max,
time.3 = time.3/time_max,
time.4 = time.4/time_max,
time.5 = time.5/time_max,
time.6 = time.6/time_max,
time.7 = time.7/time_max,
time.8 = time.8/time_max,
time.9 = time.9/time_max,
time.10 = time.10/time_max,
time.11 = time.11/time_max,
time.12 = time.12/time_max,
time.13 = time.13/time_max,
time.14 = time.14/time_max,
time.15 = time.15/time_max
)

And this answer would be 10 times longer if there were 150 possible actions rather than 15 possible actions. By consolidating the data in time.1–time.15 into a single time variable, the pivot_longer() step prevents us from having to type out so many column names.

Third, it’s usually easier to widen a long tibble than it is to convert one wide tibble into another wide tibble. Indeed, recall that all roads lead to Rome, where “Rome” represents the longest possible tibble configuration: every wide tibble can stretch into the longest form, and the longest form can widen out into every other form. So storing your data in “long mode” makes it just one pivot_wider() away from any other tibble configuration. For example, define the following:

```r
wide_example_1 <-
  alibaba_long %>%
pivot_wider(
    id_cols = action,
    names_from = action_num,
    values_from = time,
    values_fn = mean
  )
```
Now note that `long_example` is just one short pivot step away from both `wide_example_1` and `wide_example_2`:

```r
# Derive wide_example_1 from long_example
long_example %>%
  pivot_wider(
    names_from = action_num,
    values_from = mean_time
  )

# Derive wide_example_2 from long_example
long_example %>%
  pivot_wider(
    names_from = action,
    values_from = mean_time
  )
```

However, converting `wide_example_1` to `wide_example_2` requires two pivots:

```r
wide_example_1
wide_example_2 <-
alibaba_long %>%
pivot_wider(
  id_cols = action_num,
  names_from = action,
  values_from = time,
  values_fn = mean
)

wide_example_2
long_example <-
alibaba_long %>%
group_by(action, action_num) %>%
summarise(
  mean_time = mean(time),
  .groups = "drop"
)

long_example
```
And fourth, since they have fewer columns, long tibbles tend to have fewer opportunities for “missing data” and thus fewer NA values. For example, `wide_example_1` and `wide_example_2` both have five NA values, but `long_example` has none.

For these four reasons, it’s usually easiest to `pivot_longer()` your data upfront and work primarily in long mode. Indeed, there’s usually only two reasons to widen a tibble: (i) to configure it for `ggplot()` or (ii) to perform a computation that references multiple contemporaneous variables. For example, suppose I wanted to calculate the difference between the `action = "DEPARTURE"` and `action = "ARRIVAL"` mean_time values, for each value of `action_num`. We could derive this from `long_example` without widening the tibble:

```r
long_example %>%
  group_by(action_num) %>%
  summarise(
    time_diff =
      weighted.mean(mean_time, action == "DEPARTURE") -
      weighted.mean(mean_time, action == "ARRIVAL")
  )
```

However, the solution is more elegant if we use `pivot_wider()`:

```r
long_example %>%
  pivot_wider(
    names_from = action,
    values_from = mean_time
  ) %>%
  mutate(time_diff = DEPARTURE - ARRIVAL)
```
3.4.3 Aggregating Pivots

Sometimes a `pivot_wider()` operation shortens a tibble faster than it widens it so that the reconfigured table doesn’t have enough cells to comfortably store all the information. For example, consider the following tibble:

```r
R_from_july_2011 <-
  ufc %>%
  filter(
    floor_date(date, "month") == "2011-07-01"
  ) %>%
  select(
    R_fighter,
    red_win,
    result
  )

R_from_july_2011
```

```
> # A tibble: 11 x 3
>   R_fighter red_win result
>   <chr>     <lgl>   <fct>
> 1 Dominick Cruz TRUE   U-DEC
> 2 Wanderlei Silva FALSE KO/TKO
> 3 Dennis Siver TRUE   U-DEC
> 4 Tito Ortiz TRUE   SUB
> 5 Carlos Condit TRUE   KO/TKO
> 6 Melvin Guillard TRUE   KO/TKO
> 7 George Sotiropoulos FALSE KO/TKO
> 8 Brian Bowles TRUE   U-DEC
> 9 Brad Tavares FALSE KO/DEC
>10 Anthony Njokuani TRUE   U-DEC
>11 Jeff Hougland TRUE   U-DEC
```

And now let’s pivot this tibble so that the values of `red_win` run along the rows and the values of `result` run along the columns:

```r
R_from_july_2011 %>%
  pivot_wider(
    names_from = result,
    values_from = R_fighter
  )
```

224
> Warning: Values from `R_fighter` are not uniquely identified; output will contain list-cols.
> * Use `values_fn = list` to suppress this warning.
> * Use `values_fn = {summary_fun}` to summarise duplicates.
> * Use the following dplyr code to identify duplicates.
> ```r
> {data} %>%
>   dplyr::group_by(red_win, result) %>%
>   dplyr::summarise(n = dplyr::n(), .groups = "drop") %>%
>   dplyr::filter(n > 1L)
> # A tibble: 2 x 4
> red_win `U-DEC` `KO/TKO` SUB
> <lgl> <list> <list> <list>
> # A tibble: 2 x 4
> red_win `U-DEC` `KO/TKO` SUB
> <lgl> <int> <int> <int>
> 1 TRUE 5 2 1
> 2 FALSE 1 2 NA
>```

As you see, we now get the warning that “Values are not uniquely identified.” The problem is that our initial tibble had 11 `R_fighter` values, but only two distinct `red_win` values and three distinct `result` values. Hence, the pivoted table has only $2 \times 3 = 6$ cells to store 11 text strings. Accordingly, to cram all the data into this table, R makes each cell store a list of text strings rather than a single text string. Lists are basically glorified vectors (we'll cover them in the next section). But they allow you to store multiple objects in a single tibble cell. For example, the first cell in the second column houses `list(c("Dominick Cruz", "Dennis Siver", "Brian Bowles", "Anthony Njokuani", "Jeff Hougland"))` (i.e., all the fights with `red_win = TRUE` and `result = 'U-DEC').

Whereas the previous `pivot_wider()` call used lists to retain all the information in the original tibble, the following `pivot_wider()` retains only the number of observations in each case:

```
R_from_july_2011 %>%
  pivot_wider(
    names_from = result,
    values_from = R_fighter,
    values_fn = length
  )
# A tibble: 2 x 4
red_win `U-DEC` `KO/TKO` SUB
<lgl> <int> <int> <int>
1 TRUE 5 2 1
2 FALSE 1 2 NA
```
For example, the first value in the second column is now `length(c("Dominick Cruz", "Dennis Siver", "Brian Bowles", "Anthony Njokuani", "Jeff Hougland")) = 5`. The bottom-right value is `NA` because there's no vector of observations with `red_win = FALSE` and `result = "SUB"` that R can take the `length()` of. And when a cell has no corresponding vector, R assigns it the default value of `NA`. However, we can change this default value to zero with the `values_fill = 0` option:

```r
R_from_july_2011 %>%
  pivot_wider(
    names_from = result,
    values_from = R_fighter,
    values_fn = length,
    values_fill = 0
  )
```

> # A tibble: 2 x 4
> red_win `U-DEC` `KO/TKO` SUB
> <lgl> <int> <int> <int>
> 1 TRUE 5 2 1
> 2 FALSE 1 2 0

Now instead of `length()`, we can pass in any aggregating function we like into the `values_fn` option. For example, the following tells R to use `str_flatten()` to compress the multiple text strings assigned to a cell into one long text string:

```r
R_from_july_2011 %>%
  pivot_wider(
    names_from = result,
    values_from = R_fighter,
    values_fn = str_flatten,
    values_fill = "NO DATA"
  )
```

> # A tibble: 2 x 4
> red_win `U-DEC` `KO/TKO` SUB
> <lgl> <chr> <chr> <chr>
> 1 TRUE Dominick Cruz Dennis Siver Brian Bowles Anthony Njokuani~ Carlos ~ Tito~
> 2 FALSE Brad Tavares Wanderl~ NO D~

For example, the first cell in the second column now reads `str_flatten(c("Dominick Cruz", "Dennis Siver", "Brian Bowles", "Anthony Njokuani", "Jeff Hougland")) = "Dominick`
Cruz, Dennis Siver, Brian Bowles, Anthony Njokuani, Jeff Hougland. And the values_fill = "NO DATA" option tells R to make a cell store "NO DATA" if it has no other corresponding text strings.

And we can even pass in home-made functions (see section 2.2.5):

```r
paste_last_names <-
  . %>%
  word(2) %>%
  str_flatten(collapse = ", ", )

R_from_july_2011 %>%
pivot_wider(
  names_from = result,
  values_from = R_fighter,
  values_fn = paste_last_names
)
```

> # A tibble: 2 x 4
>   red_win `U-DEC` `KO/TKO` SUB
> 1 TRUE Cruz, Siver, Bowles, Njokuani, Hougland Condit, Guillard Ortiz
> 2 FALSE Tavares Silva, Sotiropoulos <NA>

Now the first cell in the second column reads `c("Dominick Cruz", "Dennis Siver", "Brian Bowles", "Anthony Njokuani", "Jeff Hougland") %>% word(2) %>% str_flatten(collapse = ", ", ) = "Cruz, Siver, Bowles, Njokuani, Hougland".

Note, while these aggregating pivots are pretty slick we can always yield the same result by combining a traditional `pivot_wider()` call with a `summarise()` step. For example, the following outputs the same tibble (albeit with its rows rearranged):

```r
R_from_july_2011 %>%
group_by(red_win, result) %>%
summarise(
  names = paste_last_names(R_fighter),
  .groups = "drop"
) %>%
pivot_wider(
  names_from = result,
  values_from = names
)
```
Exercise 3.51. The following reports the average time in the round that each finishing move took place, by finishing_round and gender:

```
ufc %>%
  filter(!is.na(finish_round_time), !is.na(finish_details)) %>%
  group_by(finish_details, finish_round, gender) %>%
  summarise(mean_time = mean(finish_round_time), .groups = "drop") %>%
  pivot_wider(names_from = c(finish_round, gender), values_from = mean_time)
```

Recreate this tibble without the `group_by()` and `summarise()` steps. (Your rows and columns may be ordered differently.)
• Use the `id_cols = finish_details` option of `pivot_wider()` to drop all non-relevant variables. For example, `x %>% pivot_wider(id_cols = a, names_from = b, values_from = c)` is equivalent to `x %>% select(a, b, c) %>% pivot_wider(names_from = b, values_from = c).

**Exercise 3.52.** We will now make a big table that reports the median fighter age in each `weight_class` in each `location`. The rows of our output will vary by `location` and the columns will vary by `weight_class`.

• Use `pivot_longer()` to consolidate the values in `R_age` and `B_age` into a common column called `age`.
  
  – Your tibble should have 8222 rows after this step.

• Pipe the result into `pivot_wider()`. Use `id_cols = location` to specify the row variable and `names_from = weight_class` to specify the column variable. Also use the `values_from` and `values_fn` options.

• Your final tibble should have 145 rows and 14 columns.

### 3.4.4 Ungroup Data before Saving

We could have avoided the `group_by()` step at the beginning of exercise 3.43 if we hadn’t included an `ungroup()` step at the end of exercise 3.42. However, you should commit to always saving your data in an ungrouped state because otherwise you’ll constantly be having to check whether your input tibble is grouped or not. Consistently leaving your named tibbles ungrouped helps you standardize your workflow, which will help you both in terms of speed and precision.

### 3.4.5 Invert Dependency in Plots

The outputs in section 3.3.4 depict the distribution of `time` as a function of the `score` color. However, this is a bit unusual because `time` is actually the “independent variable” and `score` the “dependent variable”—i.e., `score` responds to `time` and not the other way around. So it may seem more intuitive to plot the distribution of `score` as a function of `time`. However, such plots would be less satisfying because it’s harder to depict the distribution of a discrete variable and to “color by” a continuous variable. Hence, although `score` actually depends on `time`, the plots are more informative if we illustrate how `time` depends on `score` in our sample.

### 3.5 Solutions

3.1
```r
point_height_plot <-
    layerless_height_plot +
    geom_point(size = .01)

point_height_plot

jitter_height_plot <-
    layerless_height_plot +
    geom_jitter(size = .01, height = 1, width = 1)

jitter_height_plot

3.2

ejitter_reach_plot <-
ejitter_height_plot +
aes(x = R_reach, y = B_reach) +
labs(
    x = "Red Reach",
    y = "Blue Reach"
)
jitter_reach_plot

3.3

heat_map_plot <-
ufc %>%
ggplot +
aes(
    x = R_weight,
    y = B_weight
) +
geom_density2d_filled() +
labs(
    x = "Weight of Fighter in the Red Corner",
    y = "Weight of Fighter in the Blue Corner"
)

heat_map_plot
```
3.4

```r
ufc %>%
mutate(
  weight_class =
    fct_reorder(weight_class, R_weight, mean)
) %>%
ggplot +
aes(
  x = date,
  color = weight_class,
  linetype = weight_class
) +
geom_freqpoly(bins = 20) +
facet_wrap(
  vars(gender),
  ncol = 1
) +
scale_color_grey() +
theme_bw() +
theme(strip.background = element_rect(color = "white", fill = "white"))
```

3.5

```r
ufc %>%
filter(weight_class != "Catch Weight") %>%
mutate(
  weight_class =
  weight_class %>%
    str_replace("Women's", "W."),
  str_remove("weight")
) %>%
ggplot +
aes(
  x = abs(B_odds - R_odds),
  y = fight_time
) +
geom_point(size = .2) +
geom_smooth(method = "lm") +
facet_wrap(
  vars(weight_class),
  scales = "free",
)
\( ncol = 4 \)

3.6–3.7

```r
ufc %>%
  mutate(
    house_take = 1/R_odds + 1/B_odds - 1,
    country =
      location %>%
        str_extract("[^,]*") %>%
        fct_relevel(rev)
  ) %>%
  ggplot +
  aes(
    x = house_take,
    y = country
  ) +
  geom_violin() +
  labs(
    x = "",
    y = "House Take"
  )
```

3.8–3.10

```r
finish_details_plot <-
ufc %>%
  filter(
    result != "DQ",
    !is.na(finish_details),
    !is.na(finish_round_time)
  ) %>%
  group_by(finish_details) %>%
  mutate(
    finish_details =
      ifelse(
        n() < 18,
        str_c(result, "Other", sep = " ")
      ,
      finish_details
    )
```

232
```r
ungroup %>%
mutate(
  finish_details =
    fct_reorder(
      finish_details, finish_round_time, median,
      .desc = TRUE
    )
)

%>%
ggplot +
aes(
  x = finish_round_time,
  y = finish_details,
  fill = result
)
%>
geom_boxplot() +
theme_bw() +
labs(
  x = "Time in Round",
  y = "Finishing Move"
) +
theme(
  legend.position = "top",
  legend.title = element_blank()
)

finish_details_plot

3.11

new_plot_data <-
finish_details_plot %>%
pluck("data") %>%
mutate(
  upset =
    ifelse(
      red_win,
      R_odds > B_odds,
      R_odds <= B_odds
    ),
  winner =
    ifelse(
      red_win,
      R_odds > B_odds,
      R_odds <= B_odds
    )
)
```
upset,
"Underdog Won",
"Favored Fighter Won"
)
)

finish_details_plot %>%
  new_plot_data +
  facet_wrap(vars(winner)) +
  theme(strip.background = element_rect(color = "white", fill = "white"))

3.12

stance_plot %>% {
  ufc %>%
    group_by(location, weight_class, B_stance, R_stance) %>%
    summarise(spread = min(B_rounds_fought - R_rounds_fought))
}

3.13

make_stance_plot <- function(sample) {
  data_to_plot <-
    sample %>%
    mutate(spread = B_age - R_age)

    stance_plot %>% data_to_plot
}

# or even better:
make_stance_plot <-
  . %>%
  mutate(spread = B_age - R_age) %>%
  {stance_plot %>% .}

ufc %>%
  filter(R_stance != B_stance) %>%
  make_stance_plot

ufc %>%
  filter(R_age < 28) %>%
  make_stance_plot
new_plot_data <-
    finish_details_plot %>%
    pluck("data") %>%
    group_by(weight_class) %>%
    mutate(
      R_aggregate_weight = median(R_weight),
      B_aggregate_weight = median(B_weight)
    )

heat_map_plot %+
new_plot_data +
geom_point(
  aes(
    x = R_aggregate_weight,
    y = B_aggregate_weight
  )
)

#Here's an example plot:
formula_1 %>%
mutate(position_bin = ntile(start, 6)) %>%
group_by(position_bin) %>%
mutate(
  position =
    str_c("start <= ", max(start))
) %>%
group_by(flag_d) %>%
mutate(
  flag_d =
    ifelse(n() < 18000, "Other", flag_d)
) %>%
ungroup %>%
mutate(
  flag_d =
    flag_d %>%
    fct_relevel("Other", after = Inf),
  position =
    position %>%
as.factor %>%
  fct_reorder(start, mean)
)

%>

mutate(lap_count = max(lap)) %>%
group_by(date, driver)
filter(max(lap) == lap_count) %>%

mutate(seconds = seconds/mean(seconds),
  percent_race = 2 * ntile(lap, 50)) %>
group_by(circuit)
mutate(seconds = seconds/mean(seconds),
  percent_race = 2 * ntile(lap, 50)) %>
group_by(circuit)

summarise(seconds = median(seconds)) %>
ggplot +
aes(
  x = percent_race,
  y = seconds,
  color = position
) +
geom_line() +
facet_wrap(
  vars(flag_d),
  scales = "free"
) +
theme_minimal() +
labs(
  x = "Percent of Race Completed",
  y = "Lap Time Relative to Median Value",
  legend = "NULL"
) +
theme(
  legend.title = element_blank(),
  legend.position = "top"
)
3.17

```r
planet_1 %>%
pivot_longer(
  cols = c(radius, dist, year),
  names_to = "stat",
  values_to = "value"
)
```

3.18

```r
planet_2 %>%
pivot_wider(
  names_from = planet,
  values_from = value
)
```

3.19

```r
planet_4 <-
  planet_2 %>%
pivot_wider(
  names_from = c(planet, stat),
  values_from = value
)
```

3.20

```r
planet_1 %>%
pivot_longer(
  cols = c(radius, dist),
  names_to = "stat",
  values_to = "value"
)
```

3.21

237
planet_4 %>%
  pivot_longer(
    cols = everything(),
    names_to = c("planet", "stat"),
    names_sep = ",",
    values_to = "value"
  )

3.22

ufc %>%
  pivot_longer(
    cols = c(R_fighter, B_fighter),
    names_to = "corner_color",
    values_to = "fighter_name"
  ) %>%
  count(fighter_name) %>%
  ggplot +
  aes(x = n) +
  geom_histogram()

3.23

planet_1 %>%
  pivot_longer(
    cols = -planet,
    names_to = "stat",
    values_to = "value"
  ) %>%
  pivot_wider(
    names_from = planet,
    values_from = value
  )

3.24

ufc %>%
  filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
  select(starts_with(c("R_", "B_")) & where(is.numeric)) %>%
pivot_longer(
  col = everything()
)

pivot_wider(
  names_from = corner_color,
  values_from = stat
)

3.26

ufc %>%
  select(fight_id, R_wins, B_wins, R_losses, B_losses) %>%
  pivot_longer(
    cols = -fight_id,
    names_to = c("corner_color", "stat"),
    names_sep = ",",
    values_to = "val"
  ) %>%
  pivot_wider(
    names_from = stat,
    values_from = val
  ) %>%
  mutate(
    corner_color =
      fct_recode(
        corner_color,
        `Red-Corner Fighter` = "R",
        `Blue-Corner Fighter` = "B"
      )
  ) %>%
  ggplot +
  aes(x = wins, y = losses) +
  geom_jitter(
    size = .01,
    height = .3,
    width = .3
  ) +
  facet_wrap(
vars(corner_color),
col = 2,
scales = "free"
) +
labs(x = "Wins", y = "Losses") +
theme(
  panel.background = element_rect(fill = "white", colour = "white"),
  strip.background = element_rect(color = "white", fill = "white"),
  strip.text.x = element_text(size = 12)
)

3.27

formula_1 %>%
group_by(circuit) %>%
mutate(speed_rank = row_number(seconds)) %>%
ungroup %>%
filter(speed_rank <= 5) %>%
arrange(speed_rank) %>%
pivot_wider(
  id_cols = circuit,
  names_from = speed_rank,
  values_from = team
)

# or

formula_1 %>%
group_by(circuit) %>%
mutate(speed_rank = row_number(seconds)) %>%
ungroup %>%
filter(speed_rank <= 5) %>%
arrange(speed_rank) %>%
select(circuit, speed_rank, team) %>%
pivot_wider(
  id_cols = circuit,
  names_from = speed_rank,
  values_from = team
)

3.28
\[
\text{formula}_1 \%>\% \\
\quad \text{filter(year(date) } \geq 2022\text{)} \%>\% \\
\quad \text{select}-c(lap, seconds)\%>\% \\
\quad \text{distinct } \%>\% \\
\quad \text{pivot}_{\text{longer}}( \\
\quad \quad c("start", "end"), \\
\quad \quad \text{names_to} = "measure", \\
\quad \quad \text{values_to} = "position" \\
\quad ) \%>\% \\
\quad \text{arrange(position)} \%>\% \\
\quad \text{pivot}_{\text{wider}}( \\
\quad \quad \text{id_cols} = c(\text{circuit}, \text{measure}), \\
\quad \quad \text{names_from} = \text{position}, \\
\quad \quad \text{values_from} = \text{driver} \\
\quad ) \\
\] \\
\#or \\
\text{formula}_1 \%>\% \\
\quad \text{filter(year(date) } \geq 2022\text{)} \%>\% \\
\quad \text{select}-c(lap, seconds)\%>\% \\
\quad \text{distinct } \%>\% \\
\quad \text{pivot}_{\text{longer}}( \\
\quad \quad c("start", "end"), \\
\quad \quad \text{names_to} = "measure", \\
\quad \quad \text{values_to} = "position" \\
\quad ) \%>\% \\
\quad \text{arrange(position)} \%>\% \\
\quad \text{select}(\text{circuit, measure, position, driver}) \\
\quad \text{pivot}_{\text{wider}}( \\
\quad \quad \text{names_from} = \text{position}, \\
\quad \quad \text{values_from} = \text{driver} \\
\quad ) \\
\] \\

3.29–3.31 \\
\text{ufc} \%>\% \\
\quad \text{mutate}( \\
\quad \quad \text{house\_take} = \frac{1}{\text{R\_odds}} + \frac{1}{\text{B\_odds}} - 1, \\
\quad \quad \text{date\_bucket} = \\
\quad \quad \text{date} \%>\% \\
\quad \quad \text{ntile}(200), \)
```r
weight_bucket =
  (R_weight + B_weight) %>%
  ntile(2) %>%
  as.factor %>%
  fct_recode(
    light_fights = "1",
    heavy_fights = "2"
  )
)
%>%
group_by(date_bucket) %>%
mutate(date = median(date)) %>%
group_by(date, weight_bucket) %>%
summarise(house_take_mean = mean(house_take)) %>%
pivot_wider(
  names_from = weight_bucket,
  values_from = house_take_mean
)
%>%
ggplot +
aes(
  x = light_fights,
  y = heavy_fights,
  color = date
) +
geom_path() +
theme_bw()

3.32

ufc %>%
  mutate(
    house_take = 1/R_odds + 1/B_odds - 1,
    date_bucket =
      date %>%
      ntile(100),
    weight_bucket =
      (R_weight + B_weight) %>%
      ntile(2) %>%
      as.factor %>%
      fct_recode(
        light_fights = "1",
        heavy_fights = "2"
      )
  )
```

```r
) %>%
group_by(date_bucket) %>%
mutate(date = median(date)) %>%
group_by(date, weight_bucket) %>%
summarise(house_take_mean = mean(house_take)) %>%
ggplot +
aes(x = date, y = house_take_mean, color = weight_bucket) %>%
geom_line()
```

3.33–3.35

```r
red_blue_scatterplot <-
ufc %>%
select(
  fight_id, red_win,
  contains(c("attempt", "achieve"))
) %>%
pivot_longer(
  cols = -c(fight_id, red_win),
  names_to = c("corner_color", "move", "outcome"),
```
names_sep = "_",
values_drop_na = TRUE
)

pivot_wider(
  names_from = corner_color,
  values_from = value
)

mutate(
  red_win =
    red_win %>%
    as.factor %>%
    fct_recode(
      'Red Win' = "TRUE",
      'Blue Win' = "FALSE"
    ),
  move =
    move %>%
    fct_relevel("strong", "weak", "td") %>%
    fct_recode(
      'Strong Strike' = "strong",
      'Weak Strike' = "weak",
      Takedown = "td",
      Submission = "sub"
    ),
  outcome = str_to_title(outcome)
)

ggplot +
aes(x = R, y = B, color = red_win) +
  geom_point(size = .02) +
  facet_wrap(
    vars(outcome, move),
    scales = "free",
    nrow = 2
  ) +
  theme_bw() +
  theme(
    legend.position = "top",
    legend.title = element_blank(),
    strip.background = element_rect(color = "white", fill = "white")
  ) +
  labs(
    x = "Red Fighter",
    y = "Blue Fighter"
  )
3.36

# The takedown facets are poorly scaled under facet_grid:
red_blue_scatterplot +
  facet_grid(
    rows = vars(outcome),
    cols = vars(move),
    scales = "free"
  )

# Switching the rows and columns doesn't help:
red_blue_scatterplot +
  facet_grid(
    rows = vars(move),
    cols = vars(outcome),
    scales = "free"
  )

3.37

wide_punch %>%
  pivot_longer(
    cols = -fight_id,
    names_to = c("color", ".value", "outcome"),
    names_sep = "_
  )

3.38

#1
wide_punch %>%
  pivot_longer(
    cols = -fight_id,
    names_to = c("color", "strength", ".value"),
    names_sep = "_
  )
```r
#2
wide_punch %>%
  pivot_longer(
    cols = -fight_id,
    names_to = c(".value", "strength", "outcome"),
    names_sep = "_
  )

3.39

#1
fight_counts %>%
  pivot_longer(
    everything(),
    names_to = c("wday", "month", "year"),
    names_sep = "_
    values_to = "fight_count"
  ) %>%
  pivot_wider(
    names_from = wday,
    values_from = fight_count
  )

#2
fight_counts %>%
  pivot_longer(
    everything(),
    names_to = c("wday_month", "year"),
    values_to = "fight_count",
    names_pattern = "([_.])([^_.]+)"
  ) %>%
  pivot_wider(
    names_from = wday_month,
    values_from = fight_count
  )

3.51

ufc %>%
  filter(
```
!is.na(finish_round_time),
!is.na(finish_details)
)

%>%
pivot_wider(
  id_cols = finish_details,
  names_from = c(finish_round, gender),
  values_from = finish_round_time,
  values_fn = mean
)

3.52

ufc %>%
pivot_longer(
  c(R_age, B_age),
  values_to = 'age'
)

%>%
pivot_wider(
  id_cols = location,
  names_from = weight_class,
  values_from = age,
  values_fn = median
)
Chapter 4

Multi-Column Operations and Joins

4.1 Introduction

After this chapter you'll know all the key techniques for manipulating tibbles. Indeed, we only have two data wrangling topics yet to cover: applying operations across multiple columns and joining tibbles. We will address the former topic in sections 4.2.2 and 4.2.3, and the latter topic in section 4.2.5.

We'll begin, in section 4.2.2, by learning how to refer to multiple tibble columns. R has a suite of “tidy select” functions that allow us to select various subsets of columns in a tidy fashion. We've actually used some of these tidy select functions before. For example, consider the following code, from section 3.2.7:

```r
numeric_stats <-
    ufc %>%
    select(  
        fight_id,  
        where(is.numeric) & starts_with(c("R_", "B_"))
    )
```

The `where()` and `starts_with()` are both tidy select functions. There are several other functions like this, which together comprise an elegant scheme for specifying variables.

Next, in section 4.2.3 we turbocharge functions like `mutate()`, `summarise()`, `filter()`, and `pivot_longer()` by nesting tidy select calls within them. This is a new method for applying operations across multiple columns is tremendously powerful.\footnote{For more on this topic see `vignette("colwise")` and `vignette("programming")`.
After, in section 4.2.5 I discuss basic tibble joining and in section 4.2.6 I discuss more advanced “mutating joins,” which adds columns, and “filtering joins,” which remove rows. I then provide a case study that that uses tibble joining to study networks.\(^2\)

Finally, this chapter has three labs. The first, in section 4.3, repeats the analysis of Bray et al. [2023] to show that customers are less sensitive to price changes than typical supermarket data would have you believe. This lab has some light joins and multi-column operations. The second lab, in section 4.4, repeats the analysis of Bray et al. [2019b] to illustrate that the grocery supply chain suffers runs on inventory when the upstream supplier looks like it may stock out. This lab will comprise a lot of basic joins, as the data initially come in six raw tables. The third lab, in section 4.5, repeats the analysis of Astashkina et al. [2023] to show that home cleaners do not tend to “disintermediate” from the gig-economy platform that links them up with customers. This lab will require a more advanced mutating join.

Finally, I provide several comments in section 4.6. In section 4.6.1 I argue that you should use multi-column operations sparingly, because it’s usually simpler to pivot_longer() the tibble so that all the relevant data are in a single column. In section 4.6.2 I warn against one of the most common sources of bugs: joins that unintentionally duplicate rows. In section 4.6.3 I explain that joins allow better data base design, as they enable us to save our data across a collection of tibbles, each focusing on a different aspect of the problem, rather than lumping everything in one sprawling table. In section 4.6.4 I argue that you shouldn’t be shy about reiterating your group_by() groupings. In section 4.6.5 I illustrate a new technique that enables us to treat tibble rows as vectors of data, like we usually do for tibble columns.

For the lecture, you can work in the code.R file of the wrangle project you created in exercise 1.31, and for the labs you can work in the code.R files of the rationing project you created in exercise 1.32 and the disintermediation project you created in ??.

### 4.2 Lecture

#### 4.2.1 Data

In exercise 1.31 of section 1.3 you created three tibbles: `oversight_quantity`, `oversight_price`, and `chess_panel`. The former two describe a pricing experiment, which I will discuss in section 4.3, and the final tibble reports the following 14 variables for 14,322 chess games played on lichess.org:

- **game**: Distinct game identification number.
- **rated**: Logical indicating whether the outcome of the game influences the players’ ratings.
- **moves**: The games’ moves in chess notation. For example, `game = "msIqfn6q"` has `moves = "e3 e5 Bc4 Qf6 Qf3 Qg6 Qd5 Nh6"`, which indicates that white first moved a

\(^2\)For more on joins see vignette("two-table").
pawn to square e3, then black moved a pawn to square e5, then white moved a bishop to c4, then black moved its queen to f6, etc.

- **turns**: Number of moves the game comprises. For example, *turns = 20* indicates that the game ended after both players moved ten pieces.
- **victory_status**: Game outcome, either "draw", "mate", "outoftime", or "resign".
- **winner**: Winner of the game, either "white", "black", or "draw".
- **increment_code**: The amount of time allotted to each player. Two clocks keep track of the two players’ remaining time. The white player’s turn to move and the black clock runs down when it’s the black player’s turn to move. If one player runs out of time, the other player wins. The *increment_code* values have the form *x_y*, where *x* is the number of minutes initially put on each clock and *y* is the number of seconds added to a player’s clock after they make a move. For example, in a game with *increment_code = "15_0"* the players both have a 15-minute budget to make all their moves, whereas in a game with *increment_code = "15_2"* the players both start with 15 minutes and are given an extra 2 seconds after each move.
- **white_id, black_id**: Distinct player identification numbers.
- **white_rating, black_rating**: Player ratings. Better players have higher scores.
- **opening_eco**: Classification of the game’s opening line—i.e., sequence of beginning moves—as cataloged by the *Encyclopaedia of Chess Openings*. For example, opening_eco codes A80–A99 correspond to the Dutch Defense and codes B20–B99 correspond to the Sicilian Defense.
- **opening_name**: Detailed name for the game’s opening line. For example, the games with *opening_name = "Alekhine Defense: Scandinavian Variation"* all have moves values that start with "e4 Nf6 d3".
- **opening_ply**: Number of moves that comprise the game’s opening line. This is roughly the number of moves until the players deviate from the traditional script.

### 4.2.2 Tidy Select

Stuffing all the moves in *chess_panel* into a single *moves* column makes them difficult to analyze. So, we will create a new tibble that assigns a distinct column to each of the first 200 moves:

```r
first_200_moves <-
  chess_panel %>%
  select(game, moves) %>%
  separate(
    moves,
    str_c("move_", 1:200),
    sep = " ",
  )
```

See section 2.4.3 for more on the *separate()* function.

---

3See section 2.4.3 for more on the *separate()* function.
first_200_moves %>%
  select(1:9) %>%
  head

> # A tibble: 4 x 9
>   game  move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
>   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 l1NXvwaE d4  Nc6  e4  e5  f4  f6  dxe5  fxe5
> 2 mIICvQHh e4  e5  d3  d6  Be3  c6  Be2  b5
> 3 kWKvrqYL d4  d5  Nf3  Bf5  Nc3  Nf6  Nc6  b4  Ng4
> 4 9tXo1AUZ e4  e5  Nf3  d6  d4  Nc6  d5  Nb4

For example, game = "iyPuQJxx" has turns = 15 and thus has character strings in move_1–move_15 and NA terms in move_16–move_200. Likewise define

first_8_moves <-
  first_200_moves %>%
  select(1:9)

first_8_moves

> # A tibble: 14,322 x 9
>   game  move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
>   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 l1NXxvaE d4  Nc6  e4  e5  f4  f6  dxe5  fxe5
> 2 mIICvQHh e4  e5  d3  d6  Be3  c6  Be2  b5
> 3 kWKvrqYL d4  d5  Nf3  Bf5  Nc3  Nf6  Nc6  b4  Ng4
> 4 9tXo1AUZ e4  e5  Nf3  d6  d4  Nc6  d5  Nb4
> 5 qwU9rasv d4  d5  e4  dxe4  Nc3  Nf6  f3  exf3
> 6 dwF3DJHO e4  e5  Bc4  Nc6  Nf3  Nd4  d3  Nxf3+
> 7 afoMwnLg e4  d5  exd5  Qxd5  Nc3  Qe5+  Be2  Na6
> 8 HgKLPsz e3  e6  d4  d6  Bd3  c6  Nf3  Be7
> 9 2fEjSei6 e4  e6  Qh5  g6  Qe5  Nf6  d4  d6
>10 u7i6d0aJ e4  e5  Nf3  Nc6  Bc4  Nf6  Nc3  Bc5
> # ... with 14,312 more rows

We can easily `select()` the white moves of `first_8_moves` (as the first mover, White makes all the odd moves):
first_8_moves %>%
  select(move_1, move_3, move_5, move_7)

> # A tibble: 14,322 x 4
>     move_1 move_3 move_5 move_7
>       <chr> <chr> <chr> <chr>
> 1     d4    e4    f4  dxe5
> 2     e4    d3   Be3   Be2
> 3     d4   Nf3   Nc3   Bf4
> 4     e4   Nf3    d4    d5
> 5     d4    e4   Nc3    f3
> 6     e4   Bc4   Nf3    d3
> 7     e4  exd5   Nc3   Be2
> 8     e3    d4   Bd3   Nf3
> 9     e4   Qh5   Qe5    d4
>10     e4   Nf3   Bc4   Nc3
> # ... with 14,312 more rows

But doing the same for first_200_moves would require listing 100 distinct columns:

first_200_moves %>%
  select(
    move_1, move_3, move_5,
    move_7, move_9, move_11, move_13, move_15, move_17, move_19
  )

A smarter solution is to select() all the variables that end in a "1", "3", "5", "7", or "9":

first_200_moves %>%
  select(ends_with(c("1", "3", "5", "7", "9")))

This solution uses the “tidy select” scheme for specifying variables. The tidy select conventions give you tremendous column-picking flexibility. For example,

- select(move_1:move_10) picks the first ten moves,
• `select(-(move_1:move_10))` picks all but the first ten moves,
• `select(move_1:move_10, move_191:move_200)` picks the first and last ten moves,
• `select(last_col(24):last_col())` picks the last 25 columns,
• `select(everything())` picks all columns,
• `select(contains("20"))` picks move_20, move_120, and move_200,
• `select(starts_with("move_9"))` picks move_9 and move_90–move_99,
• `select(ends_with(c("29", "44")))` picks move_29, move_129, move_44, and move_144,
• `select(where(is.character))` picks the character variables, or
• `select(where(.x %> str_detect("Nxf6") %>% any(na.rm=TRUE)))` picks the columns that contain the string "Nxf6".

The last two examples use `where()` to assign TRUE or FALSE to each column and then we `select()` the TRUE columns. We can derive the logicals from a function, such as `is.character`, or from a functional assembly line, such as `~.x %>% str_detect("Nxf6") %>% any(na.rm=TRUE)`. Starting the assembly line with `~.x` tells R to apply it to every tibble column. Hence, the last example makes R evaluate

• `game %>% str_detect("Nxf6") %>% any(na.rm=TRUE),`
• `move_1 %>% str_detect("Nxf6") %>% any(na.rm=TRUE),`
• `move_2 %>% str_detect("Nxf6") %>% any(na.rm=TRUE),` etc.

And it keeps the columns for which the result is TRUE.

**Exercise 4.1.** Use `where()` and `is.numeric()` to select the numeric columns of `chess_panel`.

**Exercise 4.2.** Use `&` and `ends_with()` to select all the black moves in `first_200_moves` between the 20th and 60th turns (inclusive). (The black player makes all the even-numbered moves.)

**Exercise 4.3.** Find four different ways to `select()` all but the game column of `first_8_moves`.

The tidy select convention isn’t limited to `select()`. In fact, it’s been rolled out to most tidyverse functions. For example, we can reposition all the numeric columns of `chess_panel` to the end:

```r
chess_panel %>%
  relocate(
    where(is.numeric),
    .after = last_col()
  )
```
And we can pivot `first_8_moves` by its black moves:

```r
first_8_moves %>%
  pivot_longer(
    cols = ends_with(c("0", "2", "4", "6", "8")),
    names_to = "move_number",
    values_to = "black_move"
  )
```

The tidy select protocol is especially powerful when combined with the functional quartet, as we’ll see in the next sections.

### 4.2.3 Multivariable Operations

We can capitalize the moves in `first_8_moves` with:

```r
first_8_moves %>%
  mutate(
    move_1 = str_to_upper(move_1),
    move_2 = str_to_upper(move_2),
    move_3 = str_to_upper(move_3),
    move_4 = str_to_upper(move_4),
    move_5 = str_to_upper(move_5),
    move_6 = str_to_upper(move_6),
    move_7 = str_to_upper(move_7),
    move_8 = str_to_upper(move_8)
  ) %>%
  head
```

```r
> # A tibble: 6 x 9
>   game  move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
>  <chr>  <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 l1NXwvaE D4 NC6 E4 E5 F4 F6 DXE5 FXE5
> 2 mIIcvQHh E4 E5 D3 D6 BE3 C6 BE2 B5
> 3 kWKrqYL D4 D5 NF3 BF5 NC3 NF6 BF4 NG4
> 4 9tXo1AUZ E4 E5 NF3 D6 D4 NC6 D5 NB4
> 5 qw9rasv D4 D5 E4 DXE4 NC3 NF6 F3 EXF3
> 6 dwF3JH0 E4 E5 BC4 NC6 NF3 ND4 D3 NXF3+
```

But extending this solution to `first_200_moves` would be impractical, requiring 200 `str_to_upper()` calls. A better solution is to tidy select the columns to transform:
`first_200_moves`%>%
mutate(
  across(
    starts_with("move"),
    str_to_upper
  )
) %>%
head

```r
# A tibble: 6 x 201
#  game    move_1    move_2    move_3    move_4    move_5    move_6    move_7    move_8    move_9    move_10
#<chr>    <chr>    <chr>    <chr>    <chr>    <chr>    <chr>    <chr>    <chr>    <chr>    <chr>
# 1 1lnXvw- D4 NC6 E4 E5 F4 F6 DXE5 FXE5 FXE5 NXE5
# 2 mLVCvQ- E4 E5 D3 D6 BE3 C6 BE2 B5 ND2 A5
# 3 kWXvrq- D4 D5 NF3 BF5 NC3 NF6 BF4 NG4 E3 NC6
# 4 9txo1A- E4 E5 NF3 D6 D4 NC6 D5 NB4 A3 NA6
# 5 qwUKra- D4 D5 E4 DXE4 NC3 NF6 F3 EXF3 NXF3 NC6
# 6 dwF3DJ- E4 E5 BC4 NC6 NF3 ND4 D3 NXF3+ QXF3 NF6
#... with 190 more variables: move_11 <chr>, move_12 <chr>, move_13 <chr>,...
# # move_14 <chr>, move_15 <chr>, move_16 <chr>, move_17 <chr>, move_18 <chr>,
# # move_19 <chr>, move_20 <chr>, move_21 <chr>, move_22 <chr>, move_23 <chr>,
# # move_24 <chr>, move_25 <chr>, move_26 <chr>, move_27 <chr>, move_28 <chr>,
# # move_29 <chr>, move_30 <chr>, move_31 <chr>, move_32 <chr>, move_33 <chr>,
# # move_34 <chr>, move_35 <chr>, move_36 <chr>, move_37 <chr>, move_38 <chr>,
# # move_39 <chr>, move_40 <chr>, move_41 <chr>, move_42 <chr>, ...
```

The `across()` function applies `str_to_upper()` to every column that `starts_with("move")`. We can likewise use `across()` to tidy select our `group_by()` groupings:

```
#group_by() first three moves:
first_8_moves %>%
group_by(across(move_1:move_3)) %>%
summarise(num_of_fourth_moves = n_distinct(move_4)) %>%
head

# A tibble: 6 x 4
# Groups: move_1, move_2 [3]
#  move_1    move_2    move_3    num_of_fourth_moves
#<chr>    <chr>    <chr>    <int>
# 1 a3      b6      d4         1
# 2 a3      c6      b4         1
```

256
> 3 a3  c6  Nf3  1
> 4 a3  d5  b4  1
> 5 a3  d5  c3  1
> 6 a3  d5  d4  2

#group_by() white_rating and black_rating
chess_panel %>%
group_by(across(ends_with("_rating"))) %>%
summarise(white_win_frac = mean(winner == "white")) %>%
head

> # A tibble: 6 x 3
> # Groups: white_rating [6]
> white_rating black_rating white_win_frac
> <dbl> <dbl> <dbl>
> 1 784 970 1
> 2 793 961 0
> 3 808 973 1
> 4 813 841 1
> 5 842 1076 0
> 6 857 958 0

And we can use across() to summarise() many columns at once:

first_200_moves %>%
summarise(
    across(
        across(
            move_5:move_8,
            n_distinct
        )
    )
)

> # A tibble: 1 x 4
> move_5 move_6 move_7 move_8
> <int> <int> <int> <int>
> 1 154 195 243 282

As you see, across() has two inputs: a tidy selection of columns and a function to apply to these columns. Hence, the code above applies n_distinct() to columns move_5, move_6, move_7 and move_8.
However, note that I write \texttt{n\_distinct} rather than \texttt{n\_distinct()}. This is an important distinction because for a given function, \texttt{fn()}, the expressions \texttt{fn} and \texttt{fn()} evaluate to two very different things: the former references the \textit{function} itself, whereas the latter references the \textit{output of the function}. For example, consider the following:

```r
#the following defines x as the function today()
x <- today

#I can now call x just like I call today()
x()
```

> [1] "2023-02-28"

```r
#indeed, R now considers x a function
class(x)
```

> [1] "function"

```r
#in contrast, the following defines y as the output of today()
y <- today()

#y is now just today's date

y
```

> [1] "2023-02-28"

```r
#indeed, R considers y a Date:
class(y)
```

> [1] "Date"

Accordingly, to pass in functions, rather than their outputs, we always leave off the parentheses from the functions we nest in \texttt{across()}. For example, \texttt{n\_distinct()} returns 0, since the the parentheses contain zero distinct items; accordingly, R would interpret this
first_200_moves %>%
  summarise(
    across(
      move_5:move_8,
      n_distinct()
    )
  )

as this:

first_200_moves %>%
  summarise(
    across(
      move_5:move_8,
      0
    )
  )

which would yield an error, since 0 is not a function. Replacing n_distinct() with n_distinct makes the across() call receive a proper function in its second slot, which fixes this problem.

**Exercise 4.4.** Use summarise() and across() to get the last() values of every column whose name contains() a ".". Your output should be a one-row tibble with column names victory_status, increment_code, white_id, white_rating, black_id, black_rating, opening_eco, opening_name, and opening_ply.

- Embed across() within summarise(), and embed last() within across().
- Drop the parentheses after last().

**Exercise 4.5.** Use mutate(), across(), and where() to apply as.roman() of every column in chess_panel that is.numeric().

And in addition to pre-defined functions like first() and sd(), we can apply custom transformations to our variables with *anonymous functions*. An anonymous functions is a general expression that starts with ~ and uses .x to refer to the inputted argument. For example, the following are equivalent:
# method 1:
first_200_moves %>%
  summarise(
    move_101 = move_101 %>% is.na %>% mean,
    move_102 = move_102 %>% is.na %>% mean
  )

> # A tibble: 1 x 2
> move_101  move_102
>    <dbl>    <dbl>
> 1   0.860   0.865

# method 2:
first_200_moves %>%
  summarise(
    across(
      c(move_101, move_102),
      ~ .x %>% is.na %>% mean # this is an anonymous function
    )
  )

> # A tibble: 1 x 2
> move_101  move_102
>    <dbl>    <dbl>
> 1   0.860   0.865

Here’s another anonymous function example:

first_8_moves %>%
  mutate(
    across(
      ends_with(c("1", "3", "5", "7", "9")),
      ~ str_c("w", .x, sep="_")
    ),
    across(
      ends_with(c("0", "2", "4", "6", "8")),
      ~ str_c("b", .x, sep="_")
    )
  ) %>%
head(4)
The first `across()` call redefines `move_1` to `str_c("w", move_1, sep="_")`, `move_3` to `str_c("w", move_3, sep="_")`, etc., and the second `across()` call redefines `move_2` to `str_c("b", move_2, sep="_")`, `move_4` to `str_c("b", move_4, sep="_")`, etc. As you see the `.x` is a stand in for the selected columns.

**Exercise 4.6.** Chess notation uses "+" to indicate a check—i.e., a threat upon the opponent’s king—and "#" to indicate a checkmate. Use `mutate()` and `across()` to replace in `move_1`–`move_200` all instances of "+" with "_check" and all instance of "#" with "_checkmate".

- Use `.x %>%` to start an assembly line that comprises two `str_replace()` calls.
- "+" has a special meaning when it's interpreted as a regular expression. To tell R not to interpret this symbol as a regular expression, refer to it as "\+" within the `str_replace()` expression.

**Exercise 4.7.** Integers should generally be saved as integers, not doubles. Accordingly, we will now convert the numeric columns of `chess_panel` that do not have decimal points to integers.

- Pipe `chess_panel` into a `mutate()`.
- Within the `mutate()` call, embed an `across()` call.
- Use the first argument of `across()` to pick a column if (i) it `is.numeric()` and (ii) it has no decimal component.
  - Impose the two criteria with two `where()` operations connected by a `&`.
  - Note that `~ all(.x == as.integer(.x), na.rm = TRUE)` is an anonymous function that returns `TRUE` if `all()` of the entries of a numeric column have no decimal part.
- Use the second argument of `across()` to apply `as.integer()`.
  - Write `as.integer()` rather than `as.integer()`.
- Overwrite the old `chess_panel` with the new tibble.

**Exercise 4.8.** Text variables that have few distinct values should generally be saved as factors, not character strings. Accordingly, we will now convert the character string columns of `chess_panel` that have fewer than a dozen distinct values to factors.

- Pipe `chess_panel` into a `mutate()`.
- Within the `mutate()` call, embed an `across()` call that applies `as.factor()` to all variables that (i) are characters and (ii) have no more than 12 distinct values.
  - Use `where(is.character)` to identify the character string columns.
  - Embed a functional assembly line within `where()` to identify the columns with fewer than 12 distinct values. Note that `c("a", "b", "c") %>% n_distinct %>% {. <= 2} = FALSE.`
– Use & to connect the two conditions.
– Write as.factor rather than as.factor().

• Overwrite the old chess_panel with the new tibble.

Exercise 4.9. Rook moves begin with an "R", knight moves an "N", bishop moves a "B", and queen moves begin with a "Q". However, king moves begin either with a "K" or an "O". The latter denotes a “castle” move in which the king jumps two spaces, rather than the usual one. Specifically, under a king-side castle—a move denoted by "O-O"—the white king moves from square e1 to g1 and the black king from e8 to g8, and under a queen-side castle—a move denoted by "O-O-O"—the white king moves from square e1 to c1 and the black king from e8 to c8. Rather than this inconsistent "O-O" and "O-O-O" notation, we’ll express these castle moves with the same piece-location notation used for the other moves. That is, we will change "O-O" to "Kg1" if the king is white and "Kg8" if black, and change "O-O-O" to "Kc1" if the king is white and "Kc8" if black.

• Pipe first_200_moves into a mutate() operation.
• Include two across() steps within the mutate(), the first corresponding to the white moves and the second to the black moves.
  – Recall how we appended "w_" to white moves and "b_" to black moves.
• In the white across() change "O-O-O" to "Kc1" and "O-O" to "Kg1".
  – Recall how we changed "#" to "_checkmate" and "+" to "_check".
  – Change "O-O-O" before "O-O". If you do it the other way around then your first step will change "O-O-O" to "Kg1-O".
• In the black across() change "O-O-O" to "Kc8" and "O-O" to "Kg8".

Whereas across() gives us multi-column versions of , if_all() and if_any() give us multi-column versions of filter():

```r
#Get games whose first four moves are made by knights:
first_8_moves %>%
  filter(
    if_all(
      move_1:move_4, #start with "N". (Knight moves start with "N".)
      -.x %>% str_detect("^N")
    )
  )
```
> # A tibble: 2 x 9
> game move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 47xEFe4o Nc3 Nf6 Nf3 Nc6 e4 e5 Bb5 d6
> 2 6qMK07pP Nf3 Nf6 Nc3 Nc6 e4 e5 Nd5 Nxe4

#Get games with action in the "a" column within in first three moves
first_8_moves %>%
  filter(
    if_any(          #keep if_any() of game's first 3 moves contain "a"
      move_1:move_3,
      ~.x %>% str_detect("a")
    )
  ) %>%
  head(4)

> # A tibble: 4 x 9
> game move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 O3wdIwg7 e4 e5 a3 Nf6 Nc3 d5 exd5 Nxd5
> 2 GstYv2mJ h4 e5 a4 d5 b4 Nc6 b5 Na5
> 3 QnXuc72h e4 a6 Nf3 b6 d4 Bb7 Nc3 h6
> 4 XLvG0Fxw h3 e5 a3 Nf6 e3 d5 Nc3 a6

In the first example, R uses str_detect("^N") to turn vectors move_1, move_2, move_3, and move_4 into four corresponding logical vectors, and then it keeps the rows for which all four vectors are TRUE. In the second example, R uses str_detect("a") to turn vectors move_1, move_2, and move_3 into three corresponding logical vectors, and then it keeps the rows for which any of the three vectors is TRUE.

For another example, the following picks the games in which the first piece was captured in the fortieth move (piece captures are denoted by an "x"):

first_200_moves %>%
  filter(
    str_detect(move_40, "x"),  #capture in move_40
    if_all(                #no captures in move_1-move_39
      move_1:move_39,
      ~ str_detect(.x, "x", negate=TRUE)  #negate option makes function return
                                            #TRUE when it doesn't find "x"
    )
  ) %>%
  select(game)
See section 4.6.5 for a more flexible—but more cumbersome—way of executing multi-column filters.

**Exercise 4.10.** We will now filter() chess_panel to ensure that there are no NA terms in all of the character string columns.

- Embed `where(is.character)` and `-.-.x %>% is.na %>% `!` inside of `if_all()`, and embed that inside of `filter()`.
  - Look what `c(1, NA, 3) %>% is.na %>% `!` yields.

**Exercise 4.11.** Moves that begin with an "O" denote a “castle.” We will now select games in which White castled.

- `filter()` `first_200_moves` to keep a game if _any_ column that `ends_with()` an odd number begins with an "O".
- The regular expression "^O" is shorthand for begins with an "O". So the moves with castles are those for which you can `str_detect("^O")`.

**Exercise 4.12.** Select the games of `first_200_moves` in which Black checkmated with a knight.

- Use regular expression "^N.+#$" to identify moves that begin with an "N" and end in a "#".

**Exercise 4.13.** Select the games of `first_200_moves` in which White never used a bishop and Black never used a rook.

- Include two `if_all()` calls in one `filter()`.
  - `str_detect("^B", negate=TRUE)` and `str_detect("^R", negate=TRUE)` respectively identify the moves that are not made by a bishop and those not made by a rook.

### 4.2.4 Case Study: Pawn Promotions

When pawns make it to the other side of the board—i.e., when white pawns arrive at eighth row or black pawns at the first row—they are “promoted” to any piece of the player’s choosing. Pawn promotions are easy to identify because they are the only moves whose notation incorporates an equals sign. For example, the 62nd move of `game = "jwiDvtUw"` is "b1=Q", which indicates that a black pawn got promoted to a queen after reaching square b1. We will now create the following plot, which illustrates the fraction of moves that promote a pawn, as a function of the move number.
Exercise 4.14. We need at least 200 observations to get a reasonable estimate of the pawn promotion fraction, so use `where()` to `select()` the columns of `first_200_moves` that have at least 200 non-NA terms.

- Use an anonymous function that begins with `. %>% is.na.`
- Note that `c(TRUE, FALSE) %>% `!` = c(FALSE, TRUE)` and `199:201 %>% { . >= 200 } = c(FALSE, TRUE, TRUE)`.

Exercise 4.15. `summarise()` the result of the previous exercise to calculate the fraction values for which you can `str_detect()` an "=" (after removing the NA terms).

- Use an an anonymous function to apply `str_detect("=") %>% mean(na.rm = TRUE)` to all the -game columns.
- The resulting tibble should have one row and 156 columns. For example, `move_80` has 4665 non-NA values, 40 of which contain the symbol "="; hence, the output should have `40/4665 = 0.0086` in its `move_80` column.

Exercise 4.16. `pivot_longer()` the result of the previous exercise.

- Pull down everything().
- Set the `values_to "frac"`.
- Set the `names_to "move_num"`.
- Include options `names_pattern = "move_(.+)", names_transform = list(move_num = as.integer)` to remove the "move_" prefix and save the move number as an integer.
Exercise 4.17. `mutate()` the result of the previous exercise to define `player` as a variable that is "White" in the odd moves and "Black" in the even moves.

- The `%%` symbol represents the remainder function. For example, `1:6 %% 2 = c(1, 0, 1, 0, 1, 0)`.
- Use `ifelse()`.

Exercise 4.18. `ggplot()` `frac` as a function of `move_num` with `geom_point()` and `geom_smooth(se = FALSE, span = 0.25)`. Give each player type a different color.

- Give the axes appropriate `labs()`.
- Tidy the `ggplot` with `theme_minimal()` and `theme(legend.position = "top", legend.title = element_blank())`.

4.2.5 Basic Joins

Limiting yourself to a single table of data is like limiting yourself to a single pair of shoes—you can do it, but not gracefully or comfortably. However, if we don’t pool all our data into one big, catch-all table then we’re going to need a way to cross-reference information stored in different tibbles. Our tool for combining data across tibbles is the join, which merges two tibbles into one.

Let me illustrate with an example. The following tibble records the number of moves until the first piece is captured (i.e., until there’s a move with an "x" in it):

```r
time_to_first_kill <-
  first_200_moves %>%
  pivot_longer(
    -game,
    names_to = "move_number",
    names_pattern = "move_(.+)",
    names_transform = list(move_number = as.integer)
  ) %>%
  filter(str_detect(value, "x")) %>%
  group_by(game) %>%
  summarise(first_kill_move = min(move_number))

time_to_first_kill %>% head

> # A tibble: 6 x 2
>   game first_kill_move
>   <chr>         <int>
> 1 005lW0Xz         6
```
For example, game = "005lW0Xz" has first_kill_move = 6, which indicates that the first piece was captured in the sixth move. We can add first_kill_move to chess_panel by merging this tibble with time_to_first_kill:

```
chess_panel %>%
  inner_join(time_to_first_kill) %>%
glimpse
```

> Joining, by = "game"

> Rows: 14,242
> Columns: 15
> $ game <chr> "l1NXvwaE", "mIIcVQhh", "KWKvrqYL", "9tXo1AUZ", "qwU9r~
> $ rated <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE-
> $ moves <chr> "d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 fxe5 Nxe5 Qd4 Nc6 Qe5+ N-
> $ turns <dbl> 16, 61, 61, 95, 33, 66, 119, 39, 31, 31, 43, 52, 66, 1-
> $ victory_status <chr> "resign", "mate", "mate", "mate", "resign", "resign", ~
> $ winner <chr> "black", "white", "white", "white", "black", "black", ~
> $ increment_code <chr> "5+10", "5+10", "20+0", "30+3", "10+0", "15+0", "10+0"-
> $ white_id <chr> "a-00", "ischia", "daniamurashov", "nik221107", "capa_~
> $ white_rating <dbl> 1322, 1496, 1439, 1523, 1520, 1439, 1381, 1381, 1381, ~
> $ black_id <chr> "skinnerua", "a-00", "adivanov2009", "adivanov2009", "~
> $ black_rating <dbl> 1261, 1500, 1454, 1469, 1423, 1392, 1209, 1272, 1607, ~
> $ opening_eco <chr> "B00", "C20", "D02", "C41", "D00", "C50", "B01", "A00"~
> $ opening_name <chr> "Nimzowitsch Defense: Kennedy Variation", "King's Pawn~
> $ opening_ply <dbl> 4, 3, 3, 5, 10, 6, 4, 1, 2, 8, 7, 8, 5, 4, 4, 4, 4, ~
> $ first_kill_move <int> 7, 13, 18, 19, 4, 8, 3, 15, 11, 17, 10, 21, 17, 8, 17,~

The 'Joining, by = "game" message at the top indicates that the inner_join() operation merges chess_panel with time_to_first_kill by their one common variable, game. In other words, it matches up all rows in chess_panel with all rows in time_to_first_kill that share the same game value, and then it pastes together these matched rows, with the chess_panel values on the left and the time_to_first_kill values on the right. Accordingly, for this merge we call chess_panel the left-hand tibble (LHT) tibble and time_to_first_kill the right-hand tibble (RHT).

Unfortunately, this inner_join() deletes 80 rows of data: chess_panel has 14,322 rows but the merged tibble has only 14,242 rows. What's happening is that there are 80 games
that do not have any no piece captures, and thus do not appear in time_to_first_kill. Accordingly, there are 80 rows in the LHT that have no counterparts in RHT. And since inner_join() outputs only successfully merged rows— with elements from both the LHT and the RHT—the unmatched observations of chess_panel get dropped. However, we can keep these unmatched rows if we use left_join() instead of inner_join(). For example, the following merged tibble has 14,322 rows, as desired:

```r
chess_panel %>%
  left_join(time_to_first_kill) %>%
dim
```

> Joining, by = "game"

> [1] 14322 15

By design, left_join() keeps all the LHT intact: it matches the tibbles as best as possible, and then throws in the unmatched rows from the LHT, populating the missing right-hand values with NA terms. For example, the game = "A7ywZFZL" row of the merged tibble has first_kill_move = NA because this game does not appear in time_to_first_kill. In addition to left_join() there is also right_join(), which keeps all the unmatched elements of the RHT, and full_join(), which keeps the unmatched elements from both tibbles. However, there are very few circumstances where you would need to use full_join(), and it’s generally deemed uncouth to use right_join(). Instead, the received etiquette is to make whatever tibble you want to preserve the LHT, so that you can stick to left_join().

**Exercise 4.19.** Use inner_join() to copy move_1–move_8 from first_8_moves to chess_panel.

**Exercise 4.20.** Use inner_join() to copy the white moves from first_200_moves to chess_panel.

- Nest the select() operation within the inner_join().

**Exercise 4.21.** We will now investigate the relationship between turns in chess_panel and first_kill_move in time_to_first_kill. Specifically, we will divide chess_panel into four equal subsamples, based on the quartiles of first_kill_move, and then plot the distribution of turns in each subsample.

- Pipe chess_panel into inner_join().
- Within the inner_join(), create a tibble that maps each game to a kill_bucket value, which is 1 if first_kill_move is below the first quartile, is 2 if first_kill_move is between the first and second quartiles, etc.
  - Look what 1:100 %>% ntile(4) yields.
– Save kill_bucket as.factor().

• After the join, ggplot() the geom_density() of turns, and color by kill_bucket.

Exercise 4.22. We will now calculate the number of distinct move_1:move_8 values associated with each opening_eco.

• Start with first_8_moves and inner_join() it with chess_panel %>% select(game, opening_eco).
• group_by() and summarise() the result to calculate n_distinct(across(move_1:move_8)) for each opening_eco.

Exercise 4.23. Use inner_join() to restrict first_8_moves to the games with increment_code == "30+3". Your join should not increase the number of columns in first_8_moves.

• Pipe first_8_moves into inner_join().
• Within the inner_join() parentheses, insert a functional assembly line that extracts from chess_panel the game values for which increment_code == "30+3".

Exercise 4.24. We will now create a geom_bar() plot that illustrates the distribution of first_kill_move, by victory_status.

• inner_join() chess_panel and time_to_first_kill.
• Pipe the combined tibble into ggplot().
• Specify a geom_bar() plot with x = first_kill_move and y = ..prop... This latter option tells R to make a “proportional plot,” whose columns sum to one.
• facet_wrap by victory_status.

4.2.6 Mutating and Filtering Joins

We can coerce left_join() to behave like mutate(). For example, suppose we wanted a variable called games_white_won_as_white that denotes the number of games that the current white_id player won as White. We can calculate this with mutate() or with a “mutating join”:

```r
#mutate():
chess_panel %>%
  group_by(white_id) %>%
  mutate(
    games_white_won_as_white =
      sum(winner == "white")
  )
```
#mutating join:
chess_panel %>%
  left_join(
    group_by(., white_id) %>%
      summarise(
        games_white_won_as_white =
          sum(winner == "white")
      ),
    by = c("white_id" = "black_id")
  ) %>%
  replace_na(list(games_white_won_as_black = 0))

In this case the former solution is preferable. But now suppose I wanted to add a variable called games_white_won_as_black that denotes the number of games that the current white_id player won as Black. I couldn’t construct this variable with mutate() because the information required to define it resides in a group_by() group that does not contain the current observation. For example, game = "epDNgzio" has white_id = "yokocito" and black_id = "luis77"; thus it belongs to the black_id = "luis77" group, but its games_white_won_as_black value depends on information the black_id = "yokocito" group.

Nevertheless, we can define games_white_won_as_black a mutating join:

chess_panel %>%
  left_join(
    filter(., winner == "black") %>%
      count(
        black_id,
        name = "games_white_won_as_black"
      ),
    by = c("white_id" = "black_id")
  ) %>%
  replace_na(list(games_white_won_as_black = 0))

Let me explain: First, a . inside a function’s parentheses always refers to the data that got piped into the function (see section 2.2.6). Hence, R interprets filter(., winner == "black") as filter(chess_panel, winner == "black"). Second, the by = c("white_id" = "black_id") option tells the left_join() to match the white_id column of the LHT with the black_id column of the RHT. For example, it merges the white_id = "yokocito" rows of the LHT with the black_id = "yokocito" row of the RHT (which is what we wanted). Third, the replace_na() step changes the games_white_won_as_black = NA values to zero: since the RHT has no record of the players that didn’t win any games in the black position, the left_join() sets their games_white_won_as_black values to NA.

In general, mutating joins are more powerful than mutate(): a mutating join can do anything that mutate() can do, but not vice versa. But the difference isn’t that big anymore, so they don’t come up all that often.
Exercise 4.25. The following code computes the number of opponents a given player faced. For example, "-l-_jedi_knight_-l-" faced four distinct opponents—playing against "erik123678" and "kayvan0098" as White and against "kayvan0098", "anton2406", and "v_clava38" as Black—and thus has opponents_faced = 4. Embed this code in a `left_join()` to create a mutating join that adds a column to `chess_panel` called `white_opponent_count` that records the number of distinct opponents faced by White.

```
summarise(
  chess_panel,
  player = c(white_id, black_id),
  opponent = c(black_id, white_id)
)%>%
group_by(player)%>%
summarise(opponents_faced = n_distinct(opponent))%>%
head
```

> # A tibble: 6 x 2
> player opponents_faced
> <chr> <int>
> 1 -l-_jedi_knight_-l- 4
> 2 -pavel- 1
> 3 1063314 1
> 4 1111112222 3
> 5 1240100948 32
> 6 1714-2014 1

Exercise 4.26. We will now study how the number of captured pieces varies with the number of turns. But rather than the “raw” captured count, we will use the z-score version of this variable, normalizing it so it has mean zero and standard deviation one across all `opening_name` values. In other words, we will subtract by the captured count mean and then divide by the captured count standard deviation (separately for each `opening_name` grouping). But there’s a catch: we want to exclude outliers when calculating these means and standard deviations, but not when making our plots. So we will calculate these statistics “offline” and join their values back into our original tibble.

- Start with `chess_panel` and `group_by()` `opening_name`.
- Use `n()` to remove the `opening_name` values with fewer than 30 corresponding games.
- Add a column to `chess_panel` called `captured` that specifies the number of pieces captured in the given game.
  - Use `str_count(moves, "x")`.
- Pipe the result into `left_join()`. And within this function do the following:
− Use `.` to refer to the tibble that got piped into the `inner_join()` and `group_by()` `opening_name`.
− `filter()` out the outlier values by imposing `row_number(captured) > 3` and `row_number(-captured) > 3`.
− Use `summarise()` to define `captured_mean` and `captured_sd` as the mean and standard deviation of the remaining captured values.

• The `left_join()` should add `captured_mean` and `captured_sd` to the larger sample.

− By nesting them in a `left_join()`, we have not subjected our larger sample to the `row_number(captured) > 3` and `row_number(-captured) > 3` restrictions.

• Use `mutate()` to define `captured_z = (captured - captured_mean) / captured_sd`.
• Scatter plot `captured_z` by `turns`.

− Use the `size = .1` option of `geom_point()`.

**Exercise 4.27.** Suppose that in preparation for an upcoming match against the player "ivanbus", you wanted to add a new column to `chess_panel` called `ivan_turns` that is NA if "ivanbus" never played that `opening_eco`, and is otherwise equal to the `median()` number of turns among the games that "ivanbus" played with that `opening_eco`. For example, the three games that "ivanbus" played with `opening_eco = "C45"` have turns 39, 46, and 50; so all games in our final panel should have `ivan_turns = median(c(39, 46, 50)) = 46`. Your join should not decrease the number of rows in `chess_panel`.

• Pipe `chess_panel` into `left_join()`.

− We use `left_join()` because we don’t want to delete a game just because "ivanbus" didn’t play the given `opening_eco`.

• Within the `left_join()`, include a functional assembly line that derives from `chess_panel` the `median()` number of turns, by `opening_eco` for games with `white_id == "ivanbus"` or `black_id == "ivanbus"`.

− The output of this assembly line should be a tibble with columns `opening_eco` and `ivan_turns`. The `left_join()` will match this tibble with `chess_panel` by `opening_eco`.

In addition to the mutating join, there are also the filtering join. Just as the mutating join is a more potent version of `mutate()` the filtering join is a more potent version of `filter()`. We use a filtering join when the decision about whether to drop a row depends on information that does not neatly reside in one of the row’s `group_by()` groupings. For example, we can use a standard `filter()` to restrict the sample to the `opening_eco` values that have at least 10 distinct `opening_name` values:
chess_panel %>%
  group_by(opening_eco) %>%
  filter(n_distinct(opening_name) >= 10)

But now suppose we want to restrict the sample to the opening_eco values that have at least 10 distinct opening_name values among its rated = TRUE games. This problem is harder because the rated = FALSE games need to access information about the rated = TRUE games. The simplest solution is a filtering join:

```r
eco_values_to_keep <-
  chess_panel %>%
  filter(rated) %>%
  group_by(opening_eco) %>%
  summarise(n = n_distinct(opening_name)) %>%
  filter(n >= 10) %>%
  select(opening_eco)

chess_panel %>%
  inner_join(eco_values_to_keep)
```

This code saves the opening_eco values that meet the criterion in wheelbarrow tibble eco_values_to_keep, and it then takes the inner_join() of this tibble and chess_panel, matching by their one common variable, opening_eco. It is this last step that filters out the undesirable games from chess_panel. Indeed, this inner_join() acts like a filter() step, dropping the games in chess_panel whose opening_eco does not appear in eco_values_to_keep. In other words, eco_values_to_keep contains a list of permissible opening_eco values and the inner_join() step removes the games whose opening_eco is not “on the list.” In fact, this step does nothing other than filter the sample: it doesn’t merge in any new columns because the only variable in the eco_values_to_keep tibble, opening_eco, already exists in chess_panel.

Here’s an even more streamlined solution:

```r
chess_panel %>%
  semi_join(
    filter(., rated) %>%
    group_by(opening_eco) %>%
    summarise(n = n_distinct(opening_name)) %>%
    filter(n >= 10)
  )
```
First, I nested the functional assembly line inside the join to avoid constructing wheelbarrow variable `eco_values_to_keep`. Second, I replaced `inner_join()` with `semi_join()` and removed the `select()` step. `semi_join()` is like `inner_join()`, except it doesn’t copy over the RHT’s columns. Instead, it uses the RHT simply to determine which rows of the LHT to keep. Hence, we don’t have to `select()` away the temporary `n` column in this case.

In essence, `semi_join()` was designed for filtering joins and `left_join()` for mutating joins: the former can remove rows but not add columns (like `filter()`) and the latter can add columns but not remove rows (like `mutate()`). And `inner_join()` is like a mix between the two, as it can both remove rows and add columns. For example, define:

```r
mini_1 <-
  tibble(
    x = c("a", "a", "b", "b", "c"),
    y = 1:5
  )
mini_1
```

```
# A tibble: 5 x 2
x y
<chr> <int>
1 a 1
2 a 2
3 b 3
4 b 4
5 c 5
```

```r
mini_2 <-
  tibble(
    x = c("a", "c"),
    z = c("X", "Y")
  )
mini_2
```

```
# A tibble: 2 x 2
x z
<chr> <chr>
1 a X
2 c Y
```

Now we can filter out the `x = "b"` rows from `mini_1` with a `semi_join()`:
We can add the `z` column to `mini_1` with a `left_join()`:

```
mini_1 %>%
  left_join(mini_2)
```

> # A tibble: 5 x 3
>   x     y     z
>  <chr> <int> <chr>
> 1 a     1     X
> 2 a     2     X
> 3 b     3 <NA>
> 4 b     4 <NA>
> 5 c     5     Y

And we can both filter out the `x = "b"` rows and add in the `z` column with an `inner_join()`:

```
mini_1 %>%
  inner_join(mini_2)
```

> # A tibble: 3 x 3
>   x     y     z
>  <chr> <int> <chr>
> 1 a     1     X
> 2 a     2     X
> 3 c     5     Y

**Exercise 4.28.** Pull() the `chess_panel` moves for the game with the largest `first_kill_move` in `time_to_first_kill`.

- Pipe `chess_panel` into `inner_join()`.
• Within the `inner_join()` derive a one-row tibble that corresponds to the game with largest `first_kill_move`.
• `pull()` moves from the joined tibble.

Now there’s one more important join I want to teach you: `anti_join()`. This join is like the evil twin of `semi_join()`. Whereas `semi_join()` returns the rows in the LHT that correspond to rows in the RHT, `anti_join()` returns the rows in the LHT that do not correspond to rows in the RHT. Thus, `anti_join()` treats the RHT as a “hit list” that specifies the rows in the LHT to knock off. For example:

```
mini_1 %>%
anti_join(mini_2)
```

```
# A tibble: 2 x 2
  x  y
  <chr> <int>
1 b    3
2 b    4
```

`anti_join()`, like `semi_join()`, is designed for filtering joins, as the following exercise illustrates.

**Exercise 4.29.** There are 20 possible opening moves in chess. We will now filter out the games with the least popular opening move.

• Pipe `chess_panel` into `anti_join()`.
• Within the `anti_join()` call, write a functional assembly line that selects the `game` values associated with the least common opening move.
  – You can reference the opening move with `word(moves, 1)`.
  – Use `add_count()`, rather than `count()`, so that you don’t lose the `game` values (which you’ll need to match on).
  – End the assembly line with `select(game)`. This final step speeds up the code, but it doesn’t change the answer. Without this step, both the LHT and RHT would contain all 14 columns of `chess_panel`, from `game` to `opening_ply`. And `inner_join()` would match on all of them, ensuring that the left-hand `game` equaled the right-hand `game`, the left-hand `rated` equaled the right-hand `rated`, the left-hand `turns` equaled the right-hand `turns`, and so on. But if the left-hand `game` equals the right-hand `game` then all the other variables would likewise match up, since both stem from the same `chess_panel` tibble. Thus, to avoid these needless comparisons, we use `select()` to remove all but the `game` column of the RHT.
Exercise 4.30. We want to remove games in which White faces their highest-ranked black opponent or Black faces their highest-ranked white opponent. Which of the two options does this?

**Option 1**
```
# Option 1
chess_panel %>%
  group_by(white_id) %>%
  filter(black_rating < max(black_rating)) %>%
  group_by(black_id) %>%
  filter(white_rating < max(white_rating))
```

**Option 2**
```
# Option 2
chess_panel %>%
  group_by(white_id) %>%
  filter(black_rating < max(black_rating)) %>%
  semi_join(
    chess_panel %>%
      group_by(black_id) %>%
      filter(white_rating < max(white_rating)) %>%
      select(game)
  )
```

Exercise 4.31. Select all the games that share an opening_name with at least one of the games played by "wall-of-shields".

- Pipe `chess_panel` into `inner_join()`.
- Within the `inner_join()`, `filter()` out the games not played by "wall-of-shields" (as either White or Black) and select the `distinct()` `opening_name` values.

4.2.7 Case Study: Network Analysis

Joins are especially useful for analyzing networks. To illustrate, we will now calculate all the chess players that are within six degrees of separation from the player "wall-of-shields".

- A player has zero degrees of separation from themselves only.
- A player has one degree of separation from the players who play against zero-degree-of-separation players, but who are not themselves zero-degree-of-separation players.
- A player has two degrees of separation from the players who play against one-degree-of-separation players, but who are not themselves zero- or one-degree-of-separation players, etc.

For example, the only opponent "joydivisions" faced is "alkeal", who in turn played "dr_perdylo" and "jordanson". Hence, "joydivisions" has zero degrees of separation from
"joydivisions", one degree of separation from "alkeal", and two degrees of separation from "dr_perdylo" and "jordanson".

**Exercise 4.32.** How could we derive from chess_panel the players that are within six degrees of separation from "wall-of-shields"? Think about this problem at a high level before doing any coding. Write a short paragraph that sketches out your solution strategy.

OK, here’s my solution (don’t read this until you’ve mulled the problem over for yourself):

- Create a tibble called `player_graph` that records all the players that play each other. Call its columns `player_from` and `player_to`, so that a `player_from` can reach a corresponding `player_to` in “one hop.” `player_graph` reports the players that are within one degree of separation from "wall-of-shields".
- Create two copies of `player_graph`. Rename the first tibble’s columns to `player_from` and `player_intermediary` and rename the second tibble’s columns to `player_intermediary` and `player_to`. Renaming the columns doesn’t change the player graph, so the first tibble’s `player_from` can reach a corresponding `player_intermediary` in one hop, and the second tibble’s `player_intermediary` can reach a corresponding `player_to` in one hop.
- Join these two tibbles by `player_intermediary` to create a tibble with columns `player_from`, `player_intermediary`, and `player_to`. Since `player_from` can reach `player_intermediary` in one hop and `player_intermediary` can reach `player_to` in one hop, it follows that `player_from` can reach `player_to` within two hops. Accordingly, this tibble reports the players that are within two degrees of separation from "wall-of-shields".
- Repeat this process five times to calculate who a given `player_from` can access in six hops.

We will now implement this solution.

**Exercise 4.33.** `pivot_longer()` the game, white_id, and black_id columns of chess_panel into a tibble called `game_player` that lists the players associated with each game.

- Use the `names_to = NULL` option of `pivot_longer()` to disregard the words "white_id" and "black_id".
- The `head()` of `game_player` should be

```r
> # A tibble: 6 x 2
>   game     player
>   <chr>    <chr>
> 1 l1NXvwaE a-00
> 2 l1NXvwaE skinnerua
```
Exercise 4.34. `inner_join()` two copies of `game_player` together by = "game", with the suffix = `c("_from", "_to")` option. The result of this join should be a tibble with columns `game`, `player_from`, and `player_to`. Save the `distinct(player_from, player_to)` values as `player_graph`. This tibble defines the edges of our player network graph: i.e., two players are connected if and only if they share a row in this tibble.

- The `head()` of `player_graph` should be

```r
> # A tibble: 6 x 2
> player_from player_to
> <chr>   <chr>
> 1 a-00   a-00
> 2 a-00   skinnerua
> 3 skinnerua a-00
> 4 skinnerua skinnerua
> 5 ischia ischia
> 6 ischia a-00
```

`player_graph` identifies who’s “connected” to whom. By definition, a player is connected to themselves and to all the players they competed against (i.e., to the players that are zero or one degree of separation away). In other words, we can get to `player_to` from `player_from` within one hop. For example, the following establishes that "kimmmmi060" and "moomoo123" are the only players within one hop from "moomoo123":

```r
player_graph %>%
  filter(player_from == "moomoo123")
```

```r
# A tibble: 2 x 2
player_from player_to
<chr>       <chr>
1 moomoo123 kimmni060
2 moomoo123 moomoo123
```

Now let’s calculate the players that are within two hops of "moomoo123". To do so, we’ll use two copies of `player_graph`: the first to hop from "moomoo123" to an intermediary player—either "kimmmmi060" or "moomoo123"—and the second to hop from this intermediary player to a new `player_to`:

```r
```
For example, the second line indicates that "moomoo123" can jump to "francooooooo" by way of "kimmmmi060". Note, setting "player_intermediary" = "player_from" in the inner_join() makes the second hop to start from player_intermediary (i.e., the old player_to).

The following exercise will use this approach to create a tibble that records all the players that a given player_from can access within two hops.

**Exercise 4.35.** player_graph lists all the player relationships with zero or one degree of separation. We will now create an analogous tibble that lists all player relationships with zero, one, or two degrees of separation.

- rename() the player_to column of player_graph to player_intermediary.
- inner_join() the result by = c("player_intermediary", "player_from").
- Select the distinct(player_from, player_to) rows, and save the result as player_graph_degree_2.
- The head() of player_graph_degree_2 should be
We can use `player_graph_degree_2` to look up the players that "moomoo123" can access in two steps:

```r
player_graph_degree_2 %>%
  filter(player_from == "moomoo123")
```

> # A tibble: 6 x 2
>   player_from player_to
>   <chr>      <chr>
> 1 moomoo123 kimmmmi060
> 2 moomoo123 francooooooo
> 3 moomoo123 moomoo123
> 4 moomoo123 aerdna87
> 5 moomoo123 jirgu
> 6 moomoo123 always100fc

Now let’s calculate the players that "moomoo123" can access in three steps. To do so, we’ll `rename()` the `player_to` column above to `player_intermediary` and `inner_join()` in another `player_graph` to hop from these `player_intermediary` values to new `player_to` values:

```r
player_graph_degree_2 %>%
  filter(player_from == "moomoo123") %>%
  rename(player_intermediary = player_to) %>%
  inner_join(
    player_graph,
    by = c("player_intermediary" = "player_from")
  )
```

> # A tibble: 17 x 3
>   player_from player_intermediary player_to
>   <chr>      <chr>      <chr>
> 1 moomoo123 kimmmmi060 kimmmmi060
> 2 moomoo123 kimmmmi060 francooooooo
> 3 moomoo123 kimmmmi060 moomoo123
> 4 moomoo123 kimmmmi060 aerdna87
> 5 moomoo123 kimmmmi060 jirgu
For example, the twelfth row indicates that "moomoo123" can reach "ivanbus" in three hops: two hops to "aerdna87" and one hop from there.

The following exercise will use this approach to create a tibble that records all the players that a given player_from can access within three hops.

**Exercise 4.36.** Apply the previous exercise's `rename()`, `inner_join()`, and `distinct()` operations to `player_graph_degree_2` to create a tibble called `player_graph_degree_3` that lists all player relationships with zero, one, two, or three degrees of separation. This tibble should have 467566 rows and two columns: `player_from` and `player_to`.

**Exercise 4.37.** In general we can get the degree-\(n\) relationships by applying the `rename()`, `inner_join()`, and `distinct()` operations to the degree \(n-1\) relationships. To avoid copy and pasting these three functions over and over, compress them into a stand-alone function called `increase_separation()`.

- Use the . <- `%>%` convention from section 2.2.5.
- `player_graph %>% increase_separation` should yield `player_graph_degree_2` and `player_graph %>% increase_separation %>% increase_separation` should yield `player_graph_degree_3`.

**Exercise 4.38.** Combine five `increase_separation()` calls in an assembly line to find the players that are within six degrees of separation from "wall-of-shields".

- It’s a lot faster to impose the `player_from == "wall-of-shields"` condition *before* applying the `increase_separation()` steps.
- You should get 656 players.

**Exercise 4.39.** Calculate the players that are exactly six degrees of separation from "wall-of-shields". That, is identify the players that are within six degrees of separation from "wall-of-shields", but not within five degrees of separation.

- End the assembly line with `anti_join(increase_separation(.), .)`.

282
4.3 Lab: Pricing Experiment

4.3.1 Overview

We will now study a pricing experiment conducted by Eversight\(^4\), a company specializing in “AI-powered experimentation.” Specifically, Eversight runs price experiments at retail stores to estimate customer price elasticities. They have developed the infrastructure necessary to convert brick-and-mortar shops into laboratories capable of running continuous A-B price experiments.

One of the key insights from Eversight’s experiments is that demand estimates based on non-experimental data tend to over-estimate customer sensitivity to price changes because promotions drive most price changes, and promoted products are usually flagged with yellow stickers or advertized as end-cap displays. Hence, price change effects are generally amplified by coincident marketing effects, which inflate the apparent responsiveness of customers to price changes. Accordingly, companies should not use price changes that are highlighted with yellow stickers to anticipate how customers would respond to price changes that are not highlighted with yellow stickers. Unfortunately, this “yellow sticker bias” corrupts most passively collected retail data, which motivates Eversight’s active experimental approach.

You can see the yellow sticker bias in figure 4.1, which we will create in lab. The plot illustrates the outcome of 86,043 price changes. The left-most panel corresponds to traditional price changes, the middle two panels correspond to the price changes that start and end promotions, and the right-most panel corresponds to Eversight’s experimental price changes. The x-axis denotes the magnitude of the price change and the y-axis denotes the magnitude of the sales-rate change. As you see, the regression slope is significantly less steep with the test price changes than with the baseline or promotional price changes. This illustrates that we overestimate the negative relationship between price and sales when we use non-experimental data.

4.3.2 Data

We will use two tibbles from Eversight. The first denotes the amount of each product sold at each store on each date:

```r
> # A tibble: 6 x 4
> store product day quantity
> <int> <chr> <date> <dbl>
> 1 993 997377 2019-09-09 1
> 2 993 997377 2019-09-10 3
```

\(^4\)https://eversightlabs.com/
The products are all from the same category: canned food. Hence, store = 978 sold four cans of product = 995036 on date = ymd("2019-09-10"). The second tibble looks like this:

eversight_price %>% glimpse

```r
> Rows: 2,487,667
> Columns: 13
> $ store <int> 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22
> $ product <chr> "007219", "007219", "007219",...
> $ date <date> 2019-07-03, 2019-07-04, 2019-
> $ price <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
> $ Promotion_TPR_Sale <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
> $ Promotion_Print_Ad_Sale <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ Promotion <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ No_Label <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ Test <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ Promotion_Digital_Ad_Sale <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ Promotion_Print_Ad_Buy_One__Get_One_Free <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ Promotion_TPR_Clearance <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
> $ Promotion_TPR_Buy_One__Get_One_Free <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.1: Elasticity of Demand Under Various Price Changes
The first four columns denote the price of a given product on a given date at a given store, and the remaining nine columns specify the “price type,” which is either No_Label, which denotes regular prices, Test, which denotes Eversight’s experimental prices, or various flavors of promotion, e.g., Promotion_Print_Ad_Buy_One__Get_One_Free. There is only one price type at any given time, so each row has one price type value equal to one and the rest equal to zero. For example, the glimpse() output above describes a product on Promotion, but not on Promotion_TPR_Clearance, Promotion_TPR_Sale, etc.

4.3.3 Clean

Exercise 4.40. We will begin by pivoting eversight_price into a more amenable form.

- **pivot_longer()** eversight_price to remove all columns besides store, product, date, and price.
  - Set the names_to = "price_type" and the values_to = "value".
  - Use values_transform = list(value = as.logical) to turn the 1 and 0 values into TRUE and FALSE values.
- <- the result into eversight_price.
  - The result should have 22389003 rows.

Exercise 4.41. We will now tidy eversight_price and join it with eversight_quantity, so that all our data are in one organized tibble.

- **filter()** eversight_price by value and then drop the value column.
  - For example, the first row of the pivoted tibble tells us that product = "004337" at store = 9 had price_type = "Promotion" on date = ymd("2019-01-04"); the following eight columns are redundant, as they simply tell us that this product’s price type was not "No_Label", "Promotion_TPR_Clearance", etc. Your filter() should remove these redundant columns.
  - Relabel the price_type values by piping them through str_replace("No_Label", "Baseline") %>% str_remove("_.+").
    - The first operation changes "No_Label" to "Baseline" and the second removes all the text after and including the first underscore.
    - The resulting price_type vector should have distinct values "Baseline", "Promotion", and "Test".
  - **inner_join()** the resulting tibble with eversight_quantity %>% rename(date = day).
    - If we didn’t change day to date then the inner_join() would match on store and product alone, which would almost certainly crash our computer, as each LHT observation would match every RHT observation with the same store and product. For example, there are 797 observations with store == 37 and product ==
274224 in both eversight_quantity and eversight_price; if we didn’t match on
date then each of the 797 eversight_order observations would pair with each of
the 797 eversight_price observations, for a total of $797^2 = 635209$ observations
with store == 37 and product == 274224.

- <- the result into eversight.
  - eversight should have 2487667 rows and 6 columns.

4.3.4 Analyze

Exercise 4.42. We will now calculate the average sales rate associated with each posted
price.

- Start with eversight and group_by() store and product.
- Use mutate() to define price_change as a logical that indicates whether price differs
  from lag(price, order_by = date, default = 0).
  - The order_by = date option ensures that your code will work even when the
tibble rows are not chronologically arranged. (To create robust code, it’s best to
not rely on the ordering of your tibble rows.)
  - The default = 0 option ensures that the first value of price_change is TRUE,
rather than NA.
- Within the same mutate(), define price_period = cumsum(price_change).
  - For example, price_period = 4 indicates that the given product at the given
store is on its fourth price—i.e., it’s had three prior price changes.
- group_by() store, product, price_period, price, and price_type.
  - Equivalently, you could group_by() across(-c(date, quantity, price_change)).
- summarise() the result to define n = n() as the number of rows in the given group and
define sales_rate = sum(quantity)/n as the corresponding average sales quantity.
  - Use the .groups = "drop" option of summarise() to return an ungrouped tibble.
- <- the result into eversight_2.
  - This tibble should report the average sales_rate associated with 236560 different
pricing regimes. For example, the observation with store = 22, product = 017334,
and price_period = 11 indicates that this product’s eleventh price lasted for n
= 8 days, over which time the average sales_rate = 3.25.

Exercise 4.43. We will now pull forward the prior price’s data, so that we can measure
the changes between the current price and the prior price.

- Start with eversight_2 and group_by() store and product.
• Use `mutate()` and `across()` to apply anonymous function `~.x %>% lag(order_by = price_period)` to columns `n`, `sales_rate`, `price_type`, and `price`. Include the option `.names = "{.col}_lag"` in your `across()` call to assign the outputs to new columns called `n_lag`, `sales_rate_lag`, `price_type_lag`, and `price_lag`.

• `ungroup()` the result and `filter()` it down to the rows with `n >= 7` and `n_lag >= 7`.
  - This step restricts the sample to the cases in which the current price and the previous price both lasted at least one week. We impose this condition, because accurately estimating the `sales_rate` requires at least a week’s worth of data.

• `<-` the result into `eversight_3`.
  - This tibble should have 76736 rows.

---

Our `eversight_3` tibble should look like this:

```
> eversight_3 %>% glimpse

Rows: 76,736
Columns: 11
$ store <int> 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
$ product <chr> "007219", "007219", "007219", "017334", "017334", "0173-
$ price_period <int> 2, 3, 4, 2, 7, 7, 8, 9, 10, 12, 13, 14, 15, 17, 21, 22,
$ price <dbl> 0.89, 0.99, 0.89, 1.49, 1.49, 1.49, 1.00, 1.19, 1.49, 1-
$ price_type <chr> "Promotion", "Promotion", "Promotion", "Baseline", "Bas-
$ n <int> 7, 8, 13, 12, 7, 9, 7, 9, 17, 12, 7, 7, 21, 34, 35, 7, ~
$ sales_rate <dbl> 2.0000000, 1.7500000, 1.6923077, 0.6666667, 0.2857143, ~
$ n_lag <int> 49, 7, 8, 10, 7, 9, 7, 9, 8, 12, 7, 7, 14, 35, 7, ~
$ sales_rate_lag <dbl> 1.0000000, 1.7500000, 0.8571429, 0.2000000, ~
$ price_type_lag <chr> "Promotion", "Promotion", "Promotion", "Promotion", "Pr-
$ price_lag <dbl> 1.00, 0.89, 0.99, 1.00, 1.25, 1.49, 1.49, 1.00, 1.19, 0-
```

Each row corresponds to a price change, where `price`, `price_type`, and `sales_rate` describe the post-change situation, and `price_lag`, `price_type_lag`, and `sales_rate_lag` describe the pre-change situation. For example, the first observation has `price_type = "Baseline"` and `price_type_lag = "Promotion"`, and thus this price change corresponds to the end of a promotion.

Overall, there are nine ways that the price type can change:


```r
eversight_3 %>%
distinct(price_type_lag, price_type)
```

> # A tibble: 9 x 2
> 
> | price_type | price_type_lag |
> |------------|---------------|
> | <chr>      | <chr>         |
> | 1          | Promotion     |
> | 2          | Baseline      |
> | 3          | Promotion     |
> | 4          | Baseline      |
> | 5          | Test          |
> | 6          | Baseline      |
> | 7          | Test          |
> | 8          | Test          |
> | 9          | Promotion     |

But we will only consider the four listed in this tibble:

```r
cases_to_consider <-
tribble(
  ~ price_type_lag, ~ price_type, ~ label,
  "Test", "Test", "Test", "Baseline",
  "Baseline", "Baseline", "Baseline",
  "Baseline", "Promotion", "Promo Start",
  "Promotion", "Baseline", "Promo End"
) %>%
mutate(
  label =
  fct_relevel(label, "Baseline", "Promo Start")
)

cases_to_consider
```

> # A tibble: 4 x 3
> 
> | price_type_lag | price_type | label |
> |---------------|------------|------|
> | <chr>         | <chr>      | <fct>|
> | 1             | Test       | Test |
> | 2             | Baseline   | Baseline |
> | 3             | Baseline   | Promotion |
> | 4             | Promotion  | Promo End |

Exercise 4.44. We will now combine `eversight_3` with `cases_to_consider` to create the plot presented at the beginning of this section.
• `inner_join()` `eoversight_3` with `cases_to_consider`, to restrict attention to the four main types of price changes: baseline-to-baseline, test-to-test, baseline-to-promotion, and promotion-to-baseline.

• `mutate()` the result to define `price_change = log(price) - log(price_lag)` and `sales_change = log(sales_rate) - log(sales_rate_lag)`.

  – The logarithms enable us to interpret the slop of our plot’s regression lines as price elasticities.

• `ggplot()` the result, with `x = price_change` and `y = sales_change`.

• Include `geom_point(size = .05, alpha = 1/20)` and `geom_smooth(method = "lm")` layers.

• `facet_wrap()` by label.

### 4.3.5 Explore

**Exercise 4.45.** The figure you created indicates that sales go up when a promotion starts and go down when a promotion ends. However, the decrease could exceed the increase because the promotion could cannabalize future sales, as customers stock up when the price is cheap. To quantify this cannabalize effect, compare the baseline sales rates in the weeks leading up to a promotion to the baseline sales rates in the weeks following a promotion.

### 4.4 Lab: Inventory Runs

#### 4.4.1 Overview

The COVID-19 pandemic incited panic purchasing that nearly crippled our supply chains. Stores faced runs on inventory as customers scrambled to stock up their home pantries and freezers with basic essentials. Customers took photos of bare Walmart and Target shelves—a spectacle they had never before seen—and posted them to social media, reinforcing the fear that initiated the hoarding. An evil cycle began: we stockpiled because others stockpiled. And once it got going, the cycle sustained itself: we kept hoarding even after learning that the virus wouldn’t dramatically undermine our production capacity, because we feared that even if our supply chains could withstand the virus, they couldn’t withstand our hoarding. Hence, hoarding begat hoarding.

Leonard [2020] describes these precarious few months:

> Suddenly every consumer wants copious amounts of toilet paper, hand sanitizer and shelf-stable food. At the same time, the healthcare system needs ventilators and masks.

The spike in demand is evident in empty grocery store shelves and state politicians’ pleas for medical supplies.
When the retail or end-user node of the supply chain sees even a slight variation in demand, it quickly ripples and grows larger as it reaches suppliers and manufacturers. In the world of supply chain, this is known as the bullwhip effect—and we’re seeing it play out in real-time as the coronavirus leads consumers to panic buying and forces hospitals to scramble for supplies.

“Now what we have is probably the bullwhip on crack,” [said] Sarah Rathke, a partner with Squire Patton Bogg. ...

[T]he main cause of the bullwhip in the current environment is the rationing game, according to Robert Bray, an associate professor of operation at Northwestern University who published research on the effect last year.

“Customers are getting really apprehensive about the supply being secure,” Bray told Supply Chain Dive in an interview. “So they’re all kind of trying to cash out the inventory that they’re going to need in the next couple months in anticipation of this store sort of running out of stock, and it’s kind of like a self-fulfilling prophecy.”

The inventory runs we experienced at the beginning of the pandemic are the supply chain analog of the bank runs we experienced at the beginning of the 2008 financial crisis. However, whereas bank runs are rare calamities, inventory runs are actually relatively common. Indeed, most inventory runs are orders of magnitude less consequential than those induced by the pandemic. For example, Bray et al. [2019b] illustrated the smaller inventory runs that often bedevil supermarket supply chains:

The context is a Chinese grocery supply chain that spans one upstream distribution center (DC) and 73 downstream stores. If the stores were self-sacrificing, they would curtail their orders when the DC’s inventory runs low, scrimping for those in need. However, they are self-serving, and therefore, they accelerate their orders, stockpiling inventory to hedge against a potential upstream stock out.

Note, in this case it’s stores, rather than customers, that hoard inventory. And the signal that triggers these inventory runs isn’t a pandemic, but a low upstream inventory level: all stores see the DC stock levels, and all stores know that all other stores see the same information. So a low upstream inventory level will spur the stores to snap up the dwindling supplies in anticipation of a possible future shortage, slamming the distributor with a blast of simultaneous orders. However, the effect is far more subtle than anything we saw during the coronavirus. Specifically, Bray et al. [2019b] reported that the knowledge that the upstream inventory level is in the bottom 10% of the values it typically takes increases the likelihood of a given store ordering inventory by a moderate 31%.

We will replicate Bray et al.’s study to uncover these inventory runs for ourselves. Specifically, we will reproduce their primary exhibit, figure 2 on page 458, which most clearly illustrates the phenomenon. This lab will entail a lot of challenging data cleaning—indeed no other assignment in this book will require as much data wrangling. But this work will
shore up your data science foundation, instilling the sound data manipulation practices. Moreover, it is not contrived: it is precisely the data cleaning methodology that Bray et al. [2019b] employed.

Before starting on the lab, read this write-up\(^5\) of Bray’s study, so you have a better sense for what we’ll be doing.

### 4.4.2 Objective

Mark Twain famously wrote “the difference between the right word and the almost right word is really a large matter—it is the difference between the lightning and the lightning bug.” And the same goes for plots: if the right picture is worth 1,000 words then the almost right picture is probably worth about two sentences. The perfect plot concentrates a sprawling dataset to a singularity of illumination, like a magnifying glass focuses diffuse sunlight to a point of fire. Whereas a very good plot encapsulates a key facet of the data, an ideal plot brings it to life. The objective of this lab is to show you what it takes to create just the right plot. Specifically, we will recreate figure @ref, which crystalizes the phenomenon of ration gaming (which we’ll discuss below). These simple line graphs look straightforward enough to make, but nothing in data science is easy because a large sample will undermine you in many surprising ways. Indeed, a million-observation sample will have representatives for every possible problem, so you can’t take anything for granted. This lab will give you a sense for what it takes to do real data science. It’s not glamorous, but neither is real science. Real science is a struggle.

There are three fundamental challenges in this lab. The first is joining together the input tables into a large tibble that contains all the relevant variables. We must perform these joins with care to avoid creating redundancies in our resulting tibble. We will prepare for and perform these joins in exercises 4.46–4.50.

The second challenge is to address the data issues in our sample. Exercise 4.51 distinguishes between observations in which the store didn’t order inventory and observations in which we didn’t observe whether the store ordered inventory, both of which correspond to NA values after the join. Exercise 4.52 highlights two problems with our inventory series: (i) long spells with no inventory records and (ii) long spells with no change in the inventory level (which suggests an error in the records). We mitigate these issues by chopping up a product’s data into “generations” in exercise 4.53 and removing the generations with fewer than 200 observations in exercise 4.54. Specifically, if there’s a range of data that looks fishy, we give every observation in that range a different generation number, dividing the suspicious subsample into a collection of single-observation generations; and since they correspond to only one row, these degenerate generations get dropped when we impose the 200-observation requirement. Exercise 4.54 also removes products with very infrequent orders or inventory changes.

The third challenge is constructing the graph. For this, we must compute a collection of statistics, pivot them into the correct shape, and then `ggplot()` them.

\(^5\)https://insight.kellogg.northwestern.edu/article/counterintuitive-way-to-keep-shelves-stocked-and-prices-down

291
Figure 4.2: Signatures of Rationing and Gaming
These line plots depict the degree of rationing and gaming by product category. The rationing plot graphs the order fulfillment probability (the fraction of orders the DC fulfills) as a function of the DC inventory level. And the gaming plot graphs the order placement probability (the fraction of observations with a positive order quantity) as a function of the DC inventory level. We measure DC inventories in percentages, with the empirical cumulative distribution. The probability of the DC fulfilling an order is lower when the DC inventory level is in the lowest decile, which implies rationing. In contrast, the probability of the store placing an order is higher when the DC inventory level is in the lowest decile, which implies gaming.

4.4.3 Data
We will use order_dates, category, order_store, ship_store, inv_store, and inv_DC from section 1.3, exercise 1.32. These tibbles describe different aspects of the grocery supply chain:

- order_dates specifies whether, for a given date, we observe the stores’ order quantities (order_observed = TRUE) or we do not (order_observed = FALSE). Unfortunately, there’s a gap in our order data because the store lost some of these records, as Bray et al. [2019b] explained: “We do not observe orders ... from October 23, 2011 to December 31, 2012, because of a lost Excel file.”
• **category** assigns a product category to each item. This latter variable is a stock keeping unit (SKU) identification number. For example, *item* = 1895 corresponds to a 250 ml carton of “fresh, orange-flavor orange juice” (direct translation).

• **order_store** records the order quantities of 113 stores, which are identified by ID variable *store*. If *order_type* = "DC" then order records the amount of inventory that the store requested from the DC at the beginning of the given date. Alternatively, if *order_type* = "CD" then the product is “cross-docked,” which means that the DC distributes this merchandise to the stores as soon as it receives it. Hence, the *order_type* = "CD" observations do not actually correspond to store orders—rather, they are automated records that the DC’s IT system generated when deploying the cross-docked merchandise. Finally, **order_code** is an order ID number.
  
  – An **order_code** generally corresponds to multiple **item** values, since stores usually include multiple products in an order.
  
  – A given **store** did not order a given **item** on a given **date** if (i) there is no corresponding (**store**, **item**, **date**) record in **order_store** and (ii) the row in **order_dates** corresponding to the given **date** has **order_observed** = TRUE (i.e., we observe the orders placed on this **date**).
  
  – **order** measures inventories in physical units. We will always measure inventory in physical units (as opposed to, say, the retail value of the inventory).

• **ship_store** reports the inventory that arrived at the stores from the DC. The **order_code** values link these shipments with the orders that triggered them.

• **inv_store** describes the stores’ end-of-day inventory levels.

• **inv_DC** describes the DC’s end-of-day inventory levels.

  – **inv** denotes the DC’s inventories whereas **inv** denotes the stores’ inventories.

4.4.4 Merging Inputs into Sample

The **key** of a tibble is a minimal set of columns that uniquely identifies an observation. The phrase “uniquely identifies an observation” means that every row in the tibble has a unique key, so that you could use the key of a given row to “look up” the rest of the information stored in that line item. And the phrase “minimal set of columns” means that no part of the key is redundant: i.e., that if you removed any column from the key set then the remaining columns wouldn’t uniquely identify each observation. For example, the **inv_store** tibble looks like this:

```
slice(inv_store, 1000:1004)
```

```r
> # A tibble: 5 x 4
> store item date inv
> <fct> <fct> <date> <int>
> 1 1003 11042 2012-03-25 22
```

293
The key in this case is (store, item, date). First, each (store, item, date) triple appears only once in the data, a fact that we can confirm by comparing the number of rows with the number of distinct (store, item, date) values:

```
nrow(inv_store)
```

```
> [1] 16490877
```

```
inv_store %>%
  distinct(store, item, date) %>%
  nrow
```

```
> [1] 16490877
```

Since there are as many distinct (store, item, date) triples as there are observations, each triple must correspond to only one observation. Second, these three variables are minimal because no subset of them uniquely identifies an observation. For example, there are fewer distinct (item, date) pairs than observations:

```
inv_store %>%
  distinct(item, date) %>%
  nrow
```

```
> [1] 342782
```

Hence, some (item, date) pairs must repeat. And the same is true for the (store, date) and (store, item) pairs. Thus, (store, item, date) is the tibble’s shortest row identifier—i.e., its key.

A tibble’s key is usually quite easy to identify. Often, a tibble’s variables will fall in two broad categories: identification variables and measurement variables. The latter store the

---

6 Alternatively, you can check that every row has a different (store, item, date) by confirming that `inv_store %>% count(store, item, date) %>% summarise(max(n))` outputs 1. However, this method can be quite a bit slower.
basic data recordings—i.e., “what the tibble is about”—whereas the former store the “meta-data” necessary to keep track of these recording. The key is usually the collection of the identification variables. For example, identification variables store, item, and date identify what measurement variable inv actually measures.

The key test verifies that what looks like the key is actually a key. The test comprises the uniqueness and redundancy checks that we performed for inv_store. The following exercise provides another example.

**Exercise 4.46.** Determine the identification variables of inv_DC, and use this information to guess what this tibble’s key must be. Use nrow() and distinct() to confirm that your proposed key uniquely identifies the observations. Use nrow() and distinct() to confirm that your proposed key is minimal—i.e., that no subset of its columns uniquely identifies the observations.

Understanding the key is crucial because treating a non-unique collection of variables as unique is a surefire way to corrupt your analysis. The following two exercises illustrate the dangers of not checking the key.

**Exercise 4.47.** The following two tibbles house the inv_store and order_store data for store = 7511:

```r
inv_7511 <-
  inv_store %>%
  filter(store == 7511) %>%
  select(-store)

order_7511 <-
  order_store %>%
  filter(store == 7511) %>%
  select(item, date, order)
```

The first tibble has columns item, date, and inv and the second tibble has columns item, date, and order. The first tibble has key (item, date), but the latter does not because an item can be shipped multiple times on a given date. For example, order_7511 has two observations with item = 13180 and date = ymd("2013-04-06"). But let’s see what happens when we mistakenly assume that (item, date) is the key of both tibbles.

- left_join() tibbles inv_7511 and order_7511 to create a tibble called master_panel_7511.
  - Your new tibble should have 186,468 rows.
• If (item, date) is the key for inv_7511 and order_7511 then it must also be the key for master_panel_7511. Assuming this, sum across the item values of master_panel_7511 to calculate the total inventory the store holds on each date. That is, group_by(date) and define inv_total = sum(inv).
• Calculate the median() inv_total.
  – You should get 9,275 units, which is the wrong amount.

Exercise 4.48. Misidentifying the key led to a miscalculation in the previous exercise. Specifically, the (item, date) duplicates in order_7511 spawned inv duplicates in master_panel_7511, which begot some inventory level double counting. We will now correct this problem by aggregating the data in order_7511 up front, so that it has key (item, date).

• Use group_by() and summarise() to modify order_7511 so that its order variable reports the total amount of a given item ordered on a given date. By construction, the new version of order_7511 will have a different (item, date) pair in each row.
• Use your new version of order_7511 to create a new version of master_panel_7511, and calculate the corresponding median() inv_total.
  – You should get 9,271 units, which is the correct amount.

In the exercises above, a bug in the order data led to an error in the inventory calculation: redundant rows of order_7511 lead to redundant rows in master_panel_7511, which lead to redundant values of inv. And this is often how data errors work: your data transformations can amplify an error in an input table so dramatically as to contaminate your entire panel, tainting calculations that are seemingly unrelated to the initial issue. This is why it’s important to check all tables that contribute to your analysis, even those that are seemingly inconsequential.

More specifically, exercise 4.47 illustrates a common join error: the rows of one tibble matching too many rows of another tibble, creating unwanted duplicates. We can avoid these spurious duplicates by keeping track of our tibble keys. In general, if we join a collection tibbles then the key of the resulting merged tibble is never larger than the union of the keys of the constituent tibbles. For example, we would like to merge all our tibbles into a large master panel that has a different (store, item, date) combination in each row. And we can guarantee that our combined panel won’t have any (store, item, date) duplicates by ensuring that our input tibbles’ keys contain only store, item, and date. Unfortunately, this currently isn’t the case:

• the key of category is item,
• the key of order_dates is date,
• the key of inv_DC is (item, date),
• the key of inv_store is (store, item, date),
• the key of order_store is (store, item, date, order_type), and
• the key of ship_store is (item, order_code, ship).

Hence, if we join together all these tibbles the result could have a key as large as (store, item, date, order_type, order_code, ship), and could thus have (store, item, date) duplicates. To prevent such duplicates we will combine order_store and ship_store into a tibble with key (store, item, date).

**Exercise 4.49.** We will now join order_store and ship_store into a simpler tibble called order_ship.

- Start with order_store and drop the observations in which order_type = "CD".
  - These cross-docked products are not relevant to our analysis, since the stores do not order them.
- Remove the order_type column, which the previous step made redundant.
- `left_join()` in ship_store.
- Now an NA ship value indicates that the given order_store row had no ship_store match. And this indicates that there was no shipment corresponding to this order quantity—i.e., that zero units of inventory were shipped in response. Accordingly, use `replace_na()` to change these NA terms to zero.
- Use `group_by()` and `summarise()` to redefine order and ship to be the sum of all order and ship values corresponding to a given (store, item, date) triple.
  - Use `across()` to avoid writing `sum()` more than once.
  - Use the `.groups = "drop"` option of `summarise()` to ungroup the data after you `sum()` them.
  - The observation with store = 1015, item = 11488, and date = ymd("2014-02-19") should have order and ship values of 12.
  - ship does not report the inventory shipped to a given store on a given date. Instead, it reports the inventory shipped to a given store in response to the orders placed on a given date. The shipment arrives on the ship_date, which usually falls a day or two after. However, our `summarise()` steps disregards ship_date, which we won’t use.
  - order_ship should have 409,636 rows.

Now the union of the keys of category, order_dates, inv_DC, inv_store, and order_ship is (store, item, date). Hence, we can join these tibbles together without duplicating (store, item, date) values.

**Exercise 4.50.** `left_join()` together tibbles inv_store, inv_DC, order_ship, order_dates, and category in that order.

- Your code should start withinv_store %>% left_join(inv_DC).
- <- the resulting tibble into master_panel.
- master_panel should have as many rows as inv_store.
- Confirm that every row of master_panel has a unique (store, item, date).
4.4.5 Clean

We will now tidy up our sample.

Exercise 4.51. The order and ship values in master_panel are NA if (i) no inventory was ordered on the given date or (ii) order_observed == FALSE, in which case the order data is missing. However, these values should be NA only when we’re missing data—i.e., only when order_observed = FALSE. Otherwise, these values should be zero, because we explicitly observed that no inventory was ordered or shipped on that day. We will now turn the NA values of order and ship to zero when order_observed = TRUE.

- First use replace_na() to turn all the NA values in order and ship to 0.
  - Consider this example: tibble(x = c(1, 2, NA), y = c("a", NA, "b")) %>% replace_na(list(x = 0, y = "unknown")).
- Use ifelse() to set order = NA and ship = NA when order_observed == FALSE.

Exercise 4.52. We will create two plots that depict two problems with inv_store.

- Define a data-free ggplot with inv_plot <- ggplot() + aes(x = date, y = inv) + geom_point().
- Define sample_1 as the store == 1039, item == 211610 observations of inv_store and define sample_2 as the store == 7723, item == 13045 observations of inv_store.
- %+% sample_1 to inv_plot and then %+% sample_2 to inv_plot.

The first plot you created in exercise 4.52 indicates that our inventory series can have gaps, as products are temporarily discontinued. For example, store = 1039 stopped selling item = 211610 on 2012-10-27 and then started selling it again on 2013-05-11. But like a person who’s returned to his hometown after a long absence, a product that’s reintroduced to a store after a long break is, in a sense, a different product. Or at least that’s how Bray et al. [2019b] saw it, as they treated a product that was reintroduced after an absence of more than two weeks as a new “generation” of the product. And, for all intents and purposes, they treated different generations of the same product as different products. We’ll do the same.

The second plot illustrates a more subtle problem. Basically, inv_store contains spurious records for stocks that don’t actually exist, as merchandise is lost, stolen, or somehow taken from the store without its removal being logged. And this issue becomes apparent after a product is discontinued, as the “phantom inventory” lingers on the books long after the store stops ordering it. For example, store = 7723 discontinued item = 13045 on 2014-05-31, at which point it had 11 phantom units of this item on its books. And since these units never sold—because they didn’t exist—it appears that the store held a constant 11 units of this item through the end of 2015. Bray et al. [2019b] resolved this problem by starting a
new product generation whenever the inventory level remained flat for more than 60 days in a row. This correction shortens the store = 7723, item = 13045 series from ending on 2015-06-01 to ending on 2014-12-31, excising most of the degenerate data.

We’ll add a product generation variable to master_panel in the next exercise. Following Bray et al. [2019b], we will increment the value of generation when (i) the current date is more than 14 days after the preceding date or (ii) the value of inv hasn’t changed for 60 consecutive observations. We will combine store, item, and generation into an identification variable called product. Henceforth, the term “product” will denote a (store, item, generation) triple.

Exercise 4.53. We will now add the product generation to our master_panel.

- arrange() master_panel by store, item, and date.
  - The second observation should have store == 1003, item == 10210, and date == ymd("2011-04-02").
- group_by() store and item.
- Use mutate() and lag() to create a logical variable that indicates whether this observation’s date is more than 14 days later than the previous observation’s date. In other words, time_jump should be TRUE if the difference between date and lag(date) exceeds 14.
  - Generally, the first element of a lagged vector is NA. For example, 1:3 %>% lag = c(NA, 1, 2). Hence, the first value of time_jump should be NA for each store and item grouping. We will remove these NA terms later.
  - In general, you should avoid embedding constants, like the number 14, in your code. Instead, it’s preferable to define some descriptive variable upfront, such as allowable_time_gap <- 14, and then using that in your code.
- Within the same mutate() call, use lag() to define value_change as a logical variable that indicates whether this row’s inventory level differs from the previous row’s inventory level. In other words, value_change should be TRUE if inv differs from lag(inv).
  - The first element of value_change should be NA for each store and item grouping.
- Pipe the output of mutate() into replace_na() to turn the NA values in time_jump and value_change to FALSE.
- Pipe the output of replace_na() into a second mutate(), and use cumsum() to define cum_value_change as the cumulative sum of value_change. That is, cum_value_change should record the total number of times the inventory level has changed up until the given point.
- Within the same mutate() step, define stationary as a logical variable that indicates whether the value of cum_value_change in this row equals that from 60 rows back. In other words, stationary should be true TRUE if cum_value_change equals lag(cum_value_change, 60), which indicates that the inventory level didn’t change over the past 60 observations.
– Note that the first 60 values of `1:100 %>% lag(n = 60)` are NA. By a similar logic, the first 60 values of `stationary` should be NA.
– Rather than hard code the number 60 into your solution, define `flat_time <- 60` upfront and use that variable instead.

• After the `mutate()` step, `replace_na()` the values in `stationary` to FALSE.
• Use a third `mutate()` step to define `new_gen` as a logical variable that is TRUE if and only if there was a 14-day `time_jump` or the inventory level was `stationary` for the past 60 days. In other words, let `new_gen` indicate a long span of missing data or a long spell without a sale or delivery (both of which should trigger a new product generation).
• Define `gen` as the cumulative sum of `new_gen`. This variable specifies the current generation number.
• `ungroup()` the data and pipe it into `unite(product, store, item, gen, remove = FALSE)` to create a column called `product` that combines `store`, `item`, and `gen`. This variable will serve as our product identification number.
• Save the result as `master_panel`.
  – The 5000th row of `master_panel` should have `product = "1003_11222_838"`.

**Exercise 4.54.** Finally, we will remove anomalous products from our sample, in the fashion of Bray et al. [2019b]. Specifically, we will drop a `product` if (i) it has fewer than 200 observations, (ii) less than 4% of its observations have an inventory level change, or (iii) less than 2% of its observations have a positive order quantity.

• Start with `master_panel` and `group_by(product)`.
• `filter()` the result as follows:
  – Remove `product` groups that have fewer than 200 observations.
  – Remove `product` groups with `mean(value_change)` less than 0.04.
  – Remove `product` groups that have `mean(order > 0, na.rm = TRUE)` less than 0.02.
• `ungroup()` the tibble, and drop columns `time_jump`, `value_change`, `cum_value_change`, and `stationary`.
• Save the result as `master_panel`.
  – The tibble should now have 8,663 distinct `product` values.

### 4.4.6 Analyze

We will now use `master_panel` to illustrate the inventory runs endemic to our supermarket supply chains. Bray et al. [2019b] referred to the stores’ strategic hoarding as “ration gaming,” explaining that stores may game the means by which inventory is rationed. In addition to competing for customer demand, retailers must compete for vendor supply. Thus,
stores will jockey for stock in times of scarcity—they will request excess inventory when they anticipate curtailed shipments, hoping to end up with the desired amount of product. These inflated orders amplify supply chain volatility.

Bray et al. [2019b] depicted this ration gaming in figure 2 of their article, which we reproduced in figure 4.2 above. This figure captures the essence of their argument in two simple plots. Bray et al. [2019b] explained how these two graphs indicate ration gaming:

Ration gaming has two aspects: rationing—the DC curtailing shipments when its supply runs short—and gaming—the stores selfishly manipulating the inventory allocation scheme. Figure 2 depicts both phenomena.

First, the DC rations inventory. It fulfills 95% of orders when its inventory level is above the first decile, but only 36% when its inventory level is below the first decile (overall, 13% of orders go unfulfilled). And a stint of rationing can last awhile: if we define a “rationing spell” as a span of time when the estimated order fulfillment probability is less than one half, then 50% of rationing spells last at least 5 days, 10% last at least 14 days, and 1% last at least 35 days. Sample-wide, these rationing spells comprise 10% of our sample and account for 74% of unfulfilled orders. Moreover, these rationing spells are predictable because the DC inventory declines at a steady pace. ...

Second, the stores game the inventory rationing scheme. They have a 9.6% probability of ordering inventory when the DC inventory is above the first decile and a 12.8% probability of ordering inventory when the DC inventory is below the first decile (an increase of (12.8% - 9.6%)/9.6% = 31%). This phenomenon is broad: 60 out of the 61 stores that have at least 10,000 observations in our sample order more frequently when the DC inventory is in its lowest decile.

The following exercise will create analogous plots with master_panel. But our graphs will be a bit different as our sample was a bit different. (For example, our plot will feature only three product categories.)

**Exercise 4.55.** We will now create our inventory run plot.

- Remove from master_panel the observations in which Inv is NA.
- group_by() item.
- Use ntile() to create a variable called Inv_percent that specifies the percentile value of Inv. This variable should take integer values from 1 to 100, dividing each item’s data into 100 parts of roughly equal size. Higher values of Inv_percent should correspond to higher values of Inv.
  - Inv should have an average value of 10 units in the subsample with item = 10210 and Inv_percent = 17.
- Define logical variable order_positive that is TRUE if order > 0, FALSE if order = 0, and NA if order is NA.
• `group_by()` `category` and `Inv_percent`.
• Use `summarise()` and `across()` to redefine `order`, `ship`, `order_positive`, and `order_observed` to be the sum of their respective values across `(category, Inv_percent)` pairs.
  – Use the `na.rm = TRUE` option of `sum()`.
  – For `category = "Drinks"` and `Inv_percent = 19`, you should get `order = 52758` and `order_observed = 23685`.

• Define `Rationing = ship / order` as the ratio of inventory delivered to inventory requested.
• Define `Gaming = order_positive / order_observed` as the ratio of the number of days for which the order quantity is observed and positive to the number of days for which the order quantity is observed.
• Remove all variables besides `category`, `Inv_percent`, `Rationing`, and `Gaming`.
• `pivot_longer()` the tibble into one with column names `category`, `Inv_percent`, `value`, and `name`, where the last variable is a text string that equals either "Rationing" or "Gaming".
• `ggplot()` the data in the fashion of Bray et al. [2019b].

The plot you created illustrates two effects. The top panel demonstrates that the supplier is less likely to fulfill store orders when its inventory is low—and thus that stores have an incentive to track the supplier’s inventory level. And the bottom panel demonstrates that stores respond by placing more orders when it seems that the supplier may stock out—i.e., that they engage in inventory runs (albeit modest ones).

### 4.5 Lab: Disintermediation in Gig Economy

#### 4.5.1 Overview

Gig economy platforms, such as Uber or Airbnb, are middlemen, that link customers with service providers. And since there’s always an incentive to cut out a middleman, customers and service providers are often inclined to transact offline, to avoid giving the platform its cut. Astashkina et al. [2023] study the prevalence of such platform disintermediation in a European company that sells itself as the “Uber of home cleaning.” They use the company’s data, which reports the geographic distance between the cleaners and the residences they were assigned to clean—both while the cleaners are working and when they are not. You see, the platform’s app records each cleaner’s whereabouts at roughly 30-minute intervals, whenever it is running on the cleaner’s phone. And the cleaners tend to leave the app open for long periods, which allows the company to deduce whether any of the cleaners returned to any of the residences they were previously assigned to for some off-the-books cleanings (assuming that the cleaner wouldn’t have the foresight to exit out of the app before...
performing the disintermediated cleaning). The authors conclude that there is negligible disintermediation in this context.

Figure 4.2 depicts the author’s most compelling evidence for a dearth of disintermediation. This figure plots the distribution of the shortest recorded distance between a given cleaner and a given residence on a given day, across four samples of data. The Working sample comprises the (cleaner, residence, date) triples that correspond to an official, app-sanctioned cleaning. In this case, the distance values are short, as the cleaner is usually geotagged in the vicinity of the apartment, as they clean it. The Non_Working sample comprises the (cleaner, residence, date) triples that do not correspond to an official cleaning, but that lie between two official cleanings. For example, if cleaner A cleans residence X on 2017-02-02, 2017-02-08, and 2017-02-12, then the Working sample would have cleaner A, residence X observations with dates 2/2, 2/8, and 2/12, and the Non_Working sample would have cleaner A, residence X observations with dates 2/3, 2/4, 2/5, 2/6, 2/7, 2/9, 2/10, and 2/11. The idea is that a cleaner and customer wouldn’t transact on the app after disintermediating off of it, so the official cleaning on 2017-02-012 implies that cleaner A did not do any unofficial cleanings at residence X before 2017-02-013, which implies that this cleaner did not do any cleanings at this residence—official or otherwise—on the Non_Working dates. As expected, the distance values are longer in the Non_Working sample, as the cleaners have no reason to be near these residences in this case. (The authors attribute the sharp dip near zero to the fact that “the area of land that’s approximately $r$ km from an apartment decreases linearly with $r$.) Finally, the Observed comprises the observations that constitute the Working and Non_Working samples, and the Unobserved sample comprises the observations that come after the given cleaner’s last official job at the given residence. For example, the Observed sample would have cleaner A, residence X observations with dates between 2017-02-02 and 2017-02-12, and the Unobserved sample would have cleaner A, residence X observations with dates between 2017-02-13 and 2017-08-1 (which is the end of our sample). In other words, the Observed sample consists of the (cleaner, residence, date) triples for which we know whether or not a cleaning took place (e.g., there couldn’t have been a disintermediated cleaning on 2/9, because the official relationship didn’t end until 2/12), whereas the Unobserved sample consists of the (cleaner, residence, date) triples for which we don’t know whether a cleaning took place (e.g., there could have been disintermediated cleaning on 2/20, since the official relationship was over by then).

The main takeaway is that the distribution of the distance variable, which the authors call coarsened_shortest_distance, is almost exactly the same in the Unobserved and Non_Working samples, as the authors explain:

The stark similarity between these two distributions is our most compelling evidence for our null result. Intuitively, we would expect the Unobserved coarsened_shortest_distance values to match the Non_Working coarsened_shortest_distance values if no contracts disintermediated so that the cleanings stopped before the post period [(i.e., after the last official cleaning)]; we would expect them to match the Observed values if all contracts disintermediated so that the cleanings continued as before in the post period; and we would expect them to resemble a mix between the Observed and
Non_Working values if a fraction of contracts disintermediated so that the cleaners still visit the dwellings in the post period, but less often. [Figure 4.2] depicts the first of these three cases: with the characteristic dip at zero, the Unobserved coarsened_shortest_distance distribution resembles the Non_Working coarsened_shortest_distance distribution but not the Observed coarsened_shortest_distance distribution. In fact, the Unobserved sample has even fewer small coarsened_shortest_distance values than the Non_Working sample: the share of observations with a coarsened_shortest_distance value less than 25, 50, and 100 meters is 26, 15, and 7.3 times higher in the Observed sample than the Non_Working sample and is 30, 21, and 9.9 times higher in the Observed sample than the Unobserved sample. Indeed, rather than sneaking back for unofficial work, it seems the cleaners avoid their old work sites: the mean and median [distance value] is 20.3 and 7.06 km in the Non_Working sample and 23.9 and 7.84 km in the Unobserved sample.

In other words, figure 4.2 indicates that the data in the Unobserved sample are consistent with the hypothesis that the cleaners do not return to their residences after their official employment ends. We will now recreate this figure.

4.5.2 Data

Astashkina et al. [2023] start with two raw tables of data: Jobs and Distances. The former
has many columns that describe the nature of each cleaning assignment in detail. However the authors use only three variables from this table: “the cleaner IDs, the residence IDs, and the time of the cleanings—i.e., who worked where when.” The latter table reports the geographic distances between the cleaners and the apartments:

The Distances table reports the geographic distances between each cleaner and the dwellings they clean, roughly every half hour the app is open on their phone, between 2015-09-07 and 2016-06-23. More specifically, the table comprises a collection of snapshots that capture at a particular time the distance between a given cleaner and all the dwellings they have ever cleaned or will ever clean. For example, a representative distance snapshot indicates that at 04:30:03 on 2016-04-02, cleaner 14592 was 0.106, 2.05, 2.17, 5.30, 6.07, 6.68, 6.82, 11.9, 14.2, 14.6, 18.0, and 25.0 km from residences 39215, 40124, 10986, 6080, 35402, 22177, 24612, 13027, 31629, 32096, 15966, and 43196, which is all the units this cleaner is associated with. We have 4,323,897 such distance snapshots, across 76,112 cleaner–date pairs, so that a cleaner’s distance profile is recorded 4,323,897/76,112 = 57 times a day, on average.

The authors then transform Distances into a table called Shortest_Distances which comprises a collection of (cleaner, residence, date, shortest_distance) quadruples, the last variable of which specifies how close the given cleaner got to the given dwelling on the given day, when they were at their nearest. Case in point: Distances has distance = 77.0, 77.2, 77.0, 76.9, 76.8, 77.0, 77.0, 76.9, and 15.9 km at time = 01:36, 01:37, 05:29, 05:31, 05:48, 08:53, 08:54, 09:10, and 15:58 for (cleaner, residence, date) = (11452, 22177, 2016-04-21), and thus Shortest_Distances has shortest_distance = min(77.0, 77.2, 77.0, 76.9, 76.8, 77.0, 77.0, 76.9, 15.9) = 15.9 km for (cleaner, residence, date) = (11452, 22177, 2016-04-21).

Since the Distances table is too large to conveniently share, we will instead start with Shortest_Distances and Jobs, which you downloaded in section 1.3, exercise 1.33. The Shortest_Distances tibble has four self-explanatory variables: the address_id, the cleaner_id, the date, and shortest_distance, which denotes the shortest recorded distance between the cleaner and the residence for the day. The Jobs table has the cleaner_id, the address_id, time_clean, which specifies when each cleaning took place, and a bunch of other variables that we will not need.

### 4.5.3 Analyze

Figure 4.2 depicts the distribution of coarsened_shortest_distance, which Astashkina et al. [2023] defined as the value of shortest_distance rounded up to the nearest multiple of 25 and then truncated at 5,000 meters. Bucketing the distances in this fashion makes them easier to plot, because we can simply graph the fraction of observations in each bucket (like a histogram).
Exercise 4.56. Add coarsened_shortest_distance, as Astashkina et al. [2023] define it, to Shortest_Distances.

Exercise 4.57. Use Jobs to add a logical column to Shortest_Distances called cleaning that specifies whether there was a cleaning that day. For example, the observation with address_id = "290d4791", cleaner_id = 4131, and date = ymd("2015-10-22") should have cleaning = TRUE, and the observation with address_id = "290d4791", cleaner_id = 4131, and date = ymd("2015-10-23") should have cleaning = FALSE.

Astashkina et al. [2023] derived from Shortest_Distances a tibble called Shortest_Distances_Filtered, as they explain:

Next, we define a (cleaner, residence) pair's active period as the days between 69 days before its first cleaning and its last cleaning, and we define this pair's post period as the days after its last cleaning and before the end of the Jobs sample. For example, cleaner 71747 first cleaned residence 71043 on 2016-12-11 and last cleaned it on 2017-03-18, so this (cleaner, residence) pair's active period spans from 2016-12-11 - 69 days = 2016-10-03 to 2017-03-18, and its post period spans from 2017-03-19 to 2017-08-10 (which is the last date in the Jobs sample). We then limit our Shortest_Distances sample to the (cleaner, residence) pairs whose post periods last at least 90 days—i.e., whose final cleanings occurred at least 90 days before the end of the Jobs sample. This filter limits the sample to the working relationships that most likely ended on our watch, as only 7.5% of inter-cleaning times exceed 90 days. We call the resulting panel Shortest_Distances_Filtered.

Exercise 4.58. Create tibble Shortest_Distances_Filtered in the fashion of Astashkina et al. [2023].

• This tibble should contain only the (cleaner, residence) pairs with sufficiently long post periods, and should contain only the dates in the active and post periods.
• The next exercise will be easier if you add to Shortest_Distances_Filtered a variable that specifies whether a given observation corresponds to the active period or the post period.

Astashkina et al. [2023] derived four tibbles from Shortest_Distances_Filtered:

Finally, we construct four samples from Shortest_Distances_Filtered and Jobs:

• the Unobserved sample comprises the Shortest_Distances_Filtered (cleaner, residence, date, shortest_distance) quadruples that correspond to the (cleaner, residence) pair's post period,
• the Observed sample comprises the Shortest_Distances_Filtered (cleaner, residence, date, shortest_distance) quadruples that correspond to the (cleaner, residence) pair's active period,
• the Working sample comprises the Observed observations that have a corresponding Jobs work order, and
• the Non_Working sample comprises the Observed observations that do not have a corresponding Jobs work order.

Exercise 4.59. Create tibbles Unobserved, Observed, Working, and Non_Working in the fashion of Astashkina et al. [2023].

• You will not need the Jobs panel, because the cleaning variable of Shortest_Distances_Filtered indicates which days have a corresponding work order.
• Rather than shortest_distance, save coarsened_shortest_distance in these tables.

Exercise 4.60. Unfortunately, it is quite inconvenient to have our data scattered across four different tibbles. So pool Unobserved, Observed, Working, and Non_Working into a large tibble called four_samples, which has columns address_id, cleaner_id, date, coarsened_shortest_distances, and sample, the last of which takes values "Unobserved", "Observed", "Working", and "Non_Working".

Exercise 4.61. Derive figure 4.2 from four_samples.

• The plot uses a scale_y_log10(labels = scales::comma) axis scaling.
• Save your plot as dist_plot.

4.5.4 Explore

Exercise 4.62. What would figure 4.2 look like if 1% of the Unobserved observations had a disintermediated cleaning? To determine this, slice_sample(replace = TRUE) the rows of the Working and Non_Working tibbles to simulate a hypothetical Non_Working tibble, which is 99 parts Non_Working to one part Working. Your simulated Unobserved tibble should have the same number of observations as the actual Unobserved tibble, except all its observations should originate from the Working and Non_Working samples. Accordingly, whereas we don’t know how many observations in the true Unobserved sample correspond to a cleaning, we know that exactly 1% of observations in the hypothetical Unobserved sample correspond to a cleaning. Recreate dist_plot with this hypothetical Unobserved sample. Compare this plot to the plot you made in the previous exercise to gauge how likely it is that at least 1% of the true Unobserved observations have a disintermediated cleaning.

4.6 Commentary

4.6.1 Lengthen to Avoid Multi-Column Operations

It’s generally easier to work with a tibble that has fewer columns than more columns. So before using a multi-column operation, ask yourself: Can I pivot_longer() the tibble to stack all these variables into single column? The answer is usually yes. For example, we
could have avoided much of the complexity of this section by `first_200_moves` in “long mode,” like this:

```r
long_200 <-
  first_200_moves %>%
  pivot_longer(
    -game,
    names_to = "move_number",
    names_pattern = "move_(.+)",
    names_transform = list(move_number = as.integer),
    values_drop_na = TRUE
  )
long_200 %>% head
```

> # A tibble: 6 x 3
>   game move_number value
>   <chr> <int> <chr>
> 1 l1NXvwaE 1 d4
> 2 l1NXvwaE 2 Nc6
> 3 l1NXvwaE 3 e4
> 4 l1NXvwaE 4 e5
> 5 l1NXvwaE 5 f4
> 6 l1NXvwaE 6 f6

This configuration is much more manageable. For example, this tibble has no `NA` values and it gives us room to add move-specific variables, like so:

```r
long_200 %>%
  mutate(player = ifelse(move_number %% 2, "w", "b"))
```

> # A tibble: 908,970 x 4
>   game move_number value player
>   <chr> <int> <chr> <chr>
> 1 l1NXvwaE 1 d4 w
> 2 l1NXvwaE 2 Nc6 b
> 3 l1NXvwaE 3 e4 w
> 4 l1NXvwaE 4 e5 b
> 5 l1NXvwaE 5 f4 w
> 6 l1NXvwaE 6 f6 b
> 7 l1NXvwaE 7 dxe5 w

308
Also, it’s often faster to operate in long mode. For example, the following compares two ways to compute `time_to_first_kill`. The first uses `pivot_longer()` to consolidate all moves into a single column, and the second uses `c_across()`, from section 4.6.5, to search across multiple columns.

```r
library(tictoc)

# Case 1: stack data into single column

tic()

time_to_first_kill <-
  first_200_moves %>%
  pivot_longer(
    -game,
    names_to = "move_number",
    names_pattern = "move_(.*)",
    names_transform = list(move_number = as.integer)
  ) %>%
  filter(str_detect(value, "x")) %>%
  group_by(game) %>%
  summarise(first_kill_move = min(move_number))

toc() # stop timer

> 0.5 sec elapsed

# Case 2: use multi-column operation

tic() # start timer

time_to_first_kill <-
  first_200_moves %>%
  group_by(game) %>%
  summarise(
    first_kill_move =
      c_across(starts_with("move")) %>%
      str_detect("x") %>%
      which %>%
      first
  )

toc() # stop timer
```

> 309
4.6.2 Watch Out For Duplicate Rows

Be vigilant about duplicate rows when joining tibbles. I warned you about this in the lab, but it bears repeating. A row in one tibble will be copied if it matches multiple rows in the other tibble, and this copying can lead to double counting. For example, if we replaced `distinct(opening_name)` with `select(opening_name)` in our solution to exercise 4.31 then every row in `chess_panel` with `opening_name = "Rat Defense: Balogh Defense"` will have three copies, because "wall-of-shields" played that opening three times.

Row duplication from joins is one of the most common sources of programming mistakes. And unlike most programming bugs, which lead to a compilation error or to flagrantly wrong results, the influence of row duplication can be subtle enough to overlook—and hence subtle enough to bias your final results. Or, in the other extreme, row duplication can be bad enough to crash your computer. For example, if a million rows in the LHT matches a million rows in the RHT then your computer will try to create a tibble with a trillion rows, which won’t end well.

4.6.3 Use Multiple Tibbles

Joins free us from the all-data-in-one tibble paradigm. And this is crucial, because it’s usually best to split your data across multiple tibbles. Indeed, just as you usually wouldn’t cram all your data into a single Excel worksheet, you usually shouldn’t cram all your data in a single R tibble.

For example, consider our chess data: there’s no good way to store both the chess moves and match details in a single tibble (unless we use a list column, which is an advanced tool we’ll discuss in chapter 5). The current solution of compressing each game’s moves into a long character string makes individual moves inaccessible: e.g., try extracting the odd moves from "f4 e6 Kf2 Qf6 Ke1 Qxf4 h3 Bc5 c3 Qf2#". Alternatively, we could store the match details in wide mode and the chess moves in long mode, like this:

```r
chess_panel %>%
  mutate(moves = str_split(moves, pattern = " ")) %>%
  unnest(moves) %>%
  relocate(moves)
```

> # A tibble: 909,432 x 14
>   moves game rated turns victory_status winner increment_code white_id
>   <chr> <chr> <lgl> <dbl> <chr> <chr> <chr> <chr>
> 1 d4 l1NXvwaE TRUE 16 resign black 5+10 a-00

\(^7\)This `unnest()` solution exploits the power of lists, which we will cover in the next chapter.
> 2 Nc6 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 3 e4 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 4 e5 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 5 f4 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 6 f6 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 7 dxe5 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 8 fxe5 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 9 fxe5 l1NXvwaE TRUE 16 resign black 5+10 a-00
> 10 Nxe5 l1NXvwaE TRUE 16 resign black 5+10 a-00
> # ... with 909,422 more rows, and 6 more variables: white_rating <dbl>,
> # black_id <chr>, black_rating <dbl>, opening_eco <chr>, opening_name <chr>,
> # opening_ply <dbl>

But this solution would add a lot of redundancy, repeating the non-moves values for each value of moves. For example, the opening_name value of the average game is copied 63.5 times in the tibble above. The third option is to give each move its own column, as we did for first_200_moves. But since the games comprise a different number of moves, this approach would lead to a “ragged right edge” that would introduce over four million NA terms. For example, the move_200 column of first_200_moves is 99.8% NA. Further, this approach would make the dimensions of the tibble highly sensitive to outliers. For example, the longest match has 349 moves, whereas the second-longest match has only 259 moves; thus this method would introduce 90 columns, move_260–move_349, just to accommodate a single game.

Without resorting to an advanced list column, the best solution is to store the moves in one tibble and the other match details in another:

```r
chess_moves <-
  chess_panel %>%
  select(game, moves) %>%
  mutate(moves = str_split(moves, pattern = " ")) %>%
  unnest(moves) %>%
  rowid_to_column(var = "move_num")
```

> # A tibble: 909,432 x 3
> move_num game moves
> <int> <chr> <chr>
> 1 1 l1NXvwaE d4
> 2 2 l1NXvwaE Nc6
> 3 3 l1NXvwaE e4
> 4 4 l1NXvwaE e5
> 5 5 l1NXvwaE f4

311
This solution has (i) easily accessible moves, (ii) no data redundancies, and (iii) no NA terms. And since they share a common game variable, the tibbles can easily communicate with one another, via inner_join().

This scheme also gives us a place to store move-level features. For example, we can add additional columns detailing which player made the move and whether the move captured a piece:
player = ifelse(move_num %% 2, "w", "b"),
capture = str_detect(moves, "x")
)

chess_moves

> # A tibble: 909,432 x 5
> move_num game moves player capture
> <int> <chr> <chr> <chr> <lgl>
> 1 1 l1NXvwaE d4 w FALSE
> 2 2 l1NXvwaE Nc6 b FALSE
> 3 3 l1NXvwaE e4 w FALSE
> 4 4 l1NXvwaE e5 b FALSE
> 5 5 l1NXvwaE f4 w FALSE
> 6 6 l1NXvwaE f6 b FALSE
> 7 7 l1NXvwaE dxe5 w TRUE
> 8 8 l1NXvwaE fxe5 b TRUE
> 9 9 l1NXvwaE fxe5 w TRUE
> 10 10 l1NXvwaE Nxe5 b TRUE
> # ... with 909,422 more rows

In contrast, when we saved all the moves in one row, as we did in first_200_moves, there
was nowhere convenient to save these attributes. So we had to write ends_with(c("1", "3",
"5", "7", "9")) every time we wanted to refer to white moves and write str_detect("x")
every time we wanted to refer to piece-capturing moves.

4.6.4 Reiterate Groupings

Our solution to exercise 4.26 includes a redundant group_by(opening_name) step inside the
left_join(). This group_by() operation is superfluous because the prior group_by() group-
ings still apply (the tibble represented by . inherits the opening_name groupings). But I
include this extra grouping step to make the nested query self-contained, which makes the
code more robust to future modifications.

For example, suppose I later wanted to limit the sample to the increment_code val-
ues that correspond to at least 250 games. As it stands, I could do so by adding
group_by(increment_code) %>% filter(n() > 250) just before the left_join().
But without the backup group_by(opening_name) step I would instead need to add
group_by(increment_code) %>% filter(n() > 250) %>% group_by(opening_name). And it’d
be easy to overlook the latter group_by() step because it’s not obvious that the code inside
the left_join() needs it. Hence, the dependence of the internal left_join() code on an
external group_by() operation represents a liability in the code base. Thus, it’s safer to
give the nested query its own group_by(opening_name) step, so that it is self-sufficient.
This principle applies more broadly: you should make separate sections of your code as independent as possible, so that mucking around with one chuck of code doesn’t break another chunk. Hence, you should reestablish your `group_by()` groupings liberally, to help decouple the different sections of your code. Moreover, reasserting the grouping structure makes your code easier to read: e.g., the nested `group_by(opening_name)` makes it clear that `captured_mean` and `captured_sd` vary with `opening_name`.

Finally, there’s an even larger lesson here: code is a “living document,” revised over many versions. So, it’s not enough for your program to be correct—it should also be resilient, because you’ll probably tinker with it in the future, after you’ve forgotten some key details.

4.6.5 Treating Rows as Vectors

We can use a multi-column `summarise()` to count the number of pieces captured in each turn:

```r
first_8_moves %>%
  summarise(
    across(
      starts_with("move"),
      ~ .x %>%
        str_detect("x") %>%
        sum(na.rm = TRUE)
    )
  )
```

```
# A tibble: 1 x 8
move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
<int> <int> <int> <int> <int> <int> <int> <int>
1     0     0 465 989 1274 2340 2509 2455
```

Specifically, this code counts the number of moves that contain an "x", which flags a captured piece.

But suppose we want to sum across games rather than across moves. We can switch from vertical aggregation to horizontal aggregation by replacing `across()` with `c_across()`:

```r
first_8_moves %>%
  rowwise %>%
  summarise(
```
This \texttt{c_across()} call vectorizes a row’s \texttt{move_1–move_8} values. And this is new: while we’ve always treated a tibble column as a vector, we’ve never before treated a tibble row (or select elements of a row) as a vector.

The \texttt{rowwise()} call tells R to treat each row as its own \texttt{group_by()} group. In general, \texttt{c_across()} pools the relevant elements of the given \texttt{group_by()} group into a large vector. For example, removing \texttt{rowwise()} yields the total number of pieces captured across all games:

```r
> # A tibble: 14,322 x 2
> game num_captured
> <chr> <int>
> 1 l1NvwaE 2
> 2 mICvQHh 0
> 3 kWKvqYl 0
> 4 9txo1uZ 0
> 5 qwU9rasv 2
> 6 dwF3DJHO 1
> 7 afoMwnLg 2
> 8 HgKlwPsz 0
> 9 2fElSeI6 0
> 10 u7i6d0aJ 0
> # ... with 14,312 more rows

> # A tibble: 1 x 1
> num_captured
> <int>
> 1 10032
```
In this case, `c_across()` stacks all `move_1`–`move_8` values in `first_8_moves` into a long vector that gets piped into `str_detect()`. In contrast, the following counts the number of captured pieces by opening move:

```r
defines first 8 moves: move_1--move_8 values into a long vector by `group_by()` and `summarise()`. The resulting `num_captured` counts are computed using `c_across()`, `str_detect()`, and `sum()`. The output is a tibble with 20 rows, each indicating the number of captured pieces for each move.

Exercise 4.63. Modify the following code to create a tibble that reports the number of times that White checked the black king and Black checked the white king (recall that checking moves are those that contain `\+`):

```r
# A tibble: 20 x 2
  move_1 num_captured
       <chr>        <int>
   1 a3         9
   2 a4        30
   3 b3         7
   4 c3        39
   5 d3        22
   6 e3        293
   7 f3        35
   8 g3        1573
   9 h3        106
  10 i3       7574
  11 j3         6
  12 k3        32
  13 l3        28
  14 m3        23
  15 n3         5
  16 o3        10
  17 p3         3
  18 q3        37
  19 r3        201
  20 s3        1
```

For example, there are 62 matches with `move_1 = "b4"`, so `c_across()` outputs a vector of length `8 * 62 = 496` for this group.

Exercise 4.63. Modify the following code to create a tibble that reports the number of times that White checked the black king and Black checked the white king (recall that checking moves are those that contain `\+`):
Exercise 4.64. Pawn moves are those that do not begin with "R", "N", "B", "Q", "K", or "O". We will now count the number of white and black pawn moves in each game.

- Remove from first_200_moves the games for which move_200 is not NA.
  - We can’t determine the number of pawn moves in these games because they didn’t finish in the 200-move window.
- Split the resulting tibble rowwise() and summarise() it as follows:
  - Request the game column and include the .groups = "drop" option, so that the resulting tibble is ungrouped.
  - Write white_pawn_moves = c_across(ends_with(c("1", "3", "5", "7", "9"))) %>% and attach to this a functional assembly line that counts the number of elements in the input vector that starts with a character other than "R", "N", "B", "Q", "K", or "O". Use the regular expression "^[RNBPQK0]", which refers to a character string that starts with "R", "N", "B", "Q", "K", or "O".
  - Analogously define black_pawn_moves.
- Call the resulting tibble pawn_moves.

We can analogously use c_across() with mutate(). For example, the following adds a column to our tibble that specifies the number of bishop moves made by White:
```r
first_8_moves %>%
rowwise %>%
mutate(
  B_count =
    c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
    str_detect("^[B"] %>%
    sum(na.rm = TRUE)
)
```

```
> # A tibble: 14,322 x 10
> # Rowwise:
> game move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8 B_count
> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <int>
> 1 l1NXvwaE d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 0
> 2 mIICvQHh e4 e5 d3 d6 Be3 c6 Be2 b5 2
> 3 kWKvrqYL d4 d5 Nf3 Bf5 Nc3 Nf6 Bf4 Ng4 1
> 4 9txo1Auz e4 e5 Nf3 d6 d4 Nc6 d5 Nb4 0
> 5 qwU9rasv d4 d5 e4 dxe4 Nc3 Nf6 f3 exf3 0
> 6 dwF3DjH0 e4 e5 Bc4 Nc6 Nf3 Nd4 d3 Nxf3+ 1
> 7 afoMwnLg e4 d5 exd5 Qxd5 Nc3 Qe5+ Be2 Na6 1
> 8 HgKLWPsz e3 e6 d4 d6 Bd3 c6 Nf3 Be7 1
> 9 2fEjSei6 e4 e6 Qh5 g6 Qe5 Nf6 d4 d6 0
> 10 u7i6d0aJ e4 e5 Nf3 Nc6 Bc4 Nf6 Nc3 Bc5 1
> # ... with 14,312 more rows
```

In the code above, "^[B" is shorthand for “character string that begins with B”, which indicates a bishop move. And, again, the `rowwise()` function divides the tibble so each row row comprises its own `group_by()` group. Accordingly, this code compresses each row’s `move_1`, `move_3`, `move_5`, and `move_7` values into a vector that gets piped through `str_detect("^[B") %>% sum(na.rm = TRUE). The outputs then get saved as `B_count`.

We can replace `rowwise()` with `group_by(move_1)` to apply the functional assembly line once for each opening move, rather than once for each game:

```r
first_8_moves %>%
group_by(move_1) %>%
mutate(
  B_count =
    c_across(starts_with("move")) %>%
    str_detect("^[B") %>%
    sum(na.rm = TRUE)
)
```

318
Exercise 4.65. Define the “center of gravity” of a first_8_moves game as the average row number listed in move_1–move_8. Using str_extract("[1-8]") to get the move row numbers, create a tibble called center_of_gravity that equals first_8_moves with an extra center of gravity column, called c_of_g. The top of your tibble should look like this:

```r
center_of_gravity %>%
  head(10)
```

In this case all games with move_1 = "e4" have B_count = 8730, which is the number of bishop moves across all games with this opening move.
We can also use `c_across()` to create general multi-column `filter()` operations. For example, we saw in section 4.2.3 that `if_all()` enables us to keep a game if all of its first four moves were made by knights. But suppose we wanted to filter the games down to those with at least three knight moves in the first four turns. We couldn’t do this with `if_all()`, but we could do so with `c_across()`:

```
first_8_moves %>%
group_by(game) %>%
filter(
  c_across(move_1:move_4) %>%
    str_detect("^N") %>%
    sum %>%
    {. >= 3}
)
```

```
# A tibble: 74 x 9
# Groups: game [74]
   game move_1 move_2 move_3 move_4 move_5 move_6 move_7 move_8
   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
1 VVMmBXVj d4 Nf6 Nf3 Nc6 Bf4 d6 e3 e5
2 leveeyFX f4 Nc6 Nf3 Nf6 b3 g6 Bb2 Bg7
3 r0Ixs8q e4 Nc6 Nf3 Nf6 Nc3 d6 Bb5 Bd7
4 ipabyOYy Nc3 e5 Nf3 Nf6 d4 d6 d5 Nd4
5 auHFTCK1 Nf3 e6 Nc3 Nf6 d4 Bb4 e4 Nxe4
6 kttttJF8 Nf3 g3 Nc6 Bg2 d5 0-0 e5
7 asHLw3nz Nf3 d5 Nc3 Nf6 Nxd5 Nxd5 d4 Nc6
8 bgshRM97 e4 Nc6 Nc3 Nf6 d4 e5 d5 Nd4
9 rtJouIXZ Nf3 g6 e4 d6 Bc4 Bg7
10 hGksQ160 Nf3 d5 Nc3 Nf6 d4 e6 e4 c5
# ... with 64 more rows
```

For another example, the following picks the games in which Black moved its queen in more than half of its moves:

```
first_200_moves %>%
rowwise %>%
filter(
  c_across(ends_with(c("0", "2", "4", "6", "8"))) %>%
    str_detect("^Q") %>%
    mean(na.rm = TRUE) %>%
    {. > .5}
) %>%
```
Exercise 4.66. Select the games of `first_200_moves` in which a move was repeated within the first dozen moves. For example, `game = "q3NHAFFp"` should be preserved because it has "O-O" in both `move_9` and `move_12` and `game = "7rbdtPw0"` should be preserved because it has "Nxe5" in both `move_10` and `move_11`.

- Use `na.omit` to remove the NA terms from the vector.
- `c("a", "b", "c") %>% {n_distinct(.) < length(.)} = FALSE` and `c("a", "b", "c", "a") %>% {n_distinct(.) < length(.)} = TRUE`
4.7 Solutions

4.1

```r
class <- function(x) {
    if (is.numeric(x)) {
        return(TRUE)
    } else {
        return(FALSE)
    }
}
```

4.2

```r
first_200_moves %>%
  select(where(class))
```

4.3

```r
first_8_moves %>%
  select(-game)

first_8_moves %>%
  select(starts_with("move"))

first_8_moves %>%
  select(contains("move"))

first_8_moves %>%
  select(move_1:move_8)
```

4.4

```r
chess_panel %>%
  summarise(
    across(contains("_"),
       last
  )
)
```

4.5

322
chess_panel %>%
  mutate(
    across(
      where(is.numeric),
      as.roman
    )
  )

4.6

first_200_moves %>%
  mutate(
    across(
      -game,
      ~.x %>%
        str_replace("#", "_checkmate") %>%
        str_replace("\+", "_check")
    )
  )

4.7

chess_panel <-
  chess_panel %>%
  mutate(
    across(
      where(is.numeric) &
      where(~ all(.x == as.integer(.x), na.rm = TRUE)),
      as.integer
    )
  )

4.8

chess_panel <-
  chess_panel %>%
  mutate(
    across(
      where(is.character) &
      ...)
where(
    ~.x %>%
    n_distinct %>%
    [. <= 12
    ),
    as.factor
)
)

4.9

first_200_moves %>%
mutate(
    across(
      across(ends_with(c("1", "3", "5", "7", "9")), ~.x %>%
        str_replace(  
            pattern = "O-O-O",  
            replacement = "Kc1"
        ) %>%
        str_replace(  
            pattern = "O-O",  
            replacement = "Kg1"
        )
    ),
    across(ends_with(c("0", "2", "4", "6", "8")), ~.x %>%
        str_replace(  
            pattern = "O-O-O",  
            replacement = "Kc8"
        ) %>%
        str_replace(  
            pattern = "O-O",  
            replacement = "Kg8"
        )
    )
)

4.10
chess_panel %>%
  filter(
    if_all(
      where(is.character),
      ~ .x %>% is.na %>% `!`
    )
  )

4.11

first_200_moves %>%
  filter(
    if_any(
      if_any(
        ends_with(c("1", "3", "5", "7", "9")),
        ~ .x %>% str_detect("^O")
      )
    )
  )

4.12

first_200_moves %>%
  filter(
    if_any(
      ends_with(c("0", "2", "4", "6", "8")),
      ~ str_detect(.x, "N.+#")
    )
  )

4.13

first_200_moves %>%
  filter(
    if_all(
      ends_with(c("1", "3", "5", "7", "9")),
      ~ str_detect(.x, "^B", negate=TRUE)
    ),
    if_all(
      ends_with(c("0", "2", "4", "6", "8")),
      ~ str_detect(.x, "^R", negate=TRUE)
    )
  )

325
first_200_moves %>%
  select(
    where(
      ~.x %>%
        is.na %>%
        `!` %>%
        sum %>%
        {. >= 200}
    )
  )
%>%
summarise(
  across(
    -game,
    ~.x %>%
      str_detect("=") %>%
      mean(na.rm = TRUE)
  )
)%>%
pivot_longer(
  cols = everything(),
  names_to = "move_num",
  names_pattern = "move_(.+)",
  names_transform = list(move_num = as.integer),
  values_to = "frac"
)%>%
mutate(player = ifelse(move_num %% 2, "White", "Black"))
%>%
ggplot +
aes(
  color = player,
  x = move_num,
  y = frac
) +
geom_point() +
geom_smooth(se = FALSE, span = 0.25) +
labs(
  x = "Move Number",
  y = "Pawn Promotion Fraction"
) +
theme_minimal() +
theme(
  legend.position = "top",
  legend.title = element_blank()
chess_panel %>%
  inner_join(first_8_moves)

chess_panel %>%
  inner_join(
    first_200_moves %>%
      select(game, ends_with(c("1", "3", "5", "7", "9")))
  )

chess_panel %>%
  inner_join(
    time_to_first_kill %>%
      mutate(
        kill_bucket =
          first_kill_move %>%
            ntile(4) %>%
            as.factor
      )
  )

  ggplot +
  aes(x = turns, color = kill_bucket) +
  geom_density() +
  theme_minimal()

first_8_moves %>%
  inner_join(chess_panel %>%
               select(game, opening_eco)) %>%
  group_by(opening_eco) %>%
  summarise(number_of_first_8_moves = n_distinct(across(move_1:move_8)))
4.23

```r
first_8_moves %>%
  inner_join(
    chess_panel %>%
      filter(increment_code == "30+3") %>%
      select(game)
  )
```

4.24

```r
chess_panel %>%
  inner_join(time_to_first_kill) %>%
  ggplot() +
  aes(x = first_kill_move, y = ..prop..) +
  geom_bar() +
  facet_wrap(vars(victory_status))
```

4.25

```r
chess_panel %>%
  left_join(
    summarise(
      
        ,
        player = c(white_id, black_id),
        opponent = c(black_id, white_id)
      ) %>%
      group_by(player) %>%
      summarise(white_opponent_count = n_distinct(opponent)),
      by = c("white_id" = "player")
    )
```

4.26

```r
chess_panel %>%
  group_by(opening_name) %>%
  filter(n() > 30) %>%
  mutate(captured = str_count(moves, "x")) %>%
  left_join(
```
```r
group_by(. , opening_name) %>%
  filter(
    row_number(captured) > 3,
    row_number(-captured) > 3
  ) %>%
  summarise(
    captured_mean = mean(captured),
    captured_sd = sd(captured)
  )
) %>%
mutate(captured_z = (captured-captured_mean)/captured_sd) %>%
ggplot() +
aes(
  x = turns,
  y = captured_z
) +
geom_point(size=.1) +
theme_bw()

4.27

chess_panel %>%
  left_join(
    chess_panel %>%
      filter(white_id == "ivanbus" | black_id == "ivanbus") %>%
      group_by(opening_eco) %>%
      summarise(ivan_turns = median(turns))
  )

# or even better:
chess_panel %>%
  left_join( # the . below references the piped-in tibble
    filter(., white_id == "ivanbus" | black_id == "ivanbus") %>%
      group_by(opening_eco) %>%
      summarise(ivan_turns = median(turns))
  )

4.28

chess_panel %>%
  inner_join(
```
```r
time_to_first_kill %>%
  filter(first_kill_move == max(first_kill_move))
) %>%
pull(moves)
```

4.29

```r
chess_panel %>%
  anti_join(
    add_count(., m_1 = word(moves, 1)) %>%
    filter(n == min(n)) %>%
    select(game)
  )
```

4.30 Option 2 is correct. For example, consider `game = "00XPfTET"`. It is played by `white_id = "cdvh"` and `black_id = "japcan"`. This game should not get removed because "japcan" is the second-highest-rated black player that "cdvh" faced, after "capito2017", and "cdvh" is the second-highest-rated white player that "japcan" faced, after "behrooz36tehran". However, Option 1 removes this game because "japcan" is the highest-rated black player that "behrooz36tehran" faced, and so its first `filter()` removes the "behrooz36tehran" vs. "japcan" match, which makes its second `filter()` think that "cdvh" is the hardest white adversary that "japcan" faced. In contrast, the second `filter()` in Option 2 starts from the original `chess_panel` tibble, which makes it aware of the "behrooz36tehran" vs. "japcan" match.

Note that the query nested inside the `semi_join()` ends in a `select()`. This operation doesn't affect the answer, but it does speed up the code. Without this step, both the LHT and RHT would contain all 14 columns of `chess_panel`, from `game` to `opening_ply`. And `inner_join()` would match on all of them, ensuring that the left-hand game equaled the right-hand game, the left-hand `rated` equaled the right-hand `rated`, the left-hand `turns` equaled the right-hand `turns`, etc. But we know that if the left-hand game equals the right-hand game then all the other variables would likewise match up, since both stem from the same `chess_panel` tibble. Thus, to avoid these needless comparisons, we use `select()` to remove all but the `game` column of the RHT.

4.31

```r
chess_panel %>%
  inner_join(
    filter(
      white_id == "wall-of-shields" | 
```
black_id == "wall-of-shields"
)

distinct(opening_name)

4.33

```r
game_player <-
chess_panel %>%
select(game, white_id, black_id) %>%
pivot_longer(
  -game,
  names_to = NULL,
  values_to = "player"
)
```

4.34

```r
player_graph <-
game_player %>%
inner_join(
  game_player,
  by = "game",
  suffix = c("_from", "_to")
)

distinct(player_from, player_to)
```

4.35

```r
player_graph_degree_2 <-
player_graph %>%
rename(player_intermediary = player_to) %>%
inner_join(
  player_graph,
  by = c("player_intermediary" = "player_from")
)

distinct(player_from, player_to)
```

4.36
player_graph_degree_3 <-
  player_graph_degree_2 %>%
  rename(player_intermediary = player_to) %>%
  inner_join(
    player_graph,
    by = c("player_intermediary" = "player_from")
  ) %>%
  distinct(player_from, player_to)

increase_separation <-
  . %>%
  rename(player_intermediary = player_to) %>%
  inner_join(
    player_graph,
    by = c("player_intermediary" = "player_from")
  ) %>%
  distinct(player_from, player_to)

4.37

player_graph %>%
  filter(player_from == "wall-of-shields") %>%
  increase_separation %>%
  increase_separation %>%
  increase_separation %>%
  increase_separation %>%
  increase_separation %>%
  distinct(player_from, player_to)

4.38

player_graph %>%
  filter(player.x == "wall-of-shields") %>%
  increase_separation %>%
  increase_separation %>%
  increase_separation %>%
  increase_separation %>%
  increase_separation %>%
  distinct(player_from, player_to)

4.39
4.63

```r
first_200_moves %>%
  rowwise %>%
  summarise(
    game,
    white_checked_black =
      c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
      str_detect("\\\") %>%
      sum(na.rm = TRUE),
    black_checked_white =
      c_across(ends_with(c("0", "2", "4", "6", "8"))) %>%
      str_detect("\") %>%
      sum(na.rm = TRUE),
    .groups = "drop"
  )
```

4.64

```r
pawn_moves <-
  first_200_moves %>%
  filter(is.na(move_200)) %>%
  rowwise %>%
  summarise(
    game,
    white_pawn_moves =
      c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
      str_detect("^[RNBQKO]",
        negate=TRUE
      ) %>%
      sum(na.rm = TRUE),
    black_pawn_moves =
      c_across(ends_with(c("0", "2", "4", "6", "8"))) %>%
      str_detect("^[RNBQKO]",
        negate=TRUE
      )
  )
```
\)

```
sum(na.rm = TRUE),
.groups = "drop"
```

4.65

center_of_gravity <-
first_8_moves %>%
rowwise %>%
mutate(
  c_of_g =
    c_across(-game) %>%
    str_extract("[1-8]")) %>%
    as.integer %>%
    mean(na.rm = TRUE)
)

4.66

first_200_moves %>%
rowwise %>%
filter(
  c_across(move_1:move_12) %>%
  na.omit %>%
  {n_distinct(.) < length(.)}
)
Chapter 5

Lists and Linear Models

5.1 Introduction

My 80-year-old father stores all his computer files in the “desktop” folder of his computer. It’s an absolute mess, as you can imagine. Of course, the proper way to store your computer files is with an organizing hierarchy of folders and subfolders. Unfortunately, standard tibbles store information like my dad, laying everything out flat on a two-dimensional surface. In this chapter, we’ll learn how to give our tibbles a hierarchical “file structure,” with subtibbles stored inside of tibbles, like subfolders stored inside of folders.

The key for storing tibbles inside of tibbles is the list. A list is a much more general version of a vector. Vectors can store only numbers, logicals, character strings, Dates, and factors, but lists can store anything, including entire tibbles. Heretofore, all our tibble columns have been vectors, but tibble columns can also be lists. Thus, a tibble column can store a list of tibbles. In other words, we can compress an entire table of data into a single tibble cell. Nesting tibbles in this fashion is a powerful and elegant way to store data.

For example, we saw in section 4.6.3 that there’s no good way to store the moves data in chess_panel without lists. Compressing the moves values into a long character string makes individual moves inaccessible, giving each move its own column introduces a proliferation of NA terms, and giving each move its own row requires us to repeat the other columns’ data an inordinate number of times. Lists give us a tidy solution: save the moves as a list of vectors:

```r
chess_panel_list <-
  chess_panel %>%
  mutate(moves = str_split(moves, " "))
```

> A tibble: 6 x 14
chess_panel_list %>% pull(moves) %>% head(3)

> [[1]]
> [1] "d4"  "Nc6"  "e4"  "e5"  "f4"  "f6"  "dxe5" "fxe5" "fxe5" "Nxe5"
> [11] "Qd4"  "Nc6"  "Qe5+"  "Nxe5"  "c4"  "Bb4+

> [[2]]
> [1] "e4"  "e5"  "d3"  "d6"  "Be3"  "c6"  "Be2"  "b5"  "Nd2"  "a5"
> [11] "a4"  "c5"  "axb5"  "Nc6"  "bxc6"  "Ra6"  "Nc4"  "a4"  "c3"  "a3"
> [21] "Nxa3"  "Rxa3"  "Rxa3"  "c4"  "dxc4"  "d5"  "cxd5"  "Qxd5"  "exd5"  "Bxe6"
> [31] "Ra8+"  "Ke7"  "Bc5+"  "Kf6"  "Bxf8"  "Kg6"  "Bxg7"  "Kxg7"  "dxe6"  "Kh6"
> [41] "exf7"  "Nf6"  "Rhx8"  "Nh5"  "Bxh5"  "Qg5"  "Rxh7"  "Kf5"  "Qf3+"  "Ke6"
> [51] "Bg4+"  "Kd6"  "Rhx6+"  "Kc5"  "Qe3+"  "Kb5"  "c4+"  "Kb4"  "Qc3+"  "Ka4"
> [61] "Bd1#"

> [[3]]
> [1] "d4"  "d5"  "Nf3"  "Bf5"  "Nc3"  "Nf6"  "Bf4"  "Ng4"  "e3"
> [10] "Nc6"  "Be2"  "Qd7"  "0-0"  "0-0-0"  "Nb5"  "Nd4"  "Rc1"  "Nxa2"
> [19] "Ra1"  "Nb4"  "Nxa7+"  "Kb8"  "Nb5"  "Bxc2"  "Bxc7+"  "Kc8"  "Qd2"
> [28] "Qc6"  "Na7+"  "Kd7"  "Nxc6"  "bxc6"  "Bxd8"  "Kxd8"  "Qxb4"  "e5"
> [37] "Qb8+"  "Ke7"  "dxe5"  "Be4"  "Ra7+"  "Ke6"  "Qe8+"  "Kf5"  "Qxf7+"
> [46] "Nf6"  "Nh4+"  "Kg5"  "g3"  "Ng4"  "Qf4+"  "Kh5"  "Qxg4+"  "Kh6"
> [55] "Qf4+"  "g5"  "Qf6+"  "Bg6"  "Nxe6"  "Bg7"  "Qxg7#"

In this case, moves is a collection of character string vectors. For example, the first element of moves isn’t the character string "d4"; it’s the character string vector c("d4", "Nc6", "e4", "e5", "f4", "f6", "dxe5", "fxe5", "fxe5", "Nxe5", "Qd4", "Nc6", "Qe5+", "Nxe5", "c4", "Bb4+.")

I’ll begin the lecture, in section 5.2.1, with a brief primer on lists. Then, in section 5.2.2 I’ll explain how we can use lists to nest tibbles within tibbles (within tibbles within tibbles...). I’ll cover two functions, nest() and unnest(), which are like pivot_wider().
\texttt{pivot\_longer()}, except rather than create and destroy columns they create and destroy nested tibbles.

Section 5.2.3 is a case study. We’ll use our list-wrangling skills to organize and analyze a sample of 129971 wine reviews from Wine Enthusiast Magazine. Specifically, we’ll study the relationship between the score a wine receives and the “sentiment” of its description (measured by the number of positive-connotation words it contains less the number of negative-connotation words).

We’ll formalize our analysis in section 5.2.4 with a linear regression. This section covers the basics of running ordinary least squares (OLS) regressions in R and uses these techniques to develop ever more sophisticated models of Wine Enthusiast’s review scores. Since this book is more about coding than about statistics I will focus more on the low-level mechanics of efficiently executing the least squares, and less on the high-level philosophy underlying the regressions and hypotheses tests.

Combining the two topics of this lecture, sections 5.2.5 and 5.2.6 illustrates how we can use lists to streamline the execution of regressions. In short, we will use nested tibbles to run a set of regressions over various subsamples of our data.

After the lecture, we will apply what we learned about lists and regressions in the lab in section 5.3. In this lab you will replicate the primary outputs of Bray et al. [2019a]: plots that illustrate that auto parts are more prone to failure when they’re manufactured in a factory that’s situated further from the vehicle assembly plant (the theory being that more distant factories operate with less oversight).

Finally, I provide some auxiliary content in the commentary section. In section 5.4.1 I will discuss an underappreciated fact about lists. But this fact can be misleading, so I won’t mention it now. In section 5.4.2 I provide a glossary of list functions. I don’t include these functions in the lecture because they aren’t as well designed as their tibble counterparts. In section 5.4.3 I will discuss a drawback of working with lists: if we want to apply a function to a list we must usually do so element-wise (unlike vectors). That is, rather than pass an entire list into a function, we must usually pass its elements into the function one at a time. And this can be \textit{much} slower. Hence, it’s often faster to convert a list to a vector (or tibble) before subjecting it to a function.

For the lecture, you can work in the code.R file of the \texttt{lm} project you created in exercise 1.34, and for the lab you can work in the code.R file of the \texttt{cars} project you created in exercise 1.35.

5.2 Lecture

5.2.1 Lists

So far we’ve stored all our data in vectors and tibbles. But R has a third type of data structure: the \texttt{list}. A list is like a vector, except it can store multiple different data types. For example, here’s a list that describes the novel \textit{Mansfield Park}:
mansfield <-
  list(
    title = "Mansfield Park",
    pub_date = "1 July 1814" %>% dmy,
    word_count = 159381,
    protagonist = "fanny price"
  )

mansfield

> $title
> [1] "Mansfield Park"
>
> $pub_date
> [1] "1814-07-01"
>
> $word_count
> [1] 159381
>
> $protagonist
> [1] "fanny price"

A vector couldn’t store two character strings, a date, and a number, as this list does. A list can contain any R object—including other lists. For example, the following defines a list of lists:

jane <-
  list(
    novel_1 =
      list(
        title = "Sense & Sensibility",
        pub_date = "1 February 1811" %>% dmy,
        word_count = 119394,
        protagonist = "elinor dashwood"
      ),
    novel_2 =
      list(
        title = "Pride & Prejudice",
        pub_date = "28 January 1813" %>% dmy,
        word_count = 122189,
        protagonist = "elizabeth bennet"
      ),
  )
jane

> $novel_1
> $novel_1$title
> [1] "Sense & Sensibility"
>
> $novel_1$pub_date
> [1] "1811-02-01"
>
> $novel_1$word_count
> [1] 119394
>
> $novel_1$protagonist
> [1] "elinor dashwood"
>
> $novel_2
> $novel_2$title
> [1] "Pride & Prejudice"
>
> $novel_2$pub_date
> [1] "1813-01-28"
>
> $novel_2$word_count
> [1] 122189
>
> $novel_2$protagonist
> [1] "elizabeth bennet"
>
> $novel_3
> $novel_3$title
> [1] "Mansfield Park"
>
> $novel_3$pub_date
> [1] "1814-07-01"
>
> $novel_3$word_count
> [1] 159381
>
> $novel_3$protagonist
The list `jane` comprises the lists `novel_1`, `novel_2`, and `novel_3`.

We can `pluck()` out individual elements of a list:

```r
# get first element (i.e., the list novel_1)
jane %>% pluck(1)

> title
> [1] "Sense & Sensibility"
>
> pub_date
> [1] "1811-02-01"
>
> word_count
> [1] 119394
>
> protagonist
> [1] "elinor dashwood"
```

```r
# get third attribute of second book:
jane %>% pluck(2, 3)

> [1] 122189
```

```r
# get "pub_date" of "novel_3"
jane %>% pluck("novel_3", "pub_date")

> [1] "1814-07-01"
```

And we can apply a function—or a “mapping”—to each element of a list:

```r
# turn list of lists into list of tibbles
jane %>%
  map(as_tibble)
```
> $novel_1
> # A tibble: 1 x 4
>    title          pub_date word_count protagonist
>   <chr>             <date>       <dbl>         <chr>
> 1 Sense & Sensibility 1811-02-01        119394 elinor dashwood
>
> $novel_2
> # A tibble: 1 x 4
>    title          pub_date word_count protagonist
>   <chr>             <date>       <dbl>         <chr>
> 1 Pride & Prejudice 1813-01-28        122189 elizabeth bennet
>
> $novel_3
> # A tibble: 1 x 4
>    title          pub_date word_count protagonist
>   <chr>             <date>       <dbl>         <chr>
> 1 Mansfield Park 1814-07-01        159381 fanny price

#get day each book was published:
jane %>%
  map(pluck("pub_date"))

> $novel_1
> [1] "1811-02-01"
>
> $novel_2
> [1] "1813-01-28"
>
> $novel_3
> [1] "1814-07-01"

#get day of the week each book was published:
jane %>%
  map(
    ~.x %>%
    pluck("pub_date") %>%
    wday(label = TRUE)
  )

> $novel_1
341
The last two examples use anonymous functions, which we first saw in section 4.2.3. For example, the last `map()` turns a list containing `novel_1`, `novel_2`, and `novel_3` into a list containing the outputs of `str_c(round(pluck(novel_1, "word_count")/1000), "k")`, `str_c(round(pluck(novel_2, and "word_count")/1000), "k")`, and `str_c(round(pluck(novel_3, "word_count")/1000), "k")`. As you see, the `.x` stands for “each element in the list.”

**Exercise 5.1.** Use `map()` to `pluck()` the every protagonist in `jane`.

**Exercise 5.2.** Use `map()` to `pluck()` the first `word()` in every title in `jane`.

**Exercise 5.3.** Use `map()` and `str_to_upper()` to get the title values in `jane`, expressed in capital letters.

In addition to `map()`, there is `map_df()`, which is the same except it output tibbles (df stands for “data frame”).
#protagonists of each book

jane %>%
map_df(pluck("protagonist"))

> # A tibble: 1 x 3
>   novel_1       novel_2       novel_3
>   <chr>         <chr>         <chr>
> 1 elinor dashwood elizabeth bennet fanny price

#days since publication of each book

jane %>%
map_df(
  .x %>%
  pluck("pub_date") %>%
  {today() - .}
)

> # A tibble: 1 x 3
>   novel_1          novel_2          novel_3
>   <dtrn>           <dtrn>           <dtrn>
> 1 77459 days 76732 days 76213 days

**Exercise 5.4.** Use `map_df()` to tabulate each protagonist in `jane`, expressed with `str_to_title()`.

5.2.2 Tibbled Lists

Heretofore, all of our tibble columns have been vectors, but tibble columns can also be lists. For example, define

```r
mansfield_tib <-
tibble(
  stat = c("Book Title", "Pub. Date", "Word Count", "Main Character"),
  value = mansfield
)
```

mansfield_tib
This tibble’s first column is a standard character string vector, but its second column is a list that comprises two character strings, a Date, and a double. For example, the first and third elements in the value column are:

```r
mansfield_tib %>% pluck("value", 1)
```

> [1] "Mansfield Park"

```r
mansfield_tib %>% pluck("value", 3)
```

> [1] 159381

Now since tibbles can store lists and lists can store anything, it follows that tibbles can store anything—including other tibbles. We will use this fact to organize the following tibble, which stores the text of Jane Austen’s novels:

```r
rows_are_lines <-
  janeaustenr::austen_books() %>%
  group_by(book) %>%
  mutate(
    chapter =
    text %>%
    str_detect("^CHAPTER|^Chapter") %>%
    cumsum,
    line = row_number()
  ) %>%
  filter(
    chapter >= 1,
    text != ""
  ) %>%
  mutate(line = row_number()) %>%
```
ungroup %>%
relocate(text, .after = "line")

rows_are_lines %>% head

> # A tibble: 6 x 4
>    book chapter line text
>  <fct> <int> <int> <chr>
> 1 Sense & Sensibility 1   1 CHAPTER 1
> 2 Sense & Sensibility 1   2 The family of Dashwood had long been settle-
> 3 Sense & Sensibility 1   3 was large, and their residence was at Norla-
> 4 Sense & Sensibility 1   4 their property, where, for many generations-
> 5 Sense & Sensibility 1   5 respectable a manner as to engage the gener-
> 6 Sense & Sensibility 1   6 surrounding acquaintance. The late owner o-

rows_are_lines %>% distinct(book)

> # A tibble: 6 x 1
>    book
>  <fct>
> 1 Sense & Sensibility
> 2 Pride & Prejudice
> 3 Mansfield Park
> 4 Emma
> 5 Northanger Abbey
> 6 Persuasion

First let's nest() the line and text values into collection of chapter-level tibbles:

rows_are_chapters <-
rows_are_lines %>%
nest(chapter_tib = c(line, text))

rows_are_chapters %>%
head(4)

> # A tibble: 4 x 3
>    book chapter chapter_tib
>  <fct> <int>      <list>
>1 Sense & Sensibility 1 1:1
>2 Sense & Sensibility 1 1:2
>3 Sense & Sensibility 1 1:3
>4 Sense & Sensibility 1 1:4

345
The `chapter_tib` column is a list of tibbles. For example, the third element of `chapter_tib` is a tibble with the following `head()`:

```r
rows_are_chapters %>%
  pluck("chapter_tib", 3) %>%
  head
```

```r
# A tibble: 6 x 2
line text
<int> <chr>
1 307 CHAPTER 3
2 308 Mrs. Dashwood remained at Norland several months; not from any
3 309 disinclination to move when the sight of every well known spot ceased
4 310 to raise the violent emotion which it produced for a while; for when
5 311 her spirits began to revive, and her mind became capable of some other
6 312 exertion than that of heightening its affliction by melancholy
```

Now we can further `nest()` each `chapter_tib` into a collection of novel-level tibbles:

```r
rows_are_books <-
  rows_are_chapters %>%
  nest(novel_tib = c(chapter, chapter_tib))
```

```r
# A tibble: 6 x 2
book novel_tib
<fct> <list>
1 Sense & Sensibility <tibble [50 x 2]>
2 Pride & Prejudice <tibble [61 x 2]>
3 Mansfield Park <tibble [48 x 2]>
4 Emma <tibble [55 x 2]>
5 Northanger Abbey <tibble [31 x 2]>
6 Persuasion <tibble [24 x 2]>
```
# view fourth novel tibble

```r
#view fourth novel tibble
rows_are_books %>%
  pluck("novel_tib", 4) %>%
  head(4)
```

```r
> # A tibble: 4 x 2
>   chapter chapter_tib
>  <int> <list>
> 1     1 <tibble [277 x 2]>
> 2     2 <tibble [146 x 2]>
> 3     3 <tibble [160 x 2]>
> 4     4 <tibble [294 x 2]>
```

# view eighth chapter tibble of fourth novel tibble

```r
#view eighth chapter tibble of fourth novel tibble
rows_are_books %>%
  pluck("novel_tib", 4, "chapter_tib", 8) %>%
  head
```

```r
> # A tibble: 6 x 2
>   line   text
>  <int> <chr>
> 1    1490 CHAPTER VIII
> 2    1491 Harriet slept at Hartfield that night. For some weeks past she had been
> 3    1492 spending more than half her time there, and gradually getting to have
> 4    1493 a bed-room appropriated to herself; and Emma judged it best in every
> 5    1494 respect, safest and kindest, to keep her with them as much as possible
> 6    1495 just at present. She was obliged to go the next morning for an hour or
```

The `nest()` operation above bundles the chapter and chapter_tib columns of `rows_are_chapters` into the six tibbles of the list column `novel_tib`. We've now organized our data into a hierarchical structure: `rows_are_books` stores six novel-level tibbles, which store 269 chapter-level tibbles, which in turn store 62248 lines of text.

**Exercise 5.5.** `pluck()` the lines of the fifth chapter of the third book of `rows_are_books`.

**Exercise 5.6.** What’s the difference between `rows_are_lines %>% nest(chapter_tib = c(line, text))` and `rows_are_lines %>% nest(novel_tib = c(chapter, chapter_tib))` and `rows_are_lines %>% nest(novel_tib = c(chapter, line, text))`?
We can unnest() the tibbles nested in the novel_tib column:

```r
rows_are_books %>%
  unnest(novel_tib) %>%
  head
```

> # A tibble: 6 x 3
> book chapter chapter_tib
> <fct> <int> <list>
> 1 Sense & Sensibility 1 <tibble [136 x 2]>
> 2 Sense & Sensibility 2 <tibble [170 x 2]>
> 3 Sense & Sensibility 3 <tibble [138 x 2]>
> 4 Sense & Sensibility 4 <tibble [170 x 2]>
> 5 Sense & Sensibility 5 <tibble [90 x 2]>
> 6 Sense & Sensibility 6 <tibble [114 x 2]>

And we can further unnest() the tibbles nested in the chapter_tib column:

```r
rows_are_books %>%
  unnest(novel_tib) %>%
  unnest(chapter_tib) %>%
  head
```

> # A tibble: 6 x 4
> book chapter line text
> <fct> <int> <int> <chr>
> 1 Sense & Sensibility 1 1 CHAPTER 1
> 2 Sense & Sensibility 1 2 The family of Dashwood had long been settle-
> 3 Sense & Sensibility 1 3 was large, and their residence was at Norla-
> 4 Sense & Sensibility 1 4 their property, where, for many generations-
> 5 Sense & Sensibility 1 5 respectable a manner as to engage the gener-
> 6 Sense & Sensibility 1 6 surrounding acquaintance. The late owner o-

The unnest() steps lengthen the tibble from 6 rows (one per book), to 269 (one per chapter), to 62248 rows (one per line). Since they lengthen the tibble, these operations are analogous to pivot_longer(). And nest() is likewise analogous to pivot_wider(), except rather splay out the data in new columns it squirrels them away in nested tibbles.

**Exercise 5.7.** We will now divide the line values of rows_are_lines by word to create a tibble that assigns each word to a different row.

- **mutate() rows_are_lines to redefine text = str_split(text, " ").** This transforms text from a vector of 62248 character strings to a list of 62248 character string vectors.
For example, consider `c("glory follows virtue", "as if it were its shadow")` %>% str_split(" ").

- unnest() the text column.
- <- the result into rows_are_words.

**Exercise 5.8.** nest() rows_are_words three times to create a tibble that stores a collection of novel-level tibbles, that store a collection of chapter-level tibbles, that store a collection of line-level tibbles. Call this tibble rows_are_books_2. This tibble should resemble rows_are_books, but with an additional layer of nesting.

**Exercise 5.9.** pluck() the 11th word of the 50th line of the 10th chapter of the 4th book of rows_are_books_2. (You should get the word "fool").

**Exercise 5.10.** We will now create a tibble that assigns each sentence of Sense & Sensibility to its own row.

- filter() rows_are_lines to keep only the book == "Sense & Sensibility" observations.
- select() away the book and line columns.
- group_by() chapter and summarise() to redefine text = str_c(text, collapse = " "). This will consolidate the text of each chapter into a long character string.
- mutate() the result to redefine text = str_split(text, " "). Note, there are two spaces in " ", so this operation splits text whenever there are two consecutive spaces. And since there are two spaces after each sentence (with a few exceptions that we’ll ignore), this operation chops text up by sentence.
- unnest() the result by text.

### 5.2.3 Case Study: Wine Reviews

We will now study the wine reviews in wine_list (which you defined in exercise 1.34 of section 1.3). Specifically, we will test whether we can predict the review’s final score from the number of positive- and negative-connotation words in its description.

wine_list is a list of lists that describe 129,971 wine reviews from Wine Enthusiast. For example, the 100th element wine_list is the following list:

```r
wine_list %>% pluck(100)
```

```r
> $points
> [1] "88"
>
> $title
>```
This blends 20% each of all five red-Bordeaux varieties, showing grapy, plummy flavor that's juicy and soft on the palate. With a core of tobacco, tar and graphite, it complements the fruity with the savory, ending simply and elegantly.

Virginie Boone

@vboone

75

"Intreccio Library Selection"

"Bordeaux-style Red Blend"

"Napa Valley"

"Napa"

"California"

"US"

"Soquel Vineyards"

Now if you google the title of this review, you will find that these data were scraped from this website. Every record in wine_list was scraped from a page like this on the Wine Enthusiast website. And if you compare a few of these wine reviews with the corresponding wine_list listings you’ll soon see what each variable in our sample represents.

Exercise 5.11. We’ll now tibblize our data, since tibbles are easier to work with than lists.

- Convert wine_list to a tibble with map_df(~.x).


2www.winemag.com
The anonymous function `~.x` returns each element of `wine_list` exactly as it is. For example, piping `wine_list` through `map_df(~.x %>% head)` would apply `head()` to each element of `wine_list` and return the results as a tibble. And piping `wine_list` through `map_df(~.x)` is the same, except it skips the `head()` steps.

- Extract the `distinct()` rows of this tibble.
- Add an identification variable with `rowid_to_column(var = "review_id")`.
- Use `strtoi()` to translate the points and price columns from character strings to integers.
- Extract the year the grapes were picked from each review title and add it as a column called `vintage`.
  - Use the expression `str_extract("\\b199\[5-9\]\b|\\b20\[01\]\b\b") %>% strtoi`, which returns the first number between 1995 and 2019 from a text string (or `NA` if none is found). For example, "Gaul 2013 Rosé (Pfalz)" %>% `str_extract("\\b199\[5-9\]\b|\\b20\[01\]\b\b") %>% strtoi = 2013.

- `<-` the result as `wine_tib`.
  - This tibble should have 119988 rows and 15 columns.

Now that they’re in a tibble, we can more easily analyze our data. We will use text analytics to relate a review’s points with its description. First, we find that longer reviews receive higher scores:

```r
wine_tib %>%
ggplot +
aes(
  x = str_count(description),
  y = points
) +
geom_density_2d_filled() +
theme_minimal() +
labs(  
  x = "Number of Characters in Description",
  y = "Point Score"
) +
guides(fill = guide_legend(title = "Probability Density"))
```
Unfortunately, that’s about all we can say when the entire description is stuffed in a single character string. So to perform more in-depth text analytics, we will create a tibble, over the next two exercises, that lists each word in each description.

**Exercise 5.12.** We will now convert the description column from a vector of character strings to a list of character string vectors. For example, this step will change the description associated with review_id == 12932 from the character string "Flat, fruity and lacking in complexity." to the character string vector c("Flat", "fruity", "and", "lacking", "in", "complexity").

- **select()** the review_id and description columns of wine_tib.
- **Redefine description** as description %>% str_remove_all("[^ \[:alnum:\]]") %>% str_squish %>% str_split(" "). This step will convert description into a list column that stores the individual words of each review in a collection of character string vectors.
  
  - **str_remove_all("[^ \[:alnum:\]]")** removes everything besides spaces and alphanumeric characters from a character string.
  - **str_squish()** removes excess space from a character string.
  - **str_split(" ")** turns a vector of character strings into a list of character string vectors, by splitting each character string at each " ". For example, consider c("don't make me run", "I'm full of chocolate") %>% str_split(" ").

- **<-** the result into wine_description_by_word
Exercise 5.13. We have now chopped up each wine description description by word. But these words are difficult to access, as they’re stored in a collection of vectors that are saved in a list column. We will now unnest() this list column to make the individual words accessible.

- unnest() wine_description_by_word by description, so that each word has its own row.
  - The tibble should have 4849491 rows after this step.
- rename() word = description.
- <- the result into wine_word_count.
  - Since each review_id has many rows, it’s better to save these word count records in a new tibble than to inner_join() them back into wine_tib. For example, there are 94 observations with review_id == 84998; so if we added the word counts into wine_tib, then the resulting tibble would repeat this review’s other variables 94 times—e.g., it would contain 94 copies of its full description.

Now we can use our word counts to perform a sentiment analysis. To do so, we will use the sentiment_dictionary defined below, which classifies 6786 words as either "+" or "-", depending on their connotation:

```r
sentiment_dictionary <- tidytext::get_sentiments("bing") %>%
  mutate(
    sentiment = 
    fct_recode(
      sentiment,
      `\'+` = "positive",
      `\'-` = "negative"
    )
  )
sentiment_dictionary %>% tail
```

```
# A tibble: 6 x 2
   word  sentiment
   <chr> <fct>
1 zealous -
2 zealously -
3 zenith +
```

> # A tibble: 6 x 2
>   word    sentiment
>   <chr>   <fct>
> 1 zealous -
> 2 zealously -
> 3 zenith +

353
Exercise 5.14. We will now combine wine_word_count with sentiment_dictionary to add four new variables to wine_tib.

- left_join() wine_word_count with sentiment_dictionary (so that all rows of the former are preserved).
- group_by() review_id.
- summarise() the result, with the .groups = "drop" option, to define
  - word_count as the number of words in description,
  - pos_count as the number of "+"-coded words,
  - neg_count as the number of "-"-coded words, and
  - pos_minus_neg as the difference between the number of "+"-coded words and the number of "-"-coded words.
- inner_join() the result with wine_tib, to add back the other review variables.
  - This join won’t create unwanted row duplicates, because each review_id appears only once in both tibbles.
- <- the result into wine_tib_2.
  - This tibble’s mean() pos_count should be 2.75.

Now we can now see that pos_minus_neg is positively correlated with points:

```r
wine_tib_2 %>%
  group_by(country) %>%
  filter(n() >= 80) %>%
  ungroup %>%
  mutate(
    country = fct_reorder(country, country, length)
  ) %>%
  ggplot() +
  aes(
    x = pos_minus_neg,
    y = points
  ) +
  geom_jitter(shape = ".") +
```
Indeed, \textit{pos\_minus\_neg} is a surprisingly good indicator of wine quality. For example, here are the three reviews with the lowest \textit{pos\_minus\_neg} scores:

\begin{verbatim}
wine_tib_2 %>%
  filter(row_number(pos_minus_neg) <= 3) %>%
pull(description)
\end{verbatim}

> [1] "There's a funky, strange quality to San Pedro's high-end 2008 Syrahs from Elqui. Like the Kankana ultra-premium, it has a very earthy,\n>   herbaceous, chalky and dry style. It has a raw, lithe and streamlined texture with a long, structured finish. A weird but fun wine for
>   adventure seekers and wine lovers who appreciate wines with unique characteristics."
> [2] "Sweaty, raw, sour aromas are harsh. The palate is flat and chunky in feel, with pithy grapefruit, nettle, garlic and grassy flavors. The
>   finish is chewy and sharp, with bright acidity and a long, grassy finish. Not for everyone, but an interesting and unusual wine."
> [3] "This is an unusual white blend from Mallorca that uses the native Premsal grape along with 15% Muscat and 5% Chardonnay. It
>   has a sweet, floral and fruity aroma with notes of honey, apricot and peach. The palate is rich and creamy with a smooth, balanced finish.
>   A unique and enjoyable wine for fans of sweeter white wines."

These reviews have 82, 87, and 84 points, respectively. And here are the three reviews with the highest \textit{pos\_minus\_neg} scores:
These reviews have 94, 94, and 93 points, respectively.

However, we can more richly capture points with three variables than with one. So we would like to describe the review score in terms of `pos_count`, `neg_count` and `word_count`, rather than in simply in terms of `pos_minus_neg` (which is a derivative of `pos_count` and `neg_count`). However, characterizing such a multi-variate relationship requires a statistical tool that’s more powerful than a simple scatter plot or rank ordering.

### 5.2.4 Linear Regression

We will now formally model the relationship between a wine’s description sentiment and its review points with linear regression. We will run the regressions with the linear model function `lm()`, and will organize the outputs with the `tidy()` function of the broom package:

```r
library(broom) # we'll need to load this package
```

Here’s a simple regression of points on `word_count`, `pos_count`, and `neg_count`:

```r
wine_model <- 
  wine_tib_2 %>%
  lm(
    points ~ word_count + pos_count + neg_count,
    data = .
  )

wine_model %>%
tidy
```
The `lm()` function conducts our linear regression. But this function is old, and it follows some outdated conventions. First, it returns the regression results as a messy kind of list called an “lm” object. The `tidy()` function converts this disorganized list into an organized tibble. Second, whereas all modern functions reserve the first argument for the data, the `lm()` function reserves the second argument for the data. Hence, we must set the second argument of `lm()` to `data = . `to direct the `%>%` to the function’s second slot. The first slot of `lm()` is reserved for the regression formula, which is a bit of code with the form `dependent_var ~ independent_var_1 + ... + independent_var_n`. For example, the `points ~ word_count + pos_count + neg_count` formula specifies the following statistical model:

\[
\text{points} = \beta_0 + \beta_1 \cdot \text{word_count} + \beta_2 \cdot \text{pos_count} + \beta_3 \cdot \text{neg_count} + \epsilon,
\]

where \(\beta_0, \beta_1, \beta_2,\) and \(\beta_3\) are fixed coefficients and \(\epsilon\) is an unobserved error term that’s mean-zero and uncorrelated with \(\text{word_count}, \text{pos_count},\) and \(\text{neg_count}\). These errors don’t have to be normally distributed.\(^3\)

The estimate column of the tibble above provides our coefficient estimates: \(\hat{\beta}_0 = 82.4, \hat{\beta}_1 = 0.134, \hat{\beta}_2 = 0.243,\) and \(\hat{\beta}_3 = -0.0962\). The “(Intercept)” term corresponds to the additive constant \(\beta_0\), which R added to the regression on our behalf (we never have to incorporate an intercept term in our regression formula, because R adds them automatically).\(^4\) Note, these are not the true coefficient values—they’re just our best guess, given the data we have. In fact, the error in our \(i\)th estimate, \(\hat{\beta}_i - \beta_i\), is roughly normally distributed with mean zero and standard deviation \(\sigma_i\) (even when the \(\epsilon\) terms are not normally distributed). This standard deviation is called a “standard error,” and it’s reported in the `std.error` column of our output tibble. Hence, the error \(\hat{\beta}_1 - \beta_1\) has standard deviation 0.000725, whereas the error \(\hat{\beta}_3 - \beta_3\) has standard deviation 0.00851. The former estimate is thus more precise.

The statistic column of our output tibble reports the t-statistic, which equals the estimate value divided by the `std.error` value. In short, statistic measures how far the coefficient estimate is from zero, in terms of standard deviations. For example, there are 0.243 / 0.00443 = 55 standard deviations between \(\hat{\beta}_2\) and zero.

Finally, the `p.value` column reports the probability of seeing an estimate as far from zero when the true coefficient is actually zero. For example, since it’s basically impossible for a (nearly normally distributed) random variable to be 55 standard deviations above its mean,

\(^3\)A common misconception is that OLS regressions require normally distributed error terms. They do not. They simply require the error terms be mean-zero and uncorrelated with the independent variables.

\(^4\)To run a regression without an intercept term add `-1` to the regression formula. For example, `points ~ word_count + pos_count + neg_count - 1` corresponds to the following statistical model: 

\[
\text{points} = \beta_1 \text{word_count} + \beta_2 \text{pos_count} + \beta_3 \text{neg_count} + \epsilon.
\]
we find that there would be essentially no chance of getting \( \hat{\beta}_2 \geq 0.243 \) when \( \beta_1 = 0 \) (the probability is so small that R rounds it down to \( p\text{.value} = 0 \)). Hence, we can safely conclude that \( \beta_1 > 0 \)—i.e., that a review’s score increases with the number positive-association words it contains. Likewise, we can conclude that a review’s score decreases with the number negative-association words it contains, because the probability of getting \( \hat{\beta}_4 \leq -0.0962 \) when \( \beta_4 = 0 \) is \( p\text{.value} = 1.36e-29 \) (i.e., 0.0000000000000000000000000000136).

Our best prediction of points, given \( \text{word\_count} \), \( \text{pos\_count} \), and \( \text{neg\_count} \), is:

\[
\text{.fitted} = \hat{\beta}_0 + \hat{\beta}_1 \cdot \text{word\_count} + \hat{\beta}_2 \cdot \text{pos\_count} + \hat{\beta}_3 \cdot \text{neg\_count} \\
= 82.4 + 0.134 \cdot \text{word\_count} + 0.243 \cdot \text{pos\_count} - 0.0962 \cdot \text{neg\_count}
\]

(5.1)

(5.2)

This expression is called the “fitted value” of our regression. For example, the review with title = "Marchesi Antinori 2015 Villa Antinori White (Toscana)" has \( \text{word\_count} = 38 \), \( \text{pos\_count} = 1 \), and \( \text{neg\_count} = 0 \), so our best forecast for this score would be \( 82.4 + 0.134 \cdot 38 + 0.243 \cdot 1 - 0.0962 \cdot 0 = 87.74 \), which is close to the 87 points it actually received.

Rather than manually type out the equation above, we can get the fitted value by piping \( \text{wine\_model} \) into broom’s \texttt{augment()} function:

```r
wine_model %>%
  augment(data = wine_tib_2)
```

```r
> Rows: 119,988
> Columns: 25
> $ review_id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1-
> $ word_count <int> 22, 35, 25, 31, 38, 38, 28, 22, 20, 24, 26, 33, 26, ~
> $ pos_count <int> 1, 2, 3, 2, 3, 4, 1, 5, 4, 4, 2, 2, 1, 3, 6, ~
> $ neg_count <int> 1, 0, 0, 1, 2, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, ~
> $ pos_minus_neg <int> 0, 2, 3, 1, 1, 2, 4, 1, 5, 4, 4, 1, 2, 0, 3, 5, ~
> $ points <int> 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, ~
> $ title <chr> "Nicosia 2013 Vulkà Bianco (Etna)", "Quinta dos-~
> $ description <chr> "Aromas include tropical fruit, broom, brimstone-~
> $ taster_name <chr> "Kerin O'Keefe", "Roger Voss", "Paul Gregutt", "-~
> $ taster_twitter_handle <chr> @kerinokeefe", @vossroger", @paulgwine", NA,~-~
> $ designation <chr> "Vulkà Bianco", "Avidagos", NA, "Reserve Late Ha-~
> $ variety <chr> "White Blend", "Portuguese Red", "Pinot Gris", "-~
> $ region_1 <chr> "Etna", NA, "Willamette Valley", "Lake Michigan ~
> $ province <chr> "Sicily & Sardinia", "Douro", "Oregon", "Michiga-~
> $ winery <chr> "Nicosia", "Quinta dos Avidagos", "Rainstorm", "-~
> $ price <int> NA, 15, 14, 13, 65, 15, 16, 24, 12, 27, 19, 30, ~
> $ region_2 <chr> NA, NA, "Willamette Valley", NA, "Willamette Val-~
```
The output tibble is our initial wine_tib_2 regression data with five extra columns: .fitted, .resid, .std.resid, .hat, and .sigma. The data = wine_tib_2 option tells augment() to append these statistics to wine_tib_2. But we can exclude this option to get a more streamlined output:

```
wine_model %>%
  augment
```

As before, a given row in this output tibble corresponds to the analogous row in wine_tib_2. But now the output includes only wine_tib_2’s regression variables—points, word_count, pos_count, and neg_count—and the five new statistics—.fitted, .resid, .std.resid, .hat, and .sigma. The first statistic is the fitted value. For example, we find that our model predicts a score of 85.8 for the first wine (conditional on its word_count = 25, pos_count = 1, and neg_count = 1 values). And since this wine actually received 87 points, it exceeded expectations by 87 - 85.8 = 1.21 points. This value is reported in the .resid column, which gives the difference between the forecasted score, .fitted, and the actual score, points.⁵

**Exercise 5.15.** Which wine most exceeded expectations, given its word_count, pos_count, and neg_count values? In other words, what title has the largest .resid?

⁵See ?augment.lm for the definition of the other statistics.
Exercise 5.16. A regression’s $R^2$ value reports the fraction of variation in the dependent variable that it explains. Compute this statistic with $1 - \text{sd(.resid)}^2/\text{sd(points)}^2$. Confirm your calculation by comparing it with the $r.\text{squared}$ statistic yielded by `wine_model %>% glance`. (The `glance()` function, from the broom package, calculates regression summary statistics.)

Now let’s add some control variables to our regression. We will begin by including the wine vintage:

```r
wine_tib_2 %>%
  lm(
    points ~ word_count + pos_count + neg_count + vintage,
    data = .
  ) %>%
  tidy
```

<table>
<thead>
<tr>
<th>term</th>
<th>estimate</th>
<th>std.error</th>
<th>statistic</th>
<th>p.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>-112.</td>
<td>4.27</td>
<td>-26.2</td>
<td>7.37e-151</td>
</tr>
<tr>
<td>word_count</td>
<td>0.174</td>
<td>0.000919</td>
<td>189.</td>
<td>0</td>
</tr>
<tr>
<td>pos_count</td>
<td>0.266</td>
<td>0.00459</td>
<td>58.0</td>
<td>0</td>
</tr>
<tr>
<td>neg_count</td>
<td>-0.109</td>
<td>0.00876</td>
<td>-12.4</td>
<td>2.30e-35</td>
</tr>
<tr>
<td>vintage</td>
<td>0.0962</td>
<td>0.00212</td>
<td>45.4</td>
<td>0</td>
</tr>
</tbody>
</table>

The pos_count and neg_count estimates are still statistically significant, which is reassuring, and the new estimate suggests that incrementing the vintage by one year increases the predicted number of points by 0.0950. But assuming a linear relationship between points and vintage is too crude. It’s better to allow each vintage year to have its own idiosyncratic effect:

```r
wine_model_2 <-
wine_tib_2 %>%
  lm(
    points ~ word_count + pos_count + neg_count + as.factor(vintage),
    data = .
  )

wine_model_2 %>%
tidy
```
> # A tibble: 26 x 5
> term          estimate std.error statistic  p.value
> <chr>          <dbl>    <dbl>     <dbl>     <dbl>
> 1 (Intercept)  81.2      0.362    224.        0
> 2 word_count   0.173     0.000919 188.        0
> 3 pos_count   0.275      0.00459  60.0        0
> 4 neg_count   -0.114     0.00873  -13.1      4.64e-39
> 5 as.factor(vintage)1996 -0.590  0.477     -1.24  2.15e- 1
> 6 as.factor(vintage)1997  0.0470 0.389     0.121  9.04e- 1
> 7 as.factor(vintage)1998 -1.18    0.377     -3.14  1.71e- 3
> 8 as.factor(vintage)1999 -0.920  0.375     -2.46  1.41e- 2
> 9 as.factor(vintage)2000 -0.933  0.373     -2.50  1.22e- 2
>10 as.factor(vintage)2001 -0.913  0.374     -2.44  1.46e- 2
> # ... with 16 more rows

Converting `vintage` from a number to a factor changes the statistical model from

\[
\text{points} = \beta_0 + \beta_1 \cdot \text{word_count} + \beta_2 \cdot \text{pos_count} + \beta_3 \cdot \text{neg_count} + \beta_4 \cdot \text{vintage} + \epsilon,
\]

to

\[
\text{points} = \beta_0 + \beta_1 \cdot \text{word_count} + \beta_2 \cdot \text{pos_count} + \beta_3 \cdot \text{neg_count} + \beta_{1996} \cdot (\text{vintage} == 1996) + \cdots + \beta_{2017} \cdot (\text{vintage} == 2017) + \epsilon.
\]

In the expression above, `vintage == 1996`, ..., `vintage == 2017` are called dummy variables because they’re only smart enough to take two values: 0 or 1. These variables enable us to model the incremental effect of a given `vintage`, relative to the benchmark year of 1995 (R benchmarks against 1995 because that is the first level of `as.factor(vintage)`). For example, if the `vintage` is 1995 then all the vintage dummy variables are zero (or `FALSE`), in which case our statistical model boils down to:

\[
\text{points} = \beta_0 + \beta_1 \cdot \text{word_count} + \beta_2 \cdot \text{pos_count} + \beta_3 \cdot \text{neg_count} + \beta_{1996} \cdot 0 + \cdots + \beta_{2017} \cdot 0 + \epsilon.
\]

And if the `vintage` is 2011 then then the `vintage == 2011` dummy variable is one and all the other `vintage` dummy variables are zero, in which case our statistical model boils down to:

\[
\text{points} = \beta_0 + \beta_1 \cdot \text{word_count} + \beta_2 \cdot \text{pos_count} + \beta_3 \cdot \text{neg_count} + \beta_{1996} \cdot 0 + \cdots + \beta_{2010} \cdot 0 + \beta_{2011} \cdot 1 + \beta_{2012} \cdot 0 + \cdots + \beta_{2017} \cdot 0 + \beta_{2011} \cdot \epsilon.
\]
As you see, the 2011 model is the same as the 1995 model, but with an extra $\beta_{2011}$ offset term. Accordingly, $\hat{\beta}_{2011} = 0.619$ is our estimate for the expected point differential between a 2011 wine and an analogous 1995 wine. In other words, the $\hat{.fitted}$ value of a 2011 wine would be $\hat{\beta}_{2011} = 0.619$ points higher than that of a 1995 wine with the same word_count, pos_count, and neg_count values. And we can see this when we compare the $\hat{.fitted}$ values of "Marchesi di Barolo 1995 Barolo" and "Artadi 2011 Pagos Viejos (Rioja)", both of which have word_count = 60, pos_count = 6, and neg_count = 1:

```r
wine_model_2 %>%
  augment(
    newdata =
      wine_tib_2 %>%
        filter(
          title %in%
            c(
              "Marchesi di Barolo 1995 Barolo",
              "Artadi 2011 Pagos Viejos (Rioja)"
            )
        )
  )
%>%
  select(title, word_count, pos_count, neg_count, .fitted)
%>%
  mutate(.dif = .fitted - min(.fitted))
```

> # A tibble: 2 x 6
> title word_count pos_count neg_count .fitted .dif
> <chr> <int> <int> <int>    <dbl> <dbl>
> 1 Marchesi di Barolo 1995 Barolo  53     6    1   91.9 0.0467
> 2 Artadi 2011 Pagos Viejos (Rioja) 50     6    1   91.9 0

The example above uses the `newdata` option of `augment()` to calculate the $\hat{.fitted}$ values of the two relevant rows. The `newdata` option enables us to calculate the $\hat{.fitted}$ value associated with any combination of word_count, pos_count, neg_count, and vintage. For example, the following predicts the points value of a hypothetical review with word_count = 60, pos_count = 10, neg_count = 5, and vintage = 1999:

```r
wine_model_2 %>%
  augment(
    newdata =
      tibble(
        word_count = 60, pos_count = 10,
        neg_count = 5, vintage = 1999
      )
  )
```
And the following predicts the points value of a hypothetical review with \( \text{word\_count} = 0, \text{pos\_count} = 1000, \text{neg\_count} = -1, \) and \( \text{vintage} = 2000 \):

\[
\begin{align*}
\text{wine\_model\_2} \%>\% \\
\text{augment(} \\
\quad \text{newdata =} \\
\quad \text{tibble(} \\
\quad \quad \text{word\_count} = 0, \text{pos\_count} = 1000, \\
\quad \quad \text{neg\_count} = -10, \text{vintage} = 2000 \\
\quad ) \\
\text{)} \\
\end{align*}
\]

Of course it would be impossible for a review to comprise zero words, 1000 positive words, and -10 negative words, but our statistical model doesn’t know this. It simply calculates

\[
\begin{align*}
\text{.fitted} &= \hat{\beta}_0 + \hat{\beta}_1 \cdot 0 + \hat{\beta}_2 \cdot 1000 + \hat{\beta}_3 \cdot (-10) + \hat{\beta}_{2000} \\
&= 81.8 + 0.134 \cdot 0 + 0.270 \cdot 1000 - 0.0902 \cdot (-10) - 0.829 \\
&= 351
\end{align*}
\]

**Exercise 5.17.** Use the `newdata` option of `augment()` to calculate the mean().fitted value of the following 10,000 wine review scenarios:

\[
\begin{align*}
\text{wine\_review\_scenarios} <- \\
\text{expand\_grid(} \\
\quad \text{word\_count} = 100:109, \\
\quad \text{pos\_count} = 0:9, \\
\quad \text{neg\_count} = 0:9, \\
\quad \text{vintage} = 2000:2009 \\
\quad ) \\
\end{align*}
\]

**Exercise 5.18.** We will now show that there was consistent grade inflation—or consistent wine improvement—between 1998 and 2014.
• tidy() wine_model_2 and filter() the result to keep only the rows in which you can
str_detect() the word "vintage" in the term column.
• mutate() the result to redefine term = term %>% str_remove("as.factor\(\(vintage\)\))
  %>% strtoi.
  – This step converts character strings "as.factor(vintage)1996", ..., "as.factor(vintage)2017"
• ggplot() the result with x = term by y = estimate.
  – This plot illustrates the point estimates associated with each vintage. You should
    see a systematic increase in the estimate values between 1998 and 2014. For
    example, the estimate corresponding to vintage == 2014 is 1.02, which indicates
    that the score of a 2014 wine is a full point higher, on average, than that of a
    comparable 1995 wine (the benchmark year).

We will now add province, taster_name, and variety as additional control variables. Specifically, we will assign a regression coefficient to each value of each of these variables that has at least 1000 observations. For example, we will add dummy variable taster_name == "Matt Kettmann" to our regression equation because there are 5730 observations with this reviewer, but we will not add dummy variable taster_name == "Fiona Adams" because there are only 24 observations with this reviewer. We will disregard the values with fewer than 1000 observations by setting them to "Other":

```r
wine_tib_3 <- wine_tib_2 %>%
  group_by(province) %>%
  mutate(province = ifelse(n() >= 1000, province, "Other")) %>%
  group_by(taster_name) %>%
  mutate(taster_name = ifelse(n() >= 1000, taster_name, "Other")) %>%
  group_by(variety) %>%
  mutate(variety = ifelse(n() >= 1000, variety, "Other")) %>%
  ungroup %>%
  mutate(across(c(province, taster_name, variety), ~.x %>% fct_relevel("Other"))
)
```

The final mutate() makes "Other" the first factor level of province, taster_name, and variety, which will make it the benchmark value in our regression (like how 1995 is the benchmark year).
We're now ready to run our regression:

```r
wine_model_3 <-
wine_tib_3 %>%
  lm(
    points ~
    word_count +
    pos_count + neg_count +
    as.factor(vintage) +
    province + taster_name + variety, #don't need to as.factor() these
    data = .
  )
```

As you can see, the `pos_count` and `neg_count` values are still statistically significant. Overall, this regression has 86 independent variables:

```r
wine_model_3 %>% tidy %>%
  head
```

<table>
<thead>
<tr>
<th>term</th>
<th>estimate</th>
<th>std.error</th>
<th>statistic</th>
<th>p.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>81.6</td>
<td>0.629</td>
<td>130.</td>
<td>0</td>
</tr>
<tr>
<td>word_count</td>
<td>0.162</td>
<td>0.00104</td>
<td>156.</td>
<td>0</td>
</tr>
<tr>
<td>pos_count</td>
<td>0.242</td>
<td>0.00502</td>
<td>48.2</td>
<td>0</td>
</tr>
<tr>
<td>neg_count</td>
<td>-0.0408</td>
<td>0.00901</td>
<td>-4.52</td>
<td>0.0000613</td>
</tr>
<tr>
<td>as.factor(vintage)1996</td>
<td>-0.328</td>
<td>0.740</td>
<td>-0.444</td>
<td>0.657</td>
</tr>
<tr>
<td>as.factor(vintage)1997</td>
<td>-0.971</td>
<td>0.676</td>
<td>-1.44</td>
<td>0.151</td>
</tr>
</tbody>
</table>

As you can see, the `pos_count` and `neg_count` values are still statistically significant. Overall, this regression has 86 independent variables:

```r
wine_model_3 %>% tidy %>%
pull(term)
```

```r
> [1] "(Intercept)"                  "word_count"
> [3] "pos_count"                    "neg_count"
> [5] "as.factor(vintage)1996"        "as.factor(vintage)1997"
> [7] "as.factor(vintage)1998"        "as.factor(vintage)1999"
> [9] "as.factor(vintage)2000"        "as.factor(vintage)2001"
> [13] "as.factor(vintage)2004"        "as.factor(vintage)2005"
> [15] "as.factor(vintage)2006"        "as.factor(vintage)2007"
> [17] "as.factor(vintage)2008"        "as.factor(vintage)2009"
> [19] "as.factor(vintage)2010"        "as.factor(vintage)2011"
```
Indeed, here's our statistical model:
points $= \beta_0 + \beta_1 \cdot \text{word\_count} + \beta_2 \cdot \text{pos\_count} + \beta_3 \cdot \text{neg\_count} + \beta_{1996} \cdot (\text{vintage} == 1996) + \cdots + \beta_{2017} \cdot (\text{vintage} == 2017) + \beta_{\text{Alsace}} \cdot (\text{province} == \text{"Alsace"}) + \cdots + \beta_{\text{Veneto}} \cdot (\text{province} == \text{"Veneto"}) + \beta_{\text{Anna Lee C. Iijima}} \cdot (\text{taster\_name} == \text{"Anna Lee C. Iijima"}) + \cdots + \beta_{\text{Virginie Boone}} \cdot (\text{taster\_name} == \text{"Virginie Boone"}) + \beta_{\text{Bordeaux-style Red Blend}} \cdot (\text{variety} == \text{"Bordeaux-style Red Blend"}) + \cdots + \beta_{\text{Zinfandel}} \cdot (\text{variety} == \text{"Zinfandel"}) + \epsilon.$

This specification's dummy variables provide idiosyncratic vintage, province, and taster names.
taster_name offsets. For example, the $\hat{\beta}_{\text{Merlot}} = -0.472$ estimate indicates that we would expect -0.472 fewer points from a variety == "Merlot" wine than from a variety == "Other" wine that matches on all other dimensions. (In this case, we benchmark against variety == "Other".)

**Exercise 5.19.** Of the wine tasters with at least 1000 reviews, which is the harshest grader, according to wine_model_3?

**Exercise 5.20.** We will now depict the relationship between points and price. When we model the relationship as linear, we estimate that an extra dollar in price buys and additional 0.0175 points, in expectation:

```r
wine_tib_3 %>%
  lm(
    points ~ word_count + pos_count + neg_count + as.factor(vintage) + province + taster_name + variety + price,
    data = .
  ) %>%
  tidy %>%
  filter(term == "price")
```

> # A tibble: 1 x 5
>   term estimate std.error statistic p.value
>   <chr>   <dbl>     <dbl>      <dbl>    <dbl>
> 1 price   0.0175  0.000184   95.2       0

But this linear specification is overly restrictive, as it gives us only one degree of freedom to capture the effect of price. In contrast, we will model the effect with 99 degrees of freedom by turning price into a factor with 100 levels and giving all but the first baseline level its own parameter.

- `mutate()` wine_tib_3 to define `price_group = price %>% ntile(100) %>% as.factor`. This new factor variable divides the sample into 100 groups, based on the percentile of price. For example, the observations with the lowest 1% of price values have `price_group = "1"` and the observations with the highest 1% of price values have `price_group = "100"`.  
- Run the regression above, but with `price_group` instead of `price`.  
- `tidy()` and `filter()` the result down to the rows in which you can `str_detect()` "price_group" in the term column.  
- Pipe the resulting tibble into the following assembly line, to create the desired ggplot().
mutate(
  term = #turn term values into integers
  term %>%
    str_remove("price_group") %>%
    strtoi,
  lower_lim = estimate - qnorm(.975) * std.error,
  upper_lim = estimate + qnorm(.975) * std.error
) %>%
ggplot +
aes(
  x = term,
  y = estimate,
  ymin = lower_lim,
  ymax = upper_lim
) +
geom_point() + #this layer plots the estimate values
geom_errorbar() + #this layer plots estimates' 95% confidence intervals
theme_minimal() +
labs(
  x = "Price Percentile",
  y = "Expected Point Surplus, Relative to 1st-Percentile Wines"
)

- Use `str_remove("price_group") %>% strtoi` to convert the `term` values from character strings to integers.
- Add columns `lower_lim = estimate - qnorm(.975) * std.error` and `upper_lim = estimate + qnorm(.975) * std.error`, which characterize the estimate values' 95% confidence intervals.
- `ggplot()` the results, with `geom_point()` and `geom_errorbar()` layers, evaluated under `x = term`, `y = estimate`, `ymin = lower_lim`, and `ymax = upper_lim`.
  - The points denote the estimate values, and the error bars depict their 95% confidence intervals.
  - The plot should illustrate that the expected number of points systematically increases with the price percentile. For example, wines in the 25th, 50th, and 75th price percentiles are expected to receive two, three, and four more points than otherwise equivalent wines in the first price percentile.
  - Use `theme_minimal()` and add `labs(x = "Price %", y = "Expected Additional Points, Relative to 1st %")`.

Exercise 5.21. My friend Emily Harbaugh\(^6\) reviews wines on Instagram\(^7\). So I asked her

\(^6\)https://www.instagram.com/EmilyHarbaugh/
\(^7\)https://www.instagram.com/glougang/
to dictate to me the following two reviews.

- 2016 California Merlot, Bota Box brand:

```r
review_1 <-
str_c(
  "Smells alcholic. Yeah, it definently does have notes",
  "of alchohol. Hold on. It's like pretty jucy. It's",
  "light; its not super tannic, which is good because",
  "I don't like high tannins. I'm getting like a lot of",
  "red fruit---hold on, I'm still drinking it. It's like",
  "light---not too full. You know like sometimes you're",
  "'like 'wow that's a lot a fruit'? But this one's like",
  "lighter. It's actually not that bad. because the one",
  "'from Cosco tastes a lot like alchohol ---yeah, Costco",
  "has a $14.98 bottle of organic red wine, but it",
  "tastes more like alchohol than this one. Yeah, so, I",
  "'feel like this is not a bad boxed wine to have on hand.'",
  sep = " "
)
```

- 2017 Argentinian Malbec, National Origin brand (i.e., the Costco wine referenced in `review_1`):

```r
review_2 <-
str_c(
  "So this one has a heavier mouth feel. And it definatly",
  "smells more like alchohol, so I don't really enjoy",
  "that about that. And it seems like a little more tannic",
  "and oaky, because it feels more like there's a fuzz on",
  "my tounge after I'm done drinking it. Definatly get like",
  "a lot of dark fruits with this one---like red and black",
  "fruits. I would say it like doesn't leave like a great",
  "taste in my mouth, but it's alright for a boxed wine",
  "I would suppose. It's like, there's something like",
  "kind of spicy. I get a taste like, the one I had",
  "before was like lighter and jucier and more enjoyable",
  "when drinking than this one. This one's definately like;",
  "this one definatley smells like alchohol, and I don't",
  "like that. Does that work, do you need any other",
```
Overall, here’s everything I know about the two wines:

```
emily_wine <-
tribble(
  ~review_id, ~brand, ~vintage, ~province, ~taster_name, ~variety, ~description,
  1, "Bota Box", "2016", "California", "Other", "Merlot", review_1,
  2, "National Origin", "2017", "Mendoza Province", "Other", "Malbec", review_2
)
```

Both wines have `taster_name = "Other"` because Emily Harbaugh is not represented in our sample, and the latter wine has `province = "Mendoza Province"` because that’s the wine country of Argentina. We’ll now use our statistical model to anticipate which review would receive a higher score, if they were published by Wine Enthusiast.

- Use the methodology used to create `wine_tib_2` to create `emily_wine_2`, which equals `emily_wine`, but with extra columns `word_count`, `pos_count`, and `neg_count`.
- Pipe `wine_model_3` into `augment()`, with the `newdata = emily_wine_2` option.

This exercise serves as a cautionary tale about extrapolating a statistical model beyond the scope of its training data: since Emily’s reviews deviate drastically from those found on the Wine Enthusiast website the `.fitted` values corresponding to her reviews are nonsensical. For example, the model assigned more points to the less-preferred Costco wine because its review was longer, with more positive words. In fact, this review had such large `word_count` and `pos_count` values that its `.fitted` value exceeds 100, which is an impossible score on Wine Enthusiast’s 100-point scale. But, of course, our linear model isn’t aware of this 100-point ceiling. To accommodate such a cap we must switch from a linear model to a generalized linear model, which we’ll cover in the next chapter.

### 5.2.5 Manipulating Tibbled Lists

`wine_model_3` accounts for the reviewers’ average severity. For example, all other things equal, we would expect \( \hat{\beta}_{Anna Lee C. Iijima} - \hat{\beta}_{Kerin O'Keefe} = 0.944 - 0.843 = 0.102 \) more points from a review by Anna Lee C. Iijima than one by Kerin O’Keefe. However, our model does
not account for the myriad other ways that reviewers could differ. For example, Roger Voss may have a penchant for Rieslings, or Lauren Buzzeo may be biased against California reds, or a lengthy review by Jim Gordon may be more auspicious than one from Joe Czerwinski, or the negative words of Michael Schachner may be more damning that those of Virginie Boone. To capture their idiosyncrasies, we will create an independent statistical model for each of the 13 reviewers in `wine_tib_3`.

We could create these 13 models with 13 distinct `lm()` calls:

```r
# Model 1
model_anna <-
wine_tib_3 %>%
  filter(taster_name == "Anna Lee C. Iijima") %>%
  lm(
    points ~ variety + as.factor(vintage),
    data = .
  )

# ...

# Model 13
model_virginie <-
wine_tib_3 %>%
  filter(taster_name == "Virginie Boone") %>%
  lm(
    points ~ variety + as.factor(vintage),
    data = .
  )
```

But this approach is cumbersome. So we’ll develop a more elegant solution, based on nested tibbles. We’ll start by structuring our data by `taster_name`.

**Exercise 5.22.** `nest()` `wine_tib_3` by `sample = -taster_name` to create a tibble with two columns: `taster_name` and `sample`, the latter of which stores a collection of tibbles that house all the data corresponding to a given reviewer.

- The row with `taster_name = "Roger Voss"` should have a 23560 x 18 tibble in its `sample` column.
- `filter()` out the rows with `taster_name %in% c("Other", NA).
- `<-` the result into `wine_tib_4`.

`wine_tib_4` has a `sample` of data for each of the 13 reviewers:
Now we could pluck() out each sample for individual analysis. For example, we could get the distinct() taster_twitter_handle values of each sample with:

```r
wine_tib_4 %>%
  pluck("sample", 1) %>%
  distinct(taster_twitter_handle)
```

> # A tibble: 1 x 1
> taster_twitter_handle
> <chr>
> 1 @kerinokeefe

#...

```r
wine_tib_4 %>%
  pluck("sample", 13) %>%
  distinct(taster_twitter_handle)
```

> # A tibble: 1 x 1
> taster_twitter_handle
> <chr>
> 1 @suskostrzewa
But this approach would require calling `distinct()` 13 times. An easier way to get the `distinct()` `taster_twitter_handle` values is to implement a `rowwise()` `summarise()`:

```
#return the taster_name and the taster_twitter_handle
wine_tib_4 %>%
  rowwise(taster_name) %>% #putting taster_name inside of rowwise() tells
  summarise(              #R to report taster_name in the summarise()
    sample %>%
      distinct(taster_twitter_handle)
  ) %>%
  head(4)
```

> # A tibble: 4 x 2
> # Groups: taster_name [4]
> taster_name  taster_twitter_handle
> <fct>        <chr>
> 1 Kerin O'Keefe @kerinokeefe
> 2 Roger Voss   @vossroger
> 3 Paul Gregutt @paulgwine
> 4 Michael Schachner @wineschach

This example illustrates a useful technique: splitting `wine_tib_4` `rowwise()` enables `summarise()` to access the tibbles inside of the `sample` list column. Hence, although it's a list of tibbles, the code inside of `summarise()` treats `sample` as if it were an individual tibble. Note, this would not work if we split the data with a traditional `group_by()` in which case `sample` would appear to `summarise()` like a list. Specifically, when we divide `wine_tib_4` into traditional `group_by()` groups then each group's `sample` is a list containing a tibble, from the perspective of `summarise()`; in contrast, when we divide `wine_tib_4` into `rowwise()` groups then each groups's `sample` is a tibble, from the perspective of `summarise()`.

Let me illustrate with an example:

```
tib_A <- tibble(date_col = today())

tib_B <-
tibble(  
  number_col = 1:3,
  letter_col = c("a", "b", "c")
)

tiny_tib <- tibble(
```

373
```r
x = c("A", "B"),
y = list(first_list_element = tib_A, second_list_element = tib_B)
)
tiny_tib

> # A tibble: 2 x 2
> x  y
> <chr> <named list>
> 1 A <tibble [1 x 1]>
> 2 B <tibble [3 x 2]>

Splitting tiny_tib `rowwise()` divides the `y` list column into tibbles `tib_A` and `tib_B`, which get passed into `summarise()` one at a time in the guise of `y`. Accordingly, under a `rowwise()` split, `summarise()` considers the first value of `y` to be a tibble with one column named "date_col" and considers the second value of `y` to be a tibble with columns named "number_col" and "letter_col":

```r
tiny_tib %>%
  rowwise(x) %>%
  summarise(y_is_a_tibble = is_tibble(y))
```

> # A tibble: 2 x 2
> # Groups: x [2]
> x y_is_a_tibble
> <chr> <lgl>
> 1 A TRUE
> 2 B TRUE

```r
tiny_tib %>%
  rowwise(x) %>%
  summarise(names_of_y = names(y))
```

> # A tibble: 3 x 2
> # Groups: x [2]
> x names_of_y
> <chr> <chr>
> 1 A date_col
> 2 B number_col
> 3 B letter_col
```
But performing a `group_by()` split divides \( y \) into lists `list(first_list_element = tib_A)` and `list(second_list_element = tib_B)`, which get passed into `summarise()` one at a time. Hence, under a `group_by()` split, `summarise()` considers the first value of \( y \) to be a list with one element named "first_list_element" and considers the second value of \( y \) to be a list with one element named "second_list_element":

```
tiny_tib %>%
  group_by(x) %>%
  summarise(y_is_a_tibble = is_tibble(y))
```

> # A tibble: 2 x 2
>   x   y_is_a_tibble
>  <chr> <lgl>
> 1 A    FALSE
> 2 B    FALSE

```
tiny_tib %>%
  group_by(x) %>%
  summarise(y_is_a_list = is_list(y))
```

> # A tibble: 2 x 2
>   x   y_is_a_list
>  <chr> <lgl>
> 1 A    TRUE
> 2 B    TRUE

```
tiny_tib %>%
  group_by(x) %>%
  summarise(y_is_a_list = names(y))
```

> # A tibble: 2 x 2
>   x   y_is_a_list
>  <chr> <chr>
> 1 A    first_list_element
> 2 B    second_list_element

The upshot is that splitting a tibble of tibbles `rowwise()` enables us to work with the nested tibbles directly. For example, we can apply any tibble operation to the 13 tibbles in `sample`, so long as we `rowwise()` split `wine_tib_4` before hand:
#get first three rows of each tibble in sample

```r
wine_tib_4 %>%
  rowwise(taster_name) %>%
  summarise(sample %>% head(3))
```

> # A tibble: 39 x 19
> # Groups: taster_name [13]
> taster_name review_id word_count pos_count neg_count pos_minus_neg points
> <fct>       <int>    <int>    <int>     <int>          <int>    <int>
> 1 Kerin O'Keefe 1         22       1       1         0        87
> 2 Kerin O'Keefe 7         28       4       0         4        87
> 3 Kerin O'Keefe 14        34       1       1         0        87
> 4 Roger Voss     2         35       2       0         2        87
> 5 Roger Voss     8         20       1       0         1        87
> 6 Roger Voss     10        26       4       0         4        87
> 7 Paul Gregutt   3         25       3       0         3        87
> 8 Paul Gregutt   5         38       3       2         1        87
> 9 Paul Gregutt   22        37       3       0         3        87
>10 Michael Schach~ 6         38       3       1         2        87
> # ... with 29 more rows, and 12 more variables: title <chr>, description <chr>,
> # taster_twitter_handle <chr>, designation <chr>, variety <fct>,
> # region_1 <chr>, province <fct>, country <chr>, winery <chr>, price <int>,
> # region_2 <chr>, vintage <int>

#get 10th row of each tibble in sample

```r
wine_tib_4 %>%
  rowwise(taster_name) %>%
  summarise(sample %>% slice(10)) %>%
  head(4)
```

> # A tibble: 4 x 19
> # Groups: taster_name [4]
> taster_name review_id word_count pos_count neg_count pos_minus_neg points
> <fct>       <int>    <int>    <int>     <int>          <int>    <int>
> 1 Kerin O'Keefe 73         32       1       0         1        86
> 2 Roger Voss    66         32       3       0         3        86
> 3 Paul Gregutt  252        34       2       0         2        85
> 4 Michael Schach~ 82        45       2       2         0        86
> # ... with 12 more variables: title <chr>, description <chr>,
> # taster_twitter_handle <chr>, designation <chr>, variety <fct>,
> # region_1 <chr>, province <fct>, country <chr>, winery <chr>, price <int>,
> # region_2 <chr>, vintage <int>
# count number of distinct price points in each tibble in sample
wine_tib_4 %>%
  rowwise(taster_name) %>%
  summarise(
    sample %>%
      summarise(number_of_prices = n_distinct(price))
  ) %>%
head(4)

> # A tibble: 4 x 2
> # Groups:  taster_name [4]
> taster_name number_of_prices
> <fct>           <int>
> 1 Kerin O'Keefe  185
> 2 Roger Voss     329
> 3 Paul Gregutt   95 
> 4 Michael Schachner 178

# get min and max "Red Blend" scores from each tibble in sample
wine_tib_4 %>%
  rowwise(taster_name) %>%
  summarise(
    sample %>%
      filter(variety == "Red Blend") %>%
      summarise(
        min_score = min(points, na.rm = TRUE),
        max_score = max(points, na.rm = TRUE)
      )
  ) %>%
head(4)

> # A tibble: 4 x 3
> # Groups:  taster_name [4]
> taster_name min_score max_score
> <fct>       <int>   <int>
> 1 Kerin O'Keefe   83     99
> 2 Roger Voss      82     95
> 3 Paul Gregutt    80     97
> 4 Michael Schachner 80     96

This rowwise() trick also works with mutate():
# Add the number of rows

```r
code{wine_tib_4} %>%
  rowwise
  mutate(row_count = sample %>% nrow)
```

> # A tibble: 13 x 3
> # Rowwise:
> taster_name sample row_count
> <fct> <list> <int>
> 1 Kerin O'Keefe <tibble [9,697 x 18]> 9697
> 2 Roger Voss <tibble [23,560 x 18]> 23560
> 3 Paul Gregutt <tibble [8,868 x 18]> 8868
> 4 Michael Schachner <tibble [14,046 x 18]> 14046
> 5 Anna Lee C. Iijima <tibble [4,017 x 18]> 4017
> 6 Virginie Boone <tibble [8,708 x 18]> 8708
> 7 Matt Kettmann <tibble [5,730 x 18]> 5730
> 8 Sean P. Sullivan <tibble [4,461 x 18]> 4461
> 9 Jim Gordon <tibble [3,766 x 18]> 3766
> 10 Joe Czerwinski <tibble [4,766 x 18]> 4766
> 11 Anne Krebiehl MW <tibble [3,290 x 18]> 3290
> 12 Lauren Buzzee <tibble [1,700 x 18]> 1700
> 13 Susan Kostrzewa <tibble [1,023 x 18]> 1023
```

# Add first description

```r
code{wine_tib_4} %>%
  rowwise
  mutate(first_review =
    sample %>%
    pull(description) %>%
    first
  )
```

> # A tibble: 13 x 3
> # Rowwise:
> taster_name sample first_review
> <fct> <list> <chr>
> 1 Kerin O'Keefe <tibble [9,697 x 18]> Aromas include tropical fruit, bro-
> 2 Roger Voss <tibble [23,560 x 18]> This is ripe and fruity, a wine th-
> 3 Paul Gregutt <tibble [8,868 x 18]> Tart and snappy, the flavors of li-
> 4 Michael Schachner <tibble [14,046 x 18]> Blackberry and raspberry aromas sh-
> 5 Anna Lee C. Iijima <tibble [4,017 x 18]> Savory dried thyme notes accent su-
> 6 Virginie Boone <tibble [8,708 x 18]> Soft, supple plum envelopes an oak-
> 7 Matt Kettmann <tibble [5,730 x 18]> Building on 150 years and six gene-
> 8 Sean P. Sullivan <tibble [4,461 x 18]> Aromas of cranberry, barrel spice ~
> 9 Jim Gordon <tibble [3,766 x 18]> Very deep in color and spicy-smoky-
> 10 Joe Czerwinski <tibble [4,461 x 18]> This medium-bodied Chardonnay feat-
> 11 Anne Krebiehl MW <tibble [4,461 x 18]> Freshness characterizes the nose: ~
> 12 Lauren Buzzeo <tibble [1,700 x 18]> This is a concentrated, fairly ful-
> 13 Susan Kostrzewa <tibble [1,023 x 18]> Orange blossom and flowers lead on-

#add the mean and median review score
wine_tib_4 %>%
  rowwise %>%
  mutate(
    sample %>%
    summarise(
      mean_score = mean(points),
      median_score = median(points)
    )
  )

> # A tibble: 13 x 4
> # Rowwise:
> taster_name sample mean_score median_score
> <fct> <list> <dbl> <dbl>
> 1 Kerin O'Keefe <tibble [9,697 x 18]> 88.9 89
> 2 Roger Voss <tibble [23,560 x 18]> 88.7 88
> 3 Paul Gregutt <tibble [8,868 x 18]> 89.1 89
> 4 Michael Schachner <tibble [14,046 x 18]> 86.9 87
> 5 Anna Lee C. Iijima <tibble [4,017 x 18]> 88.4 88
> 6 Virginie Boone <tibble [8,708 x 18]> 89.2 90
> 7 Matt Kettmann <tibble [5,730 x 18]> 90.1 90
> 8 Sean P. Sullivan <tibble [4,461 x 18]> 88.8 89
> 9 Jim Gordon <tibble [3,766 x 18]> 88.6 89
> 10 Joe Czerwinski <tibble [4,766 x 18]> 88.5 89
> 11 Anne Krebiehl MW <tibble [3,290 x 18]> 90.6 90
> 12 Lauren Buzzeo <tibble [1,700 x 18]> 87.7 87
> 13 Susan Kostrzewa <tibble [1,023 x 18]> 86.6 87

The last `mutate()` doesn't need an =, because its nested `summarise()` call returns a tibble with the desired column names.

This `rowwise()` trick also works with `filter()`:
```r
# keep reviewers with an average score of at least 90
wine_tib_4 %>%
  rowwise %>%
  filter(
    sample %>%
      summarise(mean(points) > 90) # this step returns a logical
  )
```

```r
# A tibble: 2 x 2
# Rowwise:
> taster_name sample
> <fct> <list>
> 1 Matt Kettmann <tibble [5,730 x 18]>
> 2 Anne Krebiehl MW <tibble [3,290 x 18]>
```

```r
# keep reviewers with at least 100 California wine reviews
wine_tib_4 %>%
  rowwise %>%
  filter(
    sample %>%
      filter(
        province %>%
          str_detect("California")
      ) %>%
      summarise(n() >= 100)
  )
```

```r
# A tibble: 4 x 2
# Rowwise:
> taster_name sample
> <fct> <list>
> 1 Paul Gregutt <tibble [8,868 x 18]>
> 2 Virginie Boone <tibble [8,708 x 18]>
> 3 Matt Kettmann <tibble [5,730 x 18]>
> 4 Jim Gordon <tibble [3,766 x 18]>
```

**Exercise 5.23.** Use a rowwise() filter() to limit wine_tib_4 down to the reviewers who used at least five times as many positive words as negative words.

- Nest a summarise() within the filter() to evaluate \( \text{sum(pos\_count)} > 5 \times \text{sum(neg\_count)} \) for each tibble in sample.
• ungroup() the result to remove the rowwise() split.

**Exercise 5.24.** Use a rowwise() mutate() to add the a column to wine_tib_4 called median_length that reports the median() number of characters in description.

• Nest a summarise() with the mutate() to define median_length = description %>% str_count %>% median for each tibble in sample.
• ungroup() the result to remove the rowwise() split.

**Exercise 5.25.** Use a rowwise() summarise() to calculate the correlation between each reviewer’s pos_count and neg_count.

• Nest a summarise() with the summarise() to calculate pos_neg_cor = cor(pos_count, neg_count) for each tibble in sample.
• ungroup() the result to remove the rowwise() split.

### 5.2.6 Nested Tibble Regressions

We will now use a rowwise() mutate() to run an independent lm() for each reviewer:

```r
wine_tib_5 <-
wine_tib_4 %>%
rowwise %>%
mutate(
  lms =
  sample %>%
  lm(
    points ~ word_count + pos_count + neg_count,
    data = .
  ) %>%
  list
) %>%
ungroup

wine_tib_5
```

> # A tibble: 13 x 3
>   taster_name sample  lms
>   <fct>      <list>  <list>
> 1 Kerin O'Keefe <tibble [9,697 x 18]> <lm>
> 2 Roger Voss   <tibble [23,560 x 18]> <lm>
> 3 Paul Gregutt <tibble [8,868 x 18]>  <lm>
> 4 Michael Schachner <tibble [14,046 x 18]> <lm>

381
The code in the `mutate()` above ends in `list()`, because only a list column can store a collection of linear models.

Unfortunately, if you recall, `lm()` outputs an ugly sort of list. So we'll now use a `rowwise()` `mutate()` to `tidy()` to each element of the `lms` list column:

```r
wine_tib_6 <- wine_tib_5 %>%
  rowwise %>%
  mutate(lms_tidy = lms %>% tidy %>% list) %>%
  ungroup
wine_tib_6
```

> # A tibble: 13 x 4
> taster_name sample lms lms_tidy
> <fct> <list> <list> <list>
>  1 Kerin O'Keefe <tibble [9,697 x 18]> <lm> <tibble [4 x 5]>
>  2 Roger Voss <tibble [23,560 x 18]> <lm> <tibble [4 x 5]>
>  3 Paul Gregutt <tibble [8,868 x 18]> <lm> <tibble [4 x 5]>
>  4 Michael Schachner <tibble [14,046 x 18]> <lm> <tibble [4 x 5]>
>  5 Anna Lee C. Iijima <tibble [4,017 x 18]> <lm> <tibble [4 x 5]>
>  6 Virginie Boone <tibble [8,708 x 18]> <lm> <tibble [4 x 5]>
>  7 Matt Kettmann <tibble [5,730 x 18]> <lm> <tibble [4 x 5]>
>  8 Sean P. Sullivan <tibble [4,461 x 18]> <lm> <tibble [4 x 5]>
>  9 Jim Gordon <tibble [3,766 x 18]> <lm> <tibble [4 x 5]>
> 10 Joe Czerwinski <tibble [4,766 x 18]> <lm> <tibble [4 x 5]>
> 11 Anne Krebiehl MW <tibble [3,290 x 18]> <lm> <tibble [4 x 5]>
> 12 Lauren Buzzeo <tibble [1,700 x 18]> <lm> <tibble [4 x 5]>
> 13 Susan Kostrzewa <tibble [1,023 x 18]> <lm> <tibble [4 x 5]>

Again, we use `list()`, because only a list column can store a collection of tibbles.

**Exercise 5.26.** We will now get the reviewer-specific effects of `pos_count` and `neg_count` on points.
• select() the taster_name and lms_tidy columns of wine_tib_6, and unnest() by lms_tidy.
• filter() the the result down to the estimates that correspond to pos_count or neg_count.
  – You can use str_detect(term, "pos") | str_detect(term, "neg").
  – The resulting tibble should have 26 columns an 6
• To focus on the statistically significant estimates, set estimate = ifelse(p.value < .05, estimate, NA).
  – This step turns the statistically insignificant estimate values to NA.
• pivot_wider() the result into a tibble with columns taster_name, pos_count, and neg_count, the latter two of which report these variables’ statistically significant co-efficient estimates.
  – Use id_cols = taster_name to drop the superfluous columns.

The pos_count estimates are almost always significantly positive, but the neg_count estimates are rarely significantly negative. Perhaps including variety as a control variable will help. We’ll test this next.

Exercise 5.27. We will now incorporate variety into our reviewer-specific regressions.

• Modify the code that defines wine_tib_5 and wine_tib_6 to (i) run a regression with formula points ~ word_count + pos_count + neg_count + variety for each reviewer and (ii) tidy() the results.
  – Perform these two operations in one functional assembly line that translates the sample column of wine_tib_4 into a list column called lms_tidy.
  – Your tibble should have 13 rows and three columns after this step.
• ungroup() the result to remove the rowwise() split.
• unnest() the tibble, and inspect the estimate values that correspond to neg_count.
• Your estimates should be similar to before.

Exercise 5.28. We will now ascertain who is the most sentimental reviewer—the reviewer whose points are most determined by word_count, pos_count, and neg_count.

• Use a rowwise() mutate() to glance() each linear model in the lms column of wine_tib_6, and save the results in a list column called lms_glance.
• ungroup() the result to remove the rowwise() split.
• unnest() the tibble by lms_glance to append columns r.squared, adj.r.squared, sigma, ..., nobs to wine_tib_6.
• filter() the sample down to the reviewer with the largest r.squared value (i.e., the reviewer whose points are best explained by our sentiment variables.)

**Exercise 5.29.** Suppose the review of your wine comprised 40 words, four of which are negative and none of which are positive. Who would you least want to receive such a review from? In other words, who do you think would award the fewest number of points to a review with word_count = 40, pos_count = 0, and neg_count = 4?

• Use a rowwise() mutate() to augment() each linear model in the lms column of wine_tib_5, with the option newdata = tibble(word_count = 40, pos_count = 0, neg_count = 4). Save the result as a list() column called lms_augment.
• unnest() the resulting tibble by lms_augment.
• filter() the result down to the row with .fitted == min(.fitted).
• select() this row’s taster_name.

5.3 Lab: Supply Chain Proximity

5.3.1 Objective

Since it’s harder to collaborate with far-flung suppliers, Bray et al. [2019a] theorized that geographically dispersed supply chains will produce lower quality goods. They tested this hypothesis with a dataset that tracks the flow of thousands of auto parts through hundreds of component factories and auto assembly plants, and concluded that “supply chain distance lowers product quality.” Specifically, they estimated that “increasing the distance between an upstream component factory and a downstream assembly plant by an order of magnitude increases the component’s expected defect rate by 3.9%.”

The authors buttressed their claim that auto parts are more prone to failure when they’re made in factories that are further from the auto assembly plant with three sets of linear regressions, which they characterize in three plots, reproduced in figures 5.1–5.3. We’ll create an updated version of these plots with an updated sample.

5.3.2 Data

You created a tibble called cars_raw in exercise 1.35 of section 1.3. This tibble has 13 columns:

• assembler, brand, and model: Automaker, car brand, and car model. For example, the 19th row has assembler = "ZHEJIANG GEELY HOLDING GROUP CO.", brand = "VOLVO", and model = "XC70" because the Zhejiang Geely Holding Group owns the Volvo brand, which produces the XC70 station wagon.
• year: Production year of the car. We observe different car models in different years.
• module and part: High-level and low-level description of the car part.
Figure 5.1: Supply Chain Distance and Product Quality by Component Type

These scatter plots illustrate how defects respond to distance. We depict the fitted regression curve with a line, which is black when the slope is significantly different from zero and white otherwise. We depict the regression curves’ 95% confidence intervals with gray bands.

- **defects**: The number of NHTSA defect reports per 10 million vehicle-years of registration. For example, if 1 million cars were each driven for 10 years and in that time there were 5 defects reported to the NHTSA then this car part would have defects = 5. This variable is 10 million times the Defect Rate variable of Bray et al. [2019a].
- **lat_d and lon_d**: Latitude and longitude of the downstream assembly plant—i.e., where the car is built.
- **lat_u and lon_u**: Latitude and longitude of the upstream component factory—i.e., where their car part is made.

The one variable we’re missing is the distance between the upstream and downstream factories. In the end notes of their article, Bray et al. [2019a] explained that “We use Vincenty’s (1975) formula to calculate geographic distances from latitudes and longitudes.” Vincenty’s formula provides the distance between two points on a spheroid, which is a more accurate representation of the Earth than a perfect sphere. We’ll calculate this formula with the `distVincentyEllipsoid()` function of the geosphere package, which you can install and load with the following:

8https://en.wikipedia.org/wiki/Vincenty%27s_formulae
Figure 5.2: Supply Chain Distance and Product Quality by Car Brand
This figure recreates the plots of figure 5.1, by car brand.

```
library(geosphere)

The distVincentyEllipsoid() function takes two vectors, each of which contain a longitude and a latitude, and returns the distance between the points, in meters. For example, the following calculates the number of meters between Moscow and Kiev:

```
moscow_coords <- c(55.7558, 37.6173)  #(longitude, latitude)
kiev_coords <- c(50.4501, 30.5234)

distVincentyEllipsoid(moscow_coords, kiev_coords)
```

> [1] 926430.1

Unfortunately, distVincentyEllipsoid() is a old function, with an outdated syntax. So rather than distVincentyEllipsoid(lon_1, lat_1, lon_2, lat_2), we must invoke this function with the more cumbersome distVincentyEllipsoid(c(lon_1, lat_1), c(lon_2, lat_2)). This awkward notation makes constructing a distance variable somewhat tricky.
Figure 5.3: Mean Effect across the Distribution of Distance
We create this plot in four steps. First, we turn distance into a factor with 100 levels, based on this variable’s percentiles. Second, we reexpress defects in percentile terms. Third, we regress the defects percentiles on the distance, brand, and part factors. Fourth, we plot the 99 distance coefficient estimates and their corresponding 90% confidence intervals.

Exercise 5.30. We will now derive a distance variable from lon_u, lat_u, lon_d, and lat_d.

- Apply a rowwise() mutate() to cars_raw to define distance = distVincentyEllipsoid(c(lon_u, lat_u), c(lon_d, lat_d)).
  - Rather than vectors, the rowwise() grouping ensures that lon_u, lat_u, lon_d, and lat_d are treated as individual numbers, which distVincentyEllipsoid() requires.
- ungroup() the result and drop the lon_u, lat_u, lon_d, and lat_d columns.
- <- the result into cars_tib.
  - This tibble should have mean(distance = 5250756).

5.3.3 Analyze

We’ll gradually construct our version of figures 5.1 and 5.2.

Exercise 5.31. Create a geom_point(size = .1) ggplot of the x = distance and y = defects variables of cars_tib.
• `facet_wrap()` by `module`, with `scales = "free"`.
• Tidy the result with `theme(strip.background = element_blank())` and add axis `labs(x = "Distance" and y = "Defects")`.
• `<-` the result into `base_plot`.

**Exercise 5.32.** We will now streamline our plot to the 40 most common `module` values.

• `cars_tib %>% count(module) %>% filter(row_number(-n) <= 40)` yields the 40 most common `module` levels in `cars_tib`. Use this code and `inner_join()` to select the observations in `cars_tib` that correspond to these 40 most common `module` values.
• Use `fct_relevel(sort)` to alphabetize the levels of `module`.
• `<-` the result into `module_sample`.
• `%>%` `module_sample` to `base_plot` and `+` an additional `geom_smooth(method = "lm")` layer.
• Log-transform the axes with `scale_x_log10()` and `scale_y_log10()`.
• `<-` the result into `module_plot`.

**Exercise 5.33.** We will now make an analogous plot for car brands.

• Create a tibble called `brand_sample` that contains all the data in `cars_tib` that correspond to the 28 most common `brand` values. Use `fct_relevel(sort)` to alphabetize the levels of `brand`. (Do not restrict `brand_sample` to the 40 most common `module` levels.)
• `%>%` `brand_sample` into `module_plot` to create an analogous plot called `brand_plot` that has facets for the 28 most common `brand` values. The facets should be arranged alphabetically.
  – You will have to define a new `facet_wrap()` scheme.

`module_plot` and `brand_plot` differ from figures 5.1 and 5.2 in one key regard: the latter’s trend lines are black when `p.value < 0.05` and are white otherwise. We will now color our trendlines accordingly.

**Exercise 5.34.** We will now add a column to `module_sample` that indicates whether the regression of `defects` on `distance` is statistically significant.

• `nest()` `module_sample` by `module`, so that it has two columns: `module` and `module_tib`, the latter of which is a column of nested tibbles.
  – `module = "EXHAUST"` should correspond to a nested tibble with 683 rows.
• Use a `rowwise()` `mutate()` to apply `lm(log10(defects) ~ log10(distance), data = .) %>% tidy` to each tibble in the `module_tib` column and save the outputs in a list column called `reg_ests`
- This step will run a regression of log() defects on log() distance for each, module, and then it will tidy() the regression outputs.
- We log() the defects and distance values to match our log-transformed ggplot axes.

- **ungroup()** and **unnest()** the tibble by **reg_est**.
- **filter()** the resulting tibble down to the observations with **term == "log10(distance)"**.
- Use **ifelse()** to define significance as a variable that equals "Significant" if **p.value < 0.05** and equals "Insignificant" otherwise.
- **unnest()** the **module_tib** list column and <- the result into **module_sample_2**.

- This tibble should have 23762 rows.

**Exercise 5.35.** We will now color the trend lines of **module_plot** to indicate whether they are statistically significant.

- %+% **module_sample_2** to **module_plot**.
- Color code the trend lines by adding **geom_smooth(method = "lm", aes(color = significance))**.
  - Embedding the **aes()** within a **geom_smooth()** indicates that we want the color coding to apply to the regression lines but not to the scatter plot points.
- **Tidy** the plot with **theme(legend.position = 'top', legend.title = element_blank())**.
- <- the result into **module_plot_2**.

**Exercise 5.36.** We will now create a brand-based version of **module_plot_2**.

- Convert **brand_sample** into **brand_sample_2** in the same way that you converted **module_sample** into **module_sample_2**.
  - In other words, add a significance column to **brand_sample**.
- %+% **brand_sample_2** to **module_plot_2**.
- **facet_wrap()** by **brand**.

We will now create our version of figure 5.3. Whereas the regressions underlying figures 5.1 and 5.2 capture the effect of distance on defects with a single parameter, the regression underlying figure 5.3 captures this effect with 99 parameters. Specifically, it uses the following statistical model:

\[
defects\_{\text{pct}} = \beta_0 + \beta_2 \cdot (\text{dist\_pct} == 2) + \cdots + \beta_{100} \cdot (\text{dist\_pct} == 100) + \beta_{\text{AIRBAGS}} \cdot (\text{part} == \text{"AIRBAGS"}) + \cdots + \beta_{\text{WIRING}} \cdot (\text{part} == \text{"WIRING"}) + \beta_{\text{AUDI}} \cdot (\text{brand} == \text{"AUDI"}) + \cdots + \beta_{\text{VOLVO}} \cdot (\text{brand} == \text{"VOLVO"}),\]

389
where \( \text{dist} \_ \text{pct} = \text{ntile} (\text{distance}, \ n = 100) \) is the distance percentage and \( \text{defects} \_ \text{pct} = \text{percent} \_ \text{rank} (\text{defects}) \) is the defect rate percentage. However, while \( \text{dist} \_ \text{pct} \) and \( \text{defects} \_ \text{pct} \) are both expressed in percentile terms, the former takes integer values between 1 and 100 the latter takes fractional values between 0 and 1 (e.g., it has values 0.4999580, 0.5000000, and 0.5000420).

The dummy variables enable each part, brand, and \( \text{dist} \_ \text{pct} \) to have its own distinct offset. We will plot the \( \text{dist} \_ \text{pct} \) offset estimates, \( \hat{\beta}_2, \ldots, \hat{\beta}_{100} \), which denote the expected \( \text{defects} \_ \text{pct} \) when distance is in its second through hundredth percentiles, relative to the benchmark case in which the distance is in its first percentile. For example, we will get \( \hat{\beta}_{80} = 0.0732 \), which indicates that lengthening a component’s supply chain distance from the first percentile to the 80th percentile would increase its expected \( \text{defects} \_ \text{pct} \) by 0.0732. Our plot will illustrate that the \( \text{dist} \_ \text{pct} \) estimates generally increase from \( \hat{\beta}_2 \) to \( \hat{\beta}_{100} \), and hence that we should expect more defects with larger \( \text{dist} \_ \text{pct} \) values.

Exercise 5.37. We will now illustrate the relationship between the percentile of distance and the percentile of defects with a plot that’s analogous to figure 5.3.

- Use the data in \texttt{cars_tib} to run a regression with \( \text{defects} \_ \text{pct} \sim \text{as.factor} (\text{dist} \_ \text{pct}) + \text{part} + \text{brand} \), where \( \text{defects} \_ \text{pct} = \text{percent} \_ \text{rank} (\text{defects}) \) and \( \text{dist} \_ \text{pct} = \text{ntile} (\text{distance}, \ n = 100) \).
- \texttt{tidy()} the result and \texttt{filter()} out the rows that do not pertain to \( \text{dist} \_ \text{pct} \).
- Use \texttt{str_remove_all(“^[0-9]")} and \texttt{strtof()} to turn term into an integer.
  - This step should replace the character string "as.factor(dist_pct)44" with the number 44.
- Define \( \text{upper} \_ \text{lim} = \text{estimate} + \text{qnorm} (.975) \times \text{std} \_ \text{error} \) and \( \text{lower} \_ \text{lim} = \text{estimate} - \text{qnorm} (.975) \times \text{std} \_ \text{error} \), which characterize the 95% confidence interval of our estimate.
- \texttt{ggplot()} the results, depicting the \texttt{estimate} values with a \texttt{geom_point()} layer and the \texttt{upper} \_ \texttt{lim} and \texttt{lower} \_ \texttt{lim} values with a \texttt{geom_errorbar()} layer.

5.4 Commentary

5.4.1 Tibbles are Lists

Here’s an important but potentially misleading fact: tibbles are actually a special kind of list. Indeed, all tibbles are lists, but not all lists are tibbles. For example, in addition to being a tibble, \texttt{rows} \_ \texttt{are} \_ \texttt{lines} is also a list:

\begin{verbatim}
rows_are_lines %>% is_list
\end{verbatim}

\begin{verbatim}
> [1] TRUE
\end{verbatim}
Specifically, it’s a list of four elements:

```r
rows_are_lines %>% length
```

> [1] 4

The list’s elements correspond to the tibble’s columns. For example, the fourth element of the list is the character string vector stored in the `text` column.

```r
rows_are_lines %>% pluck(4) %>% head
```

> [1] "CHAPTER 1"
> [2] "The family of Dashwood had long been settled in Sussex. Their estate"
> [3] "was large, and their residence was at Norland Park, in the centre of"
> [4] "their property, where, for many generations, they had lived in so"
> [5] "respectable a manner as to engage the general good opinion of their"
> [6] "surrounding acquaintance. The late owner of this estate was a single"

In general, a tibble is a list whose elements are vectors of equal length. But having equally lengthed vectors as elements does not necessarily make a list a tibble:

```r
tiny_list <-
  list(
    x = 1:2,
    y = c("a", "b"),
    z = c(TRUE, FALSE)
  )
tiny_list
```

> `$x`
> [1] 1 2
> `$y`
> [1] "a" "b"
> `$z`
> [1] TRUE FALSE
But such a list can be turned into a tibble (unlike most other lists):

```r
> tiny_list %>% as_tibble

# A tibble: 2 x 3
  x   y   z
  <int> <chr> <lgl>
1    1    a   TRUE
2    2    b  FALSE
```

Since tibbles are lists, all the list functions defined in the next section apply to tibbles.

### 5.4.2 Common List Functions

Most standard tibble functions have a list analog. For example, the list analog of

- `glimpse()` is `str()`,
- `pull()` or `select()` is `pluck()`,
- `mutate()` is `list_modify()`,
- `filter()` is `keep()` or `discard()`,
- `summarise()` is `map()`,
- `inner_join()` is `map2()`, and
- `pivot_wider()` is `nest()` or `transpose()`, and
- `pivot_longer()` is `unnest()` or `transpose()`.

I will now illustrate each of these functions, besides `pluck()`, `map()`, `nest()`, and `unnest()`, which we covered in the lecture.

First `str()` is analogous to `glimpse()`:

```r
jane %>% str
```
Next, `list_modify()` is analogous to `mutate()`:

```r
# add a fourth book
jane %>%
  list_modify(
    novel_4 =
    list(
      title = "Emma",
      pub_date = "23 December 1815" %>% dmy,
      word_count = 155887,
      protagonist = "emma woodhouse"
    )
  )
str
```

> List of 4
> $ novel_1:List of 4
>   ..$ title : chr "Sense & Sensibility"
>   ..$ pub_date : Date[1:1], format: "1811-02-01"
>   ..$ word_count : num 119394
>   ..$ protagonist: chr "elinor dashwood"
> $ novel_2:List of 4
>   ..$ title : chr "Pride & Prejudice"
>   ..$ pub_date : Date[1:1], format: "1813-01-28"
>   ..$ word_count : num 122189
>   ..$ protagonist: chr "elizabeth bennet"
> $ novel_3:List of 4
>   ..$ title : chr "Mansfield Park"
>   ..$ pub_date : Date[1:1], format: "1814-07-01"
>   ..$ word_count : num 159381
>   ..$ protagonist: chr "fanny price"

393
$ novel_3:List of 4
  ..$ title : chr "Mansfield Park"
  ..$ pub_date : Date[1:1], format: "1814-07-01"
  ..$ word_count : num 159381
  ..$ protagonist: chr "fanny price"
$ novel_4:List of 4
  ..$ title : chr "Emma"
  ..$ pub_date : Date[1:1], format: "1815-12-23"
  ..$ word_count : num 155887
  ..$ protagonist: chr "emma woodhouse"

#remove first book

jane %>%
  list_modify(novel_1 = NULL) %>%
  str

List of 2

$ novel_2:List of 4
  ..$ title : chr "Pride & Prejudice"
  ..$ pub_date : Date[1:1], format: "1813-01-28"
  ..$ word_count : num 122189
  ..$ protagonist: chr "elizabeth bennet"
$ novel_3:List of 4
  ..$ title : chr "Mansfield Park"
  ..$ pub_date : Date[1:1], format: "1814-07-01"
  ..$ word_count : num 159381
  ..$ protagonist: chr "fanny price"

#replace first book with "Unavailable"

jane %>%
  list_modify(novel_1 = "Unavailable") %>%
  str

List of 3

$ novel_1: chr "Unavailable"
$ novel_2:List of 4
  ..$ title : chr "Pride & Prejudice"
  ..$ pub_date : Date[1:1], format: "1813-01-28"
  ..$ word_count : num 122189
  ..$ protagonist: chr "elizabeth bennet"
The first example above adds a list to `jane`. The second example removes a list from `jane` by setting it to the special keyword `NULL` (which is not to be confused with `NA`). The third example redefines `novel_1` from a list to a character string. And the last example modifies the elements of the `novel_1` list. Note that this last case departs from the standard convention. Logically, the fourth `list_modify()` should set `novel_1` to `list(publisher = "Thomas Egerton", protagonist = "marianne dashwood")`, but it doesn't. Instead, the operation collates the two elements of `list(publisher = "Thomas Egerton", protagonist = "marianne dashwood")`.
"marianne dashwood") with the four elements of the original novel_1 list. Indeed, whenever we ask list_modify() to replace one list with another list, the function takes the liberty of combining the elements of the two lists. And this is a pretty useful protocol. For example, we can use it to add an antagonist attribute to each of our lists:

```r
ejane %>%
  list_modify(
    novel_1 = list(antagonist = "John Willoughby"),
    novel_2 = list(antagonist = "George Wickham"),
    novel_3 = list(antagonist = "Mrs. Norris")
  ) %>%
  str
```

```
> List of 3
> $ novel_1:List of 5
>   ..$ title : chr "Sense & Sensibility"
>   ..$ pub_date : Date[1:1], format: "1811-02-01"
>   ..$ word_count : num 119394
>   ..$ protagonist: chr "elinor dashwood"
>   ..$ antagonist : chr "John Willoughby"
> $ novel_2:List of 5
>   ..$ title : chr "Pride & Prejudice"
>   ..$ pub_date : Date[1:1], format: "1813-01-28"
>   ..$ word_count : num 122189
>   ..$ protagonist: chr "elizabeth bennet"
>   ..$ antagonist : chr "George Wickham"
> $ novel_3:List of 5
>   ..$ title : chr "Mansfield Park"
>   ..$ pub_date : Date[1:1], format: "1814-07-01"
>   ..$ word_count : num 159381
>   ..$ protagonist: chr "fanny price"
>   ..$ antagonist : chr "Mrs. Norris"
```

Adding this antagonist feature would be more difficult if list_modify() were more literal-minded.

**Exercise 5.38.** Redefine novel_1 to list(title = "Lady Susan", protagonist = "susan").

- Use `NULL` to delete the Sense & Sensibility attributes.

**Exercise 5.39.** Use `NULL` to remove the word_count of novel_2.
Unfortunately, it’s not as easy as it should be to reference list elements within `list_modify()`. For example, the following would ideally `str_to_title()` the protagonist of `novel_1`:

```r
jane %>%
  list_modify(
    novel_1 = list(
      protagonist = novel_1 %>%
        pluck("protagonist") %>%
        str_to_title
    )
  ) %>%
str
```

> Error in pluck(., "protagonist"): object 'novel_1' not found

But, regrettably, `list_modify()` doesn’t know what to make of `novel_1` when it’s positioned to the right of the `=` sign. Here’s the (slightly dissatisfying) solution:

```r
jane %>%
  list_modify(
    novel_1 = list(
      protagonist = pluck(., "novel_1", "protagonist") %>%
        str_to_title
    )
  ) %>%
str
```

> List of 3
> $ novel_1:List of 4
> ..$ title : chr "Sense & Sensibility"
> ..$ pub_date : Date[1:1], format: "1811-02-01"
> ..$ word_count : num 119394
> ..$ protagonist: chr "Elinor Dashwood"
> $ novel_2:List of 4
> ..$ title : chr "Pride & Prejudice"
> ..$ pub_date : Date[1:1], format: "1813-01-28"
> ..$ word_count : num 122189
> ..$ protagonist: chr "elizabeth bennet"
> $ novel_3: List of 4
> ..$ title : chr "Mansfield Park"
> ..$ pub_date : Date[1:1], format: "1814-07-01"
> ..$ word_count : num 159381
> ..$ protagonist: chr "fanny price"

In this case, R interprets the . as jane, since that’s what got piped into list_modify().

**Exercise 5.40.** Add an attribute to the three lists in jane called pub_wday, which equals the wday(label = TRUE) of the pub_date.

- Within list_modify(), you can reference the first novel’s publication date with pluck(. , "novel_1", "pub_date").

**Exercise 5.41.** Use map() and list_modify() to round() the word_count values in jane to the nearest thousand. For example, the *Pride & Prejudice* word_count should change to 

122189 %>% round(digits = -3) = 122000.

**Exercise 5.42.** Use map() and list_modify() to str_to_lower() the title values in jane.

---

keep() is analogous to filter(). It applies to each element in the list a function—or an anonymous function—that yields a logical, and it keeps the elements for which this logical is TRUE.

```r
# get the character strings in mansfield
mansfield %>%
  keep(is.character)
```

```r
> $title
> [1] "Mansfield Park"
> >
> > $protagonist
> > [1] "fanny price"
```

```r
# get books published after 1812
jane %>
  keep(
    ~.x %>
    pluck("pub_date") %>
```
year %>%
(. > 1812)
) %>%
str

> List of 2
> $ novel_2:List of 4
> ..$ title : chr "Pride & Prejudice"
> ..$ pub_date : Date[1:1], format: "1813-01-28"
> ..$ word_count : num 122189
> ..$ protagonist: chr "elizabeth bennet"
> $ novel_3:List of 4
> ..$ title : chr "Mansfield Park"
> ..$ pub_date : Date[1:1], format: "1814-07-01"
> ..$ word_count : num 159381
> ..$ protagonist: chr "fanny price"

# get books with "&" in the title
jane %>%
keep(
  ~.x %>%
    pluck("title") %>%
    str_detect("&")
) %>%
str

> List of 2
> $ novel_1:List of 4
> ..$ title : chr "Sense & Sensibility"
> ..$ pub_date : Date[1:1], format: "1811-02-01"
> ..$ word_count : num 119394
> ..$ protagonist: chr "elinor dashwood"
> $ novel_2:List of 4
> ..$ title : chr "Pride & Prejudice"
> ..$ pub_date : Date[1:1], format: "1813-01-28"
> ..$ word_count : num 122189
> ..$ protagonist: chr "elizabeth bennet"

The second and third examples use an anonymous function, rather than a stand-alone function, to determine which elements to keep. For example, the last example tells R to evaluate novel_1 %>% pluck("title") %>% str_detect("&"), novel_2 %>% pluck("title")
str_detect("&"), and novel_3 %>% pluck("title") %>% str_detect("&"), and to retain the books for which this output is TRUE. But the ~.x %>% term is unnecessary if we're applying a single function without options. For example, the first example tells R to apply is.character() to each attribute in mansfield and keep those for which this is TRUE.

In addition to `keep()` there is `discard()`, which drops the elements that meet the specified condition:

```r
# remove the character strings in mansfield
mansfield %>%
  discard(is.character)
```

```r
> $pub_date
> [1] "1814-07-01"
>
> $word_count
> [1] 159381
```

```r
# remove books published after 1812
jane %>%
  discard(~.x %>%
    pluck("pub_date") %>%
    year %>%
    {. > 1812}
) %>%
  str
```

```r
> List of 1
> $ novel_1:List of 4
> ... title : chr "Sense & Sensibility"
> ... pub_date : Date[1:1], format: "1811-02-01"
> ... word_count : num 119394
> ... protagonist: chr "elinor dashwood"
```

```r
# remove books with "&" in the title
jane %>%
  discard(~.x %>%
```
```r
pluck("title") %>%
  str_detect("&")
)
```

> List of 1
> $ novel_3:List of 4
> ..$ title : chr "Mansfield Park"
> ..$ pub_date : Date[1:1], format: "1814-07-01"
> ..$ word_count : num 159381
> ..$ protagonist: chr "fanny price"

**Exercise 5.43.** Use `is.Date()` to `discard()` the Dates in `mansfield`.

**Exercise 5.44.** `keep()` only the books of `jane` that are at least 120000 words long.

**Exercise 5.45.** Use `map_df()`, `keep()`, and `is.character()` to tabulate all the character strings in `jane`.

The list version of `inner_join()` is `map2()`, which takes two lists of equal length and combines their elements with a functional assembly line. For example, the following adds a `first_sentence` attribute to `jane`:

```r
s_1 <- "The family of Dashwood had long been settled in Sussex."
s_2 <-
  str_c(
    "It is a truth universally acknowledged,",
    "that a single man in possession of a good fortune,",
    "must be in want of a wife.",
    sep = " "
  )
jane %>%
  map2(
    list(s_1, s_2, NULL),
    ~.x %>% list_modify(first_sentence = .y)
  )
```

> $novel_1
> $novel_1$title

401
Sense & Sensibility

1811-02-01
119394
Elinor Dashwood
The family of Dashwood had long been settled in Sussex.

Pride & Prejudice

1813-01-28
122189
Elizabeth Bennet
It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

Mansfield Park

1814-07-01
159381
Fanny Price
The `map2()` function receives the left-hand list (LHL) `jane` and the right-hand list (RHL) `list(s_1, s_2, NULL)`. (I made the third element of the RHL `NULL`, because the first sentence of *Mansfield Park* is really long.) `map2()` also receives the functional assembly line `~.x %>% list_modify(first_sentence = .y)`, where `.x` is a place-holder for the LHL element and `.y` is a place-holder for the RHL element. Thus, the first element that `map2()` returns is

```r
list(
    title = "Sense & Sensibility",
    pub_date = "1 February 1811" %>% dmy,
    word_count = 119394,
    protagonist = "elinor dashwood"
) %>%
list_modify(first_sentence = s_1)
```

```r
> $title
> [1] "Sense & Sensibility"
> 
> $pub_date
> [1] "1811-02-01"
> 
> $word_count
> [1] 119394
> 
> $protagonist
> [1] "elinor dashwood"
> 
> $first_sentence
> [1] "The family of Dashwood had long been settled in Sussex."
```

And the third element that `map2()` returns is

```r
mansfield %>%
list_modify(first_sentence = NULL)
```

```r
> $title
> [1] "Mansfield Park"
> 
> $pub_date
> [1] "1814-07-01"
> 
> $word_count
```

403
null tells R not to bother giving this book a first_sentence attribute.

map2() allows us combine the elements in the LHL with those of the RHL in any which way, depending on how we combine the .x and .y terms in the functional assembly line. Here are some examples:

```r
# paste together strings
list("First Name", "Last Name") %>%
  map2(
    list("Gertrude", "Bray"),
    ~.x %>% str_c(.y, sep = " ")
  )
```

```
[[1]]
[1] "First Name: Gertrude"

[[2]]
[1] "Last Name: Bray"
```

```r
# pick TRUE or FALSE statements
list(
  list("earth is round", "earth is flat"),
  list("water is wet", "water is dry"),
  list("2 + 2 = 4", "2 + 2 = 5")
) %>%
  map2(
    list(TRUE, FALSE, TRUE),
    ~.x %>% {if(.y) pluck(.x, 1) else pluck(.x, 2)}
  )
```

```
[[1]]
[1] "earth is round"
```

In practice, I would use `~ if(.y) pluck(.x, 1) else pluck(.x, 2) or ~ pluck(.x, 2 - .y)` instead of `~.x %>% {if(.y) pluck(.x, 1) else pluck(.x, 2)}` in the second example. But for consistency I'll stick to the `~.x %>%` convention in this text.

---

404
> [[2]]
> [1] "water is dry"
>
> [[3]]
> [1] "2 + 2 = 4"

```r
#apply function list to number list
list(a = 10:20, b = 100:200, c = 1000:2000) %>%
  map2(
    list(mean, median, sum),
    ~.x %>% .y
  )
```

> $a
> [1] 15
>
> $b
> [1] 150
>
> $c
> [1] 1501500

In the last case, the RHL comprises three functions—as I said, lists can store anything—and the functional assembly line tells R to pipe the LHL elements into these functions. Hence, `map2()` returns `list(10:20 %>% mean, 100:200 %>% median, 1000:2000 %>% sum) = list(15, 150, 1501500).

And like `map_df()`, we analogously have `map2_df()`:

```r
#apply different functions to different vectors
list(a = 10:20, b = 100:200, c = 1000:2000) %>%
  map2_df(
    list(mean, median, sum),
    ~.x %>% .y
  )
```

> # A tibble: 1 x 3
>   a  b    c
>  <dbl> <int> <int>
>  1   15 1501500

**Exercise 5.46.** Use `map2()` to create the following list:
Exercise 5.47. Use `map2()` to `list_modify()` the protagonist attribute with a characters attribute, which should be `c("elinor dashwood", "marianne dashwood")` for `novel_1`, `c("elizabeth bennet", "mr. darcy")` for `novel_2`, and `c("fanny price", "lady bertram", "mrs. norris")` for `novel_3`. Call the resulting list `jane_2`.

Exercise 5.48. Use `map2()` to add "John Willoughby" as the antagonist of `novel_1`, "George Wickham" as the antagonist of `novel_2`, and "Mrs. Norris" as the antagonist of `novel_3`.

Exercise 5.49. Use `map2_df()` to create the following tibble:

```
> # A tibble: 1 x 3
>   novel_1          novel_2 novel_3
>   <chr>            <dbl> <chr>
> 1 Sense & Sensibility 122189 fanny price
```

Exercise 5.50. Start with `jane_2`, from exercise 5.47, and discard() the books with more than two listed characters. Then tabulate the remaining title values with `map_df()`. (The output should be a tibble comprising "Sense & Sensibility" and "Pride & Prejudice".)

- Note that `c("a", "b", "c") %>% length %>% {>.2} = TRUE.

Another list analog of `pivot_longer()` and `pivot_wider()` is `transpose()`. This function rearranges a list of lists or a list of vectors. For example, we can `transpose()` `jane` from three length-four lists to four length-three lists:

```
#original list:
jane

> $novel_1
> [1] "Sense & Sensibility"
>
> $novel_1$title
> [1] "Sense & Sensibility"
>
> $novel_1$pub_date
```
> [1] "1811-02-01"
>
> $novel_1$word_count
> [1] 119394
>
> $novel_1$protagonist
> [1] "elinor dashwood"
>
> $novel_2$title
> [1] "Pride & Prejudice"
>
> $novel_2$pub_date
> [1] "1813-01-28"
>
> $novel_2$word_count
> [1] 122189
>
> $novel_2$protagonist
> [1] "elizabeth bennet"
>
> $novel_3$title
> [1] "Mansfield Park"
>
> $novel_3$pub_date
> [1] "1814-07-01"
>
> $novel_3$word_count
> [1] 159381
>
> $novel_3$protagonist
> [1] "fanny price"

#transposed list:

jane %>>%
	ranspose

> $title
> $title$novel_1
> [1] "Sense & Sensibility"
As you see, `transpose()` flips the list nesting order: whereas `jane` organizes the data by book, `jane %>% transpose` organizes the data by attribute. Or whereas `wine_list` groups the data by bottle of wine, `wine_list %>% transpose` groups the data by wine attribute—the names of its sublists are points, title, description, taster_name, etc.

It’s often useful to `transpose()` a tibble into a more conventional list. For example, `wine_tib
organizes the data by attribute, grouping all the `taster_name` values in one vector and all the `variety` values in another vector, etc. But `wine_tib %>% transpose` groups the data by wine bottle, as `wine_list` does.

Finally, it’s sometimes it’s easier to work with a list after you `transpose()` it. Indeed, a common strategy is to `transpose()` a list, modified it, and then `transpose()` it back to the original configuration, a technique you’ll use in the following exercise.

**Exercise 5.51.** We will now remove all the non-character-string data in `jane`.

- `transpose()` `jane` so that it comprises four lists: `title`, `pub_date`, `word_count`, and `protagonist`. Each of these lists corresponds to a single data type. For example, all the elements of `title` are character strings and all the elements of `pub_date` are Dates.
- `keep()` only the lists that store character strings.
  - `pluck()` the first element and test whether it `is.character()`.
- `transpose()` the list back into one with elements `novel_1`, `novel_2`, and `novel_3`.

### 5.4.3 Vectorize Function Inputs

There’s a fixed overhead associated with every R function call. For example, it’s faster to execute `sqrt(1:1000)` than to execute

```r
sqrt(1)
sqrt(2)
# ...
sqrt(1000)
```

Whereas the former case “opens” and “closes” the `sqrt()` function only once, the latter case “opens” and “closes” `sqrt()` a thousand times, which is onerous. Indeed, since “opening” and “closing” functions is time-consuming, it’s always faster to apply a function to an entire vector than it is to apply it to each vector element separately. Here’s a numerical example:

```r
library(tictoc)

janes_words <-
  rows_are_lines %>%
  pull(line) %>%
  str_split(" ") %>%
  unlist

#Case 1:
```
In the code above, `janes_words` is a character string vector that comprises the 724151 words in Jane Austen's novels, and `tic()` and `toc()`, from the tictoc package, start and stop a stopwatch that times our code. Both Case 1 and Case 2 administer `str_to_lower()` to `janes_words`. But the first case applies the function to the full `janes_words` vector, calling `str_to_lower()` only once, whereas the latter case applies the function one word at a time, calling `str_to_lower()` 724151 distinct times. The latter method is clearly slower.

It’s easy to avoid applying a function element-wise when the data are stored in a vector, like `janes_words`, because most functions can handle vector inputs. However, most functions cannot handle list inputs. So to apply a function to a list you usually must either (i) apply it element-wise, which can be very slow, or (ii) convert the list to a vector or tibble before applying the function.
5.5 Solutions

5.1

```r
jane %>%
  map(pluck("protagonist"))
```

5.2

```r
jane %>%
  map(
    ~ x %>%
    pluck("title") %>%
    word(1)
  )
```

5.3

```r
jane %>%
  map(
    ~ x %>%
    pluck("title") %>%
    str_to_upper
  )
```

5.4

```r
jane %>%
  map_df(
    ~ x %>%
    pluck("protagonist") %>%
    str_to_title
  )
```

5.5
The former gives each chapter of each novel its own tibble, and the latter pools all lines of a novel in one tibble.

5.7

```
rows_are_words <-
  rows_are_lines %>%
  mutate(text = str_split(text, " ")) %>%
  unnest(text)
```

5.8

```
rows_are_books_2 <-
  rows_are_words %>%
  nest(line_tib = text) %>%
  nest(chapter_tib = c(line, line_tib)) %>%
  nest(novel_tib = c(chapter, chapter_tib))
```

5.9

```
rows_are_books_2 %>%
```

5.10

```
rows_are_lines %>%
  filter() %>%
  select(-c(book, line)) %>%
  mutate(text = str_split(text, " ")) %>%
  unnest(text)
```

5.11

412
wine_tib <-
  wine_list %>%
  map_df(~.x) %>%
  distinct %>%
  rowid_to_column(var = "review_id") %>%
  mutate(
    across(
      c(points, price),
      strtoi
    ),
    vintage =
    title %>%
    str_extract("\b19[9-9]\\b|\b20[01][0-9]\\b") %>%
    strtoi
  )

5.12

wine_description_by_word <-
  wine_tib %>%
  select(review_id, description) %>%
  mutate(
    description =
    description %>%
    str_remove_all("[^[:alnum:]]") %>%
    str_squish %>%
    str_split(" ")
  )

5.13

wine_word_count <-
  wine_description_by_word %>%
  unnest(description) %>%
  rename(word = description)

5.14
wine_tib_2 <-
    wine_word_count %>%
    left_join(sentiment_dictionary) %>%
    group_by(review_id) %>%
    summarise(
        word_count = n(),
        pos_count = sum(sentiment == "+", na.rm=TRUE),
        neg_count = sum(sentiment == "-", na.rm=TRUE),
        pos_minus_neg = pos_count - neg_count,
        .groups = "drop"
    ) %>%
    inner_join(wine_tib)

5.15

wine_model %>%
    augment(data = wine_tib_2) %>%
    filter(.resid == max(.resid)) %>%
    select(title)

??

wine_model %>%
    augment %>%
    summarise(R2 = 1 - sd(.resid)^2/sd(points)^2)

5.17

wine_model_2 %>%
    augment(newdata = wine_review_scenarios) %>%
    summarise(mean(.fitted))

5.18

wine_model_2 %>%
    tidy %>%
    filter(str_detect(term, "vintage")) %>%
    mutate(

term =
  term %>%
  str_remove("as.factor\(vintage\)) %>%
  strtoi
)
%>%
ggplot() +
aes(x = term, y = estimate) +
geom_point()

5.19

wine_model_3 %>%
tidy %>%
filter(str_detect(term, "taster_name")) %>%
filter(estimate == min(estimate)) %>%
mutate(term = str_remove(term, "taster_name")) %>%
pull(term)

5.20

wine_tib_3 %>%
mutate(
  price_group =
  price %>%
  ntile(100) %>%
  as.factor
)
%>%
lm(
  points ~ word_count + pos_count + neg_count + as.factor(vintage) +
  province + taster_name + variety + price_group,
  data = .
)
%>%
tidy %>%
filter(str_detect(term, "price_group"))

5.21

emily_word_count <-
emily_wine %>%
select(review_id, description) %>%
mutate(
    description =
    description %>%
    str_remove_all("[^[:alnum:]]") %>%
    str_squish %>%
    str_split(" ")%
) %>%
unnest(description) %>%
rename(word = description) %>%
count(review_id, word)

emily_wine_2 <-
emily_word_count %>%
left_join(sentiment_dictionary) %>%
group_by(review_id) %>%
summarise(
    word_count = sum(n),
    pos_count = sum(n * (sentiment == "+"), na.rm=TRUE),
    neg_count = sum(n * (sentiment == "-"), na.rm=TRUE),
    .groups = "drop"
) %>%
inner_join(emily_wine)

wine_model_3 %>%
augment(newdata = emily_wine_2) %>%
select(brand, .fitted)

5.22

wine_tib_4 <-
wine_tib_3 %>%
nest(sample = -taster_name) %>%
filter(!taster_name %in% c("Other", NA))

5.23
summarise(sum(pos_count) > 5 * sum(neg_count))
)

wine_tib_4 %>%
rowwise %>%
mutate(
  sample %>%
    summarise(
      median_length =
        description %>%
        str_count %>%
        median
    )
)

wine_tib_4 %>%
rowwise(taster_name) %>%
summarise(
  sample %>%
    summarise(pos_neg_cor = cor(pos_count, neg_count))
)

wine_tib_6 %>%
select(taster_name, lms_tidy) %>%
unnest(lms_tidy) %>%
filter(
  str_detect(term, "pos") |
  str_detect(term, "neg")
)

mutate(estimate = ifelse(p.value < .05, estimate, NA))
pivot_wider(
  id_cols = taster_name,
  names_from = term,
  values_from = estimate
)

5.27

wine_tib_4 %>%
rowwise %>%
mutate(
  lms_tidy =
    sample %>%
    lm(
      points ~ word_count + pos_count + neg_count + variety,
      data = .
    ) %>%
    tidy %>%
    list
  ) %>%
ungroup %>%
unnest(lms_tidy) %>%
filter(term %>% str_detect("neg"))

5.28

wine_tib_6 %>%
rowwise %>%
mutate(lms_glance = lms %>% glance %>% list) %>%
ungroup %>%
unnest(lms_glance) %>%
filter(r.squared == max(r.squared)) %>%
select(taster_name)

5.29

wine_tib_5 %>%
rowwise %>%
mutate(
lms_augment =
  lms %>%
  augment(
    newdata =
      tibble(word_count = 40, pos_count = 0, neg_count = 4)
  ) %>%
  list
) %>%
unnest(lms_augment) %>%
filter(.fitted == min(.fitted)) %>%
select(taster_name)
Chapter 6

Generalized Linear Models

6.1 Introduction

The generalized linear model (GLM) is a powerful framework that unifies many statistical models, such as linear regression, Poisson regression, logistic regression, multinomial regression, and gamma regression. Like ordinary least squares (OLS), a GLM characterizes the dependence of a left-hand variable, $y$, on a linear combination of right-hand variables, $\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$. However, whereas an OLS regression describes how the linear sum of independent variables influences the dependent variable’s mean and standard deviation, a GLM describes how the linear sum of independent variables influences the dependent variable’s full distribution. For example, we could use a GLM to estimate how the far-right tail of $y$ responds to changes in $x_1$, something we could never do with an OLS.

The second defining characteristic of the GLM is that the influence between the linear combination of independent variables, $\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$, and the dependent variable, $y$, is mediated by transformation function $f()$. In other words, $y$ depends not on $\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$, but on $f(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n)$. And $f()$ is something that we can choose. For example, we can set $f(x) = \exp(x)$ to force $y$ to be positive or set $f(x) = \frac{\exp(x)}{1+\exp(x)}$ to force $y$ to be between zero and one.

In the lecture, in section 6.2, we’ll cover R’s `glm()` function with an extended case study of Citi Bike’s NYC bicycle sharing platform. Specifically, we’ll analyze the long bike rentals, which are inconvenient for the company. Unfortunately, the full distribution of bike rental times has an ad hoc shape that’s hard to model. However, the right-hand tail of this distribution decays gracefully, like the gamma distribution. So we characterize the distribution of long bike rides in two steps. First, we use a gamma regression to model the bike rental duration, conditional on being in the right tail of the distribution. And second, we use a logistic regression to estimate the probability of being in the right tail of the distribution. Combining these two models yields a flexible and accurate characterization of the long-lasting bike rides.

The lab, in section 6.3, will replicate a study of safety issues at nuclear power plants conducted by Blanco et al. [2019]. We will use `glm()` to run Poisson regressions, which are
especially designed for count data (which take integer values). We will use our regression models to estimate how many more safety issues the plants would have suffered had they not implemented Probability Risk Assessments (PRAs).

For the lecture, you can work in the code.R file of the glm project you created in exercise 1.36, and for the lab you can work in the code.R file of the radiation project you created in exercise 1.37.

6.2 Lecture

6.2.1 Limits of Linear Models

The bike tibble you made in exercise 1.36 of section 1.3 looks like this:

```r
bike %>% glimpse
```

This tibble describes Citi Bike’s1 May 2018 bike-sharing rides. Two key variables are distance, which records the number of miles between coordinates (start_station_lat, start_station_long) and (end_station_lat, end_station_long), and duration, which

---

1https://citibikenyc.com/homepage
records the number of minutes between `start_time` and `end_time`. Here are their distributions:

```r
bike %>%
  pivot_longer(c(duration, distance)) %>%
  mutate(
    name =
      ifelse(
        name == "distance",
        "Distance (miles)",
        "Duration (minutes)"
      ),
    ) %>%
  ggplot() +
  aes(x = value) +
  geom_density(adjust = .2) +
  facet_wrap( 
    vars(name),
    scales = "free"
  ) +
  labs( 
    x = "",
    y = "Probability Density"
  ) +
  theme_minimal()
```

Overall, 80% of trips last less than 20 minutes. Citi Bike prefers these short trips because they enable it to execute more rentals. For example, the company would prefer four 15-
minute rentals, which help justify four subscriptions, to one hour-long rental, which helps justify only one subscription. Moreover, shorter trips mean shorter bike wait times: the more frequently the company turns over its fleet, the less time a customer has to wait for a bike to be returned.

Trips that last longer than 20 minutes are deemed to run “over time.” The overtime column reports the trip duration in excess of 20 minutes. For example, a trip with duration = 19 has overtime = 0, whereas a trip with duration = 25 has overtime = 5. To begin, we will limit our sample to the trips that ran overtime:

```r
overtime_trips <-
  bike %>%
  filter(overtime > 0)
```

We will study the drivers of overtime in our sample of overtime_trips. When explaining the variation in a random variable, it’s always best to start with a simple `lm()`. Since OLS does not require the dependent variable or statistical error term to be normally distributed—contrary to popular belief—we can use `lm()` to model the expected value of overtime, even though it’s non-negative and right-skewed.

**Exercise 6.1.** Use overtime_trips to run a linear regression of overtime on trip distance, and save the result as baseline_lm. tidy() the result to confirm that increasing a trip’s distance by one mile increases its expected overtime by 1.212 minutes (conditional on the trip running overtime).

```r
#the .fitted values are the means and .sigma is the standard deviation:
baseline_lm %>%
  augment
```

> # A tibble: 317,393 x 8
>   overtime distance .fitted .resid .hat .sigma .cooksd .std.resid
>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1  1.97  1.31  8.47 -6.50 0.00000541  9.28 0.00000133 -0.700
> 2 16.6  2.77 10.2  6.39 0.00000350  9.28 0.000000831  0.689
> 3 39.1  4.76 12.6 26.4  0.0000149  9.28 0.0000605  2.85
> 4  5.67  3.67 11.3 -5.66 0.00000667  9.28 0.00000124 -0.610
> 5 17.0  2.77 10.2  6.74 0.00000350  9.28 0.000000924  0.727
For example, the first row indicates that if distance = 1.31 then overtime has a mean equal to \( \text{fitted} = 8.47 \) and has a standard deviation equal to \( \text{sigma} = 9.28 \). But rather than the mean, suppose we wanted to calculate the median value of each overtime. For example, if distance = 1.31 then what value has exactly a 50% chance of exceeding overtime? If the overtime variables were normally distributed then their means would equal their medians, in which case the \( \text{fitted} \) values would also be the medians—i.e., the overtime values would exceed the \( \text{fitted} \) values half the time. However, only 37% of overtime values exceed their corresponding \( \text{fitted} \) value:

```r
baseline_lm %>%
  augment %>%
  summarise(mean(overtime > .fitted))
```

```r
> # A tibble: 1 x 1
> `mean(overtime > .fitted)`
> <dbl>
> 1 0.374
```

The problem is that overtime is instead heavily right skewed, which makes its mean (i.e., \( \text{fitted} \)) far exceed its median. For example, here is the distribution of overtime when distance is near 1.31:

```r
overtime_trips %>%
  filter(
    distance > 1.27,
    distance < 1.36
  ) %>%
  ggplot() +
  aes(x = overtime) +
  geom_density(adjust = .1) +
  labs(
    x = "Trip Overtime",
    y = "Probability Density"
  ) +
  theme_minimal()
```
As you see, this distribution looks nothing like a normal distribution, so there’s no easy way we can derive its median from the outputs provided by `lm()`. Indeed, `lm()` tells us nothing besides the mean and standard deviation of the dependent variable. Accordingly, it’s impossible to determine from `baseline_lm` how the median `overtime` varies with `distance`. And it’s similarly impossible to determine the probability that `overtime` exceeds 15 minutes given `distance = 1`, or the interquartile range of `overtime` when `distance = 10`, or the probability that a `distance = 3` trip has a longer `overtime` than a `distance = 30` trip. To answer these questions, we must explicitly characterize the distribution of the `overtime` conditional on `distance`. And we can assign `overtime` a distribution if we upgrade our linear model to a generalized linear model (GLM).

### 6.2.2 Generalized Linear Model

Since `overtime` is positive and right-skewed, we will model it with the gamma distribution\(^2\). This distribution is parameterized by its mean and by a shape coefficient. The shape parameter describes the silhouette of the distribution and the mean describes the scaling of the axes:

\[
\text{expand_grid}(
  \text{mean} = \text{c}(1, 10, 100),
  \text{shape} = \text{c}(.5, 1, 2, 10, 100)
) \%
\%
\text{mutate}(
  \text{case} =
\]

\(^2\text{https://en.wikipedia.org/wiki/Gamma_distribution}\)
If you compare this plot with the previous plot, you’ll see that the distribution of overtime resembles the gamma distribution with shape parameter = 1 when distance is near 1.31. And this checks out, as the following `glm()` pegs the shape parameter at 1.02399:
# estimate glm

```r
gamma_glm <-
  overtime_trips %>%
  glm(
    overtime ~ distance,
    family = Gamma(link = "identity"),
    control = list(maxit = 10^3),
    data = .
  )

# get shape parameter estimate

gamma_shape <-
  gamma_glm %>%
  summary %>%
  {1/pluck(., "dispersion")}
```

A glm with family = Gamma(link = "identity") models the dependent variable as a gamma whose shape parameter remains fixed but whose mean varies linearly with the dependent variables. Accordingly, gamma_glm models overtime as a gamma random variable with shape parameter = 1.02399 and mean = 8.28 + 0.606 * distance. For example, gamma_glm tells us that when distance = 2 the distribution of overtime looks like this:
\[
\text{gamma\_mean} <- \\
gamma\_glm \%>% \\
augment(newdata = tibble(distance = 2)) \%>% \\
pull(.fitted)
\]

```
tibble(
  x = seq(.01, 70, length.out = 10^3),
  density =
    dgamma(
      x = x,
      shape = gamma\_shape,
      scale = gamma\_mean / gamma\_shape
    )
) \%>%
```

```
ggplot() +
aes(x = x, y = density) +
geom\_line() +
theme\_minimal()
```

Compare this with the plot presented at the end of section 6.2.1 and you’ll see that our model fits the data quite well.

Since our model specifies the full distribution of our dependent variable—not just its mean and standard deviation—we can now calculate the median overtime and the probability that it exceeds 15 minutes (statistics we couldn’t derive from our \texttt{lm}()):

429
#median overtime

qgamma(
  p = 0.5,
  shape = gamma_shape,
  scale = gamma_mean / gamma_shape
)

> [1] 6.640216

#probability that overtime exceeds 15 minutes

pgamma(
  15,
  shape = gamma_shape,
  scale = gamma_mean / gamma_shape,
  lower.tail = FALSE
)

> [1] 0.2053227

Note, the scale of a gamma distribution is its mean divided by its shape parameter.

We could run a glm() with other distributions. For example, instead of gamma random variable with time-varying mean, we could model overtime as a normal random variable with time-varying mean:

#estimate glm with normal distribution

normal_glm <- overtime_trips %>%
  glm(
    overtime ~ distance,
    family = gaussian(link = "identity"),
    data = .
  )

#get standard deviation

normal_sd <- normal_glm %>%
  augment %>%
  pluck(".sigma", 1)

normal_sd

430
#get mean coefficient estimates

```r
normal_glm %>%
tidy
```

```
# A tibble: 2 x 5
#  term         estimate std.error statistic p.value
#  <chr>        <dbl>     <dbl>     <dbl>   <dbl>
#1 (Intercept)  6.88      0.0354    194.     0
#2 distance    1.21      0.0133     91.3    0
```

The normal_glm models overtime as a normally distributed random variable, with standard deviation = 9.278 and mean = 6.88 + 1.21 * distance. However, the gamma distribution fits the data better than the normal distribution, as we can see when we compare the models’ Akaike information criteria (AIC)³:

#the model with the lower AIC has a better fit:

```r
gamma_glm %>%
glance
```

```
# A tibble: 1 x 8
#  null.deviance df.null logLik    AIC    BIC deviance df.residual nobs
#1          335773 317392  -1038914. 2077833. 2077865.  331995.   317391
```

normal_glm %>%
glance

```
# A tibble: 1 x 8
#  null.deviance df.null logLik    AIC    BIC deviance df.residual nobs
#1          28040707 317392  -1157407. 2314820. 2314852. 27322728. 317391
```

Exercise 6.2. Does factoring the customer age improve our statistical model?

- Run a glm() with formula overtime ~ distance + age and family = Gamma(link = "identity").

³https://en.wikipedia.org/wiki/Akaike_information_criterion
Include the \texttt{control = list(maxit = 10^3)} option, to give the computer extra computation time.

Ignore the warnings that \texttt{glm()} returns.

• \texttt{<-} the result into \texttt{gamma_glm_2}.
• \texttt{glance()} \texttt{gamma_glm} and \texttt{gamma_glm_2} to determine which has a lower AIC.

Two feature distinguish GLMs from linear models. We’ve already discussed the first feature: GLMs specify the full distribution of the dependent variable, whereas linear models specify only the mean and standard deviation. We will now explore the second distinguishing feature: whereas linear models set the expected value of the dependent variable to a function of a linear combination of independent variables, GLMs can more generally set the expected value of the dependent variable to a function of a linear combination of independent variables. In other words, linear models specify \( E(y) = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n \), whereas GLMs can specify \( E(y) = f(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \), for some function \( f() \).

Let \( f^{-1}(\cdot) \) denote the inverse of function \( f() \), such that \( f^{-1}(f(x)) = x \) for all \( x \). For example:

• if \( f(x) = \exp(x) \) then \( f^{-1}(x) = \log(x) \) since \( f^{-1}(f(x)) = \log(\exp(x)) = x \).
• if \( f(x) = x^2 \) then \( f^{-1}(x) = \sqrt{x} \) since \( f^{-1}(f(x)) = \sqrt{x^2} = x \),
• if \( f(x) = 1/x \) then \( f^{-1}(x) = 1/x \) since \( f^{-1}(f(x)) = \frac{1}{1/x} = x \), and
• if \( f(x) = x \) then \( f^{-1}(x) = x \) since \( f^{-1}(f(x)) = f^{-1}(x) = x \).

Unfortunately, for historical reasons that predate R, we must pass inverse function \( f^{-1}(\cdot) \) into our \texttt{glm()} call, rather than \( f() \) itself. Also, the GLM literature annoyingly refers to \( f^{-1}(\cdot) \) as a “link function” rather than an inverse function. For example, when we specified \texttt{link = "identity"}, in our definition of \texttt{gamma_glm_2}, we told \texttt{glm()} to set the link function to the identity function that maps \( x \) to itself, \( f^{-1}(x) = x \); and since the inverse of the identity function is the identity function, this was an indirect way of telling \texttt{glm()} to set the transformation function to the identity function, \( f(x) = x \). Hence, setting \texttt{link = "identity"} is a roundabout way of telling \texttt{glm()} to set the mean dependent variable to a linear combination of the independent variables, \( \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n \).

But now suppose we wanted to set the mean dependent variable to \( \exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \). This specification uses transformation function \( f(x) = \exp(x) \) and thus uses link function \( f^{-1}(x) = \log(x) \). Accordingly, we run this model with \texttt{link = "log"}:

\begin{verbatim}
gamma_glm_exp <-
  overtime_trips %>%
  glm(
    overtime ~ distance + age,
    family = Gamma(link = "log"),
\end{verbatim}
control = list(maxit = 10^3),
data = .
)

# get shape parameter estimate
gamma_shape_exp <-
gamma_glm_exp %>%
summary %>%
{1/pluck(., "dispersion")}

gamma_shape_exp

> [1] 1.005337

# get mean coefficient estimates
gamma_glm_exp %>%
tidy

> # A tibble: 3 x 5
> term   estimate std.error statistic p.value
> <chr> <dbl>     <dbl>     <dbl>      <dbl>
> 1 (Intercept) 2.11     0.00708     298.       0
> 2 distance  0.0829     0.00143      58.1      0
> 3 age      -0.000958   0.000159     -6.03  0.00000000160

gamma_glm_exp models overtime as a gamma random variable with shape parameter = 1.005337 and mean = exp(2.11 + 0.0829 * distance - 0.000958 * age). For example, the average overtime when distance = 2 and age = 30 is

gamma_glm_exp %>%
augment(  
  newdata = tibble(distance = 2, age = 30),
  type.predict = "response"
) %>%
pull(.fitted)

> [1] 9.447587
The `type.predict = "response"` option ensures that `augment()` returns \( \exp(2.11 + 0.0829 \times 2 - 0.000958 \times 30) = 9.448 \) rather than \( 2.11 + 0.0829 \times 2 - 0.000958 \times 30 = 2.246 \). In general, including `type.predict = "response"` tells `augment()` to return \( \hat{f}(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_n x_n) \), which is our best estimate of the mean, rather than \( \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_n x_n \), which is largely irrelevant.

**Exercise 6.3.** We will now illustrate how the 95th percentile of overtime varies with trip distance and customer age.

- `augment()` `gamma_glm_exp` with `newdata = expand_grid(distance = seq(0, 20, length.out = 100), age = seq(20, 70, length.out = 100))` and `type.predict = "response"`.
- `mutate()` the result to define `percent_95` as the output of the gamma quantile function, `qgamma()`, evaluated with percentile \( p = .95 \), `shape = gamma_shape_exp`, and `scale = .fitted / gamma_shape_exp`.
  - We defined `gamma_shape_exp` right after creating `gamma_glm_exp`.
  - Rather than its mean, `qgamma()` expects you to pass in the gamma’s scale, which is the ratio of its mean to its shape parameter. And in this case, the gamma’s mean is the \(.fitted\) value of our model.
- `ggplot()` the result with a `geom_contour_filled()` layer evaluated with `x = distance`, `y = age`, and `z = percent_95`.
  - You should get linear contour lines, since you’re using a generalized linear model. But the space between your contour lines should decrease exponentially fast, due to the \( f(x) = \exp(x) \) transformation.

**Exercise 6.4.** Which model better fits the data: `gamma_glm_2` or `gamma_glm_exp`?

- `glance()` both models, and pick the one with the smaller AIC.

Exercise 6.4 establishes that we can better explain the data when we transform our linear combination of `distance` and `age` with \( f(x) = \exp(x) \). And we can do even better with inverse function \( f(x) = 1/x \):

```r
gamma_glm_inv <-
overtime_trips %>%
glm(
  overtime ~ distance + age,
  family = Gamma(link = "inverse"),
  control = list(maxit = 10^3),
```
data = .
)

# get shape parameter estimate
gamma_shape_inv <-
gamma_glm_inv %>%
summary %>%
{1/pluck(., "dispersion")}

gamma_shape_inv

> [1] 0.9731811

# get mean coefficient estimates
gamma_glm_inv %>%
tidy

> # A tibble: 3 x 5
> term     estimate  std.error statistic  p.value
> <chr>     <dbl>     <dbl>      <dbl>    <dbl>
> 1 (Intercept)  0.129   0.000741    174.       0
> 2 distance   -0.0116  0.000131   -88.5       0
> 3 age         0.0000946 0.0000166    5.70 0.0000000119

# confirm that gamma_glm_inv has a lower AIC:
gamma_glm_inv %>% glance %>% pull(AIC)

> [1] 2074065

gamma_glm_exp %>% glance %>% pull(AIC)

> [1] 2076437

Note, the link = "inverse" option sets the link function to \( f^{-1}(x) = 1/x \), which sets
the transformation function to \( f(x) = 1/x \) (since \( f(x) = 1/x \) has inverse \( f^{-1}(x) = 1/x \)).
Accordingly, gamma_glm_inv models overtime as a gamma random variable with shape parameter \(= 0.9731\) and mean \(= \frac{1}{(0.129 - 0.0116 \times \text{distance} + 0.0000946 \times \text{age})}\).

Putting the linear combination of distance and age in the denominator may seem strange, but it's actually quite natural, especially when the shape parameter is near one. You see, when the shape parameter is one, the gamma distribution matches the exponential distribution\(^4\), which has only one parameter: the hazard rate, which is the reciprocal of the mean. Hence, imposing \(\text{link = "inverse"}\) sets the dependent variable’s hazard rate, rather than its mean, to a linear combination of the independent variables. The hazard rate of an exponential random variable specifies the probability of the variable realizing in the next instant. Accordingly, with \(\text{link = "inverse"}\), glm() treats the probability that the customer returns the bike in the next second as linear in distance and age, whereas with \(\text{link = "identity"}\), glm() treats the expected trip duration as linear in distance and age.

**Exercise 6.5.** When you're watching a scary movie and the music gets ominous, is the expected length of time until the next jump scare high or low? Is the hazard rate of a jump scare high or low?

**Exercise 6.6.** Create a gamma glm() that models the overtime hazard rate as a linear combination of trip distance, customer age, and dummy variables that indicate (i) the customer user_type and gender and (ii) the day of week and hour of day the trip began.

- Use formula overtime ~ distance + age + user_type + gender + start_wday + start_hour and family = Gamma(link = "inverse").
- <- the result into gamma_glm_inv_2.
- glance() gamma_glm_inv_2 to confirm that it explains the data better than gamma_glm_inv.
  - Show that it has a lower AIC.

**Exercise 6.7.** We will now illustrate how the probability of overtime exceeding 10 minutes varies with distance, user_type, gender, and start_hour.

- Extract the shape parameter from gamma_shape_inv_2 with summary %>% {1/pluck(., "dispersion")}. <- the result into gamma_shape_inv_2.
- Define new_scenarios as a tibble that comprises all possible combinations of distance = 0:4, age = 30, user_type = c("Customer", "Subscriber"), gender = c("male", "female"), start_wday = "Thu", and start_hour = as.character(0:23).
  - expand_grid() these vectors into a tibble with 480 rows and 6 columns.
- augment() gamma_glm_inv_2 to get the .fitted value associated with each of our new_scenarios.
  - Use type.predict = "response".

\(^4\)https://en.wikipedia.org/wiki/Exponential_distribution
\(^5\)https://en.wikipedia.org/wiki/Jump_scare
mutate() the result to set `start_hour = strtoi(start_hour)`, `distance = as.factor(distance)`, and `prob_more_than_10_given_positive = pgamma(q = 10, shape = gamma_shape_inv_2, scale = .fitted / gamma_shape_inv_2, lower.tail = FALSE)`.

- The `pgamma(lower.tail = FALSE)` function calculates the probability that a gamma with a particular shape and scale exceeds a given threshold q.

- `<-` the result into `prob_long_overtime_given_overtime`.

- `ggplot() prob_long_overtime_given_overtime`, with a `geom_line()` layer evaluated with `x = start_hour, y = prob_more_than_10_given_positive, and color = distance`. `facet_grid()` by `user_type` and `gender`.

- `<-` the result into `probability_plot`.

### 6.2.3 Logistic Regression

Characterizing the trip overtime in our full bike sample requires (i) the probability of a trip running over time and (ii) the overtime distribution, conditional on the trip running over time. We’ve modeled the second part, and now we’ll model the first part.

Our dependent variable will be the logical `positive_overtime = overtime > 0`. We can’t assign `positive_overtime` a gamma or normal distribution, because it takes only two values: `TRUE` (which R treats as 1) and `FALSE` (which R treats as 0). Hence, the best we can do is suppose that `positive_overtime` is `TRUE` with probability `p` and `FALSE` with probability `1 − p`, where the value of `p` varies with our covariates, `distance`, `age`, `user_type`, `gender`, `start_wday`, and `start_hour`. The key to our analysis is that the expected value of `positive_overtime` equals the probability that the trip ran over time: 

\[ \mathbb{E}(positive_overtime) = TRUE \cdot p + FALSE \cdot (1 − p) = 1 \cdot p + 0 \cdot (1 − p) = p. \]

Accordingly, we can derive the over time probability from a `glm()` that models the expected value of `positive_overtime`. But for our model to yield coherent probability estimates, we must choose our transformation function so that the output of \( f(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \) is always between 0 and 1. The simplest such function is the logistic function\(^6\): 

\[
    f(x) = \frac{\exp(x)}{1+\exp(x)}.
\]

The inverse of the logistic function is the logit function\(^7\): 

\[
    f^{-1}(x) = \log \left( \frac{x}{1-x} \right).
\]

We invoke this logit link function—and hence the logistic transformation function—by including `link = "logit"` in our `glm()` call:

```r
binomial_glm <-
  bike %>%
  mutate(positive_overtime = overtime > 0) %>%
  glm(
    positive_overtime ~ distance + age,
    family = binomial(link = "logit"),
```

\(^6\)https://en.wikipedia.org/wiki/Logistic_function

\(^7\)https://en.wikipedia.org/wiki/Logit
control = list(maxit = 10^3),
data = .
)

binomial_glm %>%
tidy

> # A tibble: 3 x 5
>   term     estimate std.error statistic  p.value
>   <chr>     <dbl>     <dbl>      <dbl>    <dbl>
> 1 (Intercept) -3.87   0.0102   -380.       0
> 2 distance     1.66   0.00304   546.       0
> 3 age          0.00131 0.000227   5.78 0.00000000727

The binomial() term specifies that positive_overtime has a binary distribution over TRUE and FALSE.

**Exercise 6.8.** Use the binomial_glm estimates to create an expression for the probability that a trip runs over time, as a function of distance and age.

- To confirm the accuracy of your expression, augment() binomial_glm with type.predict = "response" and get the .fitted value associated with tibble(distance = 2, age = 30).

```
binomial_glm %>%
augment(
  newdata = tibble(distance = 2, age = 30),
  type.predict = "response"
) %>%
pull(.fitted)
```

> [1] 0.3749504

**Exercise 6.9.** Create a glm() that models the probability of a trip running over time as a function of distance, customer age, and dummy variables that indicate (i) the customer user_type and gender and (ii) the day of week and hour of day the trip began.

- mutate() bike to define positive_overtime = overtime > 0.
- Run a glm() with formula positive_overtime ~ distance + age + user_type + gender + start_wday + start_hour and family = binomial(link = "logit").
- <- the result into binomial_glm_2.
• glance() binomial_glm_2 to confirm that it explains the data better than binomial_glm.
  – Show that it has a lower AIC.

Exercise 6.10. We will now illustrate how the probability of a trip running over time varies with distance, user_type, gender, and start_hour.

• augment() binomial_glm_2 to get the .fitted value associated with each of the new_scenarios defined in exercise 6.7.
  – Use type.predict = "response".
• mutate() the result to set start_hour = strtoi(start_hour) and distance = as.factor(distance).
• rename() prob_positive = .fitted.
• <- the result into prob_overtime.

• %+% prob_overtime to the probability_plot you made in exercise 6.7, and + aes(y = prob_positive) to update the y-axis.

Exercise 6.11. The probability of a trip lasting at least 30 minutes equals the probability of it running over time (i.e., lasting at least 20 minutes) times the probability of the trip having at least 10 minutes of overtime, conditional on the trip running over time. We will now combine the tibbles we made in exercises 6.7 and 6.10 to illustrate how the probability of a trip lasting at least 30 minutes varies with distance, user_type, gender, and start_hour.

• inner_join() prob_long_overtime_given_overtime and prob_overtime.
• mutate() the result to define prob_duration_at_least_30 = prob_positive * prob_more_than_10_given_positive.
• %+% the result to the probability_plot you made in exercise 6.7, and + aes(y = prob_duration_at_least_30) to update the y-axis.

Exercise 6.12. To gauge the statistical strength of our glm(), we will compare the .fitted values in the positive_overtime = TRUE subsample against those in the positive_overtime = FALSE subsample. If our model has good predictive power then the .fitted values should be much higher in the former subsample.

• augment() binomial_glm_2 with type.predict = "response" to get the .fitted value associated with each observation in bike.
• ggplot() the distribution of .fitted with a geom_density(alpha = .3) layer. Specify fill = positive_overtime to distinguish between the positive_overtime = TRUE and positive_overtime = FALSE subsamples.
  – You should find a high correlation between the expected value of positive_overtime and the actual value of positive_overtime. In other words, you should get much larger .fitted values in the subsample of trips that ran over time.
6.3 Lab: Nuclear Mishaps

6.3.1 Overview

In 1979, the core of the Three Mile Island Reactor 2 lost coolant and melted down, releasing radioactive gases and iodine into the environment. In response, the nuclear industry and federal regulators redoubled their efforts to mitigate the risks of splitting the atom. As part of this effort, the Nuclear Regulatory Commission (NRC) published a guideline for conducting a Probabilistic Risk Assessment (PRA) at a nuclear plant. According to the NRC, a PRA is intended
to estimate risk by computing real numbers to determine what can go wrong, how likely is it, and what are its consequences. Thus, PRA provides insights into the strengths and weaknesses of the design and operation of a nuclear power plant. For the type of nuclear plant currently operating in the United States, a PRA can estimate three levels of risk. A Level 1 PRA estimates the frequency of accidents that cause damage to the nuclear reactor core. This is commonly called core damage frequency (CDF). A Level 2 PRA, which starts with the Level 1 core damage accidents, estimates the frequency of accidents that release radioactivity from the nuclear power plant. A Level 3 PRA, which starts with the Level 2 radioactivity release accidents, estimates the consequences in terms of injury to the public and damage to the environment.

In 1988 the NRC mandated PRA implementation, requiring

each existing plant [to] perform a systematic examination to identify any plant-Specific [sic] vulnerabilities to severe accidents and report the results to the Commission.

The general purpose of this examination, defined as an Individual Plant Examination (IPE), is for each utility (1) to develop an appreciation of severe accident behavior, (2) to understand the most likely severe accident sequences that could occur at its plant, (3) to gain a more quantitative understanding of the overall probabilities of core damage and fission product releases, and (4) if necessary, to reduce the overall probabilities of core damage and fission product releases by modifying, where appropriate, hardware and procedures that would help prevent or mitigate severe accidents.

Did these PRAs decrease incidents at nuclear power plants? To answer this research question, Blanco et al. [2019] gauged whether the PRAs were followed by a reduction in Licensee Event Reports (LERs) filings. A LER is “A reportable event [as] determined by Title 10, Code of Federal Regulations (10 CFR). These reports describe an event with significant safety implications at the plant.” For example, here’s a LER from Peach Bottom Atomic Power Station, Unit 2:
On 8/21/09, Exelon security staff personnel determined that unescorted access was granted to a contract worker who did not provide complete work history information that was relevant to making an access authorization determination made on 9/11/08. It was determined that the individual did not provide complete information to Exelon concerning a previous incident at a non-nuclear work facility where the individual had tested positive in a Fitness-for-Duty test. In accordance with Exelon procedures, this individual would not have been granted unescorted access if this information had been known at the time that unescorted access was granted.

There was no actual adverse security consequences associated with this event. The individual’s activities at PBAPS were limited to work on non-safety related equipment.

This individual’s access had been previously terminated at PBAPS on 9/28/08 as a result of the completion of the individual’s work activities associated with refueling outage P2R17.

There were no previous similar events identified.

And here’s a LER from Palo Verde Nuclear Generating Station, Unit 3:

On March 20, 2003, Unit 3 was in MODE 1, operating at approximately 98 percent power when augmented testing revealed that one main steam safety valve (MSSV) had an as-found lift pressure above the Technical Specification limit of +/- 3 percent of design lift pressure. This MSSV is believed to have experienced setpoint drift.

The valve was reset per plant procedures to +/-1 % of the required setpoint upon completion of valve testing.

We will follow Blanco et al.’s methodology to estimate the relationship between PRAs and LER filing rates. And we will extend their analysis to estimate the relationship between PRAs and employee radiation exposure levels (which Blanco et al. did not study). Before proceeding, skim Blanco et al.’s article, focusing primarily on Sections 1–4.1.

6.3.2 Data

We will use facility, power, shutdown, ler, and radiation from section 1.3, exercise 1.37. Here’s a description of these tibbles:

- facility:
  - pwr_bwr specifies whether the reactor is a pressurized water reactor (PWR) or a boiling water reactor (BWR).
  - pra_month specifies when the facility performed its PRA.
• power:
  – generation specifies the electricity output, in megawatt hours. This variable can be negative because the plants, themselves, use energy.
  – capacity_factor specifies the actual energy output divided by the maximum energy output. This variable can exceed 1, due to changes in reactor capacity over time.

• shutdown:
  – The stop_type = "vol" observations correspond to the stoppages that Blanco et al. [2019] described as “long-term voluntary plant shutdowns, which can take more than a year. Operators implement improvements at the plant at these times.”
  – The stop_type = "invol" observations correspond to the stoppages that Blanco et al. [2019] alluded to when explaining that “Regulators imposed a forced shutdown in only 1.6% of months in the sample.”
  – The stop_type = "refuel" observations correspond to refuelings, which reactors must do “every 12 to 18 months, and this takes anywhere from 30 to 60 days.”

• ler:
  – ler specifies the total number of Licensee Event Reports filed in the given month.
  – actuation_ler, techspec_ler, degrade_ler, inoperable_ler, and safety_eqip_ler variables specify the number of LERs that are flagged with the “actuation”, “technical specifications”, “degradation”, “inoperable”, and “prevent safety equipment” Code of Federal Regulations (CFR) codes.
  – A given LER can be assigned zero or multiple CRF codes.

• radiation:
  – exposure_sum measures the total radiation that the workforce was exposed to collectively.
  – exposure_average measures the radiation that the average worker was exposed to.
  – These variables are measured Roentgen equivalent man⁸ (rem) units.

6.3.3 Clean

We will combine these tibbles into a master panel. But first we will pivot_wider() shutdown into a more convenient form:

```r
shutdown <- shutdown %>%
pivot_wider(

⁸https://www.nrc.gov/reading-rm/basic-ref/glossary/rem-roentgen-equivalent-man.html
id_cols = c(reactor, start),
names_from = stop_type,
values_from = stop_type,
values_fn = length,
values_fill = 0,
names_prefix = "shutdown_")

\>

rename(month = start)

shutdown \>

\>

%>% head

\>

%>-%>% head

> # A tibble: 6 x 5
> reactor month shutdown_vol shutdown_invol shutdown_refuel
> <fct> <date> <int> <int> <int>
> 1 Browns Ferry 2 1984-09-01 1 0 1
> 2 Calvert Cliffs 1 1989-05-01 1 0 0
> 3 Calvert Cliffs 2 1989-03-01 1 0 1
> 4 Clinton 1996-09-01 1 0 0
> 5 Crystal River 3 1996-09-01 1 0 0
> 6 D.C. Cook 1 1997-09-01 1 0 0

This tibble reports the number of shutdown_vol, shutdown_invol, and shutdown_refuel at a given reactor in a given month. For example, the first row indicates that Browns Ferry 2 had both a voluntary shutdown and a refueling shutdown in September, 1984.

**Exercise 6.13.** We will now create our combined panel.

- **left_join()** together facility, power %>% mutate(year = year(month)), ler, radiation, and shutdown in that order.
  - Adding a year column to power gives the radiation observations something to latch onto.

- The resulting tibble’s shutdown_vol, shutdown_invol, and shutdown_refuel variables will be NA unless the plant shut down in the specified month. replace_na() values in these columns to 0, since there were zero shutdowns in these months.

- **arrange()** the result by reactor and month, group_by() reactor, and then redefine shutdown_vol, shutdown_invol, and shutdown_refuel so that they report the cumsum() number of facility shutdowns.
  - Use across() to avoid writing cumsum() multiple times.
  - The last row should indicate that the Wolf Creek 1 reactor had 23 refueling shutdowns.

- Define pra as a logical that indicates whether the facility has performed its PRA by the given month.
• filter() away the observations that fall outside of 1985–1998, since Blanco et al. [2019] explain that they “focus on the period from 1985 to 1998 because this is when PRA was widely adopted by the nuclear industry.”
• ungroup() the tibble and define day = as.integer(month - min(month)) as the number of days since the start of the sample (i.e., since mdy("01-01-1985").
• <- the result into a tibble called combined_sample.
  – This tibble should have 16163 columns and 24 rows.

In addition to our combined_sample we will use the following three tibbles in our analysis:

```r
glm_tib <-
tibble(
  specification = 1:3,
  formula =
  c(
    ler ~ pra + reactor * day + shutdown_refuel,
    ler ~ pra + reactor * day + shutdown_refuel + shutdown_vol,
    ler ~ pra + reactor * day + shutdown_refuel + shutdown_vol + shutdown_invol
  ) %>%
  map(as.formula)
)

glm_tib_2 <-
tibble(
  dependent_variable =
  c(
    "actuation_ler",
    "techspec_ler",
    "degrade_ler",
    "inoperable_ler",
    "safety_equip_ler"
  ),
  formula =
  dependent_variable %>%
  str_c(
    "~ pra + day + reactor + shutdown_refuel",
    "+ shutdown_vol + shutdown_invol"
  ) %>%
  map(as.formula)
)
```
hypothetical_scenarios <-
  combined_sample %>%
  select(-pra) %>%
  expand_grid(pra = c(TRUE, FALSE))

glm_tib and glm_tib_2 define the regressions we will run in exercises 6.17 and 6.20, and hypothetical_scenarios specifies the cases we will consider in our counterfactual analysis. Specifically, this tibble repeats the observations of combined_sample twice, once with pra = TRUE and once with pra = FALSE. We will calculate the .fitted value associated with each of these hypothetical_scenarios to compare what the ler rate would be if every observation had pra = TRUE to what it would be if every observation had pra = FALSE.

6.3.4 Analyze

We will now gradually build up to the Poisson regressions that Blanco et al. [2019] reported in Table 2 of their manuscript.

Exercise 6.14. Use combined_sample to run an lm() (not a glm()) with formula ler ~ pra + day + reactor. tidy() the result to view the coefficient estimates.

The "praTRUE" estimate suggests that implementing a pra decreases the expected ler value by 0.245. This is the key result, and its statistical significance is strong. The "reactorArkansas Nuclear 2", ..., "reactorWolf Creek 1" estimates benchmark against the reactor = "Arkansas Nuclear 1" baseline. For example, Browns Ferry 2 has an average of 0.939 more LERs per month than Arkansas Nuclear 1 (when they have the same pra value). And finally the "day" estimate reports the temporal trend: the nuclear plants become gradually safer over time, as the average ler decreases by 0.000221 per day.

However, Blanco et al. [2019] gave each reactor their own temporal trend. We can incorporate reactor-specific temporal trends into our model by changing the regression formula from ler ~ pra + day + reactor to ler ~ pra + day * reactor:

lm_nuclear <-
  combined_sample %>%
  lm(
    ler ~ pra + day * reactor,
    data = .
  )

#first six estimates:
lm_nuclear %>% tidy %>% head
Now in addition to the estimates we had previously—"(Intercept)", "praTRUE", "day", "reactorArkansas Nuclear 2", ..., "reactorWolf Creek 1"—we have 102 reactor-level time trends: "day:reactorArkansas Nuclear 2", ..., "day:reactorWolf Creek 1". These estimates report the rate of improvement relative to the reactor = "Arkansas Nuclear 1" baseline. For example, the Watts Bar 1 reactor decreased its LER rate by 0.000750 reports per day more than Arkansas Nuclear 1 did. Put differently, after 365 days, the expected LER rate is $1.69 - 0.000196 \cdot 365 = 1.62$ at Arkansas Nuclear 1 and $1.69 - 0.000196 \cdot 365 - 0.000750 \cdot 365 = 1.34$ at Watts Bar 1 (assuming that neither facility implemented a PRA in the first year).

Unfortunately, our linear model has one glaring problem: it sometimes predicts a negative number of LERs, as we will see in the next exercise.

Exercise 6.15. augment() lm_nuclear to anticipate the expected lkr count for reactor == "Turkey Point 4" in December, 1, 1998.

Of course, this forecast is nonsense, as engineers can’t file a negative number of reports. To correct this issue, we will use a glm() with transformation function $f(x) = \exp(x)$. Since the exponential function always returns a positive number, this model ensures that the expected number of LERs—i.e., the fitted value—is positive. Specifically, our glm() will treat lkr as a Poisson random variable whose mean equals the exponentiation of a linear combination
of our independent variables. The Poisson distribution\footnote{https://en.wikipedia.org/wiki/Poisson_distribution} is the simplest distribution whose values are constrained to non-negative integers, so it’s well-suited for modeling “count data” like \( l_{er} \), which has no fractional parts. Furthermore, the “law of rare events”\footnote{https://en.wikipedia.org/wiki/Poisson_limit_theorem} suggests that \( l_{er} \) should indeed resemble a Poisson random variable.

**Exercise 6.16.** Define \( \text{glm}_nuclear \) as the \( \text{glm}() \) analog of \( \text{lm}_nuclear \). Use the Poisson distribution and the \( f(x) = \exp(x) \) transformation function (and hence the \( f^{-1}(x) = \log(x) \) link function).

- Use \texttt{family = poisson(link = "log")}.
- \texttt{tidy()} \texttt{glm_nuclear} to view its estimates.

---

The "pra\text{TRUE}" estimate is -0.153, which suggests that the expected number of LERs should only be \( \exp(-0.153) = 0.858 \) as high after implementing a PRA. For example, the "(Intercept)", "day", "reactorKewaunee", "day:reactorKewaunee" estimates are 0.585, -0.000183, -0.237, and 0.000124, respectively. Thus, the expected LER rate corresponding to \( \text{reactor} = "Kewaunee", \text{day} = 100, \text{pra} = \text{FALSE} \) is

\[
.\text{fitted} = \exp(0.585 - 0.000183 \cdot 100 - 0.237 + 0.000124 \cdot 100) = 1.41
\]

In contrast, the expected LER rate corresponding to \( \text{reactor} = "Kewaunee", \text{day} = 100, \text{pra} = \text{TRUE} \) is

\[
.\text{fitted} = \exp(0.585 - 0.153 - 0.000183 \cdot 100 - 0.237 + 0.000124 \cdot 100) \\
= \exp(-0.153) \exp(0.585 - 0.000183 \cdot 100 - 0.237 + 0.000124 \cdot 100) \\
= 0.858 \cdot 1.41 \\
= 1.21
\]

Note, the \( \exp(x + y) = \exp(x) \exp(y) \) property of the exponential function enables us to isolate \( \exp(-0.153) \).

**Exercise 6.17.** We will now run the regressions that Blanco et al. [2019] reported in Table 2.

- \texttt{mutate()} \texttt{glm_tib} to \texttt{map()} each \texttt{formula} to its corresponding \texttt{glm()}. Save the output as a list column called \texttt{estimated_glm}.
  - Within the \texttt{mutate()} write \texttt{estimated_glm = formula %>% map()}, and then embed the \texttt{glm()} call within the \texttt{map()}.
  - Specify \texttt{data = combined_sample} and \texttt{family = poisson(link = "log")} in the \texttt{glm()}.\footnote{\texttt{family = poisson(link = "log")}}
• <- the output of `mutate()` into `glm_tib`, overwriting the previous version.

**Exercise 6.18.** We will now report our estimates with a simplified version of Blanco et al.’s Table 2.

- Split `glm_tib` `rowwise()` to `tidy()` each model in the `estimated_glm` column. Save the summary statistics in a new column.
- `unnest()` the result by the new column you just made.
- `filter()` the sample down to the observations with `term == "praTRUE"`.
- `select()` the `specification`, `estimate`, `std.error`, `statistic`, and `p.value` columns.
  - Your `estimate` values should differ from those presented in Table 2 of Blanco et al. [2019], because rather than \( \hat{\beta} \) they “report the marginal effects of the results,” which is a related but different statistic.
  - This `estimate` values should indicate a strongly negative relationship between `pra` and `ler`. For example, the last `estimate` should suggest that implementing a `pra` reduces the expected `ler` by \( 1 - \exp(-0.128) = 12\% \).

**Exercise 6.19.** We will now illustrate how `pra` adoption influences the distribution of `ler`. The Poisson distribution has only one parameter: its mean. Hence, the `.fitted` values of our `glm()` fully characterize the distribution of `ler`. Accordingly, we will `augment()` our model to compare the `ler` distribution when `pra = TRUE` to when it is `FALSE`.

- `pluck("estimated_glm", 3)` from `glm_tib` and `augment()` it with `newdata = hypothetical_scenarios` and `type.predict = "response"`.
  - This step conducts a counterfactual analysis, calculating what the expected `ler` values would be in two hypothetical scenarios, the first being the case in which `pra` was always `TRUE`, and the second being the case in which `pra` was always `FALSE`.
  - The first `.fitted` value should indicate that the Arkansas Nuclear 1 reactor would have had an expected `ler` value of 1.597 in January, 1985, if it had conducted its PRA by then.
- `expand_grid()` the resulting tibble by `number_of_shutdowns = 0:5`.
  - This step will repeat each observation for each value of `number_of_shutdowns = 0:5`.
- `mutate()` the result to define `prob = dpois(number_of_shutdowns, .fitted)` as the probability of observing the specified `number_of_shutdowns`, given the observation’s independent variables.
  - The first `prob` should indicate that the Arkansas Nuclear 1 reactor would have had a 20.24% chance of not shutting down in January, 1985, if it had conducted its PRA by then.
- `ggplot()` the result with a `geom_density(alpha = .35)` layer, evaluated with `x = prob` and `fill = pra`. `facet_wrap()` by `number_of_shutdowns` with `scales = "free"`.  

448
The probability of getting \( \text{ler} = 0 \) should be higher with \( \text{pra} = \text{TRUE} \) and the probability of getting \( \text{ler} = 5 \) should be higher with \( \text{pra} = \text{FALSE} \).

After modeling the effect of PRAs on the overall LER rate, Blanco et al. [2019] estimated the effect on each specific type of LER. In other words, they estimated the influence of \( \text{pra} \) on \text{actuation_ler}, \text{techspec_ler}, \text{degrade_ler}, \text{inoperable_ler}, \text{and safety_equp_ler}. We will replicate this analysis.

**Exercise 6.20.** We will now run the regressions that Blanco et al. [2019] reported in Table 8.

- Run a Poisson regression with each formula in \text{glm_tib_2}, and save the output as a list column called \text{estimated_glm}.
  - Copy your solution to exercise 6.17.
- \text{tidy()} each element in \text{estimated_glm} and save the output as a list column called \text{tidied_glm}.
  - See how I defined \text{tidied_glm} for \text{glm_tib}.
- \text{<-} the resulting tibble into \text{glm_tib_2}, overwriting the previous version.

We can now create our own version of Blanco et al.’s Table 8:

```r
glm_tib_2 %>%
  mutate(
    tidied_glm =
      estimated_glm %>%
      map(tidy)
  ) %>%
  unnest(tidied_glm) %>%
  filter(term == "praTRUE") %>%
  select(-c(formula, estimated_glm, term))
```

> # A tibble: 5 x 5
> dependent_variable estimate std.error statistic p.value
> <chr> <dbl> <dbl> <dbl> <dbl>
><|>
In each case, we find a strongly negative relationship between pra and dependent_variable. For example, the first estimate suggests that implementing a pra decreases the expected actuation_ler by $1 - \exp(-0.385) = 31.95\%$.

**Exercise 6.21.** We will now illustrate how the distribution of the expected number of each LER type changes with PRA adoption.

- Use `mutate()` and `map()` to `augment()` each `estimated_glm` in `glm_tib_2`, with `newdata = hypothetical_scenarios` and `type.predict = "response"`. Save the outputs in a list column called `augmented_glm`.
- `unnest()` the resulting tibble by `augmented_glm`.
- `ggplot()` the result with `geom_density(alpha = .35)` layer evaluated with `x = .fitted` and `fill = pra`. `facet_wrap()` by `dependent_variable`.
  - The `pra = FALSE` distributions should be shifted to the right, as the expected number of each type of LER is larger in this case.

Blanco et al. [2019] focused on the effect of PRAs on LERs. But there’s another interesting variable in our sample: the amount of radiation employees were exposed to. We will now test whether implementing PRAs reduced radiation exposure levels.

**Exercise 6.22.** Run a GLM with formula `exposure_average ~ pra + day`. Again use the $f(x) = \exp(x)$ transformation function, but now give the dependent variable a gamma distribution (since it takes non-integer values.)

- Use `data = combined_sample` and `family = Gamma(link = "log")` in your `glm()` call.
- `tidy()` the output to view the coefficient estimates.

The gamma regression you ran in exercise 6.22 yield a statistically insignificant "praTRUE" coefficient estimate. So we fail to reject the null hypothesis that PRAs had no influence on employee radiation exposure levels.
6.3.5 Explore

**Exercise 6.23.** How does the rate of electricity generation depend on the length of time since the last refueling?

- You probably will want to model the relationship with something more flexible than a straight line, which gives you only two degrees of freedom, an intercept and a slope. For example, if you estimated the generation rate separately for each decile of the time since the last refueling then you would have 10 degrees of freedom to model the relationship.
- To better compare across reactors, you may want to redefine $\text{generation} = (\text{generation} - \text{mean(generation)}) / \text{sd(generation)}$, where the $\text{mean()}$ and $\text{sd()}$ are computed at the reactor level.

6.4 Solutions

6.1

```r
baseline_lm <-
  overtime_trips %>%
  lm(
    overtime ~ distance,
    data = .
  )
baseline_lm %>% tidy
```

6.2

```r
gamma_glm_2 <-
  overtime_trips %>%
  glm(
    overtime ~ distance + age,
    family = Gamma(link = "identity"),
    control = list(maxit = 10^3),
    data = .
  )

# Factoring age lowers the AIC, and hence improves the model:
gamma_glm %>% glance %>% pull(AIC)
gamma_glm_2 %>% glance %>% pull(AIC)
```
6.3

```r
gamma_glm_exp %>%
  augment(
    newdata =
      expand_grid(
        distance = seq(0, 20, length.out = 100),
        age = seq(20, 70, length.out = 100)
      ),
    type.predict = "response"
  ) %>%
  mutate(
    percent_95 =
      qgamma(
        p = .95,
        shape = gamma_shape_exp,
        scale = .fitted / gamma_shape_exp
      )
  ) %>%
  ggplot() +
  aes(
    x = distance,
    y = age,
    z = percent_95
  ) +
  geom_contour_filled() +
  theme_minimal()
```

6.4

```r
#gamma_glm_exp has lower AIC, and thus fits better:
gamma_glm_2 %>% glance %>% pull(AIC)
gamma_glm_exp %>% glance %>% pull(AIC)
```

6.5

When the music gets ominous, you know that something freaky will jump out at you *any second now*. So the expected length of time until the next jump scare is low, and the hazard rate of a jump scare is high. These variable are inverses of one another: when one is low the other is high.

6.6
gamma_glm_inv_2 <-
  overtime_trips %>%
  glm(
    overtime ~ distance + age + user_type + gender + start_wday + start_hour,
    family = Gamma(link = "inverse"),
    control = list(maxit = 10^3),
    data = .
  )

#gamma_glm_inv_2 model has lower AIC, and thus fits better:
gamma_glm_inv %>% glance %>% pull(AIC)
gamma_glm_inv_2 %>% glance %>% pull(AIC)

6.7

gamma_shape_inv_2 <-
  gamma_glm_inv_2 %>%
  summary %>%
  {1/pluck(., "dispersion")}

new_scenarios <-
  expand_grid(
    distance = 0:4,
    age = 30,
    user_type = c("Customer", "Subscriber"),
    gender = c("male", "female"),
    start_wday = "Thu",
    start_hour = as.character(0:23)
  )

prob_long_overtime_given_overtime <-
  gamma_glm_inv_2 %>%
  augment(
    newdata = new_scenarios,
    type.predict = "response"
  ) %>%
  mutate(
    start_hour = strtoi(start_hour),
    distance = as.factor(distance),
    prob_more_than_10_given_positive = pgamma(
      q = 10,
      #derive longer-than 10 minute
      #probability from gamma's shape
      shape = gamma_shape_inv_2$dispersion
    )
  ) %>%
shape = gamma_shape_inv_2,  # and mean (i.e., fitted value)
scale = .fitted / gamma_shape_inv_2,
lower.tail = FALSE
)

probability_plot <-
  prob_long_overtime_given_overtime %>%
  ggplot() +
  aes(
    x = start_hour,
    y = prob_more_than_10_given_positive,
    color = distance
  ) +
  geom_line() +
  facet_grid(
    cols = vars(user_type),
    rows = vars(gender)
  ) +
  theme_minimal()

probability_plot

6.8

# the following yield the same probability estimates:
distance <- 2
age <- 30
exp(-3.867568 + 1.65859 * distance + 0.0013117 * age)/(1 + exp(-3.867568 + 1.65859 * distance + 0.0013117 * age))

binomial_glm %>%
  augment(
    newdata = tibble(distance = 2, age = 30),
    type.predict = "response"
  ) %>%
  pull(.fitted)

6.9

binomial_glm_2 <-
  bike %>%
\begin{verbatim}
mutate(positive_overtime = overtime > 0) %>%
glm(
    positive_overtime ~ distance + age + user_type + gender + start_wday + start_hour,
    family = binomial(link = "logit"),
    control = list(maxit = 10^3),
    data = .
)

#gamma_glm_inv_2 model has lower AIC, and thus fits better:
binomial_glm %>% glance %>% pull(AIC)
binomial_glm_2 %>% glance %>% pull(AIC)

6.10

prob_overtime <-
    binomial_glm_2 %>%
    augment(
        newdata = new_scenarios,
        type.predict = "response"
    ) %>%
    mutate(
        start_hour = strtoi(start_hour),
        distance = as.factor(distance)
    ) %>%
    rename(prob_positive = .fitted)

probability_plot %+
prob_overtime +
aes(y = prob_positive)

6.11

data_to_plot <-
    prob_long_overtime_given_overtime %>%
    inner_join(prob_overtime) %>%
    mutate(
        prob_duration_at_least_30 =
            prob_positive * prob_more_than_10_given_positive
    )

probability_plot %+
\end{verbatim}
data_to_plot +
aes(y = prob_duration_at_least_30)

6.12

binomial_glm_2 %>%
augment(type.predict = "response") %>%
ggplot() +
aes(  
x = .fitted,
    fill = positive_overtime
  ) +
geom_density(alpha = .3) +
labs(  
x = "Prob. of Trip Running Over",
    y = "Density",
    fill = "Trip Actually Ran Over Time:
  
  theme_bw() +
theme(legend.position = "top")
Chapter 7

Spatial Data

7.1 Introduction

In this chapter you will learn how to work with geographic data. This is an area that R excels in. The two primary packages for geocomputing in R are sp and sf. The former package is old school: it’s cumbersome but it underpins much of the existing mapping technology in R. The latter package is the new mapping gold standard. It’s easier to work—e.g., it plays nicely with the tidyverse—and it receives more developer attention. We will use this more modern package.

The sf package can be a bit confusing at first, but it the gives you tremendous power once you get the hang of it. Comprising dozens of functions, it has a built-in solution for almost any geometric data problem you may have. All the functions in this package begin with st_, which stands for “spatial temporal.” This this cheat sheet¹ can help you keep them straight.²

The sf package is named after the Simple Features standard for specifying spatial data, which is recognized by both the Open Geospatial Consortium (OGC) and the International Organization for Standardization (ISO). The standard boils geometry down to its simple features: points, lines, and polygons. The package defines three primary objects:

- sfg objects are a collection of basic simple features, such as POINT or MULTIPOLYGON. They can fit in individual cells of a tibble.
- sfc objects are a collection of sfg objects stacked into a column of data. These objects can serve as a column variable in a tibble.
- sf tables are a special type of tibble that contain an sfc column for storing geometric objects and other columns for storing other data.

¹https://github.com/rstudio/cheatsheets/blob/master/sf.pdf
²Other useful reference materials are Geocomputation with R³, Using Spatial Data with R⁴, Drawing beautiful maps programmatically with R, sf and ggplot²⁵, Spatial Manipulation with sf: Cheat Sheet⁶, and vignettes returned by browseVignettes("sf").
We will cover sfg objects in section 7.2.1 of the lecture, and will cover sfc and sf objects in section 7.2.2. Finally, we will cover geometric joins st_intersection() and st_difference() in section 7.2.3. These functions are like inner_join() and anti_join() except they merge tibbles by the geometric objects they comprise. Specifically, they deem two rows “a match” if their geometric objects overlap.

In the lab in section 7.3 we will replicate the empirical study of Bastani and de Zegher [2019]. We will use satellite data to study illegal fishing practices in the open ocean. This is a geometric problem because different rules apply in different regions of the ocean. You can do this lab work in the code.R file of the fishing project you created in exercise 1.38.

7.2 Lecture

7.2.1 Geometric Objects

Download and load the sf and spData packages with the following:

```r
library(sf)
library(spData)
```

The latter package gives us some geometric objects, which we’ll apply the sf package’s functions to. In the sf package, the fundamental geometric unit is the simple feature geometry, which has class sfg. These sfg objects are collections of points, lines, and polygons. For example, `st_linestring()` turns a two-column matrix into squiggly line with corresponding (x, y) coordinates:

```r
crooked_line <-
  runif(20) %>%
  matrix(ncol = 2) %>%
  st_linestring

class(crooked_line)
```

> [1] "XY"    "LINESTRING" "sfg"

As you see, `crooked_line` is a LINESTRING object, which is a type of sfg object. We can `ggplot()` it with `geom_sf()`:

```r
crooked_line %>%
  ggplot() +
  geom_sf() +
  theme_bw()
```
A MULTILINESTRING is a sfg object that describes a collection of squiggly lines:

```r
data <- list(
  runif(20) %>%
    matrix(ncol = 2),
  runif(40, min = 1.5, max = 4) %>%
    matrix(ncol = 2))

multi_crooked_lines <- st_multilinestring(data)

class(multi_crooked_lines)

> [1] "XY" "MULTILINESTRING" "sfg"
```

```r
multi_crooked_lines %>%
  ggplot() +
  geom_sf() +
  theme_bw()
```
We can analogously create POINT, MULTIPOINT, POLYGON, and MULTIPOLYGON sfg objects with `st_point()`, `st_multipoint()`, `st_polygon()`, and `st_multipolygon()` options. As expected, we define a POINT object with a single (x, y) coordinate pair (e.g., `st_point(c(0, 1))`) and a MULTIPOINT object with a matrix of coordinate pairs (e.g., `st_multipoint(matrix(runif(8), 4))`). However, the POLYGON and MULTIPOLYGON objects are more difficult to define. For example, the following constructs and plots a single POLYGON object:

```r
base_triangle <-
  rbind(
    c(0, 0),
    c(1/2, 1/2),
    c(1, 0),
    c(0, 0)
  )

base_square <-
  rbind(
    c(0, 0),
    c(1, 0),
    c(1, 1),
    c(0, 1),
    c(0, 0)
  )

shape_list <-
  list(
    2 * base_square,
    base_square/2 + 1/4,
    base_triangle/6 + 1/2,
    base_square/16 + 1/3,
    2 * base_triangle + 1
  )

complex_polygon <-
  shape_list %>%
  st_polygon

complex_polygon %>%
  ggplot() +
  geom_sf() +
  theme_bw()
```
As you see, a POLYGON object is not a single polygon, but rather a collection of polygons layered on top of one another. This collection of polygons partitions the plane into an “interior” region and an “exterior” region, where the inside points are those that reside in an odd number of polygons and the outside points those that reside in an even number. We construct this object by giving `st_polygon()` a list of matrices (e.g., `shape_list`), where each matrix stores the coordinates of a polygon’s corners. Note, the first and last row of each matrix must be the same, to ensure that the polygons close off (e.g., `base_triangle` both starts and ends at point c(0, 0)).

Whereas a POLYGON object stems from a list of matrices, a MULTIPOLYGON object stems from a list of a list of matrices (or from a list of POLYGON objects). A MULTIPOLYGON object stores a collection of spatial partitions. For example, if your fields are divided into “wheat” and “corn” regions and into “organic” and “non-organic” regions, then you could save these two partitions in one MULTIPOLYGON object.

Finally, the GEOMETRYCOLLECTION object allows you save a general collection of points, lines, and polygons:

```r
list(
  complex_polygon,
  multi_crooked_lines,
  st_point(c(1, 3))
) %>%
  st_geometrycollection %>%
  ggplot() +
  geom_sf() +
  theme_bw()
```
7.2.2 Geometric Tables

Like integers, logicals, or character strings, sfg objects are basic R data elements that we will store in individual cells of a table. To make these geometric objects table-compatible, we stack them into a column of data. A column of sfg objects is called an sfc object (the “c” stands for column). For example, the following uses `st_sfc()` to combine three MULTIPOINT objects into a single sfc_MULTIPOINT object (which is a kind of sfc object).

```r
points_1 <-
  matrix(runif(8), ncol = 2) %>%
  st_multipoint

points_2 <-
  matrix(runif(16), ncol = 2) %>%
  st_multipoint

points_3 <-
  matrix(runif(64), ncol = 2) %>%
  st_multipoint

point_var <-
  st_sfc(
    points_1,
    points_2,
    points_3
  )

class(point_var)

> [1] "sfc_MULTIPOINT" "sfc"
```

We can now cast this sfc object as a tibble column, which assigns a MULTIPOINT object to every row of the table:
```r
mini_tibble <-
  tibble(
    random_data = 1:3,
    dots = point_var
  )
  
mini_tibble
```

However, we usually store geometric data in an sf table, which is just a tibble with a few extra features. For example, we can derive an sf table from mini_tibble with st_as_sf():

```r
mini_sf <-
  mini_tibble %>%
  st_as_sf
```

Changing mini_tibble to mini_sf gives the table extra metadata attributes: a bounding box that encompasses the geometric objects, a MULTIPOINT geometry type, and a two-dimensional XY coordinate system (as opposed to three-dimensional XYZ, or four-dimensional XYZM coordinate systems). The final printed attribute, the CRS, is NA because we haven’t specified the coordinate reference system (CRS). The CRS specifies how longitudes and
latitudes are measured. There are hundreds of such coordinate systems, customized to different regions. They are cataloged in the EPSG Geodetic Parameter Dataset.\textsuperscript{7} The most common CRS is EPSG:4326, otherwise known as the World Geodetic System 1984, or WGS 84. It corresponds to the standard latitudes and longitudes you see on most world maps. We can add this CRS to mini_sf with

```r
mini_sf <-
  mini_sf %>%
  st_set_crs(4326)
```

You’ll also find WGS 84 in the world sf table from the spData package, which you will use in the following exercise.

**Exercise 7.1.** We will now create a map that depicts the life expectancy of countries around the world.

- Remove the countries in world with NA lifeExp.
- Use `ntile()` to bucket the remaining countries into five groups, based on lifeExp. Call the bucketing variable `LE_group`.
- Use `summarise()` to calculate the mean value of lifeExp across each `LE_group`.
  - The mean lifeExp is 58.11 years in the `LE_group = 1` subsample.
  - This step compresses the sf table from 167 rows and 12 columns to five rows and three columns.
- Round lifeExp to the nearest hundredth of a year and save it as a factor variable.
- `ggplot()` the map with the `fill = lifeExp` option.
  - The map is much clearer when lifeExp is a factor than when it’s a number.

Exercise 7.1 illustrates one of the key differences between tibbles and sf tables. If world were a tibble then the `summarise()` operation would discard the unreferenced `geom` column. But sf tables treat geometric columns with deference, so even though we didn’t ask it to, the `summarise()` step aggregated `geom`’s polygons at the `LE_group` level. For example, compare what `summarise()` does to world as a tibble with what it does to world as an sf table:

```r
# tibble summarise()
world %>%
  as_tibble %>%
  group_by(continent) %>%
  summarise(area = sum(area_km2))
```

\textsuperscript{7}EPSG stands for the European Petroleum Survey Group, a defunct organization.
> # A tibble: 8 x 2
> continent area
> <chr> <dbl>
> 1 Africa 29946198.
> 2 Antarctica 12335956.
> 3 Asia 31252459.
> 4 Europe 23065219.
> 5 North America 24484309.
> 6 Oceania 8504489.
> 7 Seven seas (open ocean) 11603.
> 8 South America 17762592.

In the latter case, we get sfc column `geom`, which stores the polygons that characterize the continents.

The following exercises will give you some practice with this feature of sf tables.

**Exercise 7.2.** `nz` is an sf table from the spData package that describes the districts of New Zealand. Its `Sex_ratio` variable reports the number of males living in the district divided by the number of females. We will use this variable to divide the country into masculine and feminine regions.
• Create a logical variable called `mostly_men` that indicates whether `Sex_ratio >= 1`.
• `group_by()` `mostly_men` and `summarise()`. Call the resulting sf table `nz_sex`.
  – `nz_sex` should have two columns and two rows.
  – The `summarise()` operation “flattens” the `geom` column from the polygons that characterize the district partition to the polygons that characterize the coarser `mostly_men` partition.
• `ggplot()` `nz_sex` with the `fill = mostly_men` option.

**Exercise 7.3.** Partition New Zealand, in the fashion of exercise 7.2, into a `richest_part = TRUE` region, in which `Median_income >= 30000`, and a `richest_part = FALSE` region, in which `Median_income < 30000`. Call this new sf table `nz_wealth`. Like `nz_sex`, `nz_wealth` should have two columns and two rows. Plot it.

**Exercise 7.4.** Add country borders to the map you made in exercise 7.1. The regions should remain the same color, but they should now be partitioned by country, such that, for example, we can distinguish Canada from the United States.

• Change one word of your former solution.

### 7.2.3 Geometric Joins

The *intersection* of two shapes is the area that lies in both (e.g., the red-orange portion of the Mastercard logo) and the *difference* of two shapes is the area that lies in the first one and not the second one (e.g., the red portion of the Mastercard logo). The intersection operation is roughly analogous to an `inner_join()` and the difference operation to an `anti_join()`.

`st_intersection()` implements geometric intersections. It takes two sf tables and returns a third sf table that reports the geometric intersections. Specifically, if `x` and `y` are sf tables then `st_intersection(x, y)` joins every row of `x` with every row of `y` for which the `x` and `y` geometric objects intersect, and sets the new geometric variable to the corresponding overlapping region. For example, the following depicts the `st_intersection()` of `nz_sex` and `nz_wealth` (from Exercises 7.2 and 7.3):

```r
st_intersection(
  nz_sex,
  nz_wealth
) %>%
  mutate(
    mostly_men = str_c("men: ", mostly_men),
    richest_part = str_c("rich: ", richest_part)
  ) %>%
ggplot() +
geom_sf()
```

466
st_intersection() outputs an sf table with each (mostly_men, richest_part) combination because both nz_sex regions intersect both nz_wealth regions. However, the mostly_men = TRUE zone intersects the richest_part = TRUE zone only at the border between the West Coast district, which is predominantly male, and the Canterbury district, which is wealthy (the sf package treats the border points as belonging to both districts).  

In the following exercises, you will intersect nz with nz_height, an sf table from spData that lists the 101 highest peaks in New Zealand.

Exercise 7.5. We will now calculate the number of peaks that reside in each district.

- Take the st_intersection() of nz and nz_height.
  - The joined table has the same number of rows as nz_height because each peak intersects only one district.
  - Each row has a POINT geometry because the intersection of a polygon and a point is a point.

- count() the number of times that each Name occurs.
  - This count() operation doesn’t drop the geometry column, as it would with a standard tibble. Instead, it aggregates the mountain coordinates at the district level, so that the districts with multiple peaks have MULTIPOINT geometries.

Exercise 7.6. We will now plot the region of New Zealand that’s within 100 Km of any of its 101 tallest peaks.

---

8You can remove this border line with st_collection_extract("POLYGON"), which discards all geometries besides enclosed polygons.
• Apply `st_buffer(.)`, `100000`) to `nz_height` to add a 100 Km radius to around each peak, turning the individual points to circles.

  – This step turns the POINT geometries to POLYGON geometries.
  – `st_crs(nz_height)$units` confirms that lengths are measured in meters (under this CRS).

• Intersect the result with `nz`.

  – The average `t5e_fid` circle intersects `317/101 = 3.14` districts.

• `group_by(Name)` and `summarise()`.

  – This step flatten the various circles into cohesive regions, which makes the map look cleaner.

• `ggplot()` the result, with `fill = Name`.

The other key spatial join is the geometric difference, which we implement with `st_difference()`. It takes two sf tables and returns a third sf table that reports the geometric differences. Specifically, if `x` and `y` are sf tables then `st_difference(x, y)` joins every row of `x` with every row of `y` for which the `y` geometric object does not subsume the `x` geometric object, and sets the new geometric variable to the region in the `x` object but not the `y` object. For example, the following depicts the `st_difference()` of `nz_sex` and `nz_wealth`:

```r
st_difference(
  nz_sex,
  nz_wealth
) %>%
  mutate(
    mostly_men = str_c("men: ", mostly_men),
    richest_part = str_c("rich: ", richest_part)
  ) %>%
ggplot() +
geom_sf() +
facet_grid(
  rows = vars(richest_part),
  cols = vars(mostly_men)
) +
theme_bw()
```
Note, the output of `st_difference()` does not have a `mostly_men = TRUE, richest_part = FALSE` row, because there's nothing left when we subtract the `richest_part = FALSE` zone from the `mostly_men = TRUE`.

Or for another example, here's the part of New Zealand that's not within 100 Km of any of its 101 tallest peaks:

```r
nz %>%
  st_difference(
    nz_height %>%
      st_buffer(100000) %>%
      summarise
  ) %>%
  ggplot() +
  aes(fill = Name) +
  geom_sf() +
  theme_bw() +
  theme(legend.position = "none")
```
Note, the `summarise()` step after `st_buffer()` is crucial! It takes the union of the 101 circles outputted by `st_buffer()` to create a single MULTIPOLYGON that comprises all points within 100 Km of a peak. Taking the `st_difference()` with respect to this MULTIPOLYGON removes all the points within 100 Km of any peak. The result is an sf table with 16 rows—one for each district—where a given row describes the region of a given district that’s more than 100 Km from all peaks.

In contrast, if we didn’t include the `summarise()` step then the `st_difference()` operation would be performed peak by peak. The result would be an sf table with 1,616 rows—one for each (peak, district) combination—where a given row would describe the region of a given district that’s more than 100 Km from a given peak.

To reiterate, if x and y are sf tables then `st_difference(x, y)` returns the geometric difference of every combination of x and y elements. But we don’t usually want to take geometric differences with respect to individual elements of y. Instead, we usually want to take geometric differences with respect to the entirety of y—i.e., to remove the area in x that belongs to any element of y. And to do this, we must first flatten y with `summarise()` and before implementing the `st_difference()`.

The following exercises provide additional examples.

**Exercise 7.7.** Use `st_convex_hull()` and `st_difference()` to plot the region of the world that does not lie in the convex hull of any Asian country.

- Don’t forget to flatten the hull polygons before differencing them out.
Exercise 7.8. Use `st_convex_hull()` and `st_difference()` to plot the region of the world that does not lie between Ecuador and Japan. That is, remove from our map the diagonal strip that lies between these two countries.

- Be conscious of the order in which you implement the `st_convex_hull()` and `summarise()` steps.

7.3 Lab: Illegal Fishing

7.3.1 Overview

We will study a topic that relates to illegal fishing. It is transshipment of fish from one vessel to another on the open ocean, a practice that has been banned in several regions. Bastani and de Zegher [2019] explain the rationale behind these bans:

A reefer-to-vessel transshipment (simply referred to as a transshipment hereafter) is the act of offloading fish catch from a fishing vessel to a refrigerated cargo vessel (often called a reefer) on the high seas (UN FAO 2011). The cargo vessel would then bring the frozen catch back to port, ensuring that freshly-caught fish does not spoil onboard a fishing vessel that might remain far from shore for months or even years. The cargo vessel would typically also resupply the fishing vessel with fuel and provisions. This practice has obvious economic benefits, allowing fishing vessels to maximize catch as well as minimize fuel costs.

... However, transshipments also significantly reduce transparency in the seafood supply chain by masking where, how, and by whom the fish are caught. Cargo vessels can pick up catch from many fishing boats along their way, enabling them to launder contraband catch into poorly-monitored ports as legally-caught catch.

... Critically, transshipments enable fishing vessels to stay at sea for months or even years at a time, which allows fishing vessels to evade monitoring, enforcement and civil society. These conditions pave the way for human rights abuses, e.g., the Issara Institute (2017) finds that physical abuse of fishermen is three times more likely to occur on vessels that transshipped catch in the open ocean.

Amidst these concerns, civil society groups, several UN organizations, NGOs, and marine policy academics have argued that transshipments on the high seas—the open ocean that does not fall within any country’s jurisdiction—should be banned entirely.

But the efficacy of these bans is uncertain because unscrupulous captains can avoid scrutiny by “flying flags of convenience”—i.e., by registering their vessels with countries with notoriously lax enforcement. So we will follow the methodology of Bastani and de Zegher [2019] to estimate the effect of these bans. Skim the article they wrote before proceeding, so you have a general sense for what we’re going to do.
7.3.2 Data

Access the project data here\(^9\). The following exercises will help you load these tables into your R environment.

**Exercise 7.9.** The files in the `polygons/RFMO` folder specify the geographic boundaries of the Regional Fisheries Management Organizations (RFMOs). Specifically, these data characterize the dotted lines of this map\(^10\). Use `map_df()` to load these polygon maps into an sf table called `RFMO`.

- Although its name suggests that it'll output a data frame, `map_df()` will indeed output an sf table.
- Try using `dir()` with the `full.names = TRUE` option.
- Your sf table should have 32 rows and 12 columns.

**Exercise 7.10.** Use `read_rds()` to load `polygons/eez/eez.rds` into an sf table called `EEZ`. This object specifies the Exclusive Economic Zone boundaries. Bastani and de Zegher [2019] explain that “EEZs are areas in the ocean that typically stretch out 200 nautical miles from a country’s coastline; countries have special rights to fish in these zones as prescribed by the UN Convention on the Law of the Sea.” Following Bastani and de Zegher [2019], we will disregard transshipments that fall within these EEZs.

**Exercise 7.11.** Use `read_csv()` to load transshipment.csv into a tibble called `all_vessels` and load authorized.csv into a tibble called `authorized_vessels`. The former file describes all vessels that are capable of transshipping and the latter describes all vessels that are legally authorized to transship.

- Use the `col_types = "iccciTT"` option when loading transshipment.csv, to ensure that `mmsi` and `imo` are treated as integers rather than doubles (i.e., fractional numbers).

**Exercise 7.12.** Use `read_csv()` to load loiter.csv into a tibble called `loiter` and load encounter.csv into a tibble called `encounter`. The former file lists instances “where these vessels loitered at sea long enough to receive a transshipment” and the latter lists instance “where two vessels (a transshipment vessel and a fishing vessel) were in close proximity long enough to transfer catch, crew or supplies.” Following Bastani and de Zegher [2019], we will interpret these loitering and encounter activities as “detected transshipments.”

7.3.3 Clean

We’ll clean our data with the following exercises.

**Exercise 7.13.** We’ll first derive an sf table called `incidents` from `loiter` and `encounter`.

---

\(^9\)https://www.dropbox.com/sh/hh4e1c0ty12nyqf/AAAvjR01c8Wd_oVaUuOUZH7ma?dl=0

\(^10\)http://www.fao.org/figis/geoserver/factsheets/rfbs.html
• In loiter, (i) define \( \text{lat} = (\text{starting_latitude} + \text{ending_latitude}) / 2 \) and \( \text{long} = (\text{starting_longitude} + \text{ending_longitude}) / 2 \), (ii) rename transshipment_mmsi as mmsi and starting_timestamp as time, and (iii) drop all variables besides mmsi, lat, long, and time.

- Modify encounter to look like loiter (i.e., with columns mmsi, lat, long, and time).
- Use bind_rows() to combine loiter and encounter into a unified tibble called incidents. This tibble should have a new variable called activity, which is either "loiter" or "encounter", based on where the observation originated.
  - Before combining them, give loiter an activity = "loiter" variable and give encounter an activity = "encounter".

• Use st_as_sf(., coords = c(x = "long", y = "lat")) to turn incidents into an sf table.
• Use st_crs(incidents) <- st_crs(RFMO) to give incidents the coordinate reference system used by RFMO (which is the World Geodetic System 1984).

Exercise 7.14. We'll now remove from incidents the vessels that are authorized to transship.

• Use anti_join() to remove the ships in authorized_vessels from all_vessels. Call the resulting tibble unauthorized_vessels
  - Join the tibbles by IMO number.

• Use inner_join() to drop the observations in incidents that do not correspond to a ship in unauthorized_vessels.
  - Join the tibbles by MMSI number.

Exercise 7.15. Our sf tables are cumbersomely large. For example, calling plot(EEZ) will probably crash your R session. Use st_simplify(., dtolerance = .5) to streamline RFMO and EEZ. This operation will replace the complex borders with simpler approximations that are accurate to within a half angular degree of precision.

• Ignore the warning message you get on this step.

Exercise 7.16. Ewell et al. [2017] report that the RFMO zones that ban transshipments (either fully or partially) are SEAFO, IATTC, ICCAT, IOTC, GFCM, and WCPFC. However, these zones sometimes overlap with zones that do not ban transshipments. For example, zones IATTC and SPRFMO generously overlap. When I asked Professor Bastani about this, she responded by explaining that

Yes there's significant overlap in the RFMOs—when there's overlap, we go with the minimum regulation (no ban if there's any rfmo without a ban). The reason being you can always choose your vessel flag to be one that's signatory to the no-ban rfmo. And not have the ban apply.
Exercise 7.17. Following Bastani and de Zegher [2019], we will now “exclude transshipments that occurred within Exclusive Economic Zone (EEZ),” since transshipments bans do not apply there.

Exercise 7.18. Use \texttt{st.intersection()} to add the \texttt{ban} column from \texttt{RFMO} to \texttt{incidents}. This variable indicates whether an incident occurred in a region where transshipments are banned.

Exercise 7.19. To illustrate the relationship between the legality and growth rate of transshipments, we will plot where transshipments occurred each year. Our plot will depict two sources of data with two \texttt{geom_sf()} calls. The first layer will illustrate the \texttt{ban = TRUE} and \texttt{ban = FALSE} polygons of \texttt{RFMO}. And the second layer will illustrate the event points of \texttt{incidents}. 

7.3.4 Analyze

To replicate the primary result of Bastani and de Zegher [2019], we will demonstrate that the prevalence of transshipments has increased more slowly where transshipments are illegal. We will make our case with plots, which will suffice for our purposes.
• Pipe RFMO into ggplot().
• Include an `aes(fill = ban)` call inside a `geom_sf()` call. Nesting the `aes()` expression in this fashion tells R to apply the `fill = ban` option to this layer only.
• Add `scale_fill_grey()` and `theme_bw()` options.
• Call the resulting `ggplot` object `map_plot`.

We’ve now finished our first layer. Print `map_plot` to confirm that you get a globe with the two regions of RFMO depicted with two shades of gray. We will now add our second layer to `map_plot`.

• `mutate()` incidents, setting `year = year(time)`.
• `select()` the `year` and `activity` columns of incidents and call the result `incidents_to_plot`.
  – `incidents_to_plot` will receive the `geometry` column for free.
• Use `+` to add a second `geom_sf()` layer to `map_plot`. Inside this `geom_sf()` call, include options `data = incidents`, `size = .001`, and `color = "red"`.
• `facet_grid()` by `year` and `activity`.

This figure brings the data into relief. And it’s always good to start with a plot like this, provides a zoomed-out, big-picture sense for what the data look like. But, unfortunately, it’s not clear whether the red points proliferate faster in the light-gray or dark-gray zones. To make these relative growth rates more apparent, we’ll put time on the horizontal axis in the next exercise.

**Exercise 7.20.** We will now provide a scatter plot that illustrates how the fraction of transshipments that are illegal has changed over time.

• `arrange()` incidents by `activity` and `time` and then `group_by()` `activity`.
• Use `row_number()` and `round()` to create a variable called `bucket` that groups the data into 100-observation chunks. For example, you could give the first 100 rows `bucket = 0`, the second hundred rows `bucket = 1`, etc.
  – By design, the observations of a given `bucket` happen around the same time.
• `group_by()` `activity` and `bucket`, and use `summarise()` to define `mean_time` as the corresponding average `time` and define `fraction_banned` as the fraction of observations with `ban = TRUE`.
  – Your `sf` table should have 125 rows after this step.
• `ggplot()` the result with `x = mean_time` and `y = fraction_banned`. Include both `geom_point()` and `geom_smooth(method = "lm")` layers, and `facet_wrap()` by `activity`.

475
The fraction of loiter events that are illegal has clearly decreased. For a full-fledged analysis, you would probably want to supplement these graphs with some \texttt{lm()} or \texttt{glm()} regressions. But your plots should always take precedence, as a compelling graph will always be more convincing—and more illuminating—than a set of regression estimates.

7.4 Solutions

7.1

```
world %>%
  filter(!is.na(lifeExp)) %>%
  mutate(LE_group = ntile(lifeExp, 5)) %>%
  group_by(LE_group) %>%
  summarise(
    lifeExp = mean(lifeExp),
    lifeExp = round(lifeExp, 2),
    lifeExp = as.factor(lifeExp)
  ) %>%
  ggplot() +
  aes(fill = lifeExp) +
  geom_sf() +
  labs(fill = 'Life Expectancy')
```

7.2

```
nz_sex <-
  nz %>%
  mutate(mostly_men = Sex_ratio >= 1) %>%
  group_by(mostly_men) %>%
  summarise

nz_sex %>%
  ggplot() +
  aes(fill = mostly_men) +
  geom_sf() +
  theme_bw()
```

7.3
nz_wealth <-
  nz %>%
  mutate(richest_part = Median_income >= 30000) %>%
  group_by(richest_part) %>%
  summarise

nz_wealth %>%
  ggplot() +
  aes(fill = richest_part) +
  geom_sf() +
  theme_bw()

7.4
Change summarise() to mutate() in your solution to exercise 7.1.

7.5

st_intersection(
  nz,
  nz_height
) %>%
  count(Name)

7.6

nz_height %>%
  st_buffer(100000) %>%
  st_intersection(nz) %>%
  group_by(Name) %>%
  summarise %>%
  ggplot() +
  aes(fill = Name) +
  geom_sf() +
  theme_bw() +
  theme(legend.title = element_blank())

7.7

477
world %>%
st_difference(
  filter(. , continent == 'Asia') %>%
  st_convex_hull %>
  summarise
)
%>%
ggplot() +
geom_sf() +
theme_bw()

7.8

world %>%
st_difference(
  filter(. , name_long %in% c('Ecuador', 'Japan')) %>%
  summarise %>%
  st_convex_hull
)
%>%
ggplot() +
geom_sf() +
theme_bw()


Back Matter

Acknowledgments

I would like to thank everyone that contributed to the tidyverse. Besides the smartphone and mRNA vaccines, these packages have added more value to my life than any other technological innovation of the past 20 years. The consistent growth and development of these functions has been a heartening reminder of the steady march of human progress. So to the geniuses that made these amazing functions, I salute you.

I would like to thank Mike Joseph Ng and Bob McDonald for inspiring me to write this book.

Many thanks to Hamsa Bastani and Christian Blanco for walking me through the technical details of their studies.

Caleb Kwon gave me valuable feedback, which I tremendously appreciate.

Thank you Alex Frankel for reviewing the entire manuscript. I’m humbled to have such a tremendous mind catch such lowly errata.

Cover Design

The cover was designed by my twin, Jonathan Bray. Jonathan graduated from UC Davis with a degree in graphic design. He lives in Indonesia where he works as a full-time artist.

Data Copyright

I created several of the datasets in this book. Specifically, I simulated random analogues for the samples that I did not have the copyright to distribute. These simulated samples are of my own creation, but I meticulously preserved the basic structure and flavor of the original datasets—with a fleeting glance you wouldn’t be able to distinguish the synthetic data from the original data. And since the samples are structured the same, your code would be just as valid under the true datasets as under the simulated datasets.

11https://jjbrayart.com/