Homage to R
Learn the Language of Data Science

Robert L. Bray

2021-04-22
To Sarah and Frida.
Preface

This book will teach you the basics of the R programming language. R is a beautiful language, custom-made for data science.

The difference between manipulating data with code and manipulating it with a mouse—as you would with, say, Excel or Google Sheets—is like the difference between typing and writing longhand. A pen is ideal for signing checks and writing notes to loved ones but wholly inappropriate for writing contracts or novels. Likewise, Google Sheets is great for organizing the little scraps of data that garnish everyday life but wholly inappropriate for serious empirical inquiry.

The main benefit of code is that it can be read in a linear fashion. A computer program organizes and documents your workflow, laying bare your empirical analysis in the sequence in which it was performed. No matter how complex your statistical models are, your code always flows from start to end. In contrast, Excel spreadsheets tend 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 different-colored string that you see in police procedurals.

The other benefit of code is that it is automated. In contrast, creating a statistical report with a point-and-click protocol usually entails some manual labor—e.g., copying and pasting data from one spreadsheet to another—that is costly to repeat and not clearly documented. True, you can fully automate most Excel calculations, but most people do not. 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.

Granted, there are some Excel gurus who can quickly bend any spreadsheet to their will. 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 really 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 glossed 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 avoided getting 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—was introduced in 2014. For a second example, the primary data-pivoting
functions have been supplanted twice in the decade since I started using R. 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 big headache: First, I’d have
to get funding from my university. Then I’d have to figure out how to register the server
with MATLAB so they can bill me for every hour I use. (Yes, they charge by the hour.)
And then I’d have to submit monthly expense reports from MATLAB to my employer for
reimbursement. Using for-profit software thus adds an additional 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. Multiply that by a dozen data scientists and that’s real money. 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
could make your Alteryx expertise worthless. But this threat doesn’t exist for R because
no employer would begrudge you a free product.

I’ve divided this book into lectures, commentaries, and labs. The lectures provide the core
information that you must have to do the labs. The commentaries provide miscellaneous
tips, tricks, gripes, and rants. And the labs are where you will do real data science. I
emphasize real because every lab, other than the first one, replicates a published empirical
study, from raw data to primary results. These labs study

- mishaps at nuclear power plants,
- case scheduling at the labor appellate court of Rome,
- Alibaba’s package-delivery scores,
- auto part defects reported to the NHTSA,
- grocery-store inventory runs (the supply chain analog of bank runs), and
- illegal fishing on the high seas.

These labs all relate to operations management. This is the discipline that I know best, and
it pairs extremely well with data analytics—an explosion of new logistical datasets gives an
organization’s operations unprecedented visibility. Hence, we can now study the intricate workings of business processes like we never could in the past. And we interface with these processes via databases rather than wrenches and stopwatches.

I wrote this book because I believe teaching R is the most effective way for me to “level up” the students at my business school (the Kellogg School of Management). Indeed, I sincerely believe that there’s no more efficient way to give human capital to business students than to teach them R. Statistical programming is a concrete, marketable skill that 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 business will inexorably progress into the management of data.
# Contents

1 Data Structures .................................................. 1
   1.1 Introduction ................................................. 1
   1.2 Lecture ..................................................... 2
      1.2.1 RStudio ................................................. 2
      1.2.2 Variables .............................................. 5
      1.2.3 Vectors ................................................. 8
      1.2.4 Logicals, Doubles, and Integers ....................... 14
      1.2.5 Character Strings, Dates, and Factors ............... 20
      1.2.6 Tibbles ................................................. 29
   1.3 Lab: Workspaces ............................................ 42
   1.4 Commentary ................................................ 51
      1.4.1 <- vs. = ............................................... 52
      1.4.2 Saving and Loading Data ............................... 52
      1.4.3 Automating Code ..................................... 54
      1.4.4 Glossary of stringr Functions ....................... 55
   1.5 Solutions ................................................ 60

2 Functions ....................................................... 65
   2.1 Introduction ................................................. 65
   2.2 Lecture ..................................................... 66
      2.2.1 Calling Functions ..................................... 66
      2.2.2 Conditional Statements ............................... 69
      2.2.3 Functional Assembly Lines ......................... 73
      2.2.4 Defining Functions ................................... 80
## Pivots and Plots

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>145</td>
</tr>
<tr>
<td>3.2</td>
<td>Lecture</td>
<td>147</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Data</td>
<td>147</td>
</tr>
<tr>
<td>3.2.2</td>
<td><code>ggplot()</code></td>
<td>148</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Case Study: Best Places to Gamble</td>
<td>158</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Case Study: Knockouts Occur Earlier than Submissions</td>
<td>159</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Updating <code>ggplot()</code> Data</td>
<td>160</td>
</tr>
<tr>
<td>3.2.6</td>
<td><code>pivot_longer()</code> and <code>pivot_wider()</code></td>
<td>163</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Case Study: House Takes Have Increased</td>
<td>176</td>
</tr>
<tr>
<td>3.2.8</td>
<td>Case Study: Winners Punch More</td>
<td>177</td>
</tr>
<tr>
<td>3.2.9</td>
<td>Semi-Lengthening Pivots</td>
<td>179</td>
</tr>
<tr>
<td>3.3</td>
<td>Lab: Alibaba</td>
<td>187</td>
</tr>
</tbody>
</table>
# Lists and Linear Models

5.1 Introduction .................................................. 283
5.2 Lecture ......................................................... 284
  5.2.1 Lists ....................................................... 284
  5.2.2 Nested Tibbles ............................................. 301
  5.2.3 Case Study: Wine Reviews ................................. 307
  5.2.4 Linear Regression ......................................... 313
  5.2.5 Running Regression with Lists ........................... 324
5.3 Lab: Supply Chain Proximity .................................. 326
  5.3.1 Overview ................................................... 326
  5.3.2 Data ......................................................... 326
  5.3.3 Analysis .................................................... 328
5.4 Commentary ..................................................... 331
  5.4.1 Vectorize Function Inputs ................................ 331
  5.4.2 Tibbles are Lists .......................................... 334
5.5 Solutions ......................................................... 337

# Generalized Linear Models

6.1 Introduction ..................................................... 349
6.2 Lecture ......................................................... 350
  6.2.1 Limits of Linear Models ................................. 350
  6.2.2 Generalized Linear Models ............................... 352
  6.2.3 Application: Shooting Stars ......................... 363
  6.2.4 Logistic Regression ....................................... 365
6.3 Lab: Nuclear Mishaps ........................................... 368
  6.3.1 Overview ................................................... 368
  6.3.2 Sample ..................................................... 370
  6.3.3 Clean ....................................................... 371
  6.3.4 Analyze .................................................... 373
6.4 Solutions ......................................................... 377
7 Spatial Data

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>389</td>
</tr>
<tr>
<td>7.2 Lecture</td>
<td>390</td>
</tr>
<tr>
<td>7.2.1 Geometric Objects</td>
<td>390</td>
</tr>
<tr>
<td>7.2.2 Geometric Tables</td>
<td>395</td>
</tr>
<tr>
<td>7.2.3 Geometric Joins</td>
<td>399</td>
</tr>
<tr>
<td>7.3 Lab: Illegal Fishing</td>
<td>405</td>
</tr>
<tr>
<td>7.3.1 Overview</td>
<td>405</td>
</tr>
<tr>
<td>7.3.2 Import</td>
<td>406</td>
</tr>
<tr>
<td>7.3.3 Clean</td>
<td>406</td>
</tr>
<tr>
<td>7.3.4 Analyze</td>
<td>408</td>
</tr>
<tr>
<td>7.4 Solutions</td>
<td>410</td>
</tr>
</tbody>
</table>
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 1.4.3 I’ll discuss the importance of writing self-sufficient code—code that can be run in full without human intervention. And in section 1.4.4 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
Here's 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 page, and see the additional links provided here.

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). 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

---

2. https://rstudio.com/resources/cheatsheets/
4. I 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:

---

5 More 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.

6 By 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.
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, 76,
    "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)
```

```r
> [1] 10
```

```r
cos(pi)
```

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

```
> [1] 1024

#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
```

Also, 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`:

```
debt_2012 <- 16066241407385
```

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:

---

7We 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.
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

```
(26945391194615-19573444713936) / (19573444713936-16066241407385)
```

> [1] 2.101944

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:

```
pop_2012 <- 313.83 * 10^6
pop_2016 <- 322.94 * 10^6
pop_2020 <- 330.66 * 10^6
```

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

```
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
```

1.2.3 Vectors

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

```r
debt_1950 <- 0.257 #measured in trillions of dollars
pop_1950 <- 76.1 #measured in millions of people

debt_1951 <- 0.255
pop_1951 <- 77.6

debt_1952 <- 0.259
pop_1952 <- 79.2

# ...

debt_2020 <- 26.95
pop_2020 <- 330.7
```

But scattering the information across so many distinct variables makes it difficult to work with. For example, translating the units of measure from trillions of dollars to dollars and from millions of people to people would require the following 142 transformations:
And expressing debt in per capita terms would require the following 71 transformations:

```r
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()`:

```r
debt <-
c(0.257, 0.255, 0.259, 0.266, 0.271, 0.274, 0.273, 0.271, 0.276, 0.285, 0.286, 0.289, 0.298, 0.306, 0.312, 0.317, 0.320, 0.326, 0.348, 0.354, 0.371, 0.398, 0.427, 0.458, 0.475, 0.533, 0.620, 0.699, 0.772, 0.827, 0.908, 0.998, 1.142, 1.377, 1.572, 1.823, 2.125, 2.350, 2.602, 2.857, 3.233, 3.665, 4.065, 4.411, 4.693, 4.974, 5.225, 5.413, 5.526, 5.656, 5.674, 5.807, 6.228, 6.783, 7.379, 7.933, 8.507, 9.008, 10.03, 11.91, 13.56, 14.79, 16.07, 16.74, 17.82, 18.15, 19.57, 20.25, 21.52, 22.72, 26.95)
pop <-
c(152.3, 154.9, 157.6, 160.2, 163.0, 165.9, 168.9, 172.0, 174.9, 177.8, 180.7, 183.7, 186.5, 189.2, 191.9, 194.3, 196.6, 198.7, 200.7, 202.7, 205.1, 207.7, 209.9, 211.9, ...
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 `<-`.

We can `pluck()` out specific elements of these vectors:

```r
pluck(debt, 2)  # get second element debt
> [1] 0.255

pluck(pop, 4)  # get fourth element pop
> [1] 160.2
```

Or we can view their first several elements with `head()`:

```r
head(debt, 4)  # get first four debt values
> [1] 0.257 0.255 0.259 0.266

head(debt, 5)  # get first five pop values
> [1] 0.257 0.255 0.259 0.266 0.271
```

Storing all the debt values in the same place enables us to transform them in one fell swoop. For example, we can now translate the unit of measure from trillions of dollars to dollars in a single step:

```r
213.8, 216.0, 218.0, 220.2, 222.6, 225.1, 227.2, 229.5, 231.7, 233.8, 235.8, 237.9, 240.1, 242.3, 244.5, 246.8, 249.6, 253.0, 256.5, 259.9, 263.1, 266.3, 269.4, 272.6, 275.9, 279.0, 282.2, 285.0, 287.6, 290.1, 292.8, 295.5, 298.4, 301.2, 304.1, 306.8, 309.3, 311.6, 313.8, 316.0, 318.3, 320.6, 322.9, 325.0, 326.7, 328.2, 329.9
)
```

---

*The keyboard shortcut for writing the `<-` symbol is “Alt -“ (Windows and Linux) or “Option -“ (Mac).*
In the output above, \( e^{+11} \) stands for \( 10^{11} \) (i.e., 100 billion). And with two more steps we can change the \( \text{pop} \) unit of measure from millions of people to people, and express the debt in per-capita terms:

```r
debt <- debt * 10^12
head(debt)
```

> [1] 2.57e+11 2.55e+11 2.59e+11 2.66e+11 2.71e+11 2.74e+11

Note that R performs the above division element by element, setting \( \text{pluck}(\text{debt}, 1) = 2.57e+11/152300000 = 1687.459 \), \( \text{pluck}(\text{debt}, 2) = 2.55e+11/154900000 = 1646.223 \), etc.

Now, since \( \text{debt} \) encompasses all debt values, we can pass this one vector into various R functions to analyze the history of US debt:

```r
min(debt)
```

> [1] 1573.217

```r
mean(debt)
```

> [1] 17248.4
median(debt)

> [1] 7662.884

max(debt)

> [1] 81691.42

quantile(debt, probs = c(.25, .75)) # inter-quartile range

>  25%  75%  
> 1710.695 22518.335

sd(debt) # standard deviation

> [1] 20556.64

In addition to concatenation function `c()`, we can create vectors with repeat function `rep()`. For example, we can create a vector that repeats the number π five times with

rep(pi, 5)


Or we could use sequence function `seq()`. For example, the following creates a vector comprising the numbers between 1 and 10, inclusive:

seq(1, 10)

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

The following creates a vector comprising the numbers from 5 to 100, counting by 8
seq(5, 100, by = 8)

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

And the following creates a vector of five evenly spaced numbers between -.001 and .02

seq(-.001, .02, length.out = 5)

> [1]  -0.00100  0.00425  0.00950  0.01475  0.02000

And the following creates a vector that comprises three copies of the previous vector:

rep(seq(-.001, .02, length.out = 5), 3)

> [1] -0.00100  0.00425  0.00950  0.01475  0.02000 -0.00100  0.00425  0.00950  0.01475  0.02000
> [9]  0.01475  0.02000 -0.00100  0.00425  0.00950  0.01475  0.02000

Also, if x and y are numbers, then x:y returns a vector comprising the integers between them. For example, the following defines year as the years for which we have debt data:

year <- 1950:2020
year


**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. 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 element of `debt_change` should be `NA`, the second element should be `pluck(debt, 2) - pluck(debt, 1)`, the third element should be `pluck(debt, 3) - pluck(debt, 2)`, etc.

Exercise 1.5. Use `lag()` to define `debt_growth` as the vector of year-over-year fractional debt changes. The first element of `debt_growth` should be `NA`, the second element should be `(pluck(debt, 2) - pluck(debt, 1)) / pluck(debt, 1)`, the third element should be `(pluck(debt, 3) - pluck(debt, 2)) / pluck(debt, 2)`, etc.

• 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).

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
Let me illustrate with a concrete application: If `debt_change <- debt - lag(debt)` is a vector of year-over-year debt changes (see exercise 1.4), 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`:

```r
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:

```r
#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`. The preceding example used the greater-than symbol `>` to

---

9The `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.
convert numeric vector `debt_change` to logical vector `debt_increase`. We can also create
logicals with the greater-than-or-equal-to symbol `>=`, exactly-equal-to symbol `==`, less-than-or-equal-to 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

```r
any(FALSE, FALSE, FALSE, FALSE)
```

> [1] FALSE

```r
#Are ALL of the values TRUE?
all(FALSE, TRUE, TRUE, TRUE)
```

> [1] FALSE
Alternatively, we can combine logicals with the and, or, and not operators, &, |, and !:

- x & y = TRUE if both x = TRUE and y = TRUE, and otherwise equals FALSE.
- x | y = TRUE if x = TRUE or y = TRUE, and otherwise equals FALSE.
- !x = TRUE if x = FALSE, and otherwise equals FALSE.

Note that all(x, y, z) = x & y & z and any(x, y, z) = x | y | z.

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

```r
c(-1, 0, 2) %in% 1:3
```

> [1] FALSE FALSE TRUE

```r
debt %in% c(min(debt), max(debt))
```

> [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
> [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

**Exercise 1.6.** Which of the following expressions are TRUE?
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:

```r
is.double(2)
> [1] TRUE
```

```r
is.integer(2)
> [1] FALSE
```

To define a number as an integer, use `as.integer()` or write an “L” after it:

```r
is.double(2L)
> [1] FALSE
```

```r
is.integer(2L)
> [1] TRUE
```

Admittedly, this “L” convention is a bit awkward: e.g., `year <- 1914` appears more natural than `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:
Variable `number_killed` should be exactly ten. And it looks like it is:

```r
number_killed <- 10 + 10 + .1 - 10 - .1

number_killed
```

> [1] 10

But it’s not:

```r
number_killed > 10
```

> [1] TRUE

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] 1.776357e-15

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.\(^{10}\)

\(^{10}\)This is a do-as-I-say-not-as-I-do recommendation, as I violate this rule quite often.
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):

```r
battle_name <- c(
  "Battle of Fort Sumter",
  "Battle of Sewell's Point",
  "Battle of Aquia Creek",
  "Battle of Philippi (West Virginia)",
  "Battle of Big Bethel",
  "Battle of Boonville (Missouri)",
  "Battle of Hoke's Run",
  "Battle of Carthage",
  "Battle of Rich Mountain",
  "Battle of Blackburn's Ford",
  "First Battle of Bull Run or First Manassas",
  "Battle of Wilson's Creek or Oak Hills"
)
```

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:

```r
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"

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"
```
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
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()`.

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 1.4.4, 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
str_replace_na(c("a", "b", NA, "d", NA), "")
```

> [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
str_detect(
  battle_name,
  " or "
)
```

\(^{11}\)As you see, all the stringr functions begin with prefix `str_`.

---

**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
```
Third, the following removes all the instances of "Battle of " from `battle_name`:

```r
data <- c("Fort Sumter", "Sewell's Point")

head(data)
```

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

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

head(battle_name, 3)
```

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

```r
data <- 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 14, 1861")
```

23
"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():

```r
battle_start <-
  mdy(battle_start)

battle_end <-
  mdy(battle_end)
```

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:

```r
battle_end + \text{days}(1) - battle_start
```

\texttt{
> Time differences in days
> [1] 2 2 4 1 1 1 1 1 1 1
}

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}We refer to Date variables with a capital “D” so that we can use "d" as a shorthand for Dates and "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] 58448 58412 58399 58397 58390 58383 58368 58365 58359 58352 58349 58329

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")
```


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()
```

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()
```
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
```

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

```r
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:\(^\text{14}\)

\(^{13}\)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:
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.\(^\text{15}\)

### 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
c(1, 2, 3, "a", "b", "c")

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

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.
"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:

```r
battle_data <-
  tibble(
    start = battle_start,
    end = battle_end,
    casualty = battle_casualties,
    victor = battle_victor,
    name = battle_name,
    desc = battle_description
  )
```

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. 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 its glory with `View(battle_data)`.

We can get a summary of its variables with:

```r
glimpse(battle_data)
```

```r
> Rows: 12
> Columns: 6
> $ start <date> 1861-04-12, 1861-05-18, 1861-05-29, 1861-06-03, 1861-06-10, ~
> $ end <date> 1861-04-13, 1861-05-19, 1861-06-01, 1861-06-03, 1861-06-10, ~
> $ casualty <dbl> 0, 10, 10, 30, 86, 87, 114, 244, 346, 151, 4690, 2549
> $ victor <chr> "Confederate", "Inconclusive", "Inconclusive", "Union", "Conf-
> $ name <chr> "Battle 1: Fort Sumter", "Battle 2: Sewell's Point", "Battle ~
> $ desc <chr> "Beauregard takes Charleston Federal fort", "Union gunboats f-
```

```r
# or with
summary(battle_data)
```

```r
> start end casualty victor
> Min. :1861-04-12 Min. :1861-04-13 Min. : 0.0 Length:12
> 1st Qu.:1861-06-01 1st Qu.:1861-06-02 1st Qu.: 25.0 Class :character
> Median :1861-06-24 Median :1861-06-24 Median :100.5 Mode :character
> Mean :1861-06-20 Mean :1861-06-20 Mean :693.1
> 3rd Qu.:1861-07-12 3rd Qu.:1861-07-12 3rd Qu.:269.5
> Max. :1861-08-10 Max. :1861-08-10 Max. :4690.0
> name desc
> Length:12 Length:12
> Class :character Class :character
> Mode :character Mode :character
```

We can get its top three rows with:

```r
head(battle_data)
```

and messier. So we’ll use tibbles throughout this book. (Although nothing substantive would change if we switched to data frames.)
```r
head(battle_data, 3)

> # A tibble: 3 x 6
> start   end   casualty victor name desc
> <date>  <date>   <dbl>   <chr> <chr> <chr>
> 1 1861-04-12 1861-04-13 0 Confeder~ Battle 1: Fo~ Beauregard takes Charl~
> 2 1861-05-18 1861-05-19 10 Inconclu~ Battle 2: Se~ Union gunboats fight C~
> 3 1861-05-29 1861-06-01 10 Inconclu~ Battle 3: Aq~ Confederate artillery ~

We can get its bottom four rows with:

tail(battle_data, 4)

> # A tibble: 4 x 6
> start   end   casualty victor name desc
> <date>  <date>   <dbl>   <chr> <chr> <chr>
> 1 1861-07-11 1861-07-11 346 Union Battle 9: Rich M~ Union forces divide~
> 2 1861-07-18 1861-07-18 151 Confede~ Battle 10: Black~ Irvin McDowell's re~
> 3 1861-07-21 1861-07-21 4690 Confede~ Battle 11: First~ Thomas Jackson earn~
> 4 1861-08-10 1861-08-10 2549 Confede~ Battle 12: Wilso~ First major battle ~

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 desc
> <date>  <date>   <dbl>   <chr> <chr> <chr>
> 1 1861-06-03 1861-06-03 30 Union Battle 4: Phil~ Small Confederate det~
> 2 1861-07-05 1861-07-05 244 Confede~ Battle 8: Cart~ Confederate victory i~
> 3 1861-07-11 1861-07-11 346 Union Battle 9: Rich~ Union forces divide C~
> 4 1861-07-18 1861-07-18 151 Confede~ Battle 10: Bla~ Irvin McDowell's reco~

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

slice_min(battle_data, order_by = desc, n = 2)
```

32
> # A tibble: 2 x 6
>   start   end casualty victor name desc
>  <date> <date>   <dbl> <chr> <chr> <chr>
> 1 1861-04-12 1861-04-13   0  Confeder~ Battle 1: Fo~ Beauregard takes Charl~
> 2 1861-05-29 1861-06-01 10 Inconclu~ Battle 3: Aq~ Confederate artillery ~

We can get its last three rows when it's ordered by \texttt{end} with:

```r
slice_max(
  battle_data,
  order_by = \texttt{end},
  \texttt{n} = 3
)
```

> # A tibble: 3 x 6
>   start   end casualty victor name desc
>  <date> <date>   <dbl> <chr> <chr> <chr>
> 1 1861-08-10 1861-08-10 2549  Confede~ Battle 12: Wilso~ First major battle ~
> 2 1861-07-21 1861-07-21 4690  Confede~ Battle 11: First~ Thomas Jackson earn~
> 3 1861-07-18 1861-07-18   151  Confede~ Battle 10: Black~ Irvin McDowell's re~

We can get a random two rows with:

```r
slice_sample(
  battle_data,
  \texttt{n} = 2
)
```

> # A tibble: 2 x 6
>   start   end casualty victor name desc
>  <date> <date>   <dbl> <chr> <chr> <chr>
> 1 1861-06-03 1861-06-03   30  Union  Battle 4: Phili~ Small Confederate deta~
> 2 1861-07-11 1861-07-11  346  Union  Battle 9: Rich ~ Union forces divide Co~

We can get its \texttt{name} and \texttt{casualty} columns with:

```r
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
> 4 Battle 4: Philippi (West Virginia)  30
> 5 Battle 5: Big Bethel         86
> 6 Battle 6: Boonville (Missouri) 87
> 7 Battle 7: Hoke's Run         114
> 8 Battle 8: Carthage           244
> 9 Battle 9: Rich Mountain      346
>10 Battle 10: Blackburn's Ford  151
>11 Battle 11: First Bull Run or First Manassas 4690
>12 Battle 12: Wilson's Creek or Oak Hills 2549

# or with
select(
  battle_data,
  5, 3   #column numbers
)

# 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
 4 Battle 4: Philippi (West Virginia)  30
 5 Battle 5: Big Bethel         86
 6 Battle 6: Boonville (Missouri) 87
 7 Battle 7: Hoke's Run         114
 8 Battle 8: Carthage           244
 9 Battle 9: Rich Mountain      346
10 Battle 10: Blackburn's Ford  151
11 Battle 11: First Bull Run or First Manassas 4690
12 Battle 12: Wilson's Creek or Oak Hills 2549

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          desc                                      
> <date>  <date>   <chr>            <chr>                                    
> 1 1861-04-12 1861-04-13 Confederate        Beauregard takes Charleston Federal fort
> 2 1861-05-18 1861-05-19 Inconclusi- Union gunboats fight Confederate artillery
> 3 1861-05-29 1861-06-01 Inconclusi- Confederate artillery hit by naval bombard-
> 4 1861-06-03 1861-06-03 Union             Small Confederate detachment rooted out of-
> 5 1861-06-10 1861-06-10 Confederate Union attack near a church repelled
> 6 1861-06-17 1861-06-17 Union             Union forces defeat Missouri State Guard
> 7 1861-07-02 1861-07-02 Union             Patterson defeats Jackson, but squanders v-
> 8 1861-07-05 1861-07-05 Confederate Confederate victory in Missouri
> 9 1861-07-11 1861-07-11 Union             Union forces divide Confederate forces; ha-
> 10 1861-07-18 1861-07-18 Confederate Irvin McDowell's recon-in-force defeated a-
> 11 1861-07-21 1861-07-21 Confederate Thomas Jackson earned the moniker 'Stonewa-
> 12 1861-08-10 1861-08-10 Confederate First major battle west of the Mississippi
```

```r
#or with
select(
  battle_data,
  -c(5, 3)
)

> # A tibble: 12 x 4
> start   end       victor          desc                                      
> <date>  <date>   <chr>            <chr>                                    
> 1 1861-04-12 1861-04-13 Confederate        Beauregard takes Charleston Federal fort
> 2 1861-05-18 1861-05-19 Inconclusi- Union gunboats fight Confederate artillery
> 3 1861-05-29 1861-06-01 Inconclusi- Confederate artillery hit by naval bombard-
> 4 1861-06-03 1861-06-03 Union             Small Confederate detachment rooted out of-
> 5 1861-06-10 1861-06-10 Confederate Union attack near a church repelled
> 6 1861-06-17 1861-06-17 Union             Union forces defeat Missouri State Guard
> 7 1861-07-02 1861-07-02 Union             Patterson defeats Jackson, but squanders v-
> 8 1861-07-05 1861-07-05 Confederate Confederate victory in Missouri
> 9 1861-07-11 1861-07-11 Union             Union forces divide Confederate forces; ha-
>10 1861-07-18 1861-07-18 Confederate Irvin McDowell's recon-in-force defeated a-
>11 1861-07-21 1861-07-21 Confederate Thomas Jackson earned the moniker 'Stonewa-
>12 1861-08-10 1861-08-10 Confederate First major battle west of the Mississippi
```
We can get its `victor` column as a single-column tibble with:

```r
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 1.7.** Use `pull()` to calculate the `median()` casualty value.

**Exercise 1.8.** Use `pull()` and `pluck()` to get the description of the third battle.

**Exercise 1.9.** Use `slice()` and `select()` to get the description of the third battle.

**Exercise 1.10.** Use `slice_max()` and `select()` to get 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
genome}

rename(battle_data,
       First_Day = "start",
       Last_Day = "end")
```

> # A tibble: 12 x 6
> # Groups:   First_Day, Last_Day [1]
> First_Day Last_Day casualty victor name desc
> <date> <date> <dbl> <chr> <chr> <chr>
> 1 1861-04-12 1861-04-13 0 Confede~ Battle 1: Fort S~ Beauregard takes C~
> 2 1861-05-18 1861-05-19 10 Inconcl~ Battle 2: Sewell~ Union gunboats fig~
> 3 1861-05-29 1861-06-01 10 Inconcl~ Battle 3: Aquia ~ Confederate artill~
> 4 1861-06-03 1861-06-03 30 Union Battle 4: Philip~ Small Confederate ~
> 5 1861-06-10 1861-06-10 86 Confede~ Battle 5: Big Be~ Union attack near ~
> 6 1861-06-17 1861-06-17 87 Union Battle 6: Boonvi~ Union forces defea~
> 7 1861-07-02 1861-07-02 114 Union Battle 7: Hoke's~ Patterson defeats ~
> 8 1861-07-05 1861-07-05 244 Confede~ Battle 8: Cartha~ Confederate victor~
> 9 1861-07-11 1861-07-11 346 Union Battle 9: Rich M~ Union forces divid~
> 10 1861-07-18 1861-07-18 151 Confede~ Battle 10: Black~ Irvin McDowell's r~
> 11 1861-07-21 1861-07-21 4690 Confede~ Battle 11: First~ Thomas Jackson ear~
> 12 1861-08-10 1861-08-10 2549 Confede~ Battle 12: Wilso~ First major battle~
```

Note that the new names don't have quotation marks but the old names do.

We can also `relocate()` the columns of a tibble. For example, the following makes `name` the left-most column:

```r
genome}

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

> # A tibble: 12 x 6
> name start end casualty victor desc
> <chr> <date> <date> <dbl> <chr> <chr>
> 1 Battle 1: Fort S~ 1861-04-12 1861-04-13 0 Confede~ Beauregard takes C~
> 2 Battle 2: Sewell~ 1861-05-18 1861-05-19 10 Inconcl~ Union gunboats fig~
> 3 Battle 3: Aquia ~ 1861-05-29 1861-06-01 10 Inconcl~ Confederate artill~

37
The following moves start and end to the right of victor:

```r
relocate(
  battle_data,
  start, end,
  .after = victor
)
```

And the following makes casualty the right-most column:

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

**Exercise 1.11.** `rename()` victor to winner and made it the penultimate column (i.e., the second from the right).

- Use `last_col() - 1` to refer to the second-to-last column.

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:
char <-
dad <-
mom <-
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",
  "Electra", "Agamemnon", 
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 "Plouto"), and almost all the data we work with will be prepackaged in a vector.

**Exercise 1.12.** Use `tibble()` to organize the following data in a tibble:

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

**Exercise 1.13.** 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, "".

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("dat/CWSAC_civil_war_data.csv")
View(civil_war)
```

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 "dat/CWSAC_civil_war_data.csv"—your file path name will 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(
    "dat/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
17 https://www.dropbox.com/s/56007hal7fpfps5/CWSAC_civil_war_data.csv?dl=1
```
Now `read_csv()` downloads the data for us!

### 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.

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.\(^{18}\)

**Exercise 1.14.** Create an R project called `netflix` for the lecture in section 2.2. Include the following in the project's `code.R` file:

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

Confirm that `netflix` looks like this:

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

\(^{18}\)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 1.15. Create an R project called judges for the lab in section 2.3. Include the following in the project’s code.R file:

```r
court_data <-
  read_csv(
    "https://www.dropbox.com/s/6nbtr8q04lk12lt/court_data.csv?dl=1",
    col_types = "icccclllicc"
  )
```

Confirm that court_data looks like this:

```r
court_data %>% glimpse
```

Exercise 1.16. Create an R project called ufc for the lecture in section 3.2. Include the following in the project’s code.R file:
ufc <-
  read_csv(
    "https://www.dropbox.com/s/le5qsv3ty5vk5hs/ufc.csv?dl=1",
    col_types = "iDclllffifciddcdfiiddiiiiiiiiiiiiicdfiiddiiiiiiiiiiiii"
  )

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> <lgl> <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 MALE  3      U-DEC
> 5    5  2020-08-29 Las Vegas FALSE FALSE Middleweight MALE  3      U-DEC
> 6    6  2020-08-29 Las Vegas  TRUE FALSE Women's Str FEMALE  3       SUB
> # ... with 45 more variables: finish_details <chr>, finish_round <int>,
> # finish_round_time <dbl>, fight_time <dbl>, audience <lgl>, R_fighter <chr>,
> # R_odds <dbl>, R_stance <fct>, R_age <int>, R_weight <int>, R_height <dbl>,
> # R_reach <dbl>, R_wins <int>, R_losses <int>, R_draw <int>,
> # R_rounds_fought <int>, R_kd <int>, R_strong_attempt <int>,
> # R_strong_achieve <int>, R_weak_attempt <int>, R_weak_achieve <int>,
> # R_td_attempt <int>, R_td_achieve <int>, R_sub_attempt <int>,
> # R_sub_achieve <int>, B_fighter <chr>, B_odds <dbl>, B_stance <fct>,
> # B_age <int>, B_weight <int>, B_height <dbl>, B_reach <dbl>, B_wins <int>,
> # B_losses <int>, B_draw <int>, B_rounds_fought <int>, B_kd <int>,
> # B_strong_attempt <int>, B_strong_achieve <int>, B_weak_attempt <int>,
> # B_weak_achieve <int>, B_td_attempt <int>, B_td_achieve <int>,
> # B_sub_attempt <int>, B_sub_achieve <int>

Exercise 1.17. Create an R project called alibaba for the lab in section 3.3. Include the following in the project’s code.R file:

alibaba_wide <-
  read_csv(
    "https://www.dropbox.com/s/fh25g7fulmjmiei/alibaba.csv?dl=1",
    col_types = "iDclllffifciddc"
 cols( # use cols() since there are so many columns  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:

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 5 24751 889 1 254 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>,
> # action.5 <chr>, action.6 <chr>, action.7 <chr>, action.8 <chr>,
> # action.9 <chr>, action.10 <chr>, action.11 <chr>, action.12 <chr>,
> # action.13 <chr>, action.14 <chr>, action.15 <chr>, time.1 <dttm>,
> # time.2 <dttm>, time.3 <dttm>, time.4 <dttm>, time.5 <dttm>, time.6 <dttm>,
> # time.7 <dttm>, time.8 <dttm>, time.9 <dttm>, time.10 <dttm>,
> # time.11 <dttm>, time.12 <dttm>, time.13 <dttm>, time.14 <dttm>,
> # time.15 <dttm>, facility.1 <dbl>, facility.2 <dbl>, facility.3 <dbl>,
> # facility.4 <dbl>, facility.5 <dbl>, facility.6 <dbl>, facility.7 <dbl>,
> # facility.8 <dbl>, facility.9 <dbl>, facility.10 <dbl>, facility.11 <dbl>,
> # facility.12 <dbl>, facility.13 <dbl>, facility.14 <dbl>, facility.15 <dbl>,
> # shipper.1 <dbl>, shipper.2 <dbl>, shipper.3 <dbl>, shipper.4 <dbl>,
> # shipper.5 <dbl>, shipper.6 <dbl>, shipper.7 <dbl>, shipper.8 <dbl>,
> # shipper.9 <dbl>, shipper.10 <dbl>, shipper.11 <dbl>, shipper.12 <dbl>,
> # shipper.13 <dbl>, shipper.14 <dbl>, shipper.15 <dbl>

**Exercise 1.18.** Create an R project called chess for the lecture in section 4.2. Include the following in the project’s code.R file:
chess_tib <-
read_csv("https://www.dropbox.com/s/0fy8gfokd48tlak/chess_games.csv?dl=1")

Confirm that chess_tib looks like this:

chess_tib %>% glimpse

> Rows: 19,113
> Columns: 14
> $ game <chr> "TZJHljE", "l1NXvwaE", "mIICvQHh", "KWKvrqYL", "9tXolA-
> $ rated <lgl> FALSE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, TRUE-
> $ turns <dbl> 13, 16, 61, 61, 95, 3, 33, 25, 33, 39, 38, 60, 31, ~
> $ victory_status <chr> "outoftime", "resign", "mate", "mate", "mate", "draw", ~
> $ winner <chr> "white", "black", "white", "white", "white", "draw", "w-
> $ increment_code <chr> "15+2", "5+10", "5+10", "20+0", "30+3", "10+0", "10+
> $ white_id <chr> "bourgris", "a-00", "ischia", "daniamurashov", "nik2211-
> $ white_rating <dbl> 1500, 1322, 1496, 1439, 1523, 1250, 1520, 1413, 1439, 1
> $ black_id <chr> "a-00", "skinnerua", "a-00", "adivanov2009", "adivanov2-
> $ black_rating <dbl> 1191, 1261, 1500, 1454, 1469, 1002, 1423, 2108, 1392, 1
> $ moves <chr> "d4 d5 c4 c6 cxd5 e6 dxe6 fxe6 Nf3 Bb4+ Nc3 Ba5 Bf4", "-
> $ opening_eco <chr> "D10", "B00", "C20", "D02", "C41", "B27", "D00", "B00", ~
> $ opening_name <chr> "Slav Defense: Exchange Variation", "Nimzowitsch Defen-
> $ opening_ply <dbl> 5, 4, 3, 3, 5, 4, 10, 5, 6, 4, 1, 9, 3, 2, 8, 7, 8, 8, ~

Exercise 1.19. Create an R project called rationing for the lab in section 4.3. Include the following in the project’s code.R file:

category <-
read_csv("https://www.dropbox.com/s/p2hgkz28w2hdevc/category.csv?dl=1")

order_dates <-
read_csv("https://www.dropbox.com/s/vprgd8d3e4bmv/oder_dates.csv?dl=1")

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

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

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

ship_store <-
    read_csv("https://www.dropbox.com/s/5l2rmj5jlgyfb8/ship_store.csv?dl=1")

Confirm that your tibbles look the following:

category %>% glimpse

> Rows: 308
> Columns: 2
> $ item <dbl> 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, TRUE, T-

inv_DC %>% glimpse

> Rows: 414,419
> Columns: 3
> $ item <dbl> 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10-

inv_store %>% glimpse

> Rows: 16,490,877
> Columns: 4
> $ store <dbl> 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003, 1003-
> $ item <dbl> 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10210, 10210, 1-
> $ date <date> 2011-04-01, 2011-04-02, 2011-04-03, 2011-04-04, 2011-04-05, 201-
> $ inv <dbl> 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 9, 9, 9, 9, 9, 9, 9, 9, 9-

47
Exercise 1.20. Create an R project called `wine` for the lecture in section 5.2. Go to this website\(^1\) and click on the “Download” button. Doing so should download to your computer a zipped folder called “Archive”. This folder should contain three files: `winemag-data_first150k.csv`, `winemag-data-130k-v2.csv`, and `winemag-data-130k-v2.json`. Move the last file into your R project folder. We’ll load the data in this file into your workspace later.

Exercise 1.21. Create an R project called `cars` for the lab in section 5.3. Include the following in the project’s code.R file:

```r
# cars <-
read_csv("https://www.dropbox.com/s/6jvg7nb41lgxtf5/car_data.csv?dl=1")
```

Confirm that `cars` looks like this:

\(^1\)https://www.kaggle.com/zynicide/wine-reviews
cars %>% glimpse

> Rows: 23,783
> Columns: 11
> $ assembler <chr> "TOYOTA", "GENERAL MOTORS", "FORD", "TATA MOTORS", "VW GROUP~
> $ brand <chr> "TOYOTA", "CHEVROLET", "FORD", "JAGUAR", "VOLKSWAGEN", "CADI~
> $ model <chr> "AVALON", "IMPALA", "ESCAPE", "S TYPE", "EOS", "DTS", "AVALO~
> $ module <chr> "SUSPENSION", "LATCHES", "THERMAL", "WHEELS", "EXHAUST", "FA~
> $ part <chr> "SUSPENSION SYSTEM", "HANDLES/LATCHES", "THERMAL SYSTEM", "T~
> $ defects <dbl> 1.760183e-02, 6.728293e-01, 2.542036e-02, 6.158758e-02, 1.61~
> $ lat_d <dbl> 50.33405, 38.69562, 42.44782, 52.44002, 38.76819, 44.43854, ~
> $ lon_d <dbl> -1.831658, -103.543379, -89.811046, -5.339205, -9.660167, -8~
> $ lat_u <dbl> 48.822948, 43.953415, 44.336420, 39.161418, -5.416105, 48.82~
> $ lon_u <dbl> -3.156658, -74.941079, -83.781046, -109.662805, -38.165267, ~

Exercise 1.22. Create an R project called glm for the lecture in section 6.2. Include the following in the project’s code.R file:

```r
shooting_star <-
    read_csv("https://www.dropbox.com/s/vmbmc3wnh0bmaou/shooting_star.csv?dl=1")
```

Confirm that shooting_star looks like this:

shooting_star %>% glimpse

> Rows: 10,231
> Columns: 3
> $ visibility <dbl> 0.018403658, 0.014570673, 0.013979526, 0.043606863, 0.28949~
> $ sky_glow <dbl> 8.8791984, 3.6215062, 3.6902825, 9.6152141, 0.7290642, 2.98~
> $ star_count <dbl> 3, 8, 4, 0, 5, 3, 4, 0, 0, 4, 3, 5, 0, 0, 0, 4, 2, 3, 5, 6,~

Exercise 1.23. Create an R project called radiation for the lab in section 6.3. Include the following in the project’s code.R file:
facility <-
  read_csv("https://www.dropbox.com/s/2fw57ayxtgrja28/facility.csv?dl=1")

ter <-
  read_csv("https://www.dropbox.com/s/f07kbtpu4gk2cdec/ler.csv?dl=1")

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

radiation <-
  read_csv("https://www.dropbox.com/s/4s8shy341b4e228c/radiation.csv?dl=1")

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

Confirm that your tibbles look the following:

facility %>% glimpse

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

ler %>% glimpse

> Rows: 36,667
> Columns: 8
> $ reactor_name <chr> "Arkansas Nuclear 1", "Arkansas Nuclear 1", "Arkansas-~
> $ month <date> 1974-12-01, 1975-01-01, 1975-02-01, 1975-03-01, 1975-~
> $ ler <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ actuation_ler <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ techspec_ler <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ degrade_ler <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ inoperable_ler <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-
> $ safetyEquip_ler <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Exercise 1.24. Create an R project called `fishing` for the lab in section 7.3. Save the illegal fishing data found here\(^\text{20}\) 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

\(^{20}\)https://www.dropbox.com/sh/hh4e1c8ty12nyqf/AAAvjR01cB8d_oVaUuQZ7na?dl=0
1.4.1 <- vs. =

The <- and = operators are different, but the difference between them is highly technical (e.g., see this thread\(^{21}\) and section 8.2.26 of the R Inferno book\(^{22}\)). 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).

1.4.2 Saving and Loading Data

You could use write_csv() to save a tibble as a .csv file. However, it’s more conventional—and faster—to use write_rds() to save it as an .rds file. For 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():

\(^{21}\)https://stackoverflow.com/questions/1741820/what-are-the-differences-between-and-assignment-operators-in-r
\(^{22}\)http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
loaded_tibble <- read_rds("saved_tibble.rds")

loaded_tibble

> # A tibble: 3 x 2
>   x   y
> 1 1   a
> 2 2   b
> 3 3   c

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:

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

> # A tibble: 3 x 2
>   x   y
> 1 1   a
> 2 2   b
> 3 3   c

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")`.

1.4.3 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 = "c00cfc1"` 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.23

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.

23My 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.
1.4.4 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.

\texttt{str\_c()} concatenates character strings:

\begin{verbatim}
str_c("a", c("b", "c"), sep = "_____")
\end{verbatim}

\begin{verbatim}
> [1] "a_____b" "a_____c"
\end{verbatim}

\texttt{str\_count()} counts character string elements:

\begin{verbatim}
str_count("How many spaces are here?", " ")
\end{verbatim}

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

\begin{verbatim}
str_count(
  "How many words here start with a w?",
  "\b\w\w*"
)
\end{verbatim}

\begin{verbatim}
> [1] 3
\end{verbatim}

\begin{verbatim}
str_count(
  "How many words here start with a capital w?",
  "\B\w\w*"
)
\end{verbatim}

\begin{verbatim}
> [1] 0
\end{verbatim}

In the code above, "\b\w\w*" is shorthand for “a word that starts with w” and "\B\w\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.

\texttt{str\_detect()} identifies whether a character string contains a specific substring:
str_detect(
  "Do any words start with a w?",
  "\\bw\\w*"
)

> [1] TRUE

str_detect(
  "Do any words start with a capital w?",
  "\\bW\\w*"
)

> [1] FALSE

str_extract() and str_extract_all() pull out substrings from a character string:

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

> [1] "word"

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("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:

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

> start end
> [1,] 16 19

```r
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-expresses 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?")
)

59
str_which() is like str_subset(), except it returns the location of the strings rather than the strings themselves:

```r
str_which(c("which", "words", "contain", "a", "w"), "w")
```

> [1] 1 2 5

### 1.5 Solutions

#### 1.3

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

#### 1.4

```r
debt_change <- debt - lag(debt)
```

#### 1.5

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

#### 1.7

```r
median(casualty_vec)
```
```r
desc_vector <- pull(
  battle_data, 
  desc 
)
pluck(desc_vector, 3)

> [1] "Confederate artillery hit by naval bombardment"

third_battle <- slice(
  battle_data, 3 
)
select(third_battle, desc)

> # A tibble: 1 x 1
>   desc
>  <chr>
> 1 Confederate artillery hit by naval bombardment

bloodiest <- slice_max(
  battle_data, casualty 
)
select(bloodiest, desc)
```
Thomas Jackson earned the moniker 'Stonewall'

```r
# A tibble: 1 x 1
#  desc
#  <chr>
# 1 Thomas Jackson earned the moniker 'Stonewall'
```

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

relocate(
  temp_tib,
  winner,
  .after = last_col() - 1
)

```r
> # A tibble: 12 x 6
>   start          end casualty name      winner  desc                                           
>  <date>          <date>     <dbl> <chr> <chr>            <chr>                                             
> 1 1861-04-12     1861-04-13 0    Battle 1: Fort S~ Confede~ Beauregard takes C~
> 2 1861-05-18     1861-05-19 10   Battle 2: Sewell~ Inconcl~ Union gunboats fig-
> 3 1861-05-29     1861-06-01 10   Battle 3: Aquia ~ Inconcl~ Confederate artill-
> 4 1861-06-03     1861-06-03 30   Battle 4: Philip~ Union    Small Confederate ~
> 5 1861-06-10     1861-06-10 86   Battle 5: Big Be~ Confede~ Union attack near ~
> 6 1861-06-17     1861-06-17 87   Battle 6: Boonvi~ Union    Union forces defea-
> 7 1861-07-02     1861-07-02 114  Battle 7: Hoke's~ Union    Patterson defeats ~
> 8 1861-07-05     1861-07-05 244  Battle 8: Cartha~ Confede~ Confederate victor-
> 9 1861-07-11     1861-07-11 346  Battle 9: Rich M~ Union    Union forces divid-
>10 1861-07-18     1861-07-18 151  Battle 10: Black~ Confede~ Irvin McDowell's r-
>11 1861-07-21     1861-07-21 4690 Battle 11: First~ Confede~ Thomas Jackson ear-
>12 1861-08-10     1861-08-10 2549 Battle 12: Wilso~ Confede~ First major battle-
```

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

> # A tibble: 6 x 3
>   album                   year RIAA
>   <chr>                   <int> <chr>
> 1 Southernplayalisticadillacmuzik 1994 Platinum
> 2 ATLiens                 1996 2 × Platinum
> 3 Aquemini                1998 2 × Platinum
> 4 Stankonia               2000 4 × Platinum
> 5 Speakerboxxx/The Love Below 2003 Diamond
> 6 Idlewild                2006 Platinum

1.13

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, "")

> # A tibble: 6 x 3
>   album                                              year RIAA
>   <chr>                                              <int> <chr>
> 1 This Is a Long Drive for Someone with Nothing to Think About 1996 ""
> 2 The Lonesome Crowded West                         1997 ""
> 3 The Moon & Antarctica                             2000 "Gold"
> 4 Good News for People Who Love Bad News            2004 "Platinum"
> 5 We Were Dead Before the Ship Even Sank            2007 "Gold"
> 6 Strangers to Ourselves                            2015 ""
> 5 We Were Dead Before the Ship Even Sank 2007 "Gold"
> 6 Strangers to Ourselves 2015 ""
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 section 2.2.4 I demonstrate how to turn a functional assembly line into a new stand-alone function. To do so, we pack the sequence of functions into an abstract black box, which we can invoke at our pleasure. (However, we won’t need to define many functions because R already provides us with so many.)

In section 2.2.5 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 2.2.6–2.2.9 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.

## 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

```r
now()
```

> [1] "2021-04-22 17:11:27 CDT"

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_replace` in your R console yields the documentation for function
`str_replace()`. This documentation reports that the function produces a character vector output from three inputs, or “arguments”. And, as usual, 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:

```r
str_replace(
  string = "Four legs good, two legs bad",
  pattern = "bad",
  replacement = "better"
)
> [1] "Four legs good, two legs better"
```

Or we can invoke it with

```r
str_replace(
  "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 R generally disregards carriage returns (i.e., vertical spaces). 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"
> [1] "Four legs good, two legs better"

\[
\text{str_replace}(
\quad "Four legs good, two legs bad",
\quad "bad",
\quad "better"
\)
\]

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

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

\[
\text{str_flatten(string = LETTERS)}
\]

> [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

\[
\text{str_flatten(string = LETTERS, collapse = "")}
\]

> [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

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

\[
\text{str_flatten(string = LETTERS, collapse = "&&")}
\]
Exercise 2.1. Use `?str_detect` to review the instructions for `str_detect()`. Use this function to create a vector of logicals that identify the elements of `im_burning` that contain an exclamation mark, where

```r
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?"
)
```

That is, your `str_detect()` call should output the vector `c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE)`.

**Exercise 2.2.** Use `str_detect()` to identify the elements of `im_burning` that do not contain an exclamation mark. That is, your `str_detect()` call should output the vector `c(FALSE, FALSE, TRUE, TRUE, TRUE, FALSE, TRUE)`.

### 2.2.2 Conditional Statements

There’s an important function that does not follow the standard notation: the if-else statement. The notation for these expressions looks like this:

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

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

```r
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 ":")"
```

And the following defines time_left_to_work_today as 0 if it’s Saturday or Sunday and otherwise defines it as the time until 6:00 PM this evening:

```r
define this_year's_halloween <- mdy(str_c("10/31/", year(today()))).
define next_year's_halloween <- mdy(str_c("10/31/", year(today()) + 1)).
Use an if-else statement to return this_year's_halloween - today() if this year's Halloween has yet to pass, and return next_year's_halloween - today() otherwise.

**Exercise 2.3.** We will now count the days until the next Halloween (like in The Nightmare Before Christmas).

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

In addition to the if() { } else { } protocol there’s also the ifelse() function, which executes conditional commands across vectors. 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 a lower-cased alphabet, and setting `output_vector_if_FALSE` to an upper-cased alphabet yields an alphabet that alternates between lower and upper cases:

```r
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 an example:

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

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

In this case `output_vector_if_TRUE` is not actually 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 1.14, section 1.3):

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

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

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

```r
cast_vector <- pull(netflix, cast)
cast_vector <- head(cast_vector, 8)
cast_vector
```
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” (see vignette("regular-expressions")).

**Exercise 2.4.** Use `ifelse()` and `is.na()` to change the NA values in `director_vector` to "*missing data*".

**Exercise 2.5.** Use `ifelse()` and `%in%` to create a vector comprising the letters of the alphabet, with every vowel capitalized.
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)))))
```

> [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:

```r
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.

```r
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)
```

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

In this case, variable `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:

```r
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

> [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 2.6.** 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))))`.

In general, the following two code chunks yield the same output:
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, \( 9 \%>\% \text{sqrt} = 3 \). 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 \( \text{sum() \} \)” then you should write \( 1:100 \%>\% \text{sum} \).

Note that we can drop the parentheses after our functions when we string them into an assembly line. For example, \( x \%>\% f_1 \%>\% \ldots \%>\% f_n \) and \( x \%>\% f_1() \%>\% \ldots \%>\% f_n() \) 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"
  )
```

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

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

```r
> [1] "Is it alright to boil a sentient creature alive?"
```
Exercise 2.7. What do the following three functional assembly lines output?

```r
#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
```

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"), rather than str_c("y", "x", "z") or str_c("y", "z", "x"). 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”:

```r
"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 2.8. 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 2.9. 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 2.10. The following code calculates the average number of times each character digit, from "0" to "9", appears in the past ten thousand dates:
x <- tibble(date = today() + days(-10^4:-1))
x <- mutate(x, date_char = str_remove_all(date, "-"))
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)

> # 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  2.48  1.54  1.49  0.310  0.291  0.292  0.289  0.292  0.292  0.731

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 2.11. The following code selects the longest name (in terms of the number of characters) listed in either the director or cast columns of netflix.

netflix %>%
    replace_na(list(director = ",", cast = ",")) %>%
    mutate(
        director = str_split(director, ",", ),
        cast = str_split(cast, ",", )
    ) %>%
    unnest(director) %>%
    unnest(cast) %>%
    summarise(
Perform this calculation without a functional assembly line. In other words, use a wheelbarrow variable to remove the conveyor belts.

### 2.2.4 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 code defines a function that:

```r
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
```
encapsulates the assembly line comprising four \texttt{sqrt()} operations from exercise 2.6 in the new function \texttt{sqrt_four\_times()}:\texttt{

\begin{verbatim}
sqrt_four_times <-

  \%>\%
  sqrt \%>\%
  sqrt \%>\%
  sqrt \%>\%
  sqrt
\end{verbatim}

Now \texttt{sqrt\_four\_times()}, \texttt{x \%>\% sqrt\_four\_times}, and \texttt{x \%>\% sqrt \%>\% sqrt \%>\% sqrt \%>\% sqrt} are all equivalent. For example, \texttt{3 \%>\% sqrt\_four\_times = 1.071075} and \texttt{3 \%>\% sqrt \%>\% sqrt \%>\% sqrt \%>\% sqrt = 1.071075}. Or for another example, define

\texttt{

\begin{verbatim}
weird\_function <-

  \%>\%
  log \%>\%
  sqrt\_four\_times \%>\%
  sum
\end{verbatim}

\end{verbatim}

Now \texttt{weird\_function()}, \texttt{x \%>\% weird\_function}, \texttt{x \%>\% log \%>\% sqrt\_four\_times \%>\% sum}, and \texttt{x \%>\% s log \%>\% sqrt \%>\% sqrt \%>\% sqrt \%>\% sqrt \%>\% sum} all yield the same output. For example, \texttt{weird\_function(1:100) = 107.1425} and \texttt{1:100 \%>\% log \%>\% sqrt \%>\% sqrt \%>\% sqrt \%>\% sqrt \%>\% sum = 107.1425}.

In general, we can compress an assembly line comprising functions \texttt{f\_1()}, \texttt{f\_2()}, \ldots, and \texttt{f\_n()} into function \texttt{f()} with the following:

\texttt{

\begin{verbatim}
f <-

  \%>\%
  f\_1 \%>\%
  f\_2 \%>\%
  \#...
  f\_n
\end{verbatim}

\end{verbatim}

In other words, running this code defines a function \texttt{f()} that satisfies \texttt{f(x) = x \%>\% f\_1 \%>\% f\_2 \ldots \%>\% f\_n}.

**Exercise 2.12.** Use the \texttt{\%>\%} notation to create an R function called \texttt{soft\_max()} that takes a vector \texttt{x} and applies the \texttt{exp()}, \texttt{sum()}, and \texttt{log()} functions to it, in that order. Apply \texttt{soft\_max} to vectors \texttt{-1:1}, \texttt{-4:4}, \texttt{-16:16}, and \texttt{-64:64}.\texttt{}}
Exercise 2.13. 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 2.14. 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?
2.2.5 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
f_1 <-
  . %>%
    double_string %>%
    double_string

f_2 <-
  . %>%
  f_1 %>%
  f_1

1:3 %>%
  f_2 %>%
  sum
```

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
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’s another example:

```r
\begin{verbatim}
c(.2, .8) %>%
  {- . * log(.)} %>
  sum
\end{verbatim}

> [1] 0.5004024
```

R interprets `c(.2, .8) %>% {- . * log(.)} as - c(.2, .8) * log(c(.2, .8))`, and thus returns `sum(- c(.2, .8) * log(c(.2, .8))) = 0.5004024`.

Here’s a more complex example:

```r
\begin{verbatim}
c("3", "hey") %>% {
  rep(
    pluck(.2, 2),
    pluck(.2, 1) %>% as.integer
  ) %>
  str_flatten(collapse = " ") %>
  str_to_upper
} %>
str_c("!
```

> [1] "HEY HEY HEY!"

In this case the bracketed expression contains an entire functional assembly line. And when we have multiple lines of bracketed code we usually write the brackets and code on different lines, so that, e.g., the first bracket aligns with `c("3", "hey")` rather than with `rep`. The code looks a little tidier this way.

To see how R interprets this code, replace the `.` terms with the conveyor-belt input and remove the brackets:

```r
\begin{verbatim}
rep(
  pluck(c("3", "hey"), 2),
  pluck(c("3", "hey"), 1) %>% as.integer
```

84
> [1] "HEY HEY HEY!"

We can remove the brackets when the terms are nested in a common function. For example, the following two code chunks are equivalent:

```r
# Chunk 1:
"How many characters are in each word of this question" %>%
  str_split(" ") %>%
  pluck(1) %>%
  { str_c(., str_length(.), sep = "=", collapse = " ") }

> [1] "How=3 many=4 characters=10 are=3 in=2 each=4 word=4 of=2 this=4 question=8"
```

In this case the `str_c()` parentheses serve as the brackets.

**Exercise 2.15.** 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()`.

```r
"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(
```
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 %>% {.}^.
```

#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:

```
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):

```
fib <- . %> %>% { c(pluck(.), 2), sum(.) } c(1, 2) %>% fib
```

86
> [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\(^1\).

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 <-
  . %>%
  { if(any(is.na(.))) "NA found" else .}
```

c("a", "b", NA) %>%
  na_detector

> [1] "NA found"

c("a", "b", "c") %>%
  na_detector

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

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).\(^2\)

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.\(^3\)

\(^1\)https://en.wikipedia.org/wiki/Fibonacci_number
\(^2\)See section 2.2.2 for more on if-else statements.
\(^3\)Defining `santas_calendar()` in terms of `santas_calendar()` is an example of recursion\(^4\).
```r
santas_calendar <-
  . %>% {
    if(day(.) == 25 & month(.) == 12) "Christmas" else
c(santas_calendar(. + days(1)), "Eve")
  }

mdy("12/25/2021") %>%
santas_calendar

> [1] "Christmas"

mdy("12/24/2021") %>%
santas_calendar

> [1] "Christmas"  "Eve"

mdy("12/23/2021") %>%
santas_calendar

> [1] "Christmas"  "Eve"  "Eve"

mdy("09/15/2021") %>%
santas_calendar

> [1] "Christmas"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [7] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [13] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [19] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [25] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [31] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [37] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [43] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [49] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [55] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [61] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
> [67] "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"  "Eve"
```

88
Exercise 2.16. 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 2.17. 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.

Exercise 2.18. Use a bracketed expression to define `smart_viewer()` as a function that receives a tibble and returns its `summary()` if it has fewer than five columns 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()`.

2.2.6 `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:

```
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-
```
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>
```

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.

Whereas the previous example used `mutate()` to modify old variables, the following example uses `mutate()` to construct new variables. Specifically, it defines columns `country_primary` and `country_secondary` as the first and second countries listed in the `country` text string:

```r
netflix <- netflix %>%
mutate(
  country_primary = str_remove(country, ",.*"),
  country_secondary = str_extract(country, "(?<=, )[^,]*")
)

netflix %>%
select(country, country_primary, country_secondary) %>%
slice(1, 6, 7, 12)
```

```
# A tibble: 4 x 3
  country                  country_primary country_secondary
  <chr>                    <chr>              <chr>
1 United States, India, South Korea, China United States India
2 Spain                   Spain              <NA>
3 Bulgaria, United States, Spain, Canada Bulgaria United States
4 Netherlands, Belgium, United Kingdom, United- Netherlands Belgium
```

In the code above, regular expressions ",.*" and "(?<=, )[^,]*" respectively represent “the first comma everything after it” and “everything after the first comma but before the second comma.” Hence, we create `country_primary` by removing all the text in `country` after first comma and we create `country_secondary` by extracting all the text between the first and second commas.

5We use `=` rather than `<-` within `mutate()` calls, since we never put the latter symbol inside a function’s parentheses (see section 1.4.1).
**Exercise 2.19.** Use `\texttt{\%in\%}` and `\texttt{mutate()}` to add a logical variable to `\texttt{netflix}` called `\texttt{restricted}` that is `\texttt{TRUE}` if and only if the show’s `rating` is "R", "TV-MA", or "NC-17".

**Exercise 2.20.** Use `\texttt{ifelse()}`, `\texttt{\%in\%}`, and `\texttt{mutate()}` to add a variable to `\texttt{netflix}` called `\texttt{kid_rating}` that equals `\texttt{rating}` if this value is "G", "PG", "TV-G", or "TV-PG", and otherwise equals "not child friendly".

**Exercise 2.21.** Combine `\texttt{mutate()}` with ...

- `\texttt{mdy()}` to change the `date_added` column of `\texttt{netflix}` from a character string to a Date,
- `\texttt{month(label = TRUE)}` to add a new variable that reports the month the show was added, and
- `\texttt{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 `\texttt{mutate()}` call.

We can embed functional assembly lines inside of `\texttt{mutate()}`. For example, the following defines `\texttt{last_cast_member_is_director}` as a logical that is `\texttt{TRUE}` if the last person listed in the cast is also listed as a `\texttt{director}`:

```r
\texttt{netflix \%>\% mutate(}
  \texttt{  last_cast_member_is_director =}
  \texttt{    cast \%>\%}
  \texttt{      str_extract("\b[^,]*\$") \%>\%}
  \texttt{      str_detect(director) }
  \texttt{  ) \%>\%}
\texttt{select(cast, director, last_cast_member_is_director) \%>\% slice(249, 250, 4328, 3862)}
```

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

**Exercise 2.22.** The `\texttt{duration}` values of `\texttt{netflix}` are text strings that comprise both a measurement, which is a number, and a metric, which is either "\texttt{min}" or "\texttt{Season}". For example, the first six `\texttt{duration}` values are:

```r
\texttt{netflix \%>\% select(duration) \%>\% head}
```
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".
  
  ```r
  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"`.

### 2.2.7 `summarise()`

The second quartet function is `summarise()`. As its name suggests, it computes summary statistics. For example, we can 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).

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

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

---

6We added `wday_added` to `netflix` in exercise 2.21.
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)
  )
```

```r
> # A tibble: 1 x 2
>   mean_age median_age
>       <dttm>    <dttm>
> 1  1025.35 days  935 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))
```

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

**Exercise 2.23.** We will now calculate the fraction of days spanned by our 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`.

---

7The `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"`.  

94
– 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 2.24.** We will now calculate the difference between the average age of 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.

```r
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
  )
```

> # A tibble: 1 x 3
> avg_year_movie avg_year_tv age_difference    
> <dbl>        <dbl>             <dbl>
> 1 2012. 2016. 3.20

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 2.25.** 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 indicate 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`:

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

> # A tibble: 9 x 2
> decile_val quantile
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":

```r
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
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 2.26. 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.

Exercise 2.27. 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 2.2.5.)
- Pipe the result into quantile(probs = c(0.25, .5, .75), na.rm = TRUE).

2.2.8 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 2.24:

```r
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()`:

```r
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:

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

> # A tibble: 3 x 16
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:

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

```r
> # A tibble: 3 x 4
> # Groups: type [1]
> type rating length_mean length_sd
> <chr> <chr>     <dbl>     <dbl>
> 1 Movie G     144.      18.9
> 2 Movie NC-17 142.      9.19
> 3 Movie NR    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)`. 100
But we can also define more exotic groupings. For example, the following defines grouping scheme \(\text{year\_added} = \text{year(date\_added)}\) within the \texttt{group\_by()} operation, and uses \texttt{summarise()} to calculate the mean \texttt{release\_year} for each corresponding chunk:

\[
\text{netflix} \%\%
\text{group\_by(year\_added = \text{year(date\_added)})} %>%
\text{summarise(mean\_released = \text{mean(release\_year)})} %>%
tail
\]

\[
\begin{array}{ll}
\# \text{ A tibble: 6 x 2} \\
\text{year\_added} & \text{mean\_released} \\
1 & 2015. \\
2 & 2016. \\
3 & 2017. \\
4 & 2018. \\
5 & 2019. \\
6 & 2020. \\
\end{array}
\]

As you see, Netflix must have added a lot of old shows in 2020, as the average \texttt{release\_year} of its new offerings decreased to 2008 in that year.

**Exercise 2.28.** We will now calculate the number of shows added each month.

- Start with \texttt{netflix} and \texttt{group\_by()} \texttt{month\_added} (which we defined in exercise 2.21).
- Use \texttt{summarise()} and \texttt{n()} to define \texttt{num\_shows} as the number of observations in each group.
  - The function \texttt{n()} returns the number of observations in a given \texttt{group\_by()} group.
  - Your "Jan" group should have \texttt{num\_shows} = 610.

**Exercise 2.29.** We will now calculate the fraction of shows added in each month.

- Use \texttt{summarise()} to define \texttt{num\_shows}, in the fashion of exercise 2.28.
- \texttt{ungroup()} the tibble produced by \texttt{summarise()}.
- Use \texttt{mutate()} to define \texttt{frac = num\_shows / sum(num\_shows)}.
  - The value of \texttt{frac} should sum to 1 across all groups.
  - The \texttt{ungroup()} step makes the \texttt{sum(num\_shows)} step sum across the entire tibble, rather than across the current \texttt{month\_added} group.

**Exercise 2.30.** We found in section 2.2.7 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.

`group_by()` pairs best with `summarise()`, but it works with many other functions. For example, prefacing a `mutate()` call with a `group_by()` step enables us to derive group-level variables. For example, the following defines `avg_release_year_by_rating` as the average `release_year` across distinct rating values:

```r
netflix %>%
  group_by(rating) %>%
  mutate(avg_release_year_by_rating = mean(release_year))
```

```
> # A tibble: 6,223 x 17
> # 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> Jandi~ United~ 2016-09-09 2016 TV-MA
>  3 70234439 TV Sh~ Trans~ <NA> Peter~ United~ 2018-09-08 2013 TV-Y7-
>  4 80058654 TV Sh~ Trans~ <NA> Will ~ United~ 2018-09-08 2016 TV-Y7
>  5 80125979 Movie #real~ Fernand~ Nesta~ United~ 2017-09-08 2017 TV-14
>  6 80163890 TV Sh~ Apach~ <NA> Alber~ Spain 2017-09-08 2016 TV-MA
>  7 70304989 Movie Autom~ Gabe Ib~ Anton~ Bulgar~ 2017-09-08 2014 R
>  8 80164077 Movie Fabri~ Rodrigo~ Fabri~ Chile 2017-09-08 2017 TV-MA
>  9 80117902 TV Sh ~ Fire ~ <NA> <NA> United~ 2017-09-08 2017 TV-MA
> 10 70304990 Movie Good ~ Henrik ~ James~ United~ 2017-09-08 2014 R
> # ... with 6,213 more rows, and 8 more variables: duration <chr>,
> # listed_in <chr>, description <chr>, country_primary <chr>,
> # country_secondary <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 2.31.** 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).

**Exercise 2.32.** 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 2.33.** 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.

### 2.2.9 `filter()`

The Friday and Saturday programming is clearly different. Suppose we wanted to create a tibble comprising only the Friday and Saturday shows. We could create such a 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
> 1 Domino Movie R International Movies Denmark, France, Italy 2019
> 2 Everybody K- Movie R Dramas, Internationals Spain, France, Italy 2019
> 3 Between Worlds Movie R Thrillers Spain 2018
> 4 Day of the ~ Movie R Horror Movies, Independent Bulgaria 2018

**Exercise 2.34.** Use `filter()` and `str_detect()` to list the programs directed by "Riri Riza".

**Exercise 2.35.** Use `filter()` and `str_detect()` to list the programs directed by "Riri Riza" and at least one other person.
Exercise 2.36. 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]
> title            director            country
> <chr>             <chr>               <chr>
> 1 Ricardo Quevedo: Hay gente así Raúl Campos, Jan Su~ Colombia
> 2 Arango y Sanint: Ríase el show Raúl Campos, Jan Su~ Colombia
> 3 Chloe Atom Egoyan United States, Canada, Fr~
> 4 GoldenEye Martin Campbell United Kingdom, United St~
> 5 Hellboy Guillermo del Toro United States
> 6 Todo lo que sería Lucas Lauri~ Raúl Campos, Jan Su~ Argentina

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 2.37.** 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 2.38.** 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`. 

106
• 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 2.39.** 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.

### 2.3 Lab: Italian Judiciary

#### 2.3.1 Overview

In 2011, Bray et al. [2016] conducted an experiment in the Roman Labor Court of Appeals. The aim of the experiment was to test a new court scheduling scheme, in hopes of reducing case completion times. 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. 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 Sample

We will use the court_data tibble that you defined in section 1.3, exercise 1.15. It should have 11 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 **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 sample switched from the control group to the treated group. 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 2.40.)
- **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).

---

8We won’t be able to reproduce their results exactly, as our sample differs a bit from their sample.

9Cramming 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.

10More 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.”

11More specifically, **treated** indicates whether the judge belongs to the collegio that received treatment. As Bray et al. explain, the court is divided into five collegios, only one of which implemented our scheduling policy. Three judges transferred into the treated collegio after the experiment was underway.
2.3.3 Clean

Exercise 2.40. 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 hearings.

- 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 304,941 rows after this step.
  - You can get the number of rows in a table with dim() or nrow().

Exercise 2.41. 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 use select() to remove fdate.
- rename() hdate to date.
- Use filter() and is.na() to remove values in which date is NA.
- Call the resulting tibble hearing_dates.

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

- Start with court_data and remove 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 433,322 rows.
- Use mutate() and dmy() to convert date from a character variable to a Date variable.
- group_by() cid.
- Use mutate() and row_number() to add hearing ID variable hid. This variable should be 0 for the earliest date, 1 for the second-earliest date, and so on. Hence, the hid = 0 observations should correspond to case filings and the hid > 0 observations to hearings. ungroup() the resulting tibble and use it to overwrite court_data.
– Make sure `tidy` is ordered by `date` and not by how the rows are arranged in the tibble. For example, consider the following:

```r
tibble(
  bloody_dates =
    c("July 1, 1863",
      "May 8, 1864",
      "September 18, 1863",
      "September 17, 1862",
      "May 5, 1864"
    ) %>%
    mdy
)%>%
mutate(
  correct_order = row_number(bloody_dates),
  incorrect_order = row_number()
)
```

> # A tibble: 5 x 3
> bloody_dates correct_order incorrect_order
> <date> <int> <int>
> 1 1863-07-01 2 1
> 2 1864-05-08 5 2
> 3 1863-09-18 3 3
> 4 1862-09-17 1 4
> 5 1864-05-05 4 5

**Exercise 2.42.** 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 interpret all cases that didn’t have a hearing in the last two years of our sample as closed. That is, we will set `closed = TRUE` for these cases.

- Define `cutoff_date` as the day two years before the latest `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 convert this tibble to a Date variable with `pull()`. For example, whereas `court_data %>% summarise(first(date))` outputs a tibble, `court_data %>% summarise(first(date)) %>% pull` outputs a Date.
• Group `court_data` by `cid`.
• Use `ifelse()` to set `closed = TRUE` if the latest date associated with the given `cid` precedes `cutoff_date`—i.e., if the case 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`.

**Exercise 2.43.** We will now add a variable to our tibble that indicates whether the given hearing concludes a case.

• `group_by()` `cid`.
• Use `mutate()` to define logical `last_hearing` 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`.

**Exercise 2.44.** 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 310,401 rows after this step.

**Exercise 2.45.** 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 52 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.)

### 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. 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 2.46.** 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.

---

12In 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].
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 exercise 2.47.)
- `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 208 rows.

The `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-fledged 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 2.47.** Each point in the plot we made in exercise 2.46 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 \( X = \frac{x_1 + \cdots + x_N}{N} \) 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
- \( \sigma = \sqrt{\frac{\sum (x_i - X)^2}{N}} \).

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

```r
hazard_rate_plot <- hazard_rate_plot + geom_errorbar(
  aes(
    y = hazard_rate,
    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 208 point estimates (dots) embedded within 208 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.
2.4 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 %>%
  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 %>%
```

115
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 subsquent 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 ares 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 2.10 (you won’t need to understand the code to follow my argument):

**#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)
      ),
      digits
    ),
    m = mean(digit_count)
  ),
  names_from = digits,
  values_from = m
)
```

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, \( m = \text{mean}(\text{digit\_count}) \). For another example, try to quickly determine which function the \( \text{digit\_count} = \text{str\_count}(\text{date\_char}, \text{digits}) \) 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 \( m = \text{mean}(\text{digit\_count}) \), 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 2.22 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:

```r
netflix <-
netflix %>%
separate(
  duration,
  c("duration", "duration_metric"),
  convert = TRUE
)
```

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
118
```
Second, in exercise 2.26 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 %>%
  unite(
    duration,
    c("duration", "duration_metric"),
    sep = " 
  )
```

```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 2.29 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".
Fourth, in exercise 2.31 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 2.4 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:

```r
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`:

```r
appender <- function(string, suffix)
  .%>% str_c(string, suffix)
```

```r
appender("I love R code", "ing")
"I love R codeing"
```
```r
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:

```r
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:

```r
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
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
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)
```
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:

is_one_vector_dominant(1:3, c(-1, 0, 1))

> [1] TRUE

is_one_vector_dominant(1:3, c(-1, 0, 10))

> [1] FALSE

### 2.4.5 Download Packages from CRAN

In section 2.2.6 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, ... x_20) to vector c(NA, NA, (x_1 + x_2 + 3)/3, (x_2 + x_3 + x_4)/3, ..., (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 RcppRoll. 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 Jurimetrical 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 answers.

For example, when deciding between the various moving average options, I found an article on tidyverse.org 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:

```r
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

---

13https://stackoverflow.com/
14https://dplyr.tidyverse.org/articles/window-functions.html
Or you can execute this function without loading the RcppRoll package into your workspace with:

```r
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 run `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 offorcats 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:
mini_tib <-
tibble(
    X = c(1, 5, -1, 0, -3, -2),
    Y = 
        c("a", "b", "c", "a", "b", "d") %>%
        as.factor
  )
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

The \texttt{fct_recode()} and \texttt{fct_relabel()} functions change the factor labels. \texttt{fct_recode()} allows us to change the labels by hand:

```r
mini_tib %>%
  mutate(
    Y_2 =
      fct_recode(
        Y,
        A = "a",
        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
And `fct_relabel()` allows us to change the labels by function (or functional assembly line):

```r
mini_tib
  %>%
  mutate(
    Y_2 =
      fct_relabel(
        Y,
        ~ .x %>%
          str_to_upper %>%
          str_c("!")
      )
  )
```

> # A tibble: 6 x 3
> X Y  Y_2
> <dbl> <fct> <fct>
> 1 1  a  A!
> 2 5  b  B!
> 3 -1 c  C!
> 4 0  a  A!
> 5 -3 b  B!
> 6 -2 d  D!

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)
  )

> # A tibble: 4 x 2
> `levels(Y)` `levels(Y_2)`
> <chr>     <chr>
> 1 a        b
> 2 b        c
> 3 c        d
> 4 d        a

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)
  )

> # A tibble: 4 x 3
> `levels(Y)` `levels(Y_2)` `levels(Y_3)`
> <chr>     <chr>     <chr>
> 1 a        b        d
> 2 b        d        c
> 3 c        c        a
> 4 d        a        b

In general, `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 accending 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 court_data sample we loaded in section 2.3.2 had two similar variables: fdate, which denoted when a case was filed, and 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 fdate and 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 2.11:

```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,
```
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
  ) %>%
  mutate(person = str_split(person, " ", "") %>%
    unnest(person) %>%
    summarise(
      person = last(person, order_by=str_length(person))
    )

> # A tibble: 1 x 1
> longest_name
> <chr>
> 1 Gabrielle Susanne Solheim Leithaug

2.5 Solutions

2.1
str_detect(im_burning, "!") # ignoring negate input implicitly makes it FALSE

> [1] TRUE TRUE FALSE FALSE FALSE TRUE FALSE

2.2

str_detect(im_burning, "!", negate = TRUE) # explicitly set negate to TRUE

> [1] FALSE FALSE TRUE TRUE TRUE FALSE TRUE

2.3

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()
}

2.4

ifelse(
  is.na(director_vector),
  "*missing data*",
  director_vector
)

> [1] "Richard Finn, Tim Maltby" "*missing data*" "*missing data*"
> [3] "*missing data*" "*missing data*" "*missing data*"
> [5] "Fernando Lebrija" "*missing data*" "Rodrigo Toro, Francisco Schultz"

2.5
ifelse(
    letters %in% c("a", "e", "i", "o", "u"),
    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"

2.6

2 %>%
    sqrt %>%
    sqrt %>%
    sqrt
    sqrt

> [1] 1.044274

2.9

"comrade Napoleon is always right" %>%
    str_to_title %>%
    str_c(., "!") %>%
    str_replace("Comrade", "Master") %>%
    rep(2) %>%
    str_flatten(collapse = " ")

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

2.10

tibble(date = today() + days(-10^4:-1)) %>%
    mutate(date = str_remove_all(date, "-")) %>%
    group_by(date, date_char) %>%
    summarise(digits = as.character(0:9)) %>%
    mutate(digit_count = str_count(date_char, digits)) %>%
    group_by(digit)
    summarise(m = mean(digit_count)) %>%
    pivot_wider(names_from = digits, values_from = m)
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
      )
  )
> # A tibble: 1 x 1
> longest_name
> <chr>
> 1 Gabrielle Susanne Solheim Leithaug

2.12

```r
do

soft_max <-
    . %>%
    exp %>%
    sum %>%
    log

-1:1 %>% soft_max

> [1] 1.407606

-4:4 %>% soft_max

> [1] 4.458552

-16:16 %>% soft_max

> [1] 16.45868

-64:64 %>% soft_max

> [1] 64.45868

2.13
```
```r
clean_text <-
  . %>%
  str_replace_na(replacement = "") %>%
  str_flatten(collapse = " ") %>%
  str_remove_all("[:punct:]") %>%
  str_squish %>%
  str_to_lower

tasty_breakfast %>% clean_text

> [1] "if its your job to eat a frog its best to do it first thing in the morning and if its your job to eat two frogs its best to eat the biggest one first"

brotherly_love %>% clean_text

> [1] "when ajax lifted his shield teucer would peer round and when he had hit any one in the throng the man would fall dead then teucer would hie back to ajax as a child to its mother and again duck down under his shield"

2.15

"When a man is tired of London, he is tired of life"

str_replace_all(word(., 5), "understanding")

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

2.16

logical_not <-
  . %>% {!}.

2.16

self_exponentiation <-
  . %>%
  {.^}.

2.17
```
reverse_symmetric <-
  . %>%
  {. == rev(.)} %>%
  all

2.18

smart_viewer <-
  . %>%
  { if(ncol(.) <= 4) summary(.) else glimpse(.) }

netflix %>% smart_viewer

> 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, ~

netflix %>%
  select(1:3) %>%
  smart_viewer

> show_id     type     title
> Min. :247747 Length:6223 Length:6223
> 1st Qu.:80036542 Class:character Class:character
> Median:80163530 Mode:character Mode:character
> Mean :76708794 ~
> 3rd Qu.:80244982 ~
> Max. :81235729 ~

135
2.19

```r
netflix <-
  netflix %>%
  mutate(restricted = rating %in% c("R", "TV-MA", "NC-17"))
```

2.20

```r
netflix <-
  netflix %>%
  mutate(
    kid_rating =
      ifelse(
        rating %in% c("G", "PG", "PG", "TV-PG"),
        rating,
        "not child friendly"
      )
  )
```

2.21

```r
netflix <-
  netflix %>%
  mutate(
    date_added = mdy(date_added),
    month_added = month(date_added, label = TRUE),
    wday_added = wday(date_added, label = TRUE)
  )
```

2.22

```r
netflix %>%
  mutate(
    duration_metric = word(duration, 2),
    duration =
      duration %>%
      word(1) %>%
      as.integer
  ) %>%
  select(title, duration_metric, duration) %>%
  head(4)
```
```r
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
  )

> # A tibble: 1 x 3
>   day_count show_day_count new_show_fraction
>   <int>       <int>             <dbl>
> 1    4400        1189          0.270

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
  )

> # A tibble: 1 x 3
>   avg_year_movie avg_year_tv age_difference
>       <dbl>       <dbl>       <dbl>
> 1     2012.       2016.        3.20

netflix %>%
  summarise(
    fraction_na =
      director %>%
        is.na %>%
        mean
  )
```

137
> # A tibble: 1 x 1
> fraction_na
>  <dbl>
> 1 0.315

2.26

```r
netflix %>%
  summarise(
    director %>%
      str_remove(",.") %>%
      unique %>%
      na.omit
  )
```

> # A tibble: 3,209 x 1
> `director %>% str_remove(",.") %>% unique %>% na.omit`
> <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

2.27

```r
netflix %>%
  summarise(
    quantiles =
      cast %>%
        str_count(",") %>%
        {. + 1} %>%
        quantile(
          probs = c(0.25, .5, .75),
          na.rm = TRUE
        )
  )
)
2.28

netflix %>%
group_by(month_added) %>%
summarise(num_shows = n())

2.29

netflix %>%
group_by(month_added) %>%
summarise(num_shows = n()) %>%
ungroup %>%
mutate(frac = num_shows / sum(num_shows)) %>%
head(4)
> 1 Jan 610 0.0980
> 2 Feb 378 0.0607
> 3 Mar 551 0.0885
> 4 Apr 447 0.0718

2.30

```r
netflix %>%
group_by(Fri_Sat = wday_added %in% c("Fri", "Sat")) %>%
summarise(fraction_tv = mean(type == "TV Show"))
```

> # A tibble: 2 x 2
>   Fri_Sat fraction_tv
>   <lgl>       <dbl>
> 1 FALSE       0.264
> 2 TRUE        0.391

2.31

```r
netflix <-
netflix %>%
group_by(director) %>%
mutate(director_count = n()) %>%
ungroup
```

2.32

```r
netflix <-
netflix %>%
mutate(
  director_count =
    ifelse(
      is.na(director),
      NA,
      director_count
    )
)
```

2.33
```r
netflix %>%
group_by(
  type,
  Fri_Sat = wday_added %in% c("Fri", "Sat")
) %>%
summarise(
  inexperienced_director =
    mean(director_count == 1, na.rm = TRUE)
)
```

```
# A tibble: 4 x 3
# Groups: type [2]

<table>
<thead>
<tr>
<th>type</th>
<th>Fri_Sat</th>
<th>inexperienced_director</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movie</td>
<td>FALSE</td>
<td>0.615</td>
</tr>
<tr>
<td>Movie</td>
<td>TRUE</td>
<td>0.690</td>
</tr>
<tr>
<td>TV Show</td>
<td>FALSE</td>
<td>0.671</td>
</tr>
<tr>
<td>TV Show</td>
<td>TRUE</td>
<td>0.879</td>
</tr>
</tbody>
</table>
```

```
> 2.34
```

```
netflix %>%
  filter(str_detect(director, "Riri Riza")) %>
  select(title, description)
```

```
# A tibble: 6 x 2

<table>
<thead>
<tr>
<th>title</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gie</td>
<td>Indonesian activist Soe Hok Gie experiences a political awakening</td>
</tr>
<tr>
<td>Emma' (Mother)</td>
<td>In 1950s Makassar, the life of a devoted mother is upended</td>
</tr>
<tr>
<td>The Dreamer</td>
<td>A trio of friends struggles to make it through adolescence,</td>
</tr>
<tr>
<td>The Jungle School</td>
<td>An idealistic young woman travels to the jungles of Indonesia</td>
</tr>
<tr>
<td>The Rainbow Troo</td>
<td>Two Indonesian teachers embrace an inspiring crop of gifted</td>
</tr>
<tr>
<td>What's Up With L</td>
<td>A popular high school girl strains her relationship with her</td>
</tr>
</tbody>
</table>
```

```
> 2.35
```

```
netflix %>%
  filter(
    str_detect(director, "Riri Riza"),
```

141
director != "Riri Riza"
) %>%
select(title, director)

# A tibble: 1 x 2
#  title                        director
#  <chr>                        <chr>
# 1 What's Up With Love? Rudy Soedjarwo, Riri Riza

2.36

netflix %>%
  filter(str_detect(cast, ",", negate = TRUE)) %>%
  select(title, cast) %>%
  head

# A tibble: 6 x 2
#  title                                     cast
#  <chr>                                      <chr>
# 1 Jandino: Whatever it Takes                Jandino Asporaat
# 2 Fabrizio Copano: Solo pienso en mi        Fabrizio Copano
# 3 Joaquín Reyes: Una y no más               Joaquín Reyes
# 4 Marc Maron: Too Real                      Marc Maron
# 5 Mo Gilligan: Momentum                     Mo Gilligan
# 6 Paul Hollywood's Big Continental Road Trip Paul Hollywood

2.37

netflix %>%
  group_by(director) %>%
  filter(cast %>%
    str_detect("Forest Whitaker") %>%
    any)

# A tibble: 5 x 15
# Groups: director [4]
# A tibble: 5 x 3
# Groups: director [2]
   director                  title                                 listed_in
1 Don Michael Pa~ Scorpion King 5: Book of~ Action & Adventure, Sci-Fi & Fantasy
2 Don Michael Pa~ Death Race: Beyond Anarc~ Action & Adventure
3 Martin Scorsese Raging Bull Classic Movies, Dramas, Sports Movи~
4 Martin Scorsese The Irishman Dramas
5 Don Michael Pa~ Tremors 6: A Cold Day in~ Action & Adventure, Comedies, Horro~
```
> # A tibble: 5 x 3
> # Groups: cast [3]
>  title              cast                     release_year
> 1 Krish Trish and Baltiboy: Com - Damandeep Singh Baggan, Smita Mal - 2012
> 2 Krish Trish and Baltiboy: Par - Damandeep Singh Baggan, Smita Mal - 2010
> 3 Krish Trish and Baltiboy: The - Damandeep Singh Baggan, Smita Mal - 2013
> 4 Marc Maron: Too Real  Marc Maron           2017
> 5 Jeff Dunham: Beside Himself Jeff Dunham     2019
```
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 converts formal data stored inside the tibble into metadata stored above the tibble. For example, compare the following two tibbles:

```r
rocky_planets_1 <- tribble(
- planet, ~radius, ~sun_dist, ~year,
  "Mercury", 1516, 193.1, 88,
  "Venus", 3760, 361.0, 225,
  "Earth", 3958, 499.0, 365,
  "Mars", 2106, 760.1, 687
)

rocky_planets_1
```

```r
# A tibble: 4 x 4
# Groups: planet [4]
  planet   radius  sun_dist year
  <chr>    <dbl>   <dbl> <dbl>
1 Mercury  1516.   193.  88
2 Venus    3760.   361  225
3 Earth    3958.   499  365
4 Mars     2106.   760.  687
```
rocky_planets_2 <-
tribble(
  ~stat, ~Mercury, ~Venus, ~Earth, ~Mars,
  "radius",  1516, 3760, 3958, 2106,
  "sun_dist", 193.1, 361.0, 499.0, 760.1,
  "year", 88, 225, 365, 687
)
rocky_planets_2

> # A tibble: 3 x 5
> stat  Mercury Venus Earth Mars
> <chr> <dbl> <dbl> <dbl> <dbl>
> 1 radius  1516 3760 3958 2106
> 2 sun_dist  193. 361 499 760.
> 3 year 88 225 365 687

rocky_planets_1 stores the planet names inside the tibble and the statistic names above the tibble, and vice versa for rocky_planets_2. Neither of these configurations is superior: different data arrangements will be better suited for different problems. Thus, we would like to be able to reshape our tibble to best fit the context. And this reshaping—e.g., toggling between rocky_planets_1 and rocky_planets_2—is data pivoting.

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.
3.2 Lecture

3.2.1 Data

We will study \texttt{ufc}, a sample of 4111 Ultimate Fighting Championship (UFC) bouts data painstakingly compiled by Matthew Dabbert\textsuperscript{1}. (See section 1.3, exercise 1.16.) 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 \texttt{R}_ and the variables corresponding to the blue-corner fighter begin with \texttt{B}_. I will use \texttt{X}_ to refer to the generic case: e.g., \texttt{X\_fighter} means \texttt{R\_fighter} or \texttt{B\_fighter}. Here are the variables:

- \texttt{fight\_id}: Distinct bout identification number.
- \texttt{date, location}: Date and location of the bout.
- \texttt{red\_win}: A logical indicated whether the the boxer in the red corner wins.
- \texttt{title}: A logical indicating whether this was a championship title bout.
- \texttt{weight\_class}: The weight class of the fighters.
- \texttt{gender}: Fighter gender, either "MALE" or "FEMALE".
- \texttt{rounds}: Maximum number rounds until decision, either three, four, or five.
- \texttt{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).
- \texttt{finish\_details}: The move that finished the fight, if it ended in a knockout or submission.
- \texttt{finish\_round}: The round the fight ended, a number between one and five.
- \texttt{finish\_round\_time}: Duration of the final round, in seconds.
- \texttt{fight\_time}: Duration of the fight, in seconds.
- \texttt{audience}: A logical indicating whether there was an audience in the arena.
- \texttt{X\_fighter}: Fighter name.
- \texttt{X\_odds}: The decimal odds that that this fighter will win, from bestfightodds.com. For example, if \texttt{R\_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).
- \texttt{X\_stance}: Fighting stance.
  - "Orthodox" denotes a left-shoulder-forward stance.
  - "Southpaw" denotes a right-shoulder-forward stance.

\textsuperscript{1}https://www.kaggle.com/mdabbert
- "Switch" denotes a combination of left-shoulder-forward and right-shoulder-forward stances.

- $X_{\text{age}}$, $X_{\text{weight}}$, $X_{\text{height}}$, $X_{\text{reach}}$: Fighter age (in years), weight (in pounds), and height and arm span (in centimeters).

- $X_{\text{wins}}$, $X_{\text{losses}}$, $X_{\text{draw}}$, $X_{\text{rounds_fought}}$: Total number of UFC wins, losses, draws, and rounds fought prior to the current fight.

- $X_{\text{kd}}$: Number of times this fighter knocked down the other fighter in this bout.

- $X_{\text{strong_attempt}}$, $X_{\text{strong_achieve}}$, $X_{\text{weak_attempt}}$, $X_{\text{weak_achieve}}$: Number of strong strikes attempted, strong strikes landed, weak strikes attempted, and weak strikes landed by this fighter in this bout.

- $X_{\text{td_attempt}}$, $X_{\text{td_achieve}}$: Number of take-downs attempted and achieve by this fighter in this bout.

- $X_{\text{sub_attempt}}$, $X_{\text{sub_achieve}}$: Number of submissions attempted and achieve by this fighter in this bout.

### 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.\(^2\) Hence, most of our if 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"))
  )
```

\(^2\)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 the computer to scatter plot the `R_reach` and `B_reach` variables of `ufc`, while coloring each point according to the `weight_class` variable; furthermore, it tells the computer to create separates 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 showing the distribution of spread values for different stances and groups.](image)
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 %>%
```
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 = 3, width = 3)` 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 here³. And you can find an ugly example below:

³https://plotly.com/r/
```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 %>%
```

155
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`.

156
Exercise 3.5. The following plot—which depicts the fight_time as a function of \( \text{abs}(B\_odds - R\_odds) \)—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 = \text{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`. 

\[
\text{weight\_class}
\begin{align*}
\text{--- Women's Strawweight} \\
\text{--- Women's Flyweight} \\
\text{-- Flyweight} \\
\text{~ Women's Bantamweight} \\
\text{--- Bantamweight} \\
\text{~ Women's Featherweight} \\
\text{--- Featherweight} \\
\text{--- Catch Weight} \\
\text{--- Lightweight} \\
\text{--- Welterweight} \\
\text{~ Middleweight} \\
\text{~ Light Heavyweight} \\
\text{--- Heavyweight}
\end{align*}
\]
3.2.3 Case Study: Best Places to Gamble

We will now plot the casino’s commission by country. That is, we’ll graph the amount the bookie takes on bets placed on UFC fights held in various countries.

**Exercise 3.6.** If you bet $\frac{1}{R_{odds}}$ dollars on the red fighter then you will receive $R_{odds} \times \frac{1}{R_{odds}} = 1$ dollar from the bookie if this fighter wins. Similarly, if you bet $\frac{1}{B_{odds}}$ dollars on the blue fighter then you will receive $1$ if this fighter wins. Hence, if you simultaneously bet $\frac{1}{R_{odds}}$ dollars on the red fighter and $\frac{1}{B_{odds}}$ 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, $\frac{1}{R_{odds}} + \frac{1}{B_{odds}}$, would equal the reward from placing these bets, $1$. But the casino always takes a cut, so $\frac{1}{R_{odds}} + \frac{1}{B_{odds}}$ 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"`.
  
  - The regular expression `"\b[^,]+$"` is R shorthand for “the text after the last comma.”
Within the same `mutate()` call redefine `country = fct_relevel(country, rev)`.

- This step will change `country` from a character string to a factor, and will arrange the levels in reverse alphabetical order. (The plot will look better with this ordering.)

Save the resulting tibble as `ufc_house_take`.

Exercise 3.7.

- Pipe `ufc_house_take` into `ggplot()`.
- 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.

Exercise 3.8. Combine the solutions of the last two exercises to create your plot with one functional assembly line that begins with `ufc` and ends with your `ggplot()` call. In other words, create the plot without relying on wheelbarrow variable `ufc_house_take`. Confirm that your plot looks better with the `country = fct_relevel(country, rev)` line.

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 now plot the distribution of `finish_round_time` as a function of `finish_details` and `result`.

Exercise 3.9.

- 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 and save it as `ufc_finish_details`.

Exercise 3.10.

- Pipe `ufc_finish_details` into `ggplot()`.
• Define the cast of characters, letting `finish_round_time` play the x-coordinate role, `finish_details` play the y-coordinate role, and `result` playing the fill role.
• Use `geom_boxplot()` to add a boxplot layer.
• Tidy the plot as you see fit.

Exercise 3.11.

• Combine the solutions of the previous two exercises to create the plot without using wheelbarrow variable `ufc_finish_details`.
• After you’ve recreated the plot, add an extra `mutate()` between the `ungroup()` and `ggplot()` steps. Within this `mutate()` use `fct_reorder()` to define `finish_details` as a factor whose levels are ordered according to the `median()` `finish_round_time`.
  – See exercise 3.4.
• Save the finished plot in variable `finish_details_plot` (so we can modify it in a later exercise).

3.2.5 Updating `ggplot()` Data

We usually modify `ggplots` with the `+` symbol. For example, the following creates a black and white version of the plot we made in exercise 3.3:

```r
heat_map_plot + scale_fill_grey()
```
But to add new data rather than new options we use `%+%` rather than `+`. For example, the following removes the `gender = "MALE"` observations from `heat_map_plot` and changes the graphs’ axes from measuring pounds to measuring body mass index (BMI):

```r
female_bmi_sample <-
  pluck(heat_map_plot, "data") %>%
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
)

heat_map_plot %+% females_bmi_sample
```

In the code above, `pluck(heat_map_plot, "data")` is the sample underlying `heat_map_plot` and `female_bmi_sample` is the sample that we would like to replace it with. 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`.

And now we can use a bracketed expression to remove the `female_bmi_sample` wheelbarrow variable:
This code yields the same plot. The nested brackets indicate that we want R to execute the filter() and mutate() steps before implementing the %+%.

**Exercise 3.12.** We will now facet finish_details_plot, from exercise 3.11, by whether or not there was an upset.

- Start with `finish_details_plot %>% {. %+% {}}`.
- Within the inner brackets pipe `pluck(., "data")` into `mutate()`.
- Within the `mutate()` call 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 <= B_odds`.
  - The `red_win` variable indicates who won.
- Exit the outer brackets and use `+` to add the `facet_wrap()`.

**Exercise 3.13.** Use the %+% symbol 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 3.14.** 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 3.15.** We will now add dots to `heat_map_plot` that depict the median weight of each weight class.
• Start with `heat_map_plot %>% {. %+% {}}`.
• Within the inner brackets pipe `pluck(., "data")` into `group_by(weight_class)`.
• While still in the inner brackets, pipe the grouped tibble into `mutate()` to define `R_aggregate_weight = median(R_weight)` and `B_aggregate_weight = median(B_weight)`.
• Exit the outer brackets and use `+` to add a `geom_point()` layer. Within this function call embed `aes(x = R_aggregate_weight, y = B_aggregate_weight)` to specify that we want the points at the median weights.
  – You can attach the `+` to the brackets, so your code looks like `heat_map_plot %>% { } + facet_wrap()`.

### 3.2.6 `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:

Since both facets plot the same tibble columns, it seems impossible to have one plot heights and another plot reaches: 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 plot if we pool the `R_height` and `R_reach` data in one column and the `B_height` and `B_reach` data in another column, like so:

```r
data_to_plot <-
ufc %>%
  select(fight_id, R_height, B_height, R_reach, B_reach) %>%
pivot_longer(
  cols = ~fight_id,
  names_to = c("fighter", "stat"),
  names_sep = "_",
```r
values_to = "val"

pivot_wider(
  names_from = "fighter",
  values_from = "val"
)

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 188. 185.
#9    5   Height 178. 180.
#10   5   Reach 190. 190.
# ... with 8,212 more rows

Now to create the plot we just have to set `aes(x = R, y = B)` and `facet_wrap()` by `stat`:

```r
data_to_plot %>%
ggplot +
aes(x = R, y = B) +
geom_jitter(
  size = .01,
  height = 3,
  width = 3
) +
facet_wrap(
  vars(stat),
  ncol = 2,
  scales = "free"
) +
labs(x = "Red Fighter", y = "Blue Fighter") +
theme(
Rearranging 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 columns. To add columns, I must specify:

- where R should get the new columns’ `names_from`, and
- where R should get the new columns’ `values_from`.

Accordingly, `pivot_wider()` has two key arguments: `names_from` and `values_from`. To remove columns I must specify:

- which `cols` R should remove,
- where R should move the removed columns’ `names_to`, and
- where R should move the removed columns’ `value_to`.

Accordingly, `pivot_longer()` has three key arguments: `cols`, `names_to`, and `value_to`.

For example, define

```r
ggplot(df, aes(x = Height, y = Reach)) + geom_point() + facet_wrap(~ R_fighter)
 ```
"strong_att", "strong_ach", "weak_att", "weak_ach"
), 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)
)
)%>%na.omit
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:

```r
wide_tib <-
long_tib %>%
pivot_wider(
  names_from = metric,
  values_from = measurement
)
wide_tib
```
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 recover long_tib from wide_tib by undoing the pivot_wider() with pivot_longer():

```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.

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
> 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 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`:
long_tib_2 %>%
  pivot_wider(
    names_from = c(strength, outcome),
    names_sep = "_",
    values_from = measurement
  )

> # 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 Aalon Cruz  12        2        0        0
> 2 Aaron Phillips 54       38       225       192
> 3 Aaron Riley 138       36       40        39
> 4 Aaron Rosa  99        59       212       197
> 5 Aaron Simpson 90.7     48.7     37.7      32
> 6 Abdul Razak Alhassan 99      43        3       1
> 7 Abel Trujillo  72       39       13.5      13.5
> 8 Adam Milstead  21       10       13        12
> 9 Adam Yandiev  12        1        0        0
>10 Aiemann Zahabi 112      44        1        1
> # ... with 991 more rows

This call tells R to create the new column names by pasting together the data in the strength and outcome columns, using the "_" symbol as glue.

But we can also pivot_wider() long_tib_2 into other shapes:

#Example 1:
medium_tib_1 <-
  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>     <dbl> <dbl> <dbl>
>1 Aalon Cruz  12    2    0
>2 Aaron Phillips 54   38 225
>3 Aaron Riley 138   36   40
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.

And these different shapes permit different plots. For example, only `long_tib_2` can yield this plot:
Only `wide_tib` can yield this plot:

```r
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:
medium_tib_1 %>%
ggplot() +
aes(
  x = att,
  y = ach
) +
geom_point() +
facet_wrap(vars(strength))

And only medium_tib_2 can yield this plot:

medium_tib_2 %>%
ggplot() +
aes(
  x = strong,
  y = weak
) +
geom_point() +
facet_wrap(vars(outcome))
Exercise 3.16. 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"`.

- 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())`. For example, define `mini <- tibble(x = c("a", "b", "a"), y = c(1, 2, 1))` and compare `mini %>% count(x, y)` and `mini %>% group_by(x) %>% summarise(n = n())`. See section 2.4.3 for more on `count()`.
- `ggplot()` the result with a `geom_histogram()` layer.

We can transform 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_to = c("strength", "outcome"),
  names_sep = "-_",
  values_to = "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

# 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
For another example, the following transposes `wide_tib`, converting its columns to rows and rows to columns:

```r
wide_tib %>%
  pivot_longer(
    cols = -R_fighter,
    names_to = "measure",
    values_to = "val"
  ) %>%
  pivot_wider(
    names_from = R_fighter,
    values_from = val
  ) %>%
  select(1:5)  # View first 5 columns
```

Exercise 3.17. The shortest fight in UFC history is a sight to behold\(^4\) (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_".

\(^4\)https://www.youtube.com/watch?v=z8llI5lgWvg&ab_channel=LuckyBtw
• 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 ".\_". You can't use `names_sep = ".\_"` in this case because some names, such as `R_rounds_fought` and `B_sub_achieve`, have multiple underscores.

• Use `pivot_wider()` to convert the resulting tibble into one with columns `metric`, `R`, and `B`.

Exercise 3.18. 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()` to transform the tibble into one with columns `fight_id, fighter, stat, and val`, where `fighter` 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()`.

![](image)

3.2.7 Case Study: House Takes Have Increased

We will now demonstrate that the cut the casino takes has generally increased from around 0.02 to around 0.037.

\[^5\text{\(\text{(.\_)(.+)}\)}\text{\text{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 \"\_\".}\text{\text{}}\]

176
Exercise 3.19. Pipe ufc into `mutate()`. Within the `mutate()` call do the following:

- Define `house_take = 1/R_odds + 1/B_odds - 1`.
- Define `date_bucket = ntile(date, 200)`. Note, 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`.
- Define `weight_bucket` as a variable that equals "light" when `R_weight + B_weight` is less than its median value and equals "heavy" otherwise.

Exercise 3.20. Start with the result of the previous exercise 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 3.21. `pivot_wider()` the result of the previous exercise from a tibble with columns `date`, `weight_bucket`, and `house_take_mean` to one with columns `date`, `weight_bucket`, and `house_take_mean` to one with columns `date`, `weight_bucket`, and `house_take_mean`. `ggplot()` the result, with `aes(x = light, y = heavy, color = date)` and `geom_path()`.

3.2.8 Case Study: Winners Punch More

Exercise 3.22. The plot below demonstrates that the winning fighter generally attempts and achieves more strikes and takedowns than the losing fighter. We will recreate this plot.

- Use `select()` `fight_id, red_win, X_strong_attempt, X_strong_achieve, X_weak_attempt, X_weak_achieve, X_td_attempt, and X_td_achieve` from `ufc`.
- Have `pivot_longer()` pull down all the columns of the resulting tibble besides `fight_id` and `red_win`. Use the `names_sep = "_"` option to divide the old column names into three variables, called `fighter`, `move`, and `outcome`. For example, the column previously named `R_weak_achieve` should now correspond to the row with `fighter = "R", move = "weak", and outcome = "achieve"`, and the column previously named `B_td_attempt` should now correspond to the row with `fighter = "B", move = "td", and outcome = "attempt"`. Use the `values_drop_na = TRUE` option to disregard NA values.
  - Set `cols = -c(fight_id, red_win)`.
  - After this step your tibble should have 33228 rows and six columns: `fight_id, red_win, fighter` (which is either "R" or "B"), `move` (which is either "strong", "weak", or "td"), `outcome` (which is either "attempt" or "achieve"), and `value`.
- Use `pivot_wider()` to split the data in `value` across two new variables: `R` and `B`.
  - After this step your tibble should have 16614 rows and six columns: `fight_id, red_win, move, outcome, R, and B`.
- `ggplot()` the resulting tibble with a `geom_point()` layer. And `facet_wrap()` by `vars(outcome, move)`. 

177
• Tweak the plot so that it resembles the template.
  – To clean up the plot labels use `fct_recode()` and `fct_relevel()` before entering `ggplot()`.
• Save the plot as an R object called `red_blue_scatterplot`.

Exercise 3.23. The previous exercise asked you to “`facet_wrap()` 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()`.
3.2.9 Semi-Lengthening Pivots

A `pivot_longer()` followed by a `pivot_wider()` can yield any desired tibble configuration. But this lengthen-then-widen approach has a problem: it can change our columns’ data types. For example, in exercise 3.17 we collated the statistics of the shortest fight with the following:

```r
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("fighter", "metric"),
  names_pattern = "(.\_)(.+)",
  values_to = "stat",
  values_drop_na = TRUE
) %>%
pivot_wider(
  names_from = fighter,
  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:

```r
ufc %>%
filter(fight_time == min(fight_time, na.rm = TRUE)) %>%
select(starts_with(c("R_", "B_"))) %>%
```

179
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
)

> Error: Can't combine 'R_fighter' <character> and 'R_odds' <double>.

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

180
The R and B columns are now character string vectors. Accordingly, all the numbers in this tibble are saved as 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
  )
```

> # A tibble: 2 x 13
> corner_color fighter odds stance age weight height reach wins losses draw
> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 R Jorge ~ 2.8 Ortho~ 34 170 180.34 187.~ 15 7 0
> 2 B Ben As~ 1.45~ Ortho~ 34 170 180.34 185.~ 1 0 0
> # ... with 2 more variables: rounds_fought <chr>, sub_achieve <chr>

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`.

181
This is the fundamental problem of the lengthen-then-widen pivoting strategy: by pooling the data, \texttt{pivot_longer()} gives each column the lowest-common-denominator data type. We can obviate this problem with the \textit{semi-lengthening pivot}, which is like a \texttt{pivot_longer()-cum-pivot_wider()} packed into a single \texttt{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 = "(.)(.+)",
    values_drop_na = TRUE
  )
```

> # A tibble: 2 x 21
>   corner_color fighter odds stance age weight height reach wins losses draw
> 1 R Jorge ~ 2.8 Ortho~ 34 170 180. 188. 15 7 0
> 2 B Ben As~ 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" in the \texttt{names_to} option makes this a semi-lengthening pivot. This ".value" is a special flag that tells R to keep these values in the column names. Previously, we moved the words \texttt{fighter}, \texttt{odds}, \texttt{stance}, etc. from the column names to the \texttt{metric} column with \texttt{names_to = c("corner_color", "metric")} and then immediately moved them back to the column names with \texttt{names_from = metric}. The semi-lengthening pivot doesn’t bother creating and then destroying this \texttt{metric} column; instead, it keeps the \texttt{fighter}, \texttt{odds}, \texttt{stance}, etc. values in place and selectively pulls out of the column names only what it needs: the \texttt{corner_color} values, "R" and "B". And since it doesn’t stack dissimilar variables, this semi-lengthening pivot does not corrupt our tibble’s data types.\footnote{Also, since it doesn’t create the temporary \texttt{stat} column, this semi-lengthening pivot does not need the \texttt{values_to} option.}

For another example, the following semi-lengthening pivot derives \texttt{medium_tib_1} from \texttt{wide_tib}:

```
wide_tib %>%
pivot_longer(
```

\textbf{182}
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 (i.e., everything after "_").

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("strength", "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  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
Whereas the previous semi-lengthening pivot lengthens the tibble along the strength axis, this semi-lengthening pivot lengthens it along the outcome axis.

**Exercise 3.24.** 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):

```r
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 182   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 3.25.** Modify your solution to exercise 3.24 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 3.26.** Define the following tibble:

```r
fight_counts <-
  ufc %>%
```
Recreate the following semi-lengthening pivots with traditional `pivot_longer()` and `pivot_wider()` calls:

#Example 1:
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
> # <int> <chr> <int> <int> <int> <int> <int> <int>
> 1 1 Jan 2015 11  NA  NA  NA  NA  31
> 2 2 Jan 2016 13  NA  NA  NA  NA  24
> 3 3 Jan 2017 12  NA  NA  NA  NA  11
> 4 4 Jan 2018 10  NA  NA  NA  NA  23
> 5 5 Feb 2015 9  NA  NA  NA  NA  20
> 6 6 Feb 2016 12  NA  NA  NA  NA  23
> 7 7 Feb 2017 10  NA  NA  NA  NA  19
> 8 8 Feb 2018 11  NA  NA  NA  NA  33
> 9 9 Feb 2019 11  NA  NA  NA  NA  35
>10 10 Mar 2010 11  NA 10  NA  NA  10
> # ... with 115 more rows

#Example 2:
fight_counts %>%
pivot_longer(
everything(),
  names_to = c(".value", "month", "year"),
  names_sep = "_
)
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, suppose a customer ordered a package on Monday and received it on Friday. And further suppose that the customer received four status updates pertaining to this shipment: (i) the package being consigned to the shipper, (ii) the package moving from Handan to Xingtai, (iii) the package moving from Xingtai to Shijiazhuang, and (iv) the package moving from Shijiazhuang to Beijing, where the customer lives. Bray’s result suggests that the customer would be more satisfied with this delivery if these four announcements were scrunched toward the end of the shipping window, on Thursday and Friday, than toward the beginning of the shipping window, on Monday or Tuesday (conditional on the package arriving on Friday). When the actions cluster towards

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, suppose a customer ordered a package on Monday and received it on Friday. And further suppose that the customer received four status updates pertaining to this shipment: (i) the package being consigned to the shipper, (ii) the package moving from Handan to Xingtai, (iii) the package moving from Xingtai to Shijiazhuang, and (iv) the package moving from Shijiazhuang to Beijing, where the customer lives. Bray’s result suggests that the customer would be more satisfied with this delivery if these four announcements were scrunched toward the end of the shipping window, on Thursday and Friday, than toward the beginning of the shipping window, on Monday or Tuesday (conditional on the package arriving on Friday). When the actions cluster towards
the end of the shipping horizon, the shipper's hard work is fresh in the customers' memory when they receive their package and upload 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 they receive their package. 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, and you can find summaries of it here and here.

3.3.2 Sample

We will study `alibaba_wide`, which you created in section 1.3, exercise 1.17. 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.)
- **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:

8https://insight.kellogg.northwestern.edu/article/improve-delivery-ratings
9https://www.wsj.com/articles/with-online-package-delivery-alerts-later-in-the-process-is-better-11585352291
- "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 3.27.** 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 3.28.** Use a semi-lengthening pivot to translate `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`—to `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.
• Use the option `cols = contains(".")` to tell R to eliminate the columns whose names contain a ".".
• 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.
• Use the `values_drop_na` option to disregard the NA values.
• Include the option `names_transform = list(action_num = as.integer)` to define the `action_num` variable as an integer, rather than as a character.
• Similarly use the `values_transform` option to define the `action`, `facility`, and `shipper` variables as factors.

**Exercise 3.29.** 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.
• Pipe the result into `mutate()`. Within this operation 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` value should be 0.465 after this step.
• `ungroup()` the tibble after the `mutate()` step and save it as `alibaba_long` (overwriting the old version).

Following Bray [2020], we will now remove the extreme shipments to standardize our sample.

**Exercise 3.30.** We will subject our sample to 10 data filters.

• Start with `alibaba_long` and `group_by()` order.
• With a single `filter()` operation, do the following:
  - Use `all()` or `any()` to remove the orders with a "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".
– Remove orders that correspond to multiple shipper values.
– Remove orders with \( \text{day_count} > 8 \).
– Remove orders with more than 10 or fewer than 5 posted actions.\(^{10}\)
– Remove observations with "ORDER" and "SIGNED" actions because their time values are degenerate (mechanically being either 0 or 1).

• \texttt{ungroup()} the filtered tibble and save it as \texttt{alibaba_long}.
– \texttt{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.

\textbf{Exercise 3.31.} Use \texttt{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 \texttt{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 \( \leq 2 \) 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.

\textbf{Exercise 3.32.} Facet \texttt{density_plot} by \texttt{day_count}.

• Instead of \texttt{vars(day_count)} use \texttt{vars(str_c("Day Count: ", day_count))}.

\textbf{Exercise 3.33.} Facet \texttt{density_plot} by the number of actions in a given order.

• Use \%\% and \texttt{add_count()} to add a variable to the data underlying \texttt{density_plot} called \texttt{action_count} that records the number of actions (i.e., observations) corresponding to a given order value.

\(^{10}\)Bray [2020] required all orders to have at least four posted actions, but I’ve changed this threshold to five because it’s easier to plot six facets (i.e., 5–10 actions) than seven facets (i.e., 4–10 actions).
• 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 3.34. We will now depict the time distributions with box plots.

• Use str_replace() and str_to_title() to change the action labels from ARRIVAL, CONSIGN, DEPARTURE, GOT, and SENT_SCAN to "Arrival", "Consign", "Departure", "Got", and "Scan".
• Use fct_reorder() to order the action labels by their corresponding mean(time).
  – See the ?fct_reorder documentation.
  – This change will arrange the box plots in increasing order.
  – The first label should be "Consign".
• Pipe the result into ggplot().
• Set x = time and y = score in your cast of characters.
• Add a geom_boxplot() layer.
  – The plot looks better if you use the outlier.shape = NA option.
• facet_wrap() by action.
  – Use the strip.position = "right" and ncol=1 options.
• Call this plot box_plot.

Exercise 3.35. 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.

• Use %+% to restrict the data underlying box_plot to the (shipper, category) pairs that have at least 1000 observations.
  – You should have 23 distinct (shipper, category) groups after this step.
• facet_grid() by shipper and category.
  – Rather than rows = vars(shipper) and cols = vars(category), try using rows = vars(str_c("Ship: ", shipper)) and cols = vars(str_c("Cat: ", category)).
  – Most of the facets will be empty.
  – Add theme(axis.text.x = element_text(angle = -90)) to your plot specification, to buy more room on the horizontal axis.

Exercise 3.36. 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.
• Start with alibaba_long and group_by(order, score, and action).
• 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.\(^{11}\)
  – Your plot should have 20 facets in total.

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")
```

\(^{11}\)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.
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 `rocky_planets_1`, from section 3.1, with the following tibble:

```r
unit_of_measure <-
  tribble(
    ~stat, ~unit,
    "radius", "miles",
    "sun_dist", "light-seconds",
    "year", "Earth days"
  )

unit_of_measure
```

```
> # A tibble: 3 x 2
> stat     unit
> <chr>    <chr>
> 1 radius  miles
> 2 sun_dist light-seconds
> 3 year    Earth days
```

Unfortunately, merging `unit_of_measure` and `rocky_planets_1` would be difficult, since the latter stores `radius`, `sun_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
rocky_planets_long <-
  rocky_planets_1 %>%
  pivot_longer(
    -planet,
    names_to = "stat"
  )

rocky_planets_merged <-
  rocky_planets_long %>%
  inner_join(unit_of_measure)

rocky_planets_merged
```

```r
# A tibble: 12 x 4
#  planet stat value unit
#  <chr> <chr> <dbl> <chr>
# 1 Mercury radius 1516 miles
# 2 Mercury sun_dist 193. light-seconds
# 3 Mercury year 88 Earth days
# 4 Venus radius 3760 miles
# 5 Venus sun_dist 361 light-seconds
# 6 Venus year 225 Earth days
# 7 Earth radius 3958 miles
# 8 Earth sun_dist 499 light-seconds
# 9 Earth year 365 Earth days
#10 Mars radius 2106 miles
#11 Mars sun_dist 760. light-seconds
#12 Mars year 687 Earth days
```

```r
rocky_planets_cleaned <-
  rocky_planets_merged %>%
  mutate(
    unit = str_c("(", unit, ")")
  )%>
  unite(stat, c("stat", "unit"), sep = " ")

rocky_planets_cleaned
```

```r
# A tibble: 12 x 3
#  planet stat value
#  <chr> <chr> <dbl>
# 1 Mercury radius (miles) 1516
# 2 Mercury sun_dist (light-seconds) 193.
```
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 3.29 would have looked something like this:

```r
alibaba_wide %>%
  mutate(
    time_min =
      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,
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. 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
)

wide_example_2 <-
alibaba_long %>%
pivot_wider(
  id_cols = action_num,
  names_from = action,
  values_from = time,
  values_fn = mean
)

long_example <-
alibaba_long %>%
group_by(action, action_num) %>%
summarise(
  mean_time = mean(time),
  .groups = "drop"
)
```
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 %>%
  pivot_longer(
    -action,
    names_to = "action_num",
    values_to = "mean_time",
    values_drop_na = TRUE
  )
  %>%
  pivot_wider(
    names_from = action,
    values_from = mean_time
  )
```

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:
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)
```

rownames() ~ [ For the same reason you should And, per the same principle, you should never rely on row names.

Meta

You should minimize your reliance on metadata. For example, storing information in the `rownames()` metadata makes it nearly impossible to access: e.g., you can’t refer to a tibble’s row names in `mutate()` or `left_join()`. So you should never refer to the metadata stored in ; if these row names are is flimsy and slippery, so if you must if they’re important enough to refer to they’re important enough to treat these data seriously, it’s more transparent and instead, you should dignify th= it’s better `rownames_to_column()` or `rowid_to_column()` or `rownames` . Instead, you should ]
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_from_july_2011 <-
  ufc %>%
    filter(
      floor_date(date, "month") == "2011-07-01"
    ) %>%
    select(
      R_fighter,
      red_win,
      result
    )
```

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_from_july_2011 %>%
  pivot_wider(
    names_from = result,
    values_from = R_fighter
  )
```
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
R_from_july_2011 %>%
pivot_wider(
  names_from = result,
  values_from = R_fighter,
  values_fn = length
)
```

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_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_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 CruzDennis SiverBrian BowlesAnthony NjokuaniJeff Hougland Carlos ConditMelvin Guillard Tito Ortiz
> 2 FALSE Brad Tavares Wanderlei SilvaGeorge Roop NO DATA

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 CruzDennis SiverBrian BowlesAnthony NjokuaniJeff 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 2.2.4):

203
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
> <lgl> <chr> <chr> <chr>
> 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_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
)

> # A tibble: 2 x 4
> red_win `U-DEC` `KO/TKO` SUB
> <lgl> <chr> <chr> <chr>
> 1 FALSE Tavares Silva, Sotiropoulos <NA>
> 2 TRUE Cruz, Siver, Bowles, Njokuani, Hougland Condit, Guillard Ortiz
Exercise 3.37. The following reports the average time in the round that each finishing move took place, by `finishing_round` and `gender`:

```r
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
  )
```

> # A tibble: 27 x 10
>  finish_details `1_MALE` `2_MALE` `3_MALE` `1_FEMALE` `2_FEMALE` `4_MALE`
>  <chr>            <dbl>   <dbl>   <dbl>      <dbl>      <dbl>   <dbl>
> 1 Anaconda Choke  174.    178.    174.       NA        NA     NA
> 2 Ankle Lock      NA      25.     NA        NA        NA     NA
> 3 Arm Triangle    191.    173.    182.       74        214.    232
> 4 Armbar          183.    185.    210.       186.      182.    NA
> 5 D'Arce Choke    156.    158.    147.       223       NA      67
> 6 Elbow           170.    168.    173.       190.      226.    NA
> 7 Flying Knee     113.    103.    28.        NA        NA     NA
> 8 Guillotine Choke 161.    166.    161.       211.      NA      NA
> 9 Heel Hook       107.    NA      NA        NA        NA     NA
>10 Injury          130.    NA      111.       NA        NA     NA
> # ... with 17 more rows, and 3 more variables: 3_FEMALE <dbl>, 5_MALE <dbl>,
> 5_FEMALE <dbl>

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 3.38.** 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 3.30 if we hadn’t included an `ungroup()` step at the end of exercise 3.29. 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 = 3, width = 3)
jitter_height_plot

3.2

jitter_reach_plot <-
jitter_height_plot +
aes(x = R_reach, y = B_reach) +
labs(
    x = "Red Reach",
    y = "Blue Reach"
)
3.3

```r
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
```
Weight of Fighter in the Red Corner

Weight of Fighter in the Blue Corner

level

(0.00000, 0.00005]
(0.00005, 0.00010]
(0.00010, 0.00015]
(0.00015, 0.00020]
(0.00020, 0.00025]
(0.00025, 0.00030]
(0.00030, 0.00035]
(0.00035, 0.00040]
(0.00040, 0.00045]
(0.00045, 0.00050]
(0.00050, 0.00055]
(0.00055, 0.00060]
(0.00060, 0.00065]
(0.00065, 0.00070]

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
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(~weight_class,
             scales = "free",
             ncol = 4
  )

3.6–3.8

ufc %>%
  mutate(
    house_take = 1/R_odds + 1/B_odds - 1,
    country = str_extract(location, "\b[^,]+$"),
    country = fct_relevel(country, rev)
  ) %>%
  ggplot +
  aes(
    x = house_take,
    y = country
  ) +
  geom_violin() +
  labs(
    x = "",
    y = "House Take"
  )
3.9–3.11

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
      ),
    
  ) %>%
  ungroup %>%
  mutate(
    finish_details =
      fct_reorder(
        finish_details, finish_round_time, median,
library(ggplot2)

finish_details_plot <-
  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
```r
finish_details_plot %>% {
  . %>% {
    pluck(., "data") %>%
    mutate(
      upset =
        ifelse(
          red_win, 
          R_odds > B_odds, 
          R_odds <= B_odds
        ),
      winner =
        ifelse(
          upset,
          "Underdog Won", 
          "Favored Fighter Won"
        )
    }
  }
} +
  facet_wrap(vars(winner)) +
  theme(strip.background = element_rect(color = "white", fill = "white"))
```

![Chart showing finishing moves and time in round]

3.13
```r
斯坦斯 plots

```
ufc %>%
filter(R_age < 28) %>%
make_stance_plot

3.15

heat_map_plot %>% {
  . %>% {
    pluck(., "data") %>%
    group_by(weight_class) %>%
    mutate(
      R_aggregate_weight = median(R_weight),
      B_aggregate_weight = median(B_weight)
    )
  }
} +
  geom_point(
    aes(
      x = R_aggregate_weight,
      y = B_aggregate_weight
    )
  )
3.16

**ufc %>%**

```r
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()
```
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_sep = ".",
    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 61  1
10 sub  0  0

3.19–3.21
ufc %>%
  mutate(
    house_take = 1/R_odds + 1/B_odds - 1,
    date_bucket = ntile(date, 200),
    weight_bucket =
      ifelse(
        R_weight + B_weight < median(R_weight + B_weight),
        "light",
        "heavy"
      )
  ) %>%
  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,
    y = heavy,
    color = date
  ) +
  geom_path() +
  theme_bw() +
  labs(
    x = "Take on Light Bouts",
    y = "Take on Heavy Bouts"
  )
3.22

red_blue_scatterplot <-
ufc %>%
  select(
    c(
      fight_id,
      red_win,
      R_strong_attempt,
      R_strong_achieve,
      R_weak_attempt,
      R_weak_achieve,
      R_td_attempt,
      R_td_achieve,
      B_strong_attempt,
      B_strong_achieve,
      B_weak_attempt,
      B_weak_achieve,
      B_td_attempt,
      B_td_achieve
    )
  ) %>%
pivot_longer(
    cols = -c(fight_id, red_win),
    names_to = c("fighter", "move", "outcome"),
    names_sep = "_
  ) %>%
na.omit %>%
pivot_wider(
  names_from = fighter,
```r
values_from = value

mutate(  
  red_win = ifelse(red_win, "Red Win", "Blue Win"),  
  red_win = fct_relevel(red_win, "Red Win"),  
  move = fct_relevel(move, "strong", "weak"),
  move = fct_recode(    
    move,    
    'Strong Strike' = "strong",    
    'Weak Strike' = "weak",    
    'Takedown' = "td"
  ),
  outcome = str_to_title(outcome)
)

ggplot +
aes(x = R, y = B, color = red_win) %>%
geom_point() +
facet_wrap(    
  vars(outcome, move),    
  scales = "free",    
nrow = 2
) +
theme_bw() +
theme(    
  legend.position = "top",    
  legend.title = element_blank(),    
  theme(strip.background = element_rect(color = "white", fill = "white"))
) +
labs(    
  x = "Red Fighter",    
  y = "Blue Fighter"
)

#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"
  )
Achieve
Attempt
Strong Strike
Weak Strike
Takedown

Red Fighter
Blue Fighter
Red Win
Blue Win

3.24

wide_punch %>%
  pivot_longer(
    cols = -fight_id,
    names_to = c("color", ".value", "outcome"),
    names_sep = "-_"
  )

> # 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

3.25
#1

```r
wide_punch %>%
pivot_longer(
  cols = -fight_id,
  names_to = c("color", "strength", ".value"),
  names_sep = "_"
)
```

```r
# A tibble: 11,076 x 5
fight_id color strength attempt achieve
<int> <chr> <chr> <int> <int>
 1 75 R strong 151 94
 2 75 B strong 125 66
 3 75 R weak 190 169
 4 75 B weak 32 22
 5 76 R strong 272 137
 6 76 B strong 268 102
 7 76 R weak 3 2
 8 76 B weak 9 9
 9 77 R strong 309 194
10 77 B strong 157 83
# ... with 11,066 more rows
```

#2

```r
wide_punch %>%
pivot_longer(
  cols = -fight_id,
  names_to = c(".value", "strength", "outcome"),
  names_sep = "_"
)
```

```r
# A tibble: 11,076 x 5
fight_id strength outcome R B
<int> <chr> <chr> <int> <int>
 1 75 strong attempt 151 125
 2 75 strong achieve 94 66
 3 75 weak attempt 190 32
 4 75 weak achieve 169 22
 5 76 strong attempt 272 268
 6 76 strong achieve 137 102
 7 76 weak attempt 3 9
```

223
> 8    76 weak achieve 2   9
> 9    77 strong attempt 309 157
> 10   77 strong achieve 194  83
> # ... with 11,066 more rows

3.26

#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
)

3.37

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
)

# A tibble: 27 x 10
# 1 Rear Naked Choke 145. 172. 192. 171. 177. NA
# 2 Armbars 182. 183. 186. 185 210. 239
# 3 Guillotine Choke NA 161. 211. 166. 161. NA
# 4 Punch 188. 149. 148 161. 151. 166.
# 5 Knee NA 143. 146 201. 170. NA
# 6 Triangle Choke 220 169. 176. 160. 220. 190
# 7 Heel Hook NA 107 NA NA NA NA
# 8 Kneebar NA 137. 122 181. 143 NA
# 9 Kick 34 150. 115 134. 136. 140
# ... with 17 more rows, and 3 more variables: 3_FEMALE <dbl>, 4_MALE <dbl>, 5_FEMALE <dbl>

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
)

# A tibble: 145 x 14
# 1 Las Vegas, Nev- 31 30 31 28
# 2 Abu Dhabi, Abu- 31.5 33 26.5 30.5
# 3 Jacksonville, ~ 32.5 31.5 33 31
<table>
<thead>
<tr>
<th></th>
<th>Brasilia, Dist-</th>
<th>27.5</th>
<th>32.5</th>
<th>28</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Norfolk, Virgi-</td>
<td>28</td>
<td>29.5</td>
<td>NA</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Auckland, New</td>
<td>27.5</td>
<td>28</td>
<td>30.5</td>
<td>28.5</td>
</tr>
<tr>
<td>7</td>
<td>Minneapolis, M-</td>
<td>31</td>
<td>30.5</td>
<td>NA</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Greenville, So-</td>
<td>NA</td>
<td>29</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>Rio Rancho, Ne-</td>
<td>31.5</td>
<td>34</td>
<td>29</td>
<td>NA</td>
</tr>
<tr>
<td>10</td>
<td>Sacramento, Ca-</td>
<td>28</td>
<td>27.5</td>
<td>NA</td>
<td>30.5</td>
</tr>
</tbody>
</table>

> # ... with 135 more rows, and 9 more variables: Middleweight <dbl>,
> # Women's Strawweight <dbl>, Bantamweight <dbl>, Lightweight <dbl>,
> # Catch Weight <dbl>, Heavyweight <dbl>, Women's Bantamweight <dbl>,
> # Flyweight <dbl>, Women's Featherweight <dbl>
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 4.2.4.

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 3.2.6:

```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 will 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. In fact, it’s so powerful that it makes some of my discussion on tibble joining superfluous.

1For more on this topic see vignette("colwise") and vignette("programming").
I discuss tibble joining section 4.2.4.²
And while multi-column operations make some classic joins obsolete, they are still an essential tool. I divide the topic into “mutating joins,” which add columns, and “filtering joins,” which remove rows. As the names suggest, these joins are more powerful versions of mutate() and filter().

After the lecture you’ll put your tibble joining skills to work in section 4.3’s lab, where you will merge six different data sources. And this isn’t gratuitous joining, as this is precisely how the company saved their data. The company is a Shanghai-based supermarket, and the data set describes their grocery supply chains. You will repeat the analysis of Bray et al. [2019b] to illustrate that these supply chains suffer runs on inventory when the upstream supplier looks close to stocking out.

Finally, I provide four high-level comments, a mini-lab, and an additional set of practice problems in section 4.4. In section 4.4.1 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 4.4.2 I explain that it’s sometimes easier to consolidate multiple variables in a single column with pivot_longer() than it is to use multi-column operations. In section 4.4.3 I outline an underappreciated fact about joins: they cordon off chunks of code without breaking the functional assembly line. Thus, they’re pretty useful for adding “chapter breaks” to your code. In section 4.4.4 I argue that you shouldn’t be shy about reiterating your group_by() groupings. In section 4.4.5 I provide a short assignment that illustrates the power of joins to study connected graphs. Specifically, we’ll play a version of Six Degrees of Kevin Bacon with inner_join(). But our graph’s nodes will be chess players, and its connecting edges will be chess games.

4.2 Lecture

4.2.1 Data

In exercise 1.18 of section 1.3 you created a tibble called chess_tib that detailed 19,113 chess games played on lichess.org. This tibble has 14 variables:

- game: Distinct game identification number.
- rated: Logical indicating whether the outcome of the game influences the players’ ratings.
- 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 clock counts down when it’s the white

²For more on joins see vignette("two-table").
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.

- \texttt{white_id, black_id}: Distinct player identification numbers.
- \texttt{white_rating, black_rating}: Player ratings. Better players have higher scores.
- \texttt{moves}: The games’ moves in chess notation. For example, \texttt{game = "msIqfn6q"} has \texttt{moves = "e3 e5 Bc4 Qf6 Qf3 Qg6 Qd5 Nh6"}, which indicates that white first moved a 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.
- \texttt{opening_eco}: Classification of the game’s opening line—i.e., sequence of beginning moves—as cataloged by the \textit{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.
- \texttt{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".
- \texttt{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.

We will now remove the unserious games from this tibble to create our primary sample panel of data.

**Exercise 4.1.** filter() out the games in \texttt{chess_tib} that have fewer than eight turns (a game that’s aborted before the eighth move isn’t a proper game), or more than a 250-point differential between \texttt{white_rating} and \texttt{black_rating} (these games are too lopsided to take seriously). Save this filtered sample as \texttt{chess_panel}.

### 4.2.2 Tidy Select

Suppose we wanted to distinguish the white moves listed \texttt{chess_panel} (see exercise 4.1) from the black moves. The current data configuration—which stuffs all moves into a single column—makes this difficult. So, we will create a new tibble that assigns each of the first 200 moves its own variable:

```r
first_200_moves <-
  chess_panel %>%
  select(game, moves) %>%
  separate(    
    moves, }
```
```r
str_c("move_", 1:200),
    sep = " ")

first_200_moves %>%
    select(1:9) %>%
    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 l1NXwaE 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 Bf4 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.

Since white goes first, the white player makes all the odd moves and the black player makes all the even moves. Thus, we could `select()` all the white moves with the following:

```r
first_200_moves %>%
    select(ends_with(c("1", "3", "5", "7", "9")))
```

But this solution would require listing all 100 white-move variables. A smarter solution is to `select()` all the variables that end in a "1", "3", "5", "7", or "9":

```r
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 | move_191:move_200) picks the first and last ten moves,
• select(!(move_1:move_10 | move_191:move_200)) picks all but 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(ends_with("29")) picks move_29 and move_129,
• select(starts_with("move_9")) picks move_9 and move_90-move_99,
• 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" in at least one row.

The last two examples are particularly interesting. They use where() to apply a test that assigns a TRUE or FALSE value to each column, and the outer select() operation chooses the TRUE columns. We can specify the test with a function, such as is.character, or a functional assembly line, such as ~.x %>% str_detect("Nxf6") %>% any(na.rm=TRUE). The tidy select functional assembly lines always begin with ~.x, which tells R to apply the sequence of function to every column in the tibble, rather than any one specific column. Hence, for the last example R evaluates

```r
• 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 variables for which the assembly line yields TRUE.

**Exercise 4.2.** Use where() and is.numeric() to select the numeric columns of chess_panel.

**Exercise 4.3.** Use & and ends_with() to select all the black moves between the 20th and 60th turns (inclusive).

**Exercise 4.4.** 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 select all columns that incorporate at least 10 pawn promotions.

• Pipe first_200_moves into a select() call that incorporates a nested where() call.
• Within the where() write ~.x %>% and then attach to this a functional assembly line that counts the number of elements in an inputted vector that contain the symbol "=" and returns TRUE if this number is at least 10.
  – Use str_detect() and sum(na.rm=TRUE).
— You can incorporate a test that determines whether a number is as large as 10 is within a functional assembly line with a bracketed expression (see section 2.2.5). For example, 11 %>% {. >= 10} = TRUE and 9 %>% {. >= 10} = FALSE.

4.2.3 Multivariable Operations

The tidy select convention isn’t limited to `select()`. In fact, it’s been rolled out to most tidyverse functions. For example, I can reposition all the white-move columns before all the black-move columns with:

```r
first_200_moves %>%
  relocate(
    ends_with(c("0", "2", "4", "6", "8")),
    .after = last_col()
  )
```

Or I can pivot by all numeric variables with:

```r
chess_panel %>%
  pivot_longer(where(is.numeric))
```

However, we often need the helper function `across()` to invoke the tidy select conventions. For example, we could `group_by()` every variable with

```r
chess_panel %>%
  group_by(across(everything()))
```

In contrast, `chess_panel %>% group_by(everything())` yields an error.

Here’s an example with `summarise()`:

```r
first_200_moves %>%
  summarise(
    across(
      last_col(4):last_col(),
      n_distinct
    )
  )
```
This `across()` call has two inputs: the first specifies a set of columns and the second provides a function to apply to these columns. Hence, the code above applies `n_distinct()` to the last five columns of `first_200_moves`. (Note that I write `n_distinct` rather than `n_distinct()`, as the extra parentheses yield an error in this case.)

I can also use the `~.x` `%%%` convention to apply a functional assembly line to the specified columns:

```r
first_200_moves %>%
  summarise(
    across(
      across(
        last_col(4):last_col(),
        ~.x `%%%` is.na `%%%` mean
      )
    )
  )
```

The result indicates that 99.9% of `move_196–move_200` values are NA.

The notation is the same for `mutate()`. For example, the following applies `str_to_upper()` to every variable besides `game`:

```r
first_200_moves %>%
  mutate(
    across(
      -game,
      str_to_upper
    )
  ) `%%%`
  select(1:7) `%%%`
  head
```

> # A tibble: 1 x 5
>     move_196 move_197 move_198 move_199 move_200
>    <int>    <int>    <int>    <int>    <int>
> 1    1       26       23       23       20

> # A tibble: 1 x 5
>     move_196 move_197 move_198 move_199 move_200
>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
> 1 0.998 0.998 0.998 0.998 0.998
And the following embeds two `across()` calls in our `mutate()` step to append the label "w_" to white moves and "b_" to black moves:

```r
first_200_moves %>%
  mutate(
    across(  
      ends_with(c("1", "3", "5", "7", "9")),  
      ~.x %>% str_c("w", ., sep="_")  
    ),
    across(  
      ends_with(c("0", "2", "4", "6", "8")),  
      ~.x %>% str_c("b", ., sep="_")  
    )
  )
```

Note, the . symbols in the `str_c()` calls tell the conveyor belts to feed the input character strings into the slot after the "w" and "b" labels: e.g., "Qf5" %>% str_c("w", sep="_") = "Qf5_w" and "Qf5" %>% str_c("b", ., sep="_") = "w_Qf5".

**Exercise 4.5.** 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 "_check-mate".

- Use `~.x` to specify all columns besides `game`.
- 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 4.6.** 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 `is.character()`, without the parentheses, 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", "b") %>% n_distinct %>% {. <= 2} = TRUE` and `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 4.7.** 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 that applies `as.integer()` to all variables that (i) are numeric and (ii) have no decimal component.
  - Use `is.numeric()`, without the parentheses, to identify the numeric columns.
  - Embed a functional assembly line within `where()` that returns `TRUE` if the input vector can be converted to the integer type without changing its values. Note that `c(1, 2, 3) %>% {. == as.integer(.)} %>% all(na.rm = TRUE) = TRUE` and `c(1.5, 2, 3) %>% {. == as.integer(.)} %>% all(na.rm = TRUE) = FALSE`.
  - Write `as.integer` rather than `as.integer()`.
• Overwrite the old `chess_panel` with the new tibble.

**Exercise 4.8.** 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 king moves from square e1 to g1 if white and from e8 to g8 if black, and under a queen-side castle—a move denoted by "O-O-O"—the king moves from square e1 to c1 if white and from e8 to c8 if black. 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 piece is white and "Kg8" if black, and change "O-O-O" to "Kc1" if the piece 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 "Kgl-0".

- In the black across() change "O-O-O" to "Kc8" and "O-O" to "Kg8".

Now let’s count the number of pieces captured in each turn. Moves that capture pieces are flagged with an "x": e.g., Bxc6, dxc6, and Nxe5 are piece-capturing moves, whereas Bc6, dc6, and Ne5 are not. Hence, we can embed str_detect("x") inside across() to count the number of pieces captured in move_1–move_200:

```r
first_200_moves %>%
  summarise(
    across(
      starts_with("move"),
      ~ .x %>%
        str_detect("x") %>%
        sum(na.rm = TRUE)
    )
  ) %>%
  select(1:7)
```

```
# A tibble: 1 x 7
move_1 move_2 move_3 move_4 move_5 move_6 move_7
<int> <int> <int> <int> <int> <int> <int>
 1 0 0 465 989 1274 2340 2509
```

But now instead of vertically, I want to sum horizontally—i.e., I want to calculate the number of pieces captured in a given game rather than in a given turn. I can switch from aggregating across rows to aggregating across columns by switching from across() to c_across():

```r
first_200_moves %>%
group_by(game) %>%
  summarise(
    num_captured =
      c_across(starts_with("move")) %>%
      str_detect("x") %>%
      sum(na.rm = TRUE)
  )
```

236
In this case, the `c_across()` call compresses a row’s `move_1–move_200` values into a vector that gets piped into `str_detect()`. For another example, the following sums the number of captured pieces by opening move:

```r
first_200_moves %>%
  group_by(move_1) %>%
  summarise(
    num_captured =
      c_across(starts_with("move")) %>%
      str_detect("x") %>%
      sum(na.rm = TRUE)
  )
```

> # A tibble: 20 x 2
> move_1 num_captured
> <chr> <int>
> 1 a3 213
> 2 a4 124
> 3 b3 2110
> 4 b4 902
> 5 c3 618
> 6 c4 8179
> 7 d3 818
> 8 d4 50324
> 9 e3 3781
> 10 e4 135521
> 11 f3 180
> 12 f4 1737
In this case, the `c_across()` call compresses the `move_1–move_200` values of the given `move_1` group into a vector that gets piped into `str_detect()`.

**Exercise 4.9.** Pawn moves are those that do not begin with "R", "N", "B", "Q", "K", or "O". We will now create a tibble that reports the number of white and black pawn moves.

- Remove from `first_200_moves` the games for which `move_200` is not `NA`. These games did not finish within our 200-move window, and thus we can’t determine their total pawn moves.
- `group_by()` `game`.
- Pipe the result into `mutate()`.
- Within the `mutate()` call, 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".

  - "^[RNBQKO]" is a regular expression that denotes a character string that starts with "R", "N", "B", "Q", "K", or "O". For example, "Robert is my name" %>% `str_detect(^[RNBQKO])` = TRUE and "my name is Robert" %>% `str_detect(^[RNBQKO])` = FALSE.

  - The `negate=TRUE` option reverses the output of `str_detect()`: "Robert is my name" %>% `str_detect(^[RNBQKO], negate=TRUE)` = FALSE and "my name is Robert" %>% `str_detect(^[RNBQKO], negate=TRUE)` = TRUE.

- Define `black_pawn_moves` with analogous code.
  - You can do this within the same `mutate()` call.
- After the `mutate()` step, use `relocate()` to position `white_pawn_moves` and `black_pawn_moves`.after = "game", so you can more easily view these columns.
- Call the new tibble `pawn_moves`.

---

`c_across()` also pairs extremely well with `filter()`. For example, the following selects the games in which the white player castled on the queen side, a move denoted by "0-0-0":

```r
> 13 g3 1805
> 14 g4 320
> 15 h3 79
> 16 h4 223
> 17 Na3 6
> 18 Nc3 896
> 19 Nf3 7273
> 20 Nh3 75
```
rowwise() is a function that divides a tibble so that each row comprises its own group_by() group. In this case rowwise() and group_by(game) are equivalent, since every row has a distinct game ID value. But if my tibble had some repeated rows then I would have to use rowwise() to assign them to different groups. The rowwise() grouping structure makes the filter() step implement the functional assembly line row by row. The c_across(ends_with(c("1", "3", "5", "7", "9"))) term compresses the given row's move_1, move_3, move_5, ..., move_197, and move_199 values into a vector which then gets piped into str_detect("0-0-0") %>% any. And if the assembly line returns TRUE then the row is saved.

For another example, the following selects the games in which the first piece was captured in the tenth move (recall that piece captures are denoted with an "x"):

This filter() has two conditions. The first, str_detect(move_10, "x"), a conventional single-variable restriction, requiring move_10 to contain an "x". The second condition, c_across(move_1:move_9) %>% str_detect("x", negate=TRUE) %>% all(na.rm=TRUE), is a multi-column restriction, baring move_1–move_9 from containing an "x". Note that the negate=TRUE option makes str_detect() return TRUE when it doesn’t find an "x". Also, note that this example’s all() requires the na.rm=TRUE option, whereas the previous example’s any() did not. This is because c(TRUE, TRUE, NA) %>% all = NA whereas c(TRUE, TRUE, NA) %>% any = TRUE.

Exercise 4.10. Select the games of first_200_moves in which the white player checkmated the black player with a knight.

- Knight checkmates are the moves that begins with an "N" and ends with a ":". For example, "Nec4#" and "Nxc7#".
• In R, "^N.+#$" is shorthand for "string that starts with an "N" and ends with a ".#". For example, "No, he said %$&#" %>% str_detect("N.+#") = TRUE.

**Exercise 4.11.** Select the games in which the white player never used a bishop and the black players never used a rook.

• Bishop moves are those that start with a "B" and rook moves are those that start with an "R".
• Regular expressions "^B" and "^R" denote character strings that start with a "B" and "R", respectively.
• Pipe two c_across() calls into two assembly lines, one for the white constraint and the other for the black constraint.

**Exercise 4.12.** Select the games in which the black player checked the white king more than the white player checked the black king.

• Recall that str_detect("\+") indicates whether a move is checking. For example, "Bb5+" %>% str_detect("\+)") = TRUE.
• Connect c_across(ends_with(c("1", "3", "5", "7", "9"))) to an assembly line to determine the number of white checks and connect c_across(ends_with(c("0", "2", "4", "6", "8"))) to an assembly line to determine the number of black checks. And use < to determine if the latter exceeds the former.

**Exercise 4.13.** Select the games 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 = "7rbdtPwO" 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

So far, we’ve applied all our multi-column filters rowwise(). But we can also apply them at the group level. For example, if we define an “opening line” as a game’s first six moves then the following restricts the sample to the opening lines whose corresponding games have no castling moves (i.e., moves that contain the substring "0-0"):

```r
first_200_moves %>%
  group_by(across(move_1:move_6)) %>%
  filter(
    c_across(everything()) %>%
    str_detect("0-0", negate=TRUE) %>%
    all(na.rm=TRUE)
  )
```
In this case, the `c_across(everything())` term compresses every variable of every game with a given opening line into one long vector, which gets piped into `str_detect()`.

### 4.2.4 Joins

I want to copy columns `white_pawn_moves` and `black_pawn_moves` from `pawn_moves`, which we made in exercise 4.9, to `chess_panel`. I will do so with a `join`:

```r
tibble_to_merge <-
pawn_moves %>%
  select(game, white_pawn_moves, black_pawn_moves)

chess_panel %>%
  inner_join(tibble_to_merge) %>%
glimpse
```

> Joining, by = "game"

> Rows: 14,300
> Columns: 16
> $ game <chr> "l1NXvwaE", "mIIcVQHh", "kWKrY3L", "9tXoUAU", "qwU9~
> $ rated <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALS~
> $ turns <dbl> 16, 61, 61, 95, 33, 66, 119, 39, 31, 31, 43, 52, 66, ~
> $ winner <chr> "black", "white", "white", "white", "black", "white", "w~
> $ increment_code <chr> "5+10", "5+10", "20+0", "30+3", "10+0", "15+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,~
> $ moves <chr> "d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 fxe5 Nxe5 Qd4 Qc4 Nc6 Qe5+ ~
> $ opening_eco <chr> "B00", "C20", "D02", "C41", "D00", "C50", "B01", ~
> $ opening_name <chr> "Nimzowitsch Defense: Kennedy Variation", "King's Paw~
> $ opening_ply <dbl> 4, 3, 3, 5, 10, 6, 4, 1, 2, 8, 7, 8, 8, 5, 4, 4, 4, 4, ~
> $ white_pawn_moves <int> 6, 12, 4, 13, 3, 7, 14, 5, 5, 2, 6, 4, 7, 17, 6, 9, 8~
> $ black_pawn_moves <int> 3, 10, 4, 10, 5, 7, 13, 6, 3, 4, 6, 3, 2, 9, 5, 7, 6,~
```

The `Joining, by = "game"` message at the top indicates that the `inner_join()` operation merges `chess_panel` with `tibble_to_merge` by their one common game variable. In other words, it matches up all rows in `chess_panel` with all rows in `tibble_to_merge` that share the same game value, and then it pastes together these matched rows, with the `chess_panel` values on the left and the `tibble_to_merge` values on the right. Accordingly, for this merge...
we call `chess_panel` the left-hand tibble (LHT) tibble and `tibble_to_merge` the right-hand tibble (RHT).

But there's a problem with this join: it deletes 22 rows of data, as `chess_panel` has 14,322 rows but the merged tibble has only 14,300 rows (as the output at the top reports). The issue is that we removed the 22 games that lasted at least 200 moves from `pawn_moves`, which means that there are 22 rows of `chess_panel` that have no counterparts in `tibble_to_merge`. 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:

```
chess_panel %>%
  left_join(tibble_to_merge) %>%
dim
```

> Joining, by = "game"

> [1] 14322 16

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 = "sMMTqjS2" row of the merged tibble has `white_pawn_moves = NA` and `black_pawn_moves = NA` because this game does not appear in `tibble_to_merge`. 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()`.

There's still one niggling issue with the solution above: it's use of wheelbarrow variable `tibble_to_merge`. To avoid creating such clutter we would generally nest the `select()` call within the `inner_join()` call like this:

```
chess_panel %>%
  left_join(
    pawn_moves %>%
      select(game, white_pawn_moves, black_pawn_moves)
  )
```

Any processing to the LHT could be done before the join but any processing to the RHT must be nested within the join (if we want to avoid wheelbarrow variables). Thus, our
inner_join() and left_join() calls will often encapsulate long functional assembly lines, like in the next example:

```r
chess_panel %>%
  left_join(
    chess_panel %>%
      filter(
        rated,
        turns >= 12,
        winner != "draw",
        (white_id == "ivanbus" & black_rating > 1000) | 
        (black_id == "ivanbus" & white_rating > 1000)
      ) %>%
      group_by(opening_name) %>%
      summarise(
        ivan_win_rate =
        mean(
          (white_id == "ivanbus" & winner == "white") | 
          (black_id == "ivanbus" & winner == "black")
        )
      )
  )
```

This code adds to `chess_panel` a variable called `ivan_win_rate` that indicates how proficient this player is with the given opening. Specifically, it records the fraction of “serious games” that "ivanbus" won with the given `opening_name`, where a “serious game” is one that (i) is rated, (ii) has at least a dozen moves, and (iii) is played against an opponent with a rating of at least 1000.

I call this kind of join a “mutating” join because it adds a column of data to the LHT, as `mutate()` does. In fact, we could get the same result with the following `mutate()` call:

```r
chess_panel %>%
  group_by(opening_name) %>%
  mutate(
    ivan_win_rate =
    subset(
      (white_id == "ivanbus" & winner == "white") | 
      (black_id == "ivanbus" & winner == "black"),
      rated &
      turns >= 12 &
      winner != "draw" &
    )
  )
```

243
However, I prefer the former solution because its nested assembly line is more straightforward than the latter solution’s nested assembly line. And this is usually the case: code nested within `inner_join()` will usually be simpler than analogous code nested within `mutate()` because whereas we can only use vector-level operations within `mutate()` we can tibble-level operations within `left_join()`. And tibble-level operations, such as those that comprise the functional quartet, are generally easier to follow than vector-level operations, such as `subset()` (which is the vector analog of `filter()`).

Everything we can do with `mutate()` we can also do with a mutating join, but not vice versa. For example, suppose I wanted a variable called `games_white_won_as_white` that denotes the number of games that the current `white_id` player won in the white position. I can calculate this with either `mutate()` or 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(
    chess_panel %>%
      group_by(white_id) %>%
      summarise(
        games_white_won_as_white =
          sum(winner == "white")
      )
  )
```

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 in the black position. 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 = "epDNgiO"` 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 stored in the `black_id = "yokocito"` group. However, we can define `games_white_won_as_black` with a more powerful mutating join:

```r
class(chess_panel) = "tbl_df"
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 this code: First, whenever you’re within a function’s parentheses you can refer to the data that got piped into that function with the . symbol. 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`.

**Exercise 4.14.** We will now add a column to `chess_panel` called `white_opponent_count`, which records the number of distinct opponents faced by the white player. For example, `game = "10lHsFmM"` should have `white_opponent_count = 4` because it has `white_id = "bat_of_doom"`, who played against "charzi" and "tepa_1996" from the white position and against "dog555" and "gui645" from the black position. Note, the information necessary to calculate this `white_opponent_count = 4` value is scattered across both the `white_id = "bat_of_doom"` and `black_id = "bat_of_doom"` groups, which prevents us from deriving it from `mutate()`. Accordingly, we will use a more general mutating join.

- Pipe `chess_panel` into `summarise()`, and use this function to define `player = c(white_id, black_id)` and `opponent = c(black_id, white_id)`.
- Group the result by `player`, and use a `summarise()` operation to define `white_opponent_count = n_distinct(opponent)`.3

[3] I write `filter(., winner == "black")` instead of `%% filter(winner == "black")` because, for technical reasons, the . symbol can’t be attached to a conveyor belt.
Now you have a functional assembly line that converts `chess_panel` into `white_opponent_count`. Embed this functional assembly line inside a `left_join()` call to add these `white_opponent_count` values to the `chess_panel` tibble.

- Use the `by = c("white_id" = "player")` option.

**Exercise 4.15.** 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:
  - `group_by()` `opening_name`.
    * Use the . convention to refer to the tibble that got piped into the `inner_join()`.
  - `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()`.

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 don’t
need a filtering join to restrict the sample to the opening_eco values that have at least 10 distinct opening_name values:

```r
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
eo_values_to_keep <-
  chess_panel %>%
  filter(rated) %>%
  group_by(opening_eco) %>%
  summarise(n = n_distinct(opening_name)) %>%
  filter(n >= 10) %>%
  select(opening_eco)

ches panel %>%
  inner_join(eo_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.

We can further streamline our solution:

```r
ches panel %>%
  semi_join(
    chess_panel %>%
    filter(rated) %>%
    group_by(opening_eco) %>%
  ```
I've made two changes here. First, I nested the query 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
Now we can filter out the \( x = "b" \) rows from \texttt{mini_1} with a \texttt{semi_join()}:

\begin{verbatim}
mini_1 %>%
  semi_join(mini_2)
\end{verbatim}

\begin{verbatim}
> # A tibble: 3 x 2
>   x    y
> 1 a    1
> 2 a    2
> 3 c    5
\end{verbatim}

We can add the \texttt{z} column to \texttt{mini_1} with a \texttt{left_join()}:

\begin{verbatim}
mini_1 %>%
  left_join(mini_2)
\end{verbatim}

\begin{verbatim}
> # A tibble: 5 x 3
>   x    y    z
> 1 a    1    X
> 2 a    2    X
> 3 b    3    <NA>
> 4 b    4    <NA>
> 5 c    5    Y
\end{verbatim}

And we can both filter out the \( x = "b" \) rows and add in the \texttt{z} column with an \texttt{inner_join()}:

\begin{verbatim}
mini_1 %>%
  inner_join(mini_2)
\end{verbatim}

\begin{verbatim}
> # A tibble: 3 x 3
>   x    y    z
> 1 a    1    X
> 2 a    2    X
> 3 c    5    Y
\end{verbatim}
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 4.16.** 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)`.

The `select(game)` in the solution to exercise 4.16 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 4.17.** We want to remove games in which the given white player faces their highest-ranked black opponent or the given black player faces their highest-ranked white opponent. Which of the two options does this?
# 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
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 4.18. Suppose you want to study the openings of the player "wall-of-shields" to prepare for an upcoming face-off with them. Accordingly, you wish to select all the games that share an opening_name with one of this player's games.

- Pipe chess_panel into `inner_join()`.
- Within the `inner_join()`, `filter()` out the games not played by "wall-of-shields" (in either the white or black position) and select the `distinct()` opening_name values.

What if we replaced `distinct(opening_name)` with `select(opening_name)` in our solution to exercise 4.18? This would be bad. An `inner_join()` always matches every LHT row with every possible RHT row. Hence, an LHT row will be represented multiple times in the joined sample if it matches with multiple RHT rows, and vice versa. For example, "wall-of-shields" played opening_name = "Rat Defense: Balogh Defense" three times, so if we used `select(opening_name)`, rather than `distinct(opening_name)`, then the RHT would have three rows with opening_name = "Rat Defense: Balogh Defense", which would mean that every row in the LHT with this opening would have three copies in the final tibble.

Here's an illustration:

short_1 <-
tibble(
x = c("a", "b", "c"),
y = 1:3
)
short_1

> # A tibble: 3 x 2
> x y
> <chr> <int>
> 1 a 1
> 2 b 2
> 3 c 3

short_2 <-
tibble(  
x = c("a", "a", "a", "b")
  )
short_2

> # A tibble: 4 x 1
> x
> <chr>
> 1 a
> 2 a
> 3 a
> 4 b

short_1 %>%
  inner_join(short_2)

> # A tibble: 4 x 2
> x y
> <chr> <int>
> 1 a 1
> 2 a 1
> 3 a 1
> 4 b 2

In addition to killing the \( x = "c" \) row, this join triples the \( x = "a" \) row. So watch out for repeated rows when you join.
4.3 Lab: Inventory Runs

4.3.1 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.

4.3.2 Sample

We will use order_dates, category, order_store, ship_store, inv_store, and inv_DC from section 1.3, exercise 1.19. 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.

### 4.3.3 Clean

Company data is almost always buggy, so you should thoroughly vet any table that you receive from a firm. Indeed, a few proactive checks will inevitably save you time, as it’s easier to catch issues upfront than to trace them back through multiple layers of analysis.

One test that you should always run is the key test. They key of a table is a minimal set of columns that uniquely identifies an observation. The phrase “uniquely identifies an observation” means that every row in the table 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` table looks like this:

```r
slice(inv_store, 1000:1004)
```

```r
# A tibble: 5 x 4
store item date inv
<dbl> <dbl> <date> <dbl>
1 1003 11042 2012-03-25 22
2 1003 11042 2012-03-26 19
3 1003 11042 2012-03-27 18
4 1003 11042 2012-03-28 17
5 1003 11042 2012-03-29 16
```
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:

```r
nrow(inv_store)
```

```r
eval(ee)

> [1] 16490877
```

```r
inv_store %>%
  distinct(store, item, date) %>%
  nrow
```

```r
eval(ee)

> [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:

```r
inv_store %>%
  distinct(item, date) %>%
  nrow
```

```r
eval(ee)

> [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 table's shortest row identifier—i.e., its key.

A table's key is usually quite easy to identify. Often, a table's variables will fall in two broad categories: identification variables and measurement variables. The latter store the basic data recordings—i.e., “what the table is about”—whereas the former store the “metadata” 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 4.19.

- Determine the identification variables of `inv_DC`, and use this information to guess what this table’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 exercise illustrates the dangers of not checking the key.

Exercise 4.20.

- Load the `item`, `date`, and `inv` columns of `inv_store` pertaining to `store == 7511` into a tibble called `inv_3722`.
- Load the `item`, `date`, and `order` columns of `order_store` pertaining to `store == 7511` into a tibble called `order_7511`.
- `inv_3722` clearly has one measurement variable, `inv`, and two identification variables, `item` and `date`. So `(item, date)` is probably its key. Use `nrow()` and `distinct()` to confirm this supposition.
- `order_7511` has the same structure, so we’ll assume that its key is also `(item, date)`.
- `left_join()` tibbles `inv_3722` and `order_7511` to create a tibble called `master_panel_7511`.
  - `master_panel_7511` should have 186,468 rows.
- If `(item, date)` is the key for `inv_7511` and `order_7511` then it’s also 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.
  - The median value of these daily aggregates should be 9,275 units of inventory.
- The previous calculation is predicated on the assumption that each row of `order_7511` has a distinct `(item, date)` pair. Use `count()` to demonstrate that this is untrue.
- Use `group_by()` and `summarise()` to modify `order_7511` so that `order` reports the total amount of a given `item` ordered on a given `date`.
  - By construction, `order_7511` has key `(item, date)`.
  - `order_7511` should have 4,954 rows.
- Use your new version of `order_7511` to create a new version of `master_panel_7511` and then recalculate the total inventory the store holds on each date.
The new median aggregate inventory level should change to 9,271 which is the correct answer.

In the problem above, a bug in the order data lead to an error in the inventory calculation. Specifically, redundant rows of order_7511 lead to redundant rows in master_panel_7511, which lead to some inventory values being double counted. 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 master panel, tainting calculations that are unrelated to the initial issue. Hence, it’s essential to thoroughly check all tables that contribute to your analysis, even those that are seemingly inconsequential.

After inspecting your input tables, you’ll probably want to left_join() them into a master panel, to store all your important variables in one convenient location. But before assembling this catchall table, you should anticipate its key—it’s primary unit of observation. Sometimes this key will comprise the keys of all the constituent input tables. But usually we’ll want to drop some of these key variables, to simplify the master panel. For example, order_store has key (store, item, date, order_code), but we don’t want order_code in our master panel because it simply accounts for the circumstance in which a store orders the same item multiple times in one day, an event that happens less than 1% of the time, as the follow query establishes.

```r
order_store %>%
  count(store, item, date) %>%
  summarise(mean(n > 1))
```

```r
# A tibble: 1 x 1
`mean(n > 1)`
<dbl>
1 0.00778
```

Moreover, order_code is extraneous to our analysis: all we will need is the total amount of a given item ordered by a given store on a given date, and the amount the DC ships in response. Hence, we can aggregate the order and shipping data to a tibble with key (store, item, date) without losing any material information. And since these three variables also compose the inv_store and inv_DC keys, they will be sufficient to serve as our master panel's key.

The following exercise combines order_store and ship_store into a simpler tibble called order_ship.

**Exercise 4.21.**
• Drop the observations in `order_store` with `order_type == "CD"`, and remove the `order_type` column. In other words, disregard the products that are cross-docked at the DC.
• `left_join()` in `ship_store`.
• `ship` will be `NA` when the DC did not fulfill the given `order`. Use `replace_na()` to change these `NA` terms to zero, since zero units were shipped in this case.
• 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.
• `ungroup()` the tibble, and assign it the name `order_ship`.
  – `order_ship` should have 409,636 rows.

Our input tables now have consistent observations: `order_ship` and `inv_store` report by `(store, item, date)` and `inv_DC` by `(item, date)`. So we could merge these input tables to create a master panel (repeating Inv, from `inv_DC`, for each value of `store`).

However, we always want to delay constructing our master panel as long as possible, because its large size makes it cumbersome to work with. So we won’t merge our tables until we’ve finished all the data processing that we can do before merging our tables. And the following exercise illustrates that there’s still some work we can do on `inv_store`.

**Exercise 4.22.** We will create two plots that depict two problems with `inv_store`.

• Use `ggplot()` to scatter-plot `inv` by `date` for the observations with `store == 1039` and `item == 211610`. After looking at this plot, save it as an R object called `inv_plot`.
  – Use the `scale_y_continuous(limits = c(0, NA))` option to make the y-axis start from zero.
• Create an analogous plot for the observations with `store == 7723` and `item == 13045`.
  – Do not redefine the plot from scratch. Rather, use the `inv_plot %+% {.}` convention, where you put the derivation of the new data inside the brackets.
The first plot you created in exercise 4.22 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 intense 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 2014-12-31 to ending on 2015-06-01, excising most of the degenerate data.

We’ll add a product generation variable to `inv_store` 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 4.23.**

- Use `arrange()` to order `inv_store` 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`.
- Create logical variable `time_jump` 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 and only if there’s more than a two-week gap in the inventory records.
  - Recall: We shouldn’t embed constants, like the number 14, in our code. Instead, we should define some descriptive variable upfront, such as `allowable_time_gap <- 14`, and then use that in our code.
• Create logical variable `value_change` that indicates whether this row’s inventory level differs from the previous row’s inventory level. Use `replace_na()` to turn `NA` values `FALSE`.
• Using `cumsum()`, 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.
• Define logical variable `stationary` that indicates whether the value of `cum_value_change` in this row equals that from 60 rows back. In other words, let `stationary` be `TRUE` if and only if the inventory level hasn’t changed over the past 60 observations.
• Define logical variable `new_gen` that is `TRUE` if and only if `time_jump` or `stationary` are `TRUE`. 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.
• Use `unite(., remove = FALSE)` to create a column called `product` that combines `store`, `item`, and `gen`. This variable will serve as our product identification number.
  – The first observation should have `product == "1003_10210_0"`.

We will remove undesirable products in the next problem. First, we will disregard `product` values with fewer than 200 observations. This filter ensures that that all but the first 60 days of any long stretch of constant inventory is thrown out: once the inventory level has been flat for 60 days, each subsequent day with the same inventory level starts a new generation, and so these latter generations each comprise a single observation. Second, to further reduce the influence of phantom inventory, we will drop `product` values for which the inventory level changes less than 4% of the time.

**Exercise 4.24.**

• Start with `inv_store` and use `inner_join()` and `count()` to filter out the `product` values that have fewer than 200 observations. This filter restricts the sample to product generations with sufficiently long inventory series.
  – Your tibble should have 13,944,803 rows after this filter.
• Use `inner_join()` to filter out the `product` values for which `value_change == TRUE` for less than 4% of observations. This filter restricts the sample to product generations with a sufficiently dynamic inventory series.
  – Your tibble should have 13,823,473 rows after this filter.
• Drop all columns besides `product`, `store`, `item`, `date`, and `inv`. 
Now that we’ve cleaned our input tables, we’re ready to merge them into the master panel that we will use for our analysis.

**Exercise 4.25.**

- Create a list comprising `inv_store`, `inv_DC`, `order_ship`, and `order_dates` in that order.
- Use `reduce(., left_join)` to merge the four tables in this list into a tibble called `master_panel`.
  - `master_panel` should have nine columns and 13,823,473 rows.
- Your `order` and `ship` values will be `NA` if (i) no inventory was ordered on the given date or (ii) `order_observed == FALSE`. Use `ifelse()` and `is.na()` to turn the `NA` values in `order` and `ship` to zero if `order_observed == TRUE`. That is, set `order == 0` and `ship == 0` if we explicitly observed that no inventory was ordered.
  - `order` and `ship` should be all integers in the `order_observed == TRUE` subsample and should be all `NA` terms in the `order_observed == FALSE` subsample.
- Drop the `order_observed` variable.

Now we’re almost finished—we just have to tidy up `master_panel` a bit. And this is common: constructing a master panel usually requires both a pre-merge and a post-merge round of cleaning. Indeed, while we always do as much data processing as possible before joining our input tables, there will usually be some processing steps that refer to data from multiple input tables and thus can be performed only after our joins. For example, we will remove products with very few orders, a filter we couldn’t have performed pre-merge, since it depends on `product`, from `inv_store`, and `order`, from `order_ship`. exercise 4.26 will motivate this filtering step and exercise 4.27 will execute it.

**Exercise 4.26.**

- For each `product`, calculate the number of times that `order > 0`.
- Depict the distribution of this statistic with a histogram. Use the `binwidth = 1` option of `geom_histogram()` to plot each value individually.
  - The x-axis should extend a bit beyond 150.

The plot you made should highlight a critical issue: over a third of our products have no orders! Most of these products correspond to the cross-docked items (with `order_type == "CD"`) that we removed in exercise 4.21. But sometimes a particular `generation` value doesn’t have any orders. An example of this is `product == "1003_212576_36"`: there are orders for `item == 212576` at `store == 1003`, but none that coincide with `generation == 36`. Including products with very few orders will muddy our analysis, so we will keep a `product` only if a sufficient proportion of its rows report an order.
Exercise 4.27.

- Use `group_by()` and `summarise()` to calculate for each product in `master_panel` the fraction of observations in which `order > 0`.
- Select the `product` values for which this fraction exceeds 0.02.
  - Use the `na.rm = TRUE` option of `mean()`.
- `inner_join()` these distinct `product` values with `master_panel` to filter your `master_panel` accordingly.
  - Nest the query of the preceding two steps inside the `inner_join()` parentheses so that the filter can be performed without breaking the `%>%` pipe.
  - `master_panel` should have 8,663 distinct `product` values after this filter.

4.3.4 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 reproduce below. 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. ...
Figure 4.1: 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.

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 different as the tuning parameters in our data filters were different. For example, we limited our sample to products with at least 200 observations in exercise 4.24, whereas Bray et al. [2019b] limited theirs to products with at least 500 observations.
Exercise 4.28.

- 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`.
- `inner_join()` in the `category` tibble.
- Define logical variable `order_positive` that is TRUE if `order > 0`, FALSE if`order == 0`, and NA if `order` is NA.
- Define logical variable `order_observed` that is TRUE if and only if `order` is not 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_total` 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`.
- Use `pivot_longer()` to express the tibble in the "long format" that `ggplot()` expects. The new table should have 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 with `x = Inv_percent`, `y = value`, and `color = category`, and `facet_wrap()` by `name`.
- Label the plot as Bray et al. [2019b] did.

The plot you created should indicate both rationing and gaming: the DC should be less likely to fulfill orders and the stores more likely to place orders when the DC has less inventory.
4.4 Commentary

4.4.1 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. The current solution of compressing all moves into a single column makes the individual moves inaccessible. For example, there’s no easy way to derive from chess_panel the number of captured white pieces. An alternative would be 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 longest match has 349 moves, whereas the second-longest match has only 259 moves; thus this approach would introduce 90 columns, move_260–move_349, just to accommodate a single game. What’s more, the variable names wouldn’t be stable in this case, as the largest move number would increase as you update your sample with new games; and adding new column names would invariably break some aspect of your code. The third solution would be to express 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
> 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>,

This `unnest()` solution exploits the power of lists, which we will cover in the next chapter.
But this solution would add a lot of redundancy, needlessly repeating the `game-opening_ply` values for each move.

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
> 6 6 l1NXvwaE f6
> 7 7 l1NXvwaE dxe5
> 8 8 l1NXvwaE fxe5
> 9 9 l1NXvwaE fxe5
> 10 10 l1NXvwaE Nxe5
> # ... with 909,422 more rows

match_details <-
  chess_panel %>%
  select(-moves)

match_details

> # A tibble: 14,322 x 13

---

Another good solution is to save the chess moves as a list column within `chess_panel`. We'll discuss this approach in section 5.2.2.
> game rated turns victory_status winner increment_code white_id white_rating
> <chr> <lgl> <dbl> <chr> <chr> <chr> <chr> <dbl>
> 1 l1NXv~ TRUE 16 resign black 5+10 a-00 1322
> 2 mIICv~ TRUE 61 mate white 5+10 ischia 1496
> 3 kWKvr~ TRUE 61 mate white 20+0 danielmu~ 1439
> 4 9tXo1~ TRUE 95 mate white 30+3 nik2211~ 1523
> 5 qwU9r~ TRUE 33 resign white 10+0 capa_jr 1520
> 6 dwF3D~ TRUE 66 resign black 15+0 ehabfan~ 1439
> 7 afoMw~ TRUE 119 mate white 10+0 daniel~ 1381
> 8 HgKLW~ FALSE 39 mate white 20+60 daniel~ 1381
> 9 2fEjS~ FALSE 31 resign black 8+0 daniel~ 1381
> 10 u7i6d~ FALSE 31 mate white 15+15 shivang~ 1094
> # ... with 14,312 more rows, and 5 more variables: black_id <chr>,
> # black_rating <dbl>, opening_eco <chr>, opening_name <chr>,
> # opening_ply <dbl>

This solution has (i) easily accessible moves, (ii) no NA terms, and (iii) no data redundancies. 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:

```r
chess_moves <-
chess_moves %>%
mutate(
  player = ifelse(move_num %% 2, "w", "b"),
  capture = str_detect(moves, "x")
)
```

```r
# 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
```
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.

### 4.4.2 Lengthen to Avoid Multi-Column Operations

It’s often easier to avoid multi-column operations by lengthen the tibble. For example, the `c_across()` calls in our solution to exercise 4.12 are a bit awkward:

```r
first_200_moves %>%
  rowwise %>%
  filter(
    c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
      str_detect("\+") %>%
      sum(na.rm=TRUE) <
    c_across(ends_with(c("0", "2", "4", "6", "8"))) %>%
      str_detect("\+") %>%
      sum(na.rm=TRUE)
  )
```

However, if I pipe `first_200_moves` into `pivot_longer()` then I can derive the solution with more standard single-column operations:

```r
first_200_moves %>%
  pivot_longer(-game) %>%
  group_by(game) %>%
  mutate(player = ifelse(row_number() %% 2 == 0, "w", "b")) %>%
  filter(str_detect(value, "\+")) %>%
  summarise(dif = sum(player == "b") - sum(player == "w")) %>%
  filter(dif > 0) %>%
  select(game)
```

### 4.4.3 Partition Code with Joins

We could solve exercise 4.16 without a join:
chess_panel %>%
  add_count(m_1 = word(moves, 1)) %>%
  filter(n != min(n)) %>%
  select(-m_1, -n)

And this join-free solution is probably preferable. But now let’s embed the filter in a larger assembly line:

#joined solution
chess_panel %>%
  fun_1 %>%
  fun_2 %>%
  fun_3 %>%
  anti_join(#removes most uncommon opening
    add_count(. = word(moves, 1)) %>%
    filter(n == min(n)) %>%
    select(game)
  ) %>%
  fun_4 %>%
  fun_5 %>%
  fun_6

#join-free solution
chess_panel %>%
  fun_1 %>%
  fun_2 %>%
  fun_3 %>%
  add_count(m_1 = word(moves, 1)) %>%
  filter(n != min(n)) %>%
  select(-m_1, -n) %>%
  fun_4 %>%
  fun_5 %>%
  fun_6

In the first case the anti_join() groups the add_count(), filter() and select() operations into a distinct package, which we can mentally abstract into a discrete operation that “removes most uncommon opening.” In the latter case, it’s not as easy to see that the add_count(), filter(), and select() operations work together, since they meld into the other functions. And a comment like #removes most uncommon opening wouldn’t help the join-free solution, because it wouldn’t be clear where the job starts and ends.

In this fashion, joins can be a good way to divide an assembly line into distinct chunks without breaking the pipe, like in the following:
input_data %>%
  inner_join( # do x
    fun_1(.) %>%
    fun_2 %>%
    fun_3
  ) %>%
inner_join( # do y
  fun_4(.) %>%
  fun_5 %>%
  fun_6
) %>%
inner_join( # do z
  fun_7(.) %>%
  fun_8 %>%
  fun_9
)

The `inner_join()` steps above help organize the code, dividing functions `fun_1()`–`fun_9()` into three distinct steps.

4.4.4 Reiterate Groupings

Our solution to exercise 4.15 includes a redundant `group_by(opening_name)` step inside the `left_join()`. This `group_by()` operation is superfluous because the prior `group_by()` groupings 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` values 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`.

271
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.

### 4.4.5 Network Analysis with Joins

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". The players with one degree of separation are those that played a match against "wall-of-shields"; the players with two degrees of separation are those that didn’t play against "wall-of-shields", but played against someone who did, etc.

**Exercise 4.29.** Apply `pivot_longer()` to `chess_panel` to derive a tibble called `game_player` that has two columns: `game` and `player`. This tibble should list the players associated with a given match. For example, there should be two observations with `game` = "yrAas0Kj": one with `player` = "jamboger" and the other with `player` = "schaaksmurf3".

**Exercise 4.30.** Use `inner_join()` to merge two copies of `game_player` by = "game". The result of this join should be a tibble with columns `game`, `player.x`, and `player.y`. Save the distinct (`player.x`, `player.y`) pairs of this table as a tibble called `player_player`. This tibble lists all the players that are “connected” to a given player. Two players are “connected” if there’s zero or one degree of separation between them (by definition a player has zero degrees of separation with themselves). For example, the players that are connected to "mirco25" are "mirco25", "daniel_likes_chess", and "userx100". Thus, there should be three observations with `player.x` = "mirco25" and three observations with `player.y` = "mirco25".

**Exercise 4.31.** `player_player` 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.

- `inner_join()` `player_player` with itself, by = c("player.y" = "player.x").
  - The result should be a tibble with variables `player.x`, `player.y`, and `player.y.y`. The `player.y` column lists the players that are within one degree of separation from the given `player.x`. And the `player.y.y` column lists the players that are within one degree of separation from the given `player.y`, and thus within two degrees of separation from the given `player.x`.

- Select the `distinct()` (player.x, player.y.y) pairs.
- `rename()` `player.y.y` to `player.y`.
- Save the result as `player_player_degree_2`.
  - `player_player` and `player_player_degree_2` have the same form, except the latter has nearly 10 times as many observations (there are way more two-degree-of-separation connections than one-degree-of-separation connections).
**Exercise 4.32.** Create a tibble called `player_player_degree_3` that lists all player relationships with zero, one, two, or three degrees of separation. This tibble should have columns `player.x` and `player.y`. Construct `player_player_degree_3` from `player_player` rather than from `player_player_degree_2`.

- Repeat exercise 4.31’s `inner_join()`, `distinct()`, and `rename()` operations twice. The first time translates `player_player` to `player_player_degree_2` and the second time translates `player_player_degree_2` to `player_player_degree_3`.
- `player_player_degree_3` should have 1,734,276 rows.

**Exercise 4.33.** It should now be clear that we can get all the relationships that are within $n$ degrees of separation by executing exercise 4.31’s `inner_join()`, `distinct()`, and `rename()` operations $n - 1$ times. However, rather than copy-and-paste these three functions over and over again, compress the functional assembly line that comprises them into a stand-alone function called `increase_separation()`.

- Use the `. <- %>%` convention from section 2.2.4.
- `player_player %>% increase_separation` should yield `player_player_degree_2` and `player_player %>% increase_separation %>% increase_separation` should yield `player_player_degree_3`.

**Exercise 4.34.** Use `increase_separation()` to find the players that are within six degrees of separation from "wall-of-shields".

- It’s a lot faster to impose the `player.x == "wall-of-shields"` condition *before* applying the `increase_separation()` steps.
- You should get 656 players.

**Exercise 4.35.** 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.

- Do not invoke `increase_separation()` more than five times.
- R interprets `x %>>% anti_join(fun(.), .)` as `anti_join(fun(x), x)`.

### 4.5 Solutions

4.1
chess_panel <-
  chess_tib %>%
  filter(
    turns >= 8,
    abs(white_rating - black_rating) <= 250
  )

4.2

chess_panel %>%
  select(where(is.numeric))

4.3

first_200_moves %>%
  select(
    move_20:move_60 &
    ends_with(c("0", "2", "4", "6", "8"))
  )

4.4

first_200_moves %>%
  select(
    where(
      ~ .x %>%
        str_detect("=") %>%
        sum(na.rm=TRUE) %>%
        [. >= 10}
    )
  )

4.5

first_200_moves %>%
  mutate(
    across(
      -game,
\begin{verbatim}
~.x %>%
  str_replace("#", "_checkmate") %>%
  str_replace("\+", "_check")

chess_panel <-
  chess_panel %>%
  mutate(
    across(
      is.character &
      where(~.x %>%
        n_distinct %>%
        ~ . <= 12
      ),
      as.factor
    )
  )

chess_panel <-
  chess_panel %>%
  mutate(
    across(
      is.numeric &
      where(~.x %>%
        ~ . == as.integer(.) %>%
        all(na.rm = TRUE)
      ),
      as.integer
    )
  )
\end{verbatim}
first_200_moves %>%
mutate(
  across(
    ends_with(c("1", "3", "5", "7", "9")),
    ~.x %>%
      str_replace(
        pattern = "0-0-0",
        replacement = "Kc1"
      ) %>%
      str_replace(
        pattern = "0-0",
        replacement = "Kg1"
      )
  ),
  across(
    ends_with(c("0", "2", "4", "6", "8")),
    ~.x %>%
      str_replace(
        pattern = "0-0-0",
        replacement = "Kc8"
      ) %>%
      str_replace(
        pattern = "0-0",
        replacement = "Kg8"
      )
  )
)
)

4.9

pawn_moves <-
  first_200_moves %>%
  filter(is.na(move_200)) %>%
  group_by(game) %>%
  mutate(
    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("8", "2", "4", "6", "8"))) %>%
  str_detect("^[RNBQK0]",
    negate=TRUE
  ) %>%
  sum(na.rm = TRUE)
)
%>
relocate(
c(white_pawn_moves, black_pawn_moves),
  .after = "game"
)

4.10

first_200_moves %>%
  rowwise %>%
  filter(
    c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
      str_detect("N.#") %>%
      any
  )

4.11

first_200_moves %>%
  rowwise %>%
  filter(
    c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
      str_detect("^B", negate=TRUE) %>%
      all(na.rm=TRUE),
    c_across(ends_with(c("8", "2", "4", "6", "8"))) %>%
      str_detect("^R", negate=TRUE) %>%
      all(na.rm=TRUE)
  )

4.12
first_200_moves %>%
  rowwise %>%
  filter(
    c_across(ends_with(c("1", "3", "5", "7", "9"))) %>%
    str_detect("\+\") %>%
    sum(na.rm=TRUE)
    <
    c_across(ends_with(c("6", "2", "4", "6", "8"))) %>%
    str_detect("\+\") %>%
    sum(na.rm=TRUE)
  )

4.13

first_200_moves %>%
  rowwise %>%
  filter(
    c_across(move_1:move_12) %>%
    na.omit %>%
    {n_distinct(.) < length(.)}
  )

4.14

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")
  )

4.15
chess_panel %>%
  group_by(opening_name) %>%
  filter(n() > 30) %>%
  mutate(captured = str_count(moves, "x")) %>%
  left_join(
    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()
4.16

chess_panel %>%
  anti_join(
    add_count(.,. m_1 = word(moves, 1)) %>%
    filter(n == min(n)) %>%
    select(game)
  )

4.17 Option 2 is correct. For example, consider game = "0XPFTEt". 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.

4.18

chess_panel %>%
inner_join(
  filter(
    
    white_id == "wall-of-shields" |
    black_id == "wall-of-shields"
  ) %>%
  distinct(opening_name)
)
Chapter 5

Lists and Linear Models

5.1 Introduction

In this chapter we’ll learn about one last data structure: the list. More general than vectors and tibbles, lists give you tremendous flexibility in how you store your data. For example, you can use lists to give your data a hierarchical “file structure,” since lists can store lists that store lists (much like how folders on your hard drive can store folders that store folders). Moreover, since tibbles can store lists and lists can store tibbles, we can use lists to create tibbles that store other tibbles. In other words, we can compress an entire tibble’s worth of data into a single tibble cell. These nested tibbles are a powerful and elegant way to store data.

I’ll begin the lecture, in section 5.2.1, with a brief primer on lists. Most tibble-manipulation functions have a list analog, which I’ll introduce you to. These list-manipulation functions come from the purrr package, which gets loaded when you call `library(tidyverse)`. Unfortunately, these purrr functions aren’t as developed as the rest of the tidyverse, and their notation can be a bit awkward. But it’s not too bad.

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()` and `pivot_longer()`, except rather than create and destroy columns they create and destroy nested tibbles.

Section 5.2.3 is a mini-lab. 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, section 5.2.5 illustrates how we can use lists to streamline the execution of regressions. Specifically, it uses 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’ll conclude with two points in section 5.4.2. In section 5.4.1 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 much slower. Hence, it’s often faster to convert a list to a vector (or tibble) before subjecting it to a function. In section 5.4.2 I will discuss an important fact about lists. But this fact can be misleading, so I won’t tell it to you until you have a better understanding of lists.

5.2 Lecture

5.2.1 Lists

So far we’ve stored all our data in vectors and tibbles. But R has a third data structure: the 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 Mansfield Park:

```r
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"
```

284
We couldn’t combine two character strings, a date, and a number in a vector, but we can in a list. A list can contain any R object—including other lists. For example, the following defines a list of lists:

```r
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"
      ),
    novel_3 = mansfield
  )

jane
```

```r
> $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
```
Specifically, list `jane` has three elements: the list `novel_1`, which describes *Sense & Sensibility*, the list `novel_2`, which describes *Pride & Prejudice*, and the list `novel_3`, which describes *Mansfield Park*.

Most primary tibble functions have a list analog. For example, the list version of `glimpse()` is `str()`:

```r
jane %>% str
```

```r
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
```
The list version of `select()` is `pluck()`:

```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"
```

```r
# get first element
jane %>% pluck(1)

> $title
> [1] "Sense & Sensibility"

> $pub_date
> [1] "1811-02-01"

> $word_count
> [1] 119394

> $protagonist
> [1] "elinor dashwood"
```
Unfortunately, \texttt{pluck()}, unlike \texttt{select()}, cannot return multiple attributes.

The list version of \texttt{mutate()} is \texttt{listmodify()}:

\begin{verbatim}
#Example 1: add a new 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"
> $ 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"

#Example 2: Change protagonist of first novel
jane %>%
list_modify(

\end{verbatim}
The first example is fairly straightforward: it adds novel_4 as a fourth element of jane. But the second example is more complex. It replaces the first novel's protagonist from "elinor dashwood" to "marianne dashwood"—i.e., it replaces novel_1 with list_modify(novel_1, protagonist = "marianne dashwood"). However, to refer to novel_1 within the list_modify() call we must write pluck(., "novel_1"), which R interprets as pluck(jane, "novel_1"), since the . symbol always refers to the data that got piped into the encompassing function.

The list version of filter() is either keep() or discard(), depending on whether you want to retain or drop the elements that meet the given condition.

```r
novel_1 =
    pluck(., "novel_1") %>%
    list_modify(
        protagonist = "marianne dashwood"
    )
)
) %>%
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 "marianne 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"

#remove books published after 1813
jane %>%
discard(
    ~ .x %>%
    pluck("pub_date") %>%
    year %>%
    {. > 1813}
)
%>%
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"

#keep only the 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"

Note, that we use the -.x %>% convention, from section 4.2.2, to embed functional assembly lines inside our discard() and keep() calls. R feeds every element of jane into these assembly lines to determine which to keep() and discard(). For example, the keep() call retains the Sense & Sensibility list because novel_1 %>% pluck("title") %>% str_detect("&")
= TRUE, but it drops the *Mansfield Park* list because novel_3 %>% pluck("title") %>% str_detect("&") = FALSE.

The rough analog of the tibble pivoting functions is `transpose()`. For example, we can use this function to reshape `jane` from three length-four lists to four length-three lists:

```
jane %>%
  transpose %>%
  str
```

> List of 4
> $ title :List of 3
>   ..$ novel_1: chr "Sense & Sensibility"
>   ..$ novel_2: chr "Pride & Prejudice"
>   ..$ novel_3: chr "Mansfield Park"
> $ pub_date :List of 3
>   ..$ novel_1: Date[1:1], format: "1811-02-01"
>   ..$ novel_2: Date[1:1], format: "1813-01-28"
>   ..$ novel_3: Date[1:1], format: "1814-07-01"
> $ word_count :List of 3
>   ..$ novel_1: num 119394
>   ..$ novel_2: num 122189
>   ..$ novel_3: num 159381
> $ protagonist:List of 3
>   ..$ novel_1: chr "elinor dashwood"
>   ..$ novel_2: chr "elizabeth bennet"
>   ..$ novel_3: chr "fanny price"

Like `pivot_longer()` and `pivot_wider()`, `transpose()` rearranges the structure of our data. Specifically, it flips the nesting order of a list of lists, so that, e.g., `jane %>% transpose` organizes the data by attribute, whereas `jane` organizes the data by book. Here’s another example:

```
mini_list_of_lists <-
  list(
    a = list(x = 1, y = c("a", "b"), z = today()),
    b = list(x = "X", y = c(FALSE, TRUE, TRUE))
  )

mini_list_of_lists %>% str
```

> List of 2
$ a: List of 3
  $ x: num 1
  $ y: chr [1:2] "a" "b"
  $ z: Date[1:1], format: "2021-04-22"

$ b: List of 2
  $ x: chr "X"
  $ y: logi [1:3] FALSE TRUE TRUE

mini_list_of_lists %>%
  transpose %>%
  str

List of 3
  $ x: List of 2
  $ a: num 1
  $ b: chr "X"
  $ y: List of 2
  $ a: chr [1:2] "a" "b"
  $ b: logi [1:3] FALSE TRUE TRUE
  $ z: List of 2
  $ a: Date[1:1], format: "2021-04-22"
  $ b: NULL

The list version of summarise() is map(). But whereas summarise() applies a function for each group_by() group, map() applies a function for each element in a list:

# get day of the week each book was published:
jane %>%
  map(~
  .x %>%
  pluck("pub_date") %>%
  wday(label = TRUE)
)

$novel_1
[1] Fri
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat

$novel_2
[1] Thu
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
We again use the `~.x %>%` convention to define the operation we want to apply. But if this operation has a stand-alone function then we can use that instead:

```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>
>  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>
>  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>
>  Mansfield Park 1814-07-01     159381       fanny price
```

The rough analog of the tibble joining functions is `map2()`. Specifically, `map2()` takes two lists of equal length and combines their elements in a general way with a functional assembly line. For example, the following adds a `first_sentence` attribute to `jane`:

```
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 = " "
```

293
jane %>%
map2(
  list(s_1, s_2, NULL),
  ~.x %>% list_modify(first_sentence = .y)
)

> $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"
> $novel_1$first_sentence
> [1] "The family of Dashwood had long been settled in Sussex."
>
> $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"
> $novel_2$first_sentence
> [1] "It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife."
>
> $novel_3
> $novel_3$title
The `map2()` function receives the left-hand list (LHL) `jane` and the right-hand list (RHL) `list(s_1, s_2, NULL)`. Note, 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 

```r
~.x %>% list_modify(first_sentence = .y)
```

where `.x` is a place-holder of 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)
```

And the third element that `map2()` returns is

```
> [1] "The family of Dashwood had long been settled in Sussex."
```
Note, the `NULL` tells R not to bother giving this book a `first_sentence` attribute. (I would have used `NA`, rather than `NULL`, had I wanted a record of the missing variable.)

Note `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
#Example 1: Paste together strings
list("First Name", "Last Name") %>%
  map2(  
    list("Rob", "Bray"),
    ~.x %>% str_c(.y, sep = " ")
  )
```

> [[[1]]]  
> [1] "First Name: Rob"

> [[[2]]]  
> [1] "Last Name: Bray"

---

1In 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.
# Example 2: 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(. , 1) else pluck(. , 2)
)

> [[1]]
> [1] "earth is round"
> >
> [[2]]
> [1] "water is dry"
> >
> [[3]]
> [1] "2 + 2 = 4"

# Example 3: 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)`.
In addition to `map()` and `map2()`, there are also `map_df()` and `map2_df()`, which are the same except they output tibbles (df stands for “data frame”):

```
# Previous example, with tibble output
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  150 1501500

# Days since publication of Jane's novels
jane %>%
  map_df(
    ~.x %>%
      pluck("pub_date") %>%
      {today() - .}
  )

> # A tibble: 1 x 3
>   novel_1    novel_2   novel_3
>   <drtm>    <drtm>    <drtm>
>  1 76782 days 76055 days 75536 days

# Convert Jane to tibble
jane %>%
  map_df(~.x)
```

> # A tibble: 3 x 4
>   title                  pub_date          word_count   protagonist
>   <chr>                  <date>             <dbl>         <chr>
>  1 Sense & Sensibility  1811-02-01 119394 elinor dashwood
>  2 Pride & Prejudice    1813-01-28 122189 elizabeth bennet
>  3 Mansfield Park       1814-07-01 159381 fanny price

298
Not attaching a conveyor belt to ~.x, in the last example, tells R to return each element of jane as it is, without passing it through a functional assembly. Hence, the code above converts novel_1, novel_2, and novel_3 into the rows of a tibble.

However, not all lists are so easily tibblized. For example, suppose we replaced protagonist with a list of characters:

```r
jane <-
  list(
    novel_1 =
      list(
        title = "Sense & Sensibility",
        pub_date = "1 February 1811" %>% dmy,
        word_count = 119394,
        characters =
          c("elinor dashwood", "marianne dashwood")
      ),
    novel_2 =
      list(
        title = "Pride & Prejudice",
        pub_date = "28 January 1813" %>% dmy,
        word_count = 122189,
        characters =
          c("elizabeth bennet", "mr. darcy")
      ),
    novel_3 =
      list(
        title = "Mansfield Park",
        pub_date = "1 July 1814" %>% dmy,
        word_count = 159381,
        characters =
          c("fanny price", "lady bertram", "mrs. norris")
      )
  )

jane %>% 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
>   ..$ characters: chr [1:2] "elinor dashwood" "marianne dashwood"
> $ novel_2:List of 4
>   ..$ title : chr "Pride & Prejudice"
>   ..$ pub_date : Date[1:1], format: "1811-02-01"
>   ..$ word_count: num 122189
>   ..$ characters: chr [1:2] "elizabeth bennet" "mr. darcy"
> $ novel_3:List of 4
>   ..$ title : chr "Mansfield Park"
>   ..$ pub_date : Date[1:1], format: "1811-02-01"
>   ..$ word_count: num 159381
>   ..$ characters: chr [1:3] "fanny price" "lady bertram" "mrs. norris"
Now the previous tibbling scheme wouldn’t work because we can’t stuff an entire vector into an individual tibble cell. For example, we can’t replace "elizabeth bennet" with c("elizabeth bennet", "mr. darcy") in the fourth cell of the second row.

Of course we could tibblize these data by compressing the character string vectors into individual strings—e.g., by replacing c("elizabeth bennet", "mr. darcy") with "elizabeth bennet, mr. darcy"—or by saving these vectors in a separate tibble, as we did with moves in section 4.4.1. But my favorite way to tibblize these data is to save the character vectors in a list column:

```
jane_tib <-
jane %>%
map_df(
  ~.x %>%
  list_modify(
    characters = list(pluck(., "characters"))
  )
)
jane_tib
```

```
> # A tibble: 3 x 4
> title       pub_date      word_count characters
> <chr>        <date>        <dbl>     <list>
> 1 Sense & Sensibility 1811-02-01 119394 <chr [2]>
> 2 Pride & Prejudice 1813-01-28 122189 <chr [2]>
> 3 Mansfield Park 1814-07-01 159381 <chr [3]>
```

jane_tib is a standard tibble, but it’s fourth column is a list rather than a vector. Indeed, tibbles columns can be either vectors or lists. And whereas a given vector element can only store only one text string, a given list element can store an entire vector of text strings. For example, the second element of the character column houses the following vector:

```
2Note, pluck() is compatible with tibbles. In fact, all list operations are compatible with tibbles. See section 5.4.2.
```
```
jane_tib %>%
  pluck("characters", 2)
```

```r
> [1] "elizabeth bennet" "mr. darcy"
```

**Exercise 5.1.** We will now capitalize the character names in `jane_tib`.

- Pipe `jane_tib` into `mutate()`.
- Try setting `characters = characters %>% str_to_title` within this `mutate()` step. This will yield an error because `str_to_title()` is compatible only with character string vectors, not lists.
- Within the `mutate()` call, use `map()` to apply `str_to_title()` to each element of the `characters` list.
- Overwrite `jane_tib` with the resulting tibble.

**Exercise 5.2.** We will now add "Edward Ferrars" and "John Willoughby" to the *Sense & Sensibility* character list and "Sir Thomas Bertram" to the *Mansfield Park* character list.

- Pipe `jane_tib` into `mutate()`.
- Within the `mutate()` call, use `map2()` to append the new names to the old names.
  - `list(c("a", "b"), c("x", "y")) %>% map2(list(c("C", "D"), NULL), ~.x %>% c(.y))`
- Overwrite `jane_tib` with the resulting tibble.

### 5.2.2 Nested Tibbles

If tibbles can store lists and lists can store anything then tibbles can store anything. Most notably this means that tibbles can store other tibbles. For example:
In this case, the B column is a list that contains three tibbles.

Exercise 5.3. Use `mutate()` and `map_int()` to add a column to `nested_tib` that specifies the `nrow()` of the corresponding tibble in the B column.

- `map_int()` is like `map()`, except it returns a vector of integers rather than a list.

We will use nested tibbles to add the full text of *Sense & Sensibility*, *Pride & Prejudice*, and *Mansfield Park* to `jane_tib`. To get the text of these books, run the following code:

```r
install.packages("janeaustenr")
library(janeaustenr)
full_text <-
austen_books() %>%
group_by(book) %>%
mutate(
  chapter =
  text %>%
  str_detect("^CHAPTER|^Chapter") %>%
  cumsum
) %>%
ungroup %>%
filter(
  chapter >= 1,
  text != ""
)
```

This code should define `full_text` as a tibble with three variables: `text`, `chapter`, and `book`, the last of which takes values "Sense & Sensibility", "Pride & Prejudice", "Mansfield Park", "Emma", "Northanger Abbey", or "Persuasion". Here’s a representative four rows:
full_text %>% slice(50000:50003)

> # A tibble: 4 x 3
> text book chapter
t> <chr> <fct> <int>
> 1 both brother and sister. Could she have foreseen such a~ Northanger A~ 9
> 2 nothing should have persuaded her to go out with the ot~ Northanger A~ 9
> 3 it was, she could only lament her ill luck, and think o~ Northanger A~ 9
> 4 lost, till it was clear to her that the drive had by no~ Northanger A~ 9

We can covert this standard tibble into a set of nesting tibbles with nest():

chapter_level_tib <-
  full_text %>%
  nest(text_tib = text)

chapter_level_tib %>%
  head(4)

> # A tibble: 4 x 3
> book chapter text_tib
> <fct> <int> <list>
> 1 Sense & Sensibility 1 <tibble [136 x 1]>
> 2 Sense & Sensibility 2 <tibble [170 x 1]>
> 3 Sense & Sensibility 3 <tibble [138 x 1]>
> 4 Sense & Sensibility 4 <tibble [170 x 1]>

This operation rolls up the text values by book and chapter into a set of nested tibbles that are stored in the text_tib list column. For example, the third element of the text_tib column is a tibble that looks like this:

chapter_level_tib %>%
  pluck("text_tib", 3) %>%
  head(4)

> # A tibble: 4 x 1
> text
> <chr>
> 1 CHAPTER 3
> 2 Mrs. Dashwood remained at Norland several months; not from any
> 3 disinclination to move when the sight of every well known spot ceased
> 4 to raise the violent emotion which it produced for a while; for when
And we can roll up the book chapters into a second layer of nested tibbles with a second `nest()` operation:

```r
book_level_tib <-
  full_text %>%
  nest(text_tib = text) %>%
  nest(chapter_tib = c(chapter, text_tib))

book_level_tib
```

<table>
<thead>
<tr>
<th>book</th>
<th>chapter_tib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sense &amp; Sensibility</td>
<td>&lt;tibble [50 x 2]&gt;</td>
</tr>
<tr>
<td>Pride &amp; Prejudice</td>
<td>&lt;tibble [61 x 2]&gt;</td>
</tr>
<tr>
<td>Mansfield Park</td>
<td>&lt;tibble [48 x 2]&gt;</td>
</tr>
<tr>
<td>Emma</td>
<td>&lt;tibble [55 x 2]&gt;</td>
</tr>
<tr>
<td>Northanger Abbey</td>
<td>&lt;tibble [31 x 2]&gt;</td>
</tr>
<tr>
<td>Persuasion</td>
<td>&lt;tibble [24 x 2]&gt;</td>
</tr>
</tbody>
</table>

The `chapter_tib = c(chapter, text_tib)` term in the second `nest()` call tells R to bundle the chapter and text_tib columns into a set of tibbles nested within the `chapter_tib` list column. Hence, our data now has a three-tier hierarchy, with tibbles nested within tibbles nested within a tibble. For example, the second element of the `chapter_tib` column is a tibble that looks like this:

```r
book_level_tib %>%
  pluck("chapter_tib", 2) %>%
  head(4)
```

<table>
<thead>
<tr>
<th>chapter</th>
<th>text_tib</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;tibble [79 x 1]&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;tibble [77 x 1]&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;tibble [146 x 1]&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;tibble [96 x 1]&gt;</td>
</tr>
</tbody>
</table>

And, as before, the `text_tib` column stores a list of tibbles.

We’ll add the text of the novels to `jane_tib` in the following exercise.
Exercise 5.4. Use `left_join()` to copy the `chapter_tib` column from `book_level_tib` to `jane_tib`.

- Merge the tibbles by = c("title" = "book").

Exercise 5.5. Select the text of the fifth chapter of the third book listed in `jane_tib`. (We added this text to `jane_tib` in exercise 5.4.)

- You should be able to do this with a single `pluck()` call.

Exercise 5.6. What’s the difference between `full_text %>% nest(text_tib = text) %>% nest(chapter_tib = c(chapter, text_tib))` and `full_text %>% nest(chapter_tib = c(chapter, text))`?

As we used `nest()` to furled up the chapter and `text_tib` columns, we can use `unnest()` to unfurl them:

```r
ejane_tib %>%
  unnest(chapter_tib) %>%
  head(3)
```

<table>
<thead>
<tr>
<th>title</th>
<th>publish_date</th>
<th>word_count</th>
<th>characters</th>
<th>chapter</th>
<th>text_tib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sense &amp; Sensibility</td>
<td>1811-02-01</td>
<td>119394</td>
<td>&lt;chr [4]&gt;</td>
<td>1</td>
<td>&lt;tibble [136 x 1~]</td>
</tr>
<tr>
<td>Sense &amp; Sensibility</td>
<td>1811-02-01</td>
<td>119394</td>
<td>&lt;chr [4]&gt;</td>
<td>2</td>
<td>&lt;tibble [170 x 1~]</td>
</tr>
<tr>
<td>Sense &amp; Sensibility</td>
<td>1811-02-01</td>
<td>119394</td>
<td>&lt;chr [4]&gt;</td>
<td>3</td>
<td>&lt;tibble [138 x 1~]</td>
</tr>
</tbody>
</table>

And with a second `unnest()` we can unfurl the `text_tib` column:

```r
ejane_tib %>%
  unnest(chapter_tib) %>%
  unnest(text_tib) %>%
  head(3)
```

<table>
<thead>
<tr>
<th>title</th>
<th>publish_date</th>
<th>word_count</th>
<th>characters</th>
<th>chapter</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sense &amp; S-</td>
<td>1811-02-01</td>
<td>119394</td>
<td>&lt;chr [4]&gt;</td>
<td>1</td>
<td>CHAPTER 1</td>
</tr>
<tr>
<td>Sense &amp; S-</td>
<td>1811-02-01</td>
<td>119394</td>
<td>&lt;chr [4]&gt;</td>
<td>2</td>
<td>The family of Dashwood ha~</td>
</tr>
<tr>
<td>Sense &amp; S-</td>
<td>1811-02-01</td>
<td>119394</td>
<td>&lt;chr [4]&gt;</td>
<td>3</td>
<td>1 was large, and their resi~</td>
</tr>
</tbody>
</table>
And we could likewise unfurl the characters column:

```r
jane_tib %>%
  unnest(cols = characters) %>%
  head
```

```r
# A tibble: 6 x 5
title       pub_date    word_count characters          chapter_tib
<chr>        <date>      <dbl>     <list>                     <list>
1 Sense & Sensibility 1811-02-01 119394 <tibble [50 x 2]>     
2 Sense & Sensibility 1811-02-01 119394 <tibble [50 x 2]>     
3 Sense & Sensibility 1811-02-01 119394 <tibble [50 x 2]>     
4 Sense & Sensibility 1811-02-01 119394 <tibble [50 x 2]>     
5 Pride & Prejudice   1813-01-28 122189 <tibble [61 x 2]>     
6 Pride & Prejudice   1813-01-28 122189 <tibble [61 x 2]>     
```

So `nest()` is like `pivot_wider()`, except rather splay out the data in new columns it squirrels them away in a nested tibble. And `unnest()` is analogously like `pivot_longer()`.

**Exercise 5.7.** The `text_tib` tibbles nested within `jane_tib` look like this:

```r
jane_tib %>%
  pluck("chapter_tib", 1, "text_tib", 7)
```

```r
# A tibble: 113 x 1
text
<chr>
1 CHAPTER 7
2 Barton Park was about half a mile from the cottage. The ladies had
3 passed near it in their way along the valley, but it was screened from
4 their view at home by the projection of a hill. The house was large
5 and handsome; and the Middletons lived in a style of equal hospitality
6 and elegance. The former was for Sir John's gratification, the latter
7 for that of his lady. They were scarcely ever without some friends
8 staying with them in the house, and they kept more company of every
9 kind than any other family in the neighbourhood. It was necessary to
10 the happiness of both; for however dissimilar in temper and outward
# ... with 103 more rows
```

But these `text` values are awkward, being neither sentences nor words. We'll now change how we store this text, saving every word separately. For example, we'll transform the tibble above into a vector that reads `c("CHAPTER", "7", "Barton", "Park", "was", "about", "half", ...)`
• Use `unnest()` to unfurl the chapter_tib and text_tib nested tibbles.
• `group_by()` all variables besides `text`.
  - Use `across()`.
• Pipe the result into `summarise()` and use `str_c()` to paste the text character string vector corresponding to the given `group_by()` group into a very long character string. After this step the `text` column should be a character string vector that stores the text of the corresponding chapters. For example, the `title = "Sense & Sensibility", chapter = 45` row should have `text = "CHAPTER 45 Elinor, for some time after he left her ..."`.
  - `c("paste", "me together") %>% str_c(collapse = "") = "paste me together".
• Pipe result into `mutate()`.
• Within the `mutate()` call use `str_split(pattern = " ")` to divide each text character string into a vector of individual words. After this step the `text` column should be a list of character string vectors. For example, the `title = "Sense & Sensibility", chapter = 45` row should have `text = c("CHAPTER", "45", "Elinor", "for", "some", "time", "after", "he", "left", "her", ...)`
  - `c("please split", "this text up") %>% str_split(pattern = " ") = list(c("please", "split"), c("this", "text", "up"))`.
• Use `nest()` to furl the chapter and text columns into a set of tibbles nested within the chapter_tib list column.

### 5.2.3 Case Study: Wine Reviews

We will now use our list manipulating skills to study the wine reviews provided in `winemag-data-130k-v2.json` (see exercise 1.20 of section 1.3). Specifically, we will test whether we can predict the final score assigned by the reviewer from the number of positive-connotation words and negative-connotation words in the text of their review. To load our sample of win reviews, run the following:

```r
install.packages("RJSONIO")
library(RJSONIO)
wine_list <- fromJSON("winemag-data-130k-v2.json")
```

This code defines `wine_list` as a list of lists that describe 129,971 wine reviews from Wine Enthusiast Magazine. For example, the 100th element `wine_list` is the following list:

---

We won’t be using the data in `winemag-data_first150k.csv` or `winemag-data-130k-v2.csv`. 

---
Now if you google the title of this review, you will find that these data were scraped from
this website\textsuperscript{4}. In fact, every record in wine_list was scraped from a page like this on the (Wine Enthusiast website)[www.winemag.com]. 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.

We will use text analytics to relate the body of the review, in description, with the score of the review, in points. (This score nominally runs from 0 to 100, but there's serious grade inflation, as no wine received fewer than 80 points.) But first we must prepare our sample for analysis.

**Exercise 5.8.** A key variable that we don't currently have easy access to is the wine's vintage—i.e., the year that the grapes were picked. The vintage, however, is always listed in the title of the review. For example, the 12th review has title = "Leon Beyer 2012 Gewurztraminer (Alsace)", which indicates that this wine's vintage is 2012. Accordingly, we will now extract the year from each title to create a new vintage variable.

- Pipe wine_list into map().
- Within the map() call, write \~.x %>% list_modify().
- Within list_modify() define vintage as an integer that equals the first four-digit number that appears in the title of the review if this number is between 1985 and 2017, and define vintage = NA otherwise. For example, set vintage = 2016 when title = "Red Newt Cellars 2016 Circle Riesling (Finger Lakes)" and set vintage = NA when title = "Foxen 7200 2007 Vogelzang Vineyard Cabernet Sauvignon (Santa Ynez Valley)". (Extracting 2007 from this text string is more trouble than it's worth.)
  - Use pluck(.x, "title") to refer to the title within the list_modify() call.
- "\b[0-9]{4}\b" is a regular expression that stands for “a four-digit number book-ended by spaces” (which is how the vintages appear in the title). For example, "Kunde 2014 Reserve Zinfandel" %>% str_extract("\b[0-9]{4}\b") = "2014".
  - strtoi() turns character strings to integers.
- Save the updated listed of lists as wine_list.

**Exercise 5.9.** We will now convert wine_list to a standard tibble. However, we must remove the NULL values in this list of lists before we can tibblize it. For example, wine_list %>% pluck(1, "price") = NULL.

- Pipe wine_list into map_df().
- Within the map_df() call, use discard() and is_null() to remove the NULL elements of each nested list.

Pipe the resulting tibble into `distinct()` to remove its redundant rows. 
Use `rowid_to_column(var = "review_id")` to add an ID variable. 
Use `mutate()` and `strtoi()` to translate the points and price columns from character strings to integers. 
Save the result as `wine`. 
  - This tibble should have 119988 rows and 15 columns.

**Exercise 5.10.** We will now show that a review usually receives more points if it has a longer description. 

- Pipe `wine` into `ggplot()`
- Use `aes()` to set `x = str_length(description)` and set `y = points.`
  - `str_length(description)` is the number of characters in the `description` character string.
- Add a `geom_density_2d_filled()` layer.

The plot you made in exercise 5.10 illustrates a strong positive correlation between the length of the `description` and the number of allocated `points`. Specifically, this correlation is 

\[ \text{wine} \%>\% \text{summarise(cor(str_length(description), points))} = 0.562, \] 

which is surprisingly high.

But in addition to the length of a review, the *sentiment* of a review also matters. The following code defines `sentiment_dictionary` as a tibble that labels 6786 words with "+" or "-", depending on whether they have a positive or negative connotation:

```r
library(install.packages) 
library(tidytext)

sentiment_dictionary <-
  get_sentiments("bing") \%>\%
  mutate(
        sentiment = 
          fct_recode(
            sentiment,
            `\+' = "positive",
            `\-' = "negative"
          )
  )
```

For example, "zombie" is coded as "-" and "dazzling" as "+". The following exercises will `inner_join()` this list with the words in `description`. 

---

310
Exercise 5.11. We will now add three variables our sample: `word_count`, `pos_count`, and `neg_count`, which report the number of words, the number of "+"-coded words, and the number of "-"-coded words in `description`.

- Pipe `wine` into `mutate()`.
- Within the `mutate()` call, define

```r
word =
  description %>%
  str_replace_all("[^[:alnum:]]", " ") %>%
  str_squish %>%
  str_split(" ")
```

The `str_replace_all()` operation replaces all punctuation marks with spaces. The `str_squish()` step removes repeated spaces. And the `str_split(pattern = " ")` call divides each character string vector into a corresponding list of words. For example, the first row should have `word = list(c("Aromas", "include", "tropical", "fruit," ,...)).` *select() review_id and word and unnest by word. Save the result as `word_list`. *left_join() word_list with sentiment_dictionary. *group_by() review_id. *Use summarise() to define `word_count = n()`, `pos_count = sum(sentiment == "+", na.rm=TRUE)`, and `neg_count = sum(sentiment == "-", na.rm=TRUE). *Call the resulting tibble `word_stats`. *Define `wine_2 <- wine %>% left_join(word_stats)`.  

Exercise 5.12. We will now create a plot that illustrates that reviews with a higher `pos_count - neg_count` difference generally receive more points.

- Pipe `wine_2` into `mutate()`.
- Within `mutate()` define `score = pos_count - neg_count`, and then Winsorize\(^5\) this variable by setting `score = pmin(score, quantile(score, .995))` and `score = pmax(score, quantile(score, .005))`.
- The Winsorization caps the extreme `score` values at their 0.5% and 99.5% levels. This truncation makes the plot easier to read.
- Pipe the result into `ggplot()`.
- Add a `geom_violin()` layer with `x = points` and `y = as.factor(score)`.

The plot you made in the previous exercise illustrates that the number of positive-association words minus number of negative-association words is a strong predictor of `points`. Indeed, it’s striking how effectively the simple `pos_count - neg_count` metric distinguishes good wines from bad wines. For example, here’s the description of the four wines with `pos_count - neg_count <= -7`:

\(^5\)https://en.wikipedia.org/wiki/Winsorizing
With distinct aromas of field herbs, asphalt, gritty mineral, herbal berry and burnt toast, this is a challenging blend of Cabernet Sauvignon, Syrah, Carmenere and Petit Verdot. The palate hits hard with aggressive tannins and abrasive acidity, while flavors of baked black fruits and coconut announce loud oak. This finishes hard and gritty, with a mule’s kick.

This is an unusual white blend from Mallorca that uses the native Premsal grape along with 15% Muscat. It smells a little funky and sweaty, and while the feel is angular the palate isn’t bad. Tastes sort of fruitless, with a bit of green melon. O.K. for a weird wine, but it’s hollow and short on flavor.

Oak completely overwhelms this wine. All you can smell and taste is char, caramel and wood spice, and the tannins, which are partly from oak barrels, are dense and hard. It’s difficult even to discern any underlying fruit flavors, although if you try hard, you stumble across blackberries.

Muddled candied aromas smell uniquely unnatural and forced. This is choppy and jagged in the mouth, while a strange set of fake-oak and murky berry flavors are prominently on display.

These wines have 87, 84, 82, and 80 points, respectively. In contrast, here’s the description of the four wines with \( \text{pos}_\text{count} - \text{neg}_\text{count} \geq 14:

Peach and honeysuckle already have a notion of honeyed sweetness and ripeness on the nose. The palate boosts both fruit and floral notion to the top, where honeysuckle and rose petal notes meeting with peach on the medium-sweet palate. This is full on, joyous, juicy and rich with expressive, exuberant flavor. An underlying concentration almost feeds this with a fiery dynamic force. The finish is medium sweet but comes with a wonderfully redeeming pure lemon streak. This is ready now but will get even better with age.

A touch of freshly cut apple peel is all that appears on the shy nose. The palate is more approachable and opens into beautifully aromatic notions of ripe tart apples and blossom. This has a very harmonious balance and something like tender poise. This is absolutely fresh but it comes like a caress, touching and convincing with tenderness rather than flashy force. The palate is dry, has ample and aromatic apple fruit and immense charm. This is so easy to drink right now but will certainly evolve. It finishes with a long pure echo of lemon.

Peach and honey characterize the nose on this golden wine. These flavors also play generously on the concentrated rich midpalate where they are countered by aromatic orange and tangerine zestiness, creating a lovely contrast and exciting richness. There is just a touch of residual sweetness here to create some drama, but it works to great effect. The wine is not sweet enough for deserts but will work with Asian spices or ripe cheeses. It’s simply wonderful and so balanced.
A lovely, heady lift of lemon, peach and passion fruit dances on the nose. This rich sweet-fruited tropicality is played out fully on the medium-sweet palate where the passion fruit swirls tartly alongside sweet ripe generous peach. It is the passion-fruit aroma and tartness that gives this a real edge. This not only has sweetness but a real tropical thrill. It’s a very concentrated, pure-fruited wine with lots of promise.

These wines have 94, 94, 93, and 94 points, respectively.

5.2.4 Linear Regression

The plots we made in the previous section are suggestive, but they are not definitive because they do not control for anything. To incorporate control variables into our analysis we will use linear regression. We will run our regressions with the “linear model” function \( \text{lm}() \). Additionally, we will use the broom package—an offshoot of the tidyverse—to manipulate our regression outputs:

```r
library(broom)
```

Here’s a simple regression of points on word_count, pos_count, and neg_count:

```r
wine_model <-
wine_2 %>%
  \text{lm}(
    \text{points} ~ \text{word_count + pos_count + neg_count},
    \text{data = .}.
  )
wine_model %>%
tidy
```

> # A tibble: 4 x 5
>   term      estimate std.error statistic p.value
>   <chr>      <dbl>     <dbl>      <dbl>    <dbl>
> 1 (Intercept)  82.4   0.0283     2913.     0.
> 2 word_count  0.129   0.000703   183.       0.
> 3 pos_count  0.245   0.00435     56.3       0.
> 4 neg_count -0.0852   0.00832    -10.2 1.37e-24

The output of \( \text{lm}() \) is a weird kind of list called an “lm” object. The code above saves this list to \( \text{wine_model} \), and uses the \text{tidy()} function, from the broom package, to translate it to a standard tibble.
The second input into `lm()`, `data = .`, tells R to use `wine_2` as the regression data. By default, the `%>%` conveyor belt always feeds into a function’s first argument. But `lm()` expects to receive the regression data via its second argument. So we add `data = .` to pipe `wine_2` into this function’s second slot.

The first slot of `lm()` is reserved for the **regression formula**. For example, the `points ~ word_count + pos_count + neg_count` formula specifies the following statistical model:

\[
\text{points} = \beta_0 + \beta_1 \text{word_count} + \beta_2 \text{pos_count} + \beta_3 \text{neg_count} + \epsilon,
\]

where \(\beta_0, \beta_1, \beta_2, \text{ and } \beta_3\) are fixed coefficients and \(\epsilon\) is an unobserved error term that’s mean-zero and uncorrelated with `word_count`, `pos_count`, and `neg_count`. Note, I do not need to assume that \(\epsilon\) is normally distributed.

The `estimate` column of the output tibble provides our coefficient estimates: \(\hat{\beta}_0 = 82.4\), \(\hat{\beta}_1 = 0.129\), \(\hat{\beta}_2 = 0.245\), and \(\hat{\beta}_3 = -0.0852\). 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 \(\sigma_i\). This standard deviation is called a “standard error,” and it’s reported in the `std.error` column of our output tibble. Hence, the error \(\hat{\beta}_0 - \beta_0\) has standard deviation 0.0283, whereas the error \(\hat{\beta}_1 - \beta_1\) has standard deviation 0.000703. The latter estimate is thus more precise—i.e., it generally has a smaller error.

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 \(0.129 / 0.000703 = 183\) standard deviations between \(\hat{\beta}_1\) 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 random variable to be 183 standard deviations above its mean, we find that there would be essentially no chance of getting \(\hat{\beta}_1 \geq 0.129\) when \(\beta_1 = 0\) (the probability is so small that R rounds it down to `p.value = 0`). Hence, we can safely conclude that \(\beta_1 > 0\). We can likewise conclude that \(\beta_4 < 0\), because the probability of getting \(\hat{\beta}_4 \leq -0.0852\) when \(\beta_4 = 0\) is `p.value = 1.37e-24` (i.e., 0.00000000000000000000000137). In general, if `p.value < 0.05` then you can “reject the null hypothesis that the coefficient is zero at the \(p = 0.05\) level.”

Since \(\hat{\beta}_i\) is our best guess for \(\beta_i\), our best guess for `points`, given `word_count`, `pos_count`, and `neg_count`, is:

---

6Formulas, such as `points ~ word_count + pos_count + neg_count`, are a type of R object. We’ve already used formulas to define functional assembly lines. For example, any expression beginning with `~.x %>% pluck("pub_date") %>% (today() - .)` from section 5.2.1, is a formula. But the formulas we pass into `lm()` always have the form `dependent_var ~ independent_var_1 + ... + independent_var_n`.

7A 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.

8The "(Intercept)" term always corresponds to the additive constant \(\beta_0\). We don’t reference this term in our regression formula because R adds it automatically. However, you can remove this intercept by adding `-1` to your regression formula. For example, `points ~ word_count + pos_count + neg_count - 1` corresponds to the following statistical model: `points = \beta_1 \text{word_count} + \beta_2 \text{pos_count} + \beta_3 \text{neg_count} + \epsilon`.

---

314
𝛽0̂ +𝛽1̂ word_count + 𝛽2̂ pos_count + 𝛽3̂ neg_count
=82.4 + 0.129 ⋅ word_count + 0.245 ⋅ pos_count − 0.0852 ⋅ neg_count
This expression is called the “fitted value” of our regression. For example, the review with
title = "Marchesi Antinori 2015 Villa Antinori White (Toscana)" has word_count = 38,
pos_count = 1, and neg_count = 0, so our model suggests that its score should be around
82.4 + 0.129 * 38 + 0.245 * 1 - 0.0852 * 0 = 87.55, 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
wine_model into broom’s augment() function:

wine_model %>%
augment(data = wine_2) %>%
glimpse

> Rows: 119,988
> Columns: 24
> $ review_id

<int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1~

> $ points

<int> 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, 87, ~

> $ title

<chr> "Nicosia 2013 Vulk\xe0 Bianco

> $ description

<chr> "Aromas include tropical fruit, broom, brimstone~

> $ taster_name

<chr> "Kerin O\031Keefe", "Roger Voss", "Paul Gregutt"~

(Etna)", "Quinta ~

> $ taster_twitter_handle <chr> "@kerinokeefe", "@vossroger", "@paulgwine\xa0", ~
> $ designation

<chr> "Vulk\xe0 Bianco", "Avidagos", NA, "Reserve Late~

> $ variety

<chr> "White Blend", "Portuguese Red", "Pinot Gris", "~

> $ region_1

<chr> "Etna", NA, "Willamette Valley", "Lake Michigan ~

> $ province

<chr> "Sicily & Sardinia", "Douro", "Oregon", "Michiga~

> $ country


> $ winery

<chr> "Nicosia", "Quinta dos Avidagos", "Rainstorm", "~

> $ vintage


> $ price

<int> NA, 15, 14, 13, 65, 15, 16, 24, 12, 27, 19, 30, ~

> $ region_2

<chr> NA, NA, "Willamette Valley", NA, "Willamette Val~

> $ word_count

<int> 25, 39, 29, 34, 42, 42, 33, 22, 26, 31, 37, 33, ~

> $ pos_count

<int> 1, 2, 4, 2, 3, 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, ~

> $ .fitted

<dbl> 85.82313, 87.95619, 87.15903, 87.22717, 88.41721~

> $ .resid

<dbl> 1.17687033, -0.95618761, -0.15903408, -0.2271663~

> $ .hat

<dbl> 3.026605e-05, 1.695988e-05, 2.815231e-05, 1.3560~

> $ .sigma

<dbl> 2.601523, 2.601524, 2.601525, 2.601525, 2.601522~

> $ .cooksd

<dbl> 1.548555e-06, 5.728111e-07, 2.630307e-08, 2.5849~

> $ .std.resid

<dbl> 0.452385773, -0.367553504, -0.061132210, -0.0873~

315


The output tibble is our initial wine_2 regression data with five extra columns: .fitted, .resid, .std.resid, .hat, and .sigma. The data = wine_2 option tells augment() appends these statistics to wine_2. But we can exclude this option to get a more streamlined output:

```
wine_model %>%
augment
```

> # A tibble: 119,988 x 10
> points word_count pos_count neg_count .fitted .resid .hat .sigma .cooksd
> <int> <int> <int> <int>     <dbl>  <dbl> <dbl> <dbl>    <dbl>
> 1  87    25   1    1       85.8  1.18 3.03e-5 2.60 1.55e- 6
> 2  87    39   2    0       88.0 -0.956 1.70e-5 2.60 5.73e- 7
> 3  87    29   4    0       87.2 -0.159 2.82e-5 2.60 2.63e- 8
> 4  87    34   2    1       87.2 -0.227 1.36e-5 2.60 2.58e- 8
> 5  87    42   3    2       88.4 -1.42  2.36e-5 2.60 1.75e- 6
> 6  87    42   3    1       88.5 -1.50  8.87e-6 2.60 7.40e- 7
> 7  87    33   4    0       87.7 -0.674 2.19e-5 2.60 3.68e- 7
> 8  87    22   1    0       85.5  1.48 3.76e-5 2.60 3.03e- 6
> 9  87    26   5    0       87.0 -0.0180 4.67e-5 2.60 5.57e-10
>10  87    31   4    0       87.4 -0.417 2.48e-5 2.60 1.59e- 7
> # ... with 119,978 more rows, and 1 more variable: .std.resid <dbl>

As before, a given row in this output tibble corresponds to the analogous row in wine_2. But now the output includes only wine_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 = 1.17 points. This value is reported in the .resid column, which gives the difference between the predicted score, .fitted, and the actual score, points.9

The third broom function is glance() (which is not to be confused with glimpse()):

```
wine_model %>%
glance
```

> # A tibble: 1 x 12
> r.squared adj.r.squared sigma statistic p.value df  logLik AIC  BIC
> <dbl>     <dbl> <dbl>     <dbl> <dbl> <dbl>    <dbl> <dbl> <dbl>
> 1 0.293     0.293 2.60     16538.     0     3 -284973. 569957. 570005.
> # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>

9See ?augment.lm for the definition of the other statistics.
As you see, it produces an executive summary of the regression. The most important statistics \textit{r.squared}, which reports the fraction of the variation in the dependent variable that our model explains. For example, our \textit{r.squared} = 0.293 value indicates that we can attribute 29.3\% of the variation in \textit{points} to the variation in \textit{word_count}, \textit{pos_count}, and \textit{neg_count} (which is pretty good for just three variables).\(^{10}\)

Now let’s add some control variables to our regression. We’ll start with \textit{vintage}:

\begin{verbatim}
wine_2 %>%
  lm(
    points ~ word_count + pos_count + neg_count + vintage,
    data = .
  ) %>%
tidy
\end{verbatim}

\begin{verbatim}
> # A tibble: 5 x 5
> term    estimate std.error statistic      p.value
> <chr>    <dbl>     <dbl>       <dbl>        <dbl>
> 1 (Intercept) -107.     4.26       -25.1     3.67e-138
> 2 word_count  0.130     0.000711   182.        0.
> 3 pos_count  0.260     0.00442     58.9        0.
> 4 neg_count  -0.0754    0.00838    -8.99   2.52e- 19
> 5 vintage    0.0941    0.00212     44.4        0.
\end{verbatim}

The new estimate suggests that incrementing the \textit{vintage} year by one increases the expected number of \textit{points} by 0.0375. But assuming a linear relationship between \textit{points} and \textit{vintage} is too crude, as some years are better than others for wine. A more sophisticated specification gives each \textit{vintage} year its own idiosyncratic effect:

\begin{verbatim}
wine_model_2 <-
  wine_2 %>%
  lm(
    points ~ word_count + pos_count + neg_count + as.factor(vintage),
    data = .
  )

wine_model_2 %>%
tidy
\end{verbatim}

\begin{verbatim}
> # A tibble: 36 x 5
> term    estimate std.error statistic      p.value
> <chr>    <dbl>     <dbl>       <dbl>        <dbl>
> 1 (Intercept) -107.     4.26       -25.1     3.67e-138
> 2 word_count  0.130     0.000711   182.        0.
> 3 pos_count  0.260     0.00442     58.9        0.
> 4 neg_count  -0.0754    0.00838    -8.99   2.52e- 19
> 5 vintage    0.0941    0.00212     44.4        0.
\end{verbatim}

\(^{10}\)See \texttt{?glance.lm} for the definition of the other statistics.
> term estimate std.error statistic p.value
> <chr> <dbl> <dbl> <dbl> <dbl>
> 1 (Intercept) 83.7 1.28 65.2 0.
> 2 word_count 0.129 0.000711 181. 0.
> 3 pos_count 0.269 0.00442 60.9 0.
> 4 neg_count -0.0817 0.00835 -9.79 1.23e-22
> 5 as.factor(vintage)1986 0.613 1.96 0.313 7.54e- 1
> 6 as.factor(vintage)1987 1.64 2.22 0.735 4.62e- 1
> 7 as.factor(vintage)1988 -0.145 1.72 -0.0842 9.33e- 1
> 8 as.factor(vintage)1989 -1.59 1.66 -0.961 3.37e- 1
> 9 as.factor(vintage)1990 -0.0252 1.72 -0.0146 9.88e- 1
> 10 as.factor(vintage)1991 -3.43 1.82 -1.89 5.86e- 2
> # ... with 26 more rows

The as.factor() tells R to treat vintage as a factor, rather than an integer. And this change updates the statistical model from

\[ \text{points} = \beta_0 + \beta_1 \text{word\_count} + \beta_2 \text{pos\_count} + \beta_3 \text{neg\_count} + \beta_4 \text{vintage} + \epsilon, \]

to

\[ \text{points} = \beta_0 + \beta_1 \text{word\_count} + \beta_2 \text{pos\_count} + \beta_3 \text{neg\_count} + \beta_{1986} (\text{vintage} == 1986) + \cdots + \beta_{2017} (\text{vintage} == 2017) + \epsilon. \]

In the expression above, vintage == 1986 is a logical that indicates whether or not the vintage is 1986, where R interprets the TRUE values as 1 and the FALSE values as 0. And the \( \beta_{1986} \) coefficient records the average score of 1986 wines relative to 1985 wines (the baseline year). R benchmarks against the 1985 scores because "1985" is the first level of as.factor(vintage).\(^{11}\) For example, the \( \hat{\beta}_{1986} = 0.613 \) estimate suggests that the .fitted value of a 1986 wine would be 0.613 points higher than the .fitted value of a 1985 wine with the same word_count, pos_count, and neg_count values. And the same goes for the other years. For example, "Valserrano 2004 Rioja" is a 2004 wine with word_count == 61, pos_count==5, and neg_count == 1 and "Salon 1985 Le Mesnil Blanc de Blancs Brut Chardonnay (Champagne)" is a 1985 wine with word_count == 61, pos_count==5, and neg_count == 1. Thus, since \( \hat{\beta}_{2004} = -1.78 \) the former wine has a .fitted value that's 1.78 points less than the latter wine:

```r
wine_model_2 %>%
  augment(
    newdata =
    wine_2 %>%
  )
```

\(^{11}\)You can set this baseline to 1990 by including `mutate(vintage = as.factor(vintage), vintage = fct_relevel(vintage, "1990"))` above the `lm()` call.
filter(
  title %in%
  c(
    "Valserrano 2004 Rioja",
    str_c(
      "Salon 1985 Le Mesnil Blanc de ",
      "Blancs Brut Chardonnay (Champagne)"
    )
  )
)
)
)

select(title, .fitted)

> # A tibble: 2 x 2
> title                  .fitted
> <chr>                  <dbl>
> 1 Valserrano 2004 Rioja 91.1
> 2 Salon 1985 Le Mesnil Blanc de Blancs Brut Chardonnay (Champagne) 92.8

The example above uses the newdata option of augment() to calculate the .fitted values of the two relevant rows. The newdata option is powerful: it enables us to calculate the .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 = 40, pos_count = 35, neg_count = 5, and vintage = 1987:

wine_model_2 %>%
  augment(
    newdata =
    tibble(
      word_count = 40, pos_count = 35,
      neg_count = 5, vintage = 1987
    )
  )

> # A tibble: 1 x 5
> word_count pos_count neg_count vintage .fitted
>       <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
> 1        40        35       5     1987      99.5

And the math checks out:
\[
\text{fitted} = \hat{\beta}_0 + \hat{\beta}_1 \text{word\_count} + \hat{\beta}_2 \text{pos\_count} + \hat{\beta}_3 \text{neg\_count} \\
+ \hat{\beta}_{1986}(\text{vintage} == 1986) + \cdots + \hat{\beta}_{2017}(\text{vintage} == 2017) \\
= \hat{\beta}_0 + \hat{\beta}_1 \cdot 40 + \hat{\beta}_2 \cdot 35 + \hat{\beta}_3 \cdot 5 + \hat{\beta}_{1987} \cdot 1 \\
= 83.7 + 0.129 \cdot 40 + 0.269 \cdot 35 - 0.0817 \cdot 5 + 1.64 \cdot 1 \\
= 99.5
\]

Unfortunately, there are two problems with our treatment of vintage. First, \texttt{lm()} drops all observations with \texttt{vintage} = \texttt{NA}. Second, we do not have enough data to properly estimate all of our vintage coefficients. For example, \texttt{wine}_2 has only four rows with \texttt{vintage} = 1985 and only five rows with \texttt{vintage} = 1990. And with so few observations from 1985 and 1990, we cannot in good faith estimate \( \hat{\beta}_{1990} \), which represents the average point differential between 1990 reviews and 1985 reviews after factoring \texttt{word\_count}, \texttt{pos\_count}, and \texttt{neg\_count}. Here’s the fix:

```r
reg_tib <- 
  wine_2 %>%
  group_by(vintage) %>%
  mutate(vintage = ifelse(n() >= 100, vintage, NA)) %>%
  ungroup %>%
  replace_na(list(vintage = "Other"))

wine_model_3 <- 
  reg_tib %>%
  lm(
    points ~ word_count + pos_count + neg_count + vintage,
    data = .
  )

# Show beginning estimates:
wine_model_3 %>%
  tidy %>%
  head
```

> # A tibble: 6 x 5
> term        estimate std.error statistic  p.value
> <chr>        <dbl>     <dbl>      <dbl>    <dbl>
> 1 (Intercept) 81.9      0.151      542.     0.
> 2 word_count  0.129     0.000698   185.     0.
> 3 pos_count  0.269     0.00432     62.2     0.
> 4 neg_count  -0.0896    0.00821    -10.9 1.02e-27
> 5 vintage1998 -1.22      0.184     -6.64    3.10e-11
> 6 vintage1999 -0.967     0.179      -5.40    6.56e- 8

320
The `mutate()` step above turns all `vintage` values with fewer than 100 observations to `NA` and the `replace_na()` step transforms all `vintage = NA` values to `vintage = "Other"`. The `lm()` call treats this "Other" like any other vintage level: e.g., the "vintageOther" estimate of -0.381 indicates that the `vintage = "Other"` observations have 0.381 fewer points on average than the `vintage = "1997"` benchmark, after controlling for `word_count`, `pos_count`, and `neg_count` (the baseline year changed from 1985 to 1997). Also, setting some `vintage` values to "Other" changed this variable’s data type from integer to character string. And this means we no longer have to include `as.factor()` in the regression formula because `lm()` automatically converts all character strings to factors.

**Exercise 5.13.** We will now show that there’s been significant grade inflation over time.

- Pipe `wine_model_3` into `tidy()`, and pipe the result of that into `filter()`.
- Within the `filter()`, use `str_detect()` to keep only the rows with `term %in% c("vintage1998", "vintage1999", ..., "vintage2016")`
- `str_detect("[0-9]{4}\")` tests whether a string contains a four-digit number.
- Pipe the result into `mutate()`. Within this step do the following:
  - Define `term` as the numeric year associated with the corresponding text string. For example, if `term = "vintage1999"` then redefine `term = 1999`.
  - Define `lower_lim = estimate - qnorm(.975) * std.error` and `upper_lim = estimate + qnorm(.975) * std.error`. These are the 95% confidence intervals of our `estimate` values. (If these intervals do not contain zero then we reject the null hypothesis that the true coefficient is zero at the \( p = 0.05 \) level.)
  - Pipe the result into `ggplot()`, and use `aes()` to set `x = term`, `y = estimate`, `ymin = lower_lim`, and `ymax = upper_lim`.
  - Use `geom_point()` and `geom_errorbar()` to plot the `estimate` values and corresponding 95% confidence intervals.
  - You should see a systematic increase in the `estimate` values, which indicates that later `vintage` values receive higher scores.
Exercise 5.14. Use `wine_model_3` and `reg_tib` to get the description of the wine that most exceeded its expectations, given its `word_count`, `pos_count`, `neg_count`, and `vintage` values.

- Our model holds up well here: this wine's exorbitantly high points value is incommensurate with its subdued description.

Exercise 5.15. We will now add `province`, `taster_name`, `variety`, and `price` to our statistical model.

- Use `add_count()` to define `n_vintage`, `n_province`, `n_taster_name`, and `n_variety` as the number of observations in `wine_2` with the given `vintage`, `province`, `taster_name`, and `variety` values, respectively. For example, the observations with `province = "Oregon"` should have `n_province = 4929`.
- Use `mutate()` to redefine `vintage = ifelse(n_vintage >= 500, vintage, "Other")`. And analogously redefine `province`, `taster_name`, and `variety`.
- Within the `mutate()` call set `price = pmin(price, 200)`.
- This step truncates the maximum price to 200. Without this regularization, a few outlier price values would have an inordinate effect on the model.
- Use `replace_na()` to set the NA values of `vintage`, `province`, `taster_name`, and `variety` to "Other".
- You can do this step either inside or outside of the `mutate()`. I recommend the latter.
- You should only need one `replace_na()`.
- Save the resulting tibble as `reg_tib_2`.
- Pipe `reg_tib_2` into `lm()`, and use `points ~ word_count + pos_count + neg_count + price + vintage + province + taster_name + variety`.
- Save the output as `wine_model_4`.
- Pipe `wine_model_4` into `tidy()` to confirm that the `pos_count` and `neg_count` coefficient estimates are still statistically different from zero (strongly so).

Exercise 5.16. According to `wine_model_4`, which reviewer is the harshest grader and which is the easiest grader?

- Write code that produces the answer—do not just look it up with `View()`.
- When `lm()` converts the `taster_name` variable of `reg_tib_2` to a factor, it sets the first level to "Anna Lee C. Iijima". Hence, all `taster_name` coefficient estimates benchmark against this reviewer's average scores. For example, the estimate corresponding to Paul Gregutt is 0.422, which indicates that if Paul and Anna reviewed the same wine then we would expect the former to assign 0.422 more points, on average.

Exercise 5.17. My friend Emily Harbaugh\(^\text{12}\) reviews wines on Instagram\(^\text{13}\). So I asked her to dictate the following two reviews.

\(^\text{12}\)(https://www.instagram.com/EmilyHarbaugh/)
\(^\text{13}\)https://www.instagram.com/glougang/
• 2020 California Merlot, Bota Box brand:

Smells alcoholic. Yeah, it definitely does have notes of alcohol. Hold on. It’s like pretty juicy. It’s light; it’s 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 Costco tastes a lot like alcohol—yeah, Costco has a $14.98 bottle of organic red wine, but it tastes more like alcohol than this one. Yeah, so, I feel like this is not a bad boxed wine to have on hand.

• 2019 Argentinian Malbec, National Origin brand (i.e., the Costco wine referenced above):

So this one has a heavier mouth feel. And it definitely smells more like alcohol, 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 tongue after I’m done drinking it. Definitely 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 more juicy and more enjoyable when drinking than this one. This one’s definitely like; this one definitely smells like alcohol, and I don’t like that. Does that work, do you need any other details? A one sentence summary? It is, ahh, god, a one sentence summary... Not that great, like thick and no real like pronounced fruit flavor. I don’t like, I don’t like it. I don’t know how else to say but. Yeah, it’s just like it’s not good.

We’ll use our model to anticipate which review would have a higher score, if they were published by Wine Enthusiast. Here’s some additional information about these wines:

```r
emily_wine <- tribble(
  ~brand, ~price, ~vintage, ~province, ~taster_name, ~variety,
  "Bota Box", 20, "Other", "California", "Other", "Merlot",
  "National Origin", 15, "Other", "Mendoza Province", "Other", "Malbec"
)
```

Both wines have `vintage = "Other"` because `wine_model_4` assigns all years outside of 1999–2016 to "Other", and both wines have `taster_name = "Other"` because Emily Harbaugh is not represented in our sample. Also, the latter wine has `province = "Mendoza Province"` because that’s the wine country of Argentina.
exercise 5.17 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 Running Regression with Lists

wine_model_4 accounts for differences in the average severity of the various graders. For example, all other things equal, we would expect 1.05 fewer points from a review written by Susan Kostrzewa than one written by Anna Lee C. Iijima. 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 Buzzolo 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 the idiosyncrasies of each reviewer we will refine our approach from modeling a generic Wine Enthusiast reviewer to modeling a specific Wine Enthusiast reviewer. Specifically, we will create an independent statistical model for each of the 14 taster_name values of reg_tib_2.

Of course we could create these 14 models by writing `lm()` 14 times:

```r
#Model 1
model_anna <-
  reg_tib_2 %>%
  filter(taster_name == "Anna Lee C. Iijima") %>%
  lm(
    points ~ word_count + pos_count + neg_count +
    price + vintage + province + taster_name + variety,
    data = .
  )

#Model 2
model_anne <-
  reg_tib_2 %>%
  filter(taster_name == "Anne Krebiehl\xa0MW") %>%
  lm(
    points ~ word_count + pos_count + neg_count +
    price + vintage + province + taster_name + variety,
    data = .
  )

# ...

# Model 14
model_virginie <-
  reg_tib_2 %>%
  filter(taster_name == "Virginie Boone") %>%
  lm(
    points ~ word_count + pos_count + neg_count +
    price + vintage + province + taster_name + variety,
    data = .
  )

But the following exercise outlines a more elegant solution, based on nested tibbles.

**Exercise 5.18.** We will now create a statistical model for each `taster_name` value in `reg_tib_2`.

- Use `nest` to transform `reg_tib_2` into a tibble with two columns: `taster_name` and `taster_sample`, where the latter is a column of nested tibbles that store the data corresponding to a give reviewer.
  - The row with `taster_name = "Sean P. Sullivan"` should have a 4,461 × 21 tibble in its `taster_sample` column.
- Pipe the result into `mutate()`.
- Within `mutate()` use `map()` to run a regression with each tibble in `taster_sample`. Use `points ~ word_count + pos_count + neg_count + price + vintage + province + variety` as the formula, and save the regression outputs in a list column called `taster_models`.
- Save the resulting tibble as `taster_models_tib`.
  - This tibble should have 14 rows and 3 columns.

**Exercise 5.19.** We will now plot each reviewer’s `pos_count` and `neg_count` estimates and 95% confidence intervals.

- Pipe `taster_models_tib` into `mutate()`.
• Within the `mutate()` call do the following:
  - Define `taster_name = taster_name %>% str_replace("\xa0", " ") %>% str_remove("\u0019")`. This step removes some of the screwy characters from the reviewer names.
  - Use `map()` to apply `tidy()` to each model in the `taster_models` column. Save the corresponding tibbles in a list column called `coefficient_est`.
• `unnest()` the tibble by `coefficient_est`.
• Use `filter()` to drop the estimates that do not correspond to `pos_count` or `neg_count`.
• Use the methodology of exercise 5.13 to plot the estimate values and 95% confidence intervals with `geom_point()` and `geom_errorbar()`.
  - Set `y = taster_name`, `x = estimate`, `xmin = lower_lim`, and `xmax = upper_lim` to make the error bars run horizontally rather than vertically.
  - Use `color` to create distinct `pos_count` and `neg_count` error bars.

### 5.3 Lab: Supply Chain Proximity

#### 5.3.1 Overview

Since producing high-quality products requires collaboration across the supply chain and since it’s harder to collaborate with far-flung suppliers, Bray et al. [2019a] theorized that geographically dispersed supply chains produce lower quality goods. They confirmed this hypothesis with a dataset that tracked the flow of thousands of auto parts through hundreds of component factories and auto assembly plants, showing that, indeed, “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 supported their claims with several advanced statistical tests, but the most compelling evidence they provided are the simple plots in Figures 5, 6, and 9, which depict the results of standard linear regressions. We will recreate these plots in this lab, confirming for ourselves that auto parts are more prone to failure when their factory is further from the final auto assembly plant.

#### 5.3.2 Data

You created a tibble called `cars` in exercise 1.21 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.
• 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 calculate this formula with the geosphere package, which you can install and load with the following:

```r
install.packages('geosphere')
library(geosphere)
```

After loading this package we can calculate Vincenty’s formula with `distVincentyEllipsoid()`. This 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 Beijing and Seoul:

```r
beijing_coords <- c(116.4074, 39.9042) #(longitude, latitude)
seoul_coords <- c(126.9780, 37.5665)

beijing_coords %>% distVincentyEllipsoid(seoul_coords) #meters
```

> [1] 954356.3

In general we invoke this function with `distVincentyEllipsoid(c(lon_1, lat_1), c(lon_2, lat_2))` or `c(lon_1, lat_1) %>% distVincentyEllipsoid(c(lon_2, lat_2)).` We do not invoke it with `distVincentyEllipsoid(lon_1, lat_1, lon_2, lat_2)`, which means we cannot simply define distance with `cars %>% mutate(distance = distVincentyEllipsoid(lon_1, lat_1, lon_2, lat_2))`. Constructing this variable will require more work.

---

14For copyright reasons, I perturbed the defects, lat_d, lon_d, lat_u, and lon_u values a bit. However, the noise I added to these variables does not qualitatively change the solution.
15https://en.wikipedia.org/wiki/Vincenty%27s_formulae
17Note, `distVincentyEllipsoid()` expects coordinates to be expressed as (longitude, latitude), not as (latitude, longitude).
Exercise 5.20. We will now calculate the distance between the \((\text{lon}_u, \text{lat}_u)\) and \((\text{lon}_d, \text{lat}_d)\) points in cars

- Pipe cars into `mutate()`, and within this function do the following:
  - Use `map2()` to create a list column called `coord_u` that stores each row’s \((\text{lon}_u, \text{lat}_u)\) pair. That is, define \(\text{coord}_u = \text{list}(\text{c}(-3.16, 48.8), \text{c}(-3.16, 44.0), \ldots, \text{c}(23.1, 60.5))\).
  - Analogously define \(\text{coord}_d = \text{list}(\text{c}(-1.83, 50.3), \text{c}(-104., 38.7), \ldots, \text{c}(17.1, 48.3))\) as a list column of \((\text{lon}_d, \text{lat}_d)\) coordinate vectors.
  - Use `map2_dbl()` to apply `distVincentyEllipsoid()` to the coordinate vectors of `coord_u` and `coord_d` and save the output as `distance`. Note, `distance` should be a numeric vector, since `map2_dbl()` always returns a vector of doubles. For example, the first element of `distance` should be \(\text{c}(-3.16, 48.8) \%\% \text{distVincentyEllipsoid}(\text{c}(-1.83, 50.3)) = 192593\).
  - Redefine `distance = distance / 1000` to convert the meters to kilometers.
- `select()` away the `lat_u`, `lon_u`, `lat_d`, and `lon_d` columns, since `coord_u` and `coord_d` make them redundant.
- Overwrite `cars` with the resulting tibble.

5.3.3 Analysis

We’ll begin by reconstructing Figures 5 and 6, since scatter plots are the best means of illustrating the relationship between two variables.

Exercise 5.21. Scatter plot defects as a function of distance, `facet_wrap()` by module, and save the plot as `car_scatter`.

- Tidy the plot with `theme(strip.background = element_blank())` and `labs(x = "Distance", y = "Defects")`.

Exercise 5.22. Use `%+%' to restrict `car_scatter` to the 40 module values with the most observations, and use `scale_x_log10()` and `scale_y_log10()` to log-transform the scales (like Figure 5). Save the graph as `car_scatter_2`.

- Conduct a filtering join with `semi_join()`.
- Use a bracketed expression to avoid defining a wheelbarrow variable. That is, write `car_scatter %+% {}` and define the new tibble inside the brackets.

Exercise 5.23. Update `car_scatter_2` so that you facet by `brand` rather than by `module`. Use `%+%' to restrict the sample to the 28 brand values with the most observations. Save the result as `car_scatter_3`.

328
Now car_scatter_2 and car_scatter_3 differ from Figures 5 and 6 two notable ways. First, Bray et al. [2019a] Winsorized\(^{18}\) their data, so that their vertical axis is more limited than ours. We’ll skip this step, since the plots look fine without Winsorization. Second, Bray et al. [2019a] run an \texttt{lm()} trend line through each facet. And these trend lines have a special feature: they are color-coded based on whether the slope is statistically significantly different from zero. We could add non-color-coded trend lines by incorporating an extra \texttt{geom_smooth(method = "lm")} layer in the ggplot. But making the lines with \texttt{p.value < 0.05} a different color from those with \texttt{p.value >= 0.05} will require more work.

**Exercise 5.24.** We will now add color-coded trend lines to car_scatter_2.

- Limit cars to the 40 module values with the most observations, as you did in exercise 5.22.

- \texttt{nest()} the resulting tibble by module, so that it has only two columns: module and module\_tib, the latter of which is a column of nested tibbles.
  - The row with \texttt{module = "EXHAUST"} should have a 683 x 9 tibble nested in its module\_tib slot.

- Use \texttt{mutate()} and \texttt{map()} to run a regression with each tibble in module\_tib. Save the output as a list column called \texttt{reg\_model}.
  - For the regression formula use \texttt{I(log10(defects)) ~ I(log10(distance))}, rather than \texttt{defects ~ distance}, since we’ve log-transformed the x- and y-axes.

- Use \texttt{map()} to apply \texttt{tidy()} to each element of \texttt{reg\_model}. Save the output as a list column called \texttt{coef}.
  - Each element of \texttt{coef} should store a 2 x 5 tibble.

- \texttt{unnest()} the tibble by \texttt{coef}.
  - This step should expand your tibble from 40 x 4 to 80 x 8.

- Apply \texttt{filter(term == "I(log10(distance))")} and define significance as a variable that equals "Significant" if \texttt{p.value < 0.05} and equals "Insignificant" otherwise.

- \texttt{unnest()} the module\_tib list column.

- Save the resulting tibble as \texttt{data\_to\_plot}.

- Add \texttt{data\_to\_plot} to car_scatter_2.

- Add a color-coded line to each facet with \texttt{geom_smooth(method = "lm", aes(color = significance))}.
  - Embedding the \texttt{aes()} within \texttt{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 \texttt{theme(legend.position = 'top', legend.title = element_blank())}.

- Save the resulting plot as car_scatter_4.

\(^{18}\)https://en.wikipedia.org/wiki/Winsorizing

329
Exercise 5.25. We will now update `car_scatter_4` so that it facets by `brand` rather than by `module`.

- Define `data_to_plot_2` as the brand-based analog of `data_to_plot_2`. This tibble should correspond to the 28 brand values with the most data (like in exercise 5.23).
- Use `%>%` to add `data_to_plot_2` to `car_scatter_4`.
- `facet_wrap()` by brand (overwriting the previous faceting scheme).

We will now recreate Figure 9 of Bray et al. [2019a]. Whereas the regressions underlying Figures 5 and 6 capture the relationship between defects and distance with two parameters—an intercept and a slope—the regression underlying Figure 9 captures the relationship with 100 parameters—giving each distance percentile its own expected number of defects. This latter specification is extremely flexible, permitting distance to influence defects in a non-linear fashion: e.g., the expected rate of defects could be 1.0 when distance is in 25th percentile, 10.0 when distance is in 50th percentile, and 0.01 when distance is in the 75th percentile. Our model could accommodate this strange relationship because it gives the 25th, 50th, and 75th percentiles their own parameters.

It may seem like a paradox: using the linear model function `lm()` to model a non-linear relationship. But we’ve seen this before: when we switched from `points ~ word_count + pos_count + neg_count + vintage` to `points ~ word_count + pos_count + neg_count + as.factor(vintage)` in section 5.2.4 we went from imposing a strict linear relationship between points and vintage to allowing every vintage year to have its own idiosyncratic effect on points.\(^{19}\) We will use the same trick to allow every distance percentile to have its own idiosyncratic effect on defects.

Exercise 5.26. We will now illustrate the relationship between distance and defects with a plot that’s analogous to Figure 9 of Bray et al. [2019a].

- Pipe `cars` into `mutate()` to redefine `defects = percent_rank(defects)` and `dist_pct = ntile(distance, n = 100)`. The former operation expresses defects values in percentage terms and the latter groups distance into percentile buckets.
- Run a regression with formula `defects ~ as.factor(dist_pct) + part + brand`. With this, `lm()` will treat `as.factor(dist_pct)`, `part`, and `brand` as factors, and will estimate a parameter for all but the first level of each.
- `tidy()` the results of this regression.

\(^{19}\)Recall that replacing `vintage` with `as.factor(vintage)` replaces the \(\beta_4\) vintage term in our statistical model with \(\beta_{1986}(\text{vintage} == 1986) + \cdots + \beta_{2017}(\text{vintage} == 2017)\), where the logicals in the parentheses are interpreted as zero when they’re \text{FALSE} and one when they’re \text{TRUE}. And while `points` is not linear in `vintage`, it is linear in the 32 logicals—`vintage == 1986, vintage == 1987, \ldots, and vintage == 2017`—which is why this specification still counts as a “linear model.”
"brandVOLVO". The dist_pct estimates report the average rate of defects relative to the dist_pct = 1 benchmark. For example, the "as.factor(dist_pct)5" estimate of 0.0304 suggests that components in the fifth percentile of distance tend to have 0.0304 more defects (per ten million vehicle years) than components in the first percentile of distance, after controlling for part and brand. Similarly, the part estimates benchmark against the part = "ACTUATORS" rate of defects and the brand estimates benchmark against the brand = "ACURA" rate.

- filter() out the rows that do not pertain to dist_pct.
- Use str_remove_all("[^0-9]") to turn convert the term values into integers. For example, the first row should have term = 2.
  - "as.factor(dist_pct)11" %>% str_remove_all("[^0-9]") = "11".
- Derive lower_lim and upper_lim from estimate and std.error, as you did in exercise 5.13.
- Use geom_point() and geom_errorbar() to plot the estimate values and 95% confidence intervals (like you did in exercise 5.13).
  - Your plot should look similar to Figure 9, but your estimates values will differ as Bray et al. [2019a] pursued a slightly different methodology. For example, your plot depicts the differences relative to the dist_pct = 1 baseline, whereas their plot depicts the absolute level in the rate of defects.

5.4 Commentary

5.4.1 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:
library(tictoc)

janes_words <-
    full_text %>%
    pull(text) %>%
    str_split(" ") %>%
    unlist

#Case 1:
tic() #start timer
janes_words %>%
    str_to_lower %>%
    head

> [1] "chapter" "1"   "the"   "family" "of"   "dashwood"

toc() #stop timer

> 0.114 sec elapsed

#Case 2:
tic() #start timer
janes_words %>%
    map_chr(str_to_lower) %>%
    head

> [1] "chapter" "1"   "the"   "family" "of"   "dashwood"

toc() #stop timer

> 1.863 sec elapsed

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 \texttt{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 \texttt{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.

We used the second approach in our solution to exercise 5.11 when we exported the \texttt{word} list column to the \texttt{word_list} tibble before implementing the \texttt{left_join(sentiment_dictionary)} step. Alternatively, we could have separately applied this \texttt{left_join()} to each and every wine review:

```r
wine_2 <-
  wine %>%
  mutate(
    sentiment =
      description %>%
      str_replace_all("[^[:alnum:]]", " ") %>%
      str_squish %>%
      str_split(" ") %>%
      map(~ .x %>%
          tibble(word = .) %>%
          left_join(sentiment_dictionary) %>%
          summarise(
            word_count = n(),
            pos_count = sum(sentiment == "+", na.rm=TRUE),
            neg_count = sum(sentiment == "-", na.rm=TRUE),
            .groups = "drop"
          )
      )
  ) %>%
  unnest(sentiment)
```

And while this code is more elegant, it’s impossibly slow, because it calls \texttt{left_join(sentiment_dictionary)} 119988 times, rather than once.

On a related note, we could streamline our solution to exercise 5.11 to the following:

```r
wine_2 <-
  wine %>%
  mutate(

description %>%
      str_replace_all("[^[:alnum:]]", " ") %>%
      str_squish %>%
      str_split(" ") %>%
      map(~ .x %>%
          tibble(word = .) %>%
          left_join(sentiment_dictionary) %>%
          summarise(
            word_count = n(),
            pos_count = sum(sentiment == "+", na.rm=TRUE),
            neg_count = sum(sentiment == "-", na.rm=TRUE),
            .groups = "drop"
          )
      )
  ) %>%
  unnest(sentiment)
```
word =
    description %>%
    str_replace_all("[^[:alnum:]]", " ") %>%
    str_squish %>%
    str_split(" ")
) %>%
unnest(word) %>%
left_join(sentiment_dictionary) %>%
group_by(across(-c(word, sentiment))) %>%
summarise(
    word_count = n(),
    pos_count = sum(sentiment == "+", na.rm=TRUE),
    neg_count = sum(sentiment == "-", na.rm=TRUE),
    .groups = "drop"
)

But in this case the unnest(word) step would output a 4994756 x 16 tibble, which would strain our computer’s memory more than word_list, which is only 4994756 x 2.

### 5.4.2 Tibbles are Lists

Here’s a confusing fact: tibbles are actually a special kind of list. Indeed, all tibbles are lists, like all Californians are Americans. However, not all lists are tibbles, like not all Americans are Californians. For example, define the following tibble:

tiny_tib <-
tibble(
    x = 1:2,
    y = c("a", "b"),
    z = c(TRUE, FALSE)
)
tiny_tib

> # A tibble: 2 x 3
>   x  y   z
>  <int> <chr> <lgl>
> 1 1 a TRUE
> 2 2 b FALSE

In addition to being a tibble, tiny_tib is also a list:
Specifically, it’s a list of three elements:

```r
tiny_tib %>% length
```

> [1] 3

The list’s elements correspond to the tibble’s columns. For example, the second element of the list is the second column of the tibble:

```r
tiny_tib %>% pluck(2)
```

> [1] "a" "b"

Thus, tibbles are lists 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
tiny_list %>% is_tibble

> [1] FALSE

But such a list can be turned into a tibble (unlike most other lists):

tiny_list %>%
  as_tibble

> # A tibble: 2 x 3
>   x y  z
> 1 1 a TRUE
> 2 2 b FALSE

Moreover, tiny_list does not cease being a list when it becomes a tibble, like how an American doesn’t lose their US citizenship when they move to California.

Since tibbles are lists, all list functions apply to tibbles. For example, we can use map() to get the class() of each column in our tibble:

tiny_tib %>%
  map(class)

> $x
> [1] "integer"

> $y
> [1] "character"

> $z
> [1] "logical"

Or we could transpose() the tibble to group the data by row rather than by column:
> [[1]]
> [[1]]$x
> [1] 1
> >
> [[1]]$y
> [1] "a"
> >
> [[1]]$z
> [1] TRUE
> >
> >
> [[2]]
> [[2]]$x
> [1] 2
> >
> [[2]]$y
> [1] "b"
> >
> [[2]]$z
> [1] FALSE

5.5 Solutions

5.1

```r
jane_tib <-
  jane_tib %>%
  mutate(
    characters =
      characters %>%
      map(str_to_title)
  )
```

5.2

```r
jane_tib <-
  jane_tib %>%
  mutate(  
    characters =
      characters %>%
      map2(
```
list(
  c("Edward Ferrars", "John Willoughby"),
  NULL,
  "Sir Thomas Bertram"
),
~.x %>% c(.y)
)

5.3

nested_tib %>%
  mutate(C = B %>% map_int(nrow))

5.4

jane_tib <-
  jane_tib %>%
  left_join(
    book_level_tib,
    by = c("title" = "book")
  )

5.5

jane_tib %>%
  pluck("chapter_tib", 3, "text_tib", 5)

5.6 The former case has three layers of nesting whereas the latter case has only two layers of nesting.

5.7

jane_tib %>%
  unnest(chapter_tib) %>%
  unnest(text_tib) %>%
  group_by(across(-text)) %>%
  summarise(338
text = str_c(text, collapse = " "),
.groups = "drop"
) %>%
mutate(text = str_split(text, " ")) %>%
nest(chapter_tib = c(chapter, text))

5.8

wine_list <-
wine_list %>%
map(
  .x %>%
  list_modify(
    vintage =
      pluck(., "title") %>%
      str_extract("\b[0-9]{4}\b") %>%
      strtoi %>%
      ifelse(. < 1985 | . > 2017, NA, .)
  )
)

5.9

wine <-
wine_list %>%
map_df(~.x %>% discard(is_null)) %>%
distinct %>%
rowid_to_column(var = "review_id") %>%
mutate(
  across(
    c(points, price),
    strtoi
  )
)

5.10

wine %>%
ggplot() +
```r
aes(
    x = str_length(description),
    y = points
  ) +
geom_density_2d_filled()
```

5.11

```r
word_list <-
  wine %>%
  mutate(
    word =
      description %>%
      str_replace_all("[^[:alnum:]]", " ") %>%
      str_squish %>%
      str_split(" ")
  ) %>%
select(review_id, word) %>%
unnest(word)
```
word_stats <-
  word_list %>%
  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),
    .groups = "drop"
  )

wine_2 <-
  wine %>%
  left_join(word_stats)

5.12

wine_2 %>%
  mutate(
    score = pos_count - neg_count,
    score = pmin(score, quantile(score, .995)),
    score = pmax(score, quantile(score, .005))
  ) %>%
  ggplot() +
  aes(
    x = points,
    y = as.factor(score)
  ) +
  geom_violin()
5.13

```r
wine_model_3 %>%
tidy %>%
filter(str_detect(term, "[0-9]{4}")) %>%
mutate(
  term =
  str_replace(term, "vintage", "") %>%
  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() +
geom_errorbar()
```
5.14

```r
wine_model_3 %>%
  augment(data = reg_tib) %>%
  filter(.resid == max(.resid)) %>%
  pull(description)
```

> [1] "A big, bold wine with unbelievable power and concentration. Low yields and a dominance of Cabernet

5.15

```r
reg_tib_2 <-
  wine_2 %>%
  add_count(
    vintage,
    name = "n_vintage"
  ) %>%
  add_count(
    province,
    name = "n_province"
  )
```

343
add_count(
  taster_name,
  name = "n_taster_name"
)
%>
add_count(
  variety,
  name = "n_variety"
)
%>
mutate(
  vintage =
    ifelse(n_vintage >= 500, vintage, "Other"),
  province =
    ifelse(n_province >= 500, province, "Other"),
  taster_name =
    ifelse(n_taster_name >= 500, taster_name, "Other"),
  variety =
    ifelse(n_variety >= 500, variety, "Other"),
  price = pmin(price, 200)
)
%>
replace_na(
  list(
    province = "Other", taster_name = "Other",
    variety = "Other", vintage = "Other"
  )
)

wine_model_4 <-
  reg_tib_2 %>%
  lm(
    points ~ word_count + pos_count + neg_count +
    price + vintage + province + taster_name + variety,
    data = .
  )

wine_model_4 %>% tidy

> # A tibble: 114 x 5
> term     estimate std.error statistic p.value
> <chr>    <dbl>     <dbl>      <dbl>    <dbl>
> 1 (Intercept) 80.6 0.175     461.      0.    
> 2 word_count  0.101 0.000654 155.       0.    
> 3 pos_count   0.244 0.00385   63.2      0.    
> 4 neg_count  -0.110 0.00699  -15.7     1.36e-55
> 5 price      0.0427 0.000267 160.      0.    

344
wine_model_4 %>%
tidy %>%
  filter(str_detect(term, "taster_name")) %>%
  mutate(term = str_remove(term, "taster_name")) %>%
  summarise(
    harshest_grader =
      subset(term, estimate == min(estimate)),
    easiest_grader =
      subset(term, estimate == max(estimate))
  )

> # A tibble: 1 x 2
>   harshest_grader easiest_grader
>     <chr>       <chr>
> 1 Michael Schachner Anne KrebiehlMW

emily_word_stats <-
tibble(
  brand = c("Bota Box", "National Origin"),
  description =
    str_c(
      "Smells alcoholic. Yeah, it definitely does have notes",
      "of alcohol. Hold on. It's like pretty juicy. 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",
    )

5.16
"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 = " "
),
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 definately smells like alchohol, and I don't",
  "like that. Does that work, do you need any other",
  "details? A one sentence summary? It is, ahhh, god, a"
  "one sentence summary... Not that great, like thick and",
  "no real like pronounced fruit flavor. I don't like, I",
  "dont like it. I don't know how else to say but.",
  "Yeah, it's just like it's not good.",
  sep = " "
)
),

word =
  description %>%
  str_replace_all("[^[:alnum:]]", " ") %>%
  str_squish %>%
  str_split(" ")
) %>%
select(-description) %>%
unnest(word) %>%
left_join(sentiment_dictionary) %>%
group_by(brand) %>%
summarise(
  word_count = n(),
  pos_count = sum(sentiment == "+", na.rm=TRUE),
  neg_count = sum(sentiment == "-", na.rm=TRUE),
  .groups = "drop"

emily_wine <-
  emily_wine %>%
  left_join(emily_word_stats)

wine_model_4 %>%
augment(newdata = emily_wine

> # A tibble: 2 x 10
>   brand   price vintage province taster_name variety word_count pos_count
>  <chr>   <dbl> <chr>  <chr>      <chr>   <chr>     <int>    <int>
> 1 Bota Box  20 Other California Other Merlot 129       15
> 2 National ~ 15 Other Mendoza Pro~ Other Malbec 203       26
> # ... with 2 more variables: neg_count <int>, .fitted <dbl>

5.18

taster_models_tib <-
  reg_tib_2 %>%
  nest(taster_sample = -taster_name %>%
  mutate(
    taster_models =
      taster_sample %>
      map(~ .x %>%
        lm(
          points ~ word_count + pos_count + neg_count +
            price + vintage + province + variety,
          data = .
        )
      )
    )
  )

5.19

taster_models_tib %>%
  mutate(
    taster_name =
      taster_name %>%
   诗句文字渲染结束
str_replace("\xa0", " ") %>%
str_remove("\u0019"),
coefficient_est =
taster_models %>%
map(tidy),
) %>%
unnest(coefficient_est) %>%
filter(term %in% c("pos_count", "neg_count")) %>%
mutate(
  lower_lim = estimate - qnorm(.975) * std.error,
  upper_lim = estimate + qnorm(.975) * std.error
) %>%
ggplot() +
aes(
  y = taster_name,
  x = estimate,
  xmin = lower_lim,
  xmax = upper_lim,
  color = term
) + geom_point() +
geom_errorbar()
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 expected value, 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.

I’ll start the lecture in section 6.2.1 by illustrating a key drawback of linear models: every \texttt{lm()} outputs negative .fitted values for some range of the independent variables (since any non-flat line will be negative somewhere). And in many contexts negative values make no sense. For example, we will create a linear model that forecasts a negative number of shooting stars. I will use this issue to motivate the development of GLMs, which can be made to always output positive .fitted values.

I will define the GLM in section 6.2.2. Basically, this definition will amount to characterizing (i) the transformation function \( f() \) and (ii) the distribution of \( y \) conditional on the transformed linear combination of independent variables, \( f(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \). In section 6.2.3 we will use the GLM framework to fix the failed shooting star model we made in section 6.2.1. And finally in section 6.2.4 we will use the GLM specification to run a logistic regression with a logical dependent variable (whose values are either \texttt{TRUE} or \texttt{FALSE}).
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 our GLM framework 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).

6.2 Lecture

6.2.1 Limits of Linear Models

The shooting_star tibble you made in exercise 1.22 of section 1.3 looks like this:

```r
shooting_star %>% head
```

```
> # A tibble: 6 x 3
>   visibility sky_glow star_count
>   <dbl>   <dbl>      <dbl>
> 1  0.0184  8.88         3
> 2  0.0146  3.62         8
> 3  0.0140  3.69         4
> 4  0.0436  9.62         0
> 5  0.289  0.729         5
> 6  0.123  2.98         3
```

This tibble comprises 10231 shooting start measurements recorded by volunteers in Eastern Oregon, in the summers of 1992–1995. Each row corresponds to a four-hour observation window. The `visibility` and `sky_glow` variables record the air clarity and light pollution measured at the beginning of the observation window, and the `star_count` variable records the number of shooting stars observed during the observation window.

We want to gauge how light pollution effects our ability to see shooting stars. We’ll start by plotting the data.

**Exercise 6.1.** We will plot the distribution of `star_count` for each decile of `sky_glow` to see how the latter variable influences the former.

- Use `ntile()` to create a variable called `sky_glow_decile` that divides `shooting_star` into 10 equally sized subsamples, based on the deciles of `sky_glow`.
- Append the string "Decile: " to the beginning of each `sky_glow_decile`. And turn this variable into a factor.
- Use `geom_histogram()` to plot the distribution of `star_count`, and `facet_wrap()` by `sky_glow_decile`. 

350
The plot you made in exercise 6.1 illustrates that (i) star_count decreases with sky_glow and (ii) star_count is not normally distributed, since it’s skewed to the right. But although star_count is not normally distributed, we can still use OLS to model its expected value conditional on sky_glow. Here’s the regression:

```
shooting_star %>%
  lm(
    star_count ~ sky_glow,
    data = .
  ) %>%
  tidy
```

> # A tibble: 2 x 5
>   term      estimate std.error statistic p.value
>  <chr>       <dbl>     <dbl>      <dbl>   <dbl>
> 1 (Intercept)  4.26      0.0314     136.     0
> 2 sky_glow    -0.297     0.00412     -71.9     0

The second coefficient estimate confirms the negative correlation between star_count and sky_glow. Unfortunately, however, this linear model is quite bad, as the following exercise illustrates.

**Exercise 6.2.** We will illustrate how poorly our linear model fits the data by overlaying its regression line on a scatterplot of star_count and sky_glow. Rather than use the regression coefficients from our `lm()` call, we will plot the regression line with `geom_smooth()` (which calls `lm()`).

- Pipe `shooting_star` into `ggplot()`, with `x = sky_glow` and `y = star_count`.
- Include a `geom_point()` layer to depict the raw data.
- Include a `geom_smooth()` layer with `method = "lm"` to depict the regression line.

This plot illustrates a critical problem: the expected number of shooting stars is negative when sky_glow exceeds 13. To incorporate a non-negativity constraint in our statistical model we will upgrade our specification from a linear model to a generalized linear model (GLM).
6.2.2 Generalized Linear Models

To make the GLM concept concrete, define the following random sample:

```r
set.seed(0)  # fix random seed
N <- 10^5  # pick sample size

tib_base <-
tibble(mean_val = c(.1, .5, 1, 2, 10)) %>%
  group_by(mean_val) %>%
s summarise(
    var_gamma = rgamma(N, scale = mean_val/2, shape = 2),
    var_normal = rnorm(N, mean = mean_val, sd = 2),
    var_poisson = rpois(N, lambda = mean_val),
    .groups = "drop"
  )

tib_base %>% head
```

The `var_gamma`, `var_normal`, and `var_poisson` columns of `tib_base` are random variables with gamma\(^1\), normal\(^2\), and Poisson\(^3\) distributions. Each variable has mean `mean_val`, which takes values in \(c(.2, .5, 1, 2, 5)\).

**Exercise 6.3.** We will now plot the distributions of `var_gamma`, `var_normal`, and `var_poisson` for each value of `mean_val`.

- Use `pivot_longer()` to pull down the `var_gamma`, `var_normal`, and `var_poisson` columns of `tib_base`.
  - The resulting tibble should have columns `mean_val`, `name`, and `value`.

---

\(^1\)https://en.wikipedia.org/wiki/Gamma_distribution
\(^2\)https://en.wikipedia.org/wiki/Normal_distribution
\(^3\)https://en.wikipedia.org/wiki/Poisson_distribution
Pipe the result into ggplot(), with aes(x = value).
Add a geom_histogram(bins = 50) layer, and facet_wrap() by mean_val and name, with scales = "free".

Although they have the same means, var_gamma, var_normal, and var_poisson have very different distributions: e.g., the normal distribution is the only one that can take negative values and the only one whose spread doesn't increase with mean_val, and the Poisson distribution is the only one that's restricted to integers and the only one whose shape changes with mean_val.

Now let's add another twist:

```r
set.seed(0) # fix random seed
tib_varying_means <- tibble(
  x_1 = runif(N),
  x_2 = runif(N),
  mean_val = 0.1 + 0.5 * x_1 + 2 * x_2,
  var_gamma = rgamma(N, scale = mean_val/2, shape = 2),
  var_normal = rnorm(N, mean = mean_val, sd = 2),
  var_poisson = rpois(N, lambda = mean_val)
)
```

The var_gamma, var_normal, and var_poisson columns are still gamma, normal, and Poisson random variables with mean mean_val, but now mean_val is itself a variable. Specifically, mean_val is a linear combination of uniform random variables x_1 and x_2. And since every observation has a different mean_val, every var_gamma, var_normal, and var_poisson is drawn from a different distribution. The following exercise illustrates this point.

**Exercise 6.4.** We will now plot the distributions of var_gamma, var_normal, and var_poisson for various ranges of mean_val.

- Use ntile() to add to tib_varying_means a variable called mean_decile that specifies the decile of mean_val.
- Plot the distribution of var_gamma, var_normal, and var_poisson by mean_decile in the fashion of exercise 6.3.

Now suppose we dropped mean_val and forgot the $\beta_0 = 0.1$, $\beta_1 = 0.5$, and $\beta_2 = 2$ values that define it. Could we recover these parameters from var_normal, x_1, and x_2? We could (albeit with some estimation error):
As you see, our GLM estimates are $\hat{\beta}_0 = 0.114$, $\hat{\beta}_1 = 0.474$, and $\hat{\beta}_2 = 2.01$. The \texttt{glm()} notation is like the \texttt{lm()} notation, except with an extra \texttt{family = gaussian(link = "identity")} option. Since “Gaussian distribution” is synonymous with “normal distribution,” this option specifies that the dependent variable, \(\text{var\_normal}\), is drawn from a normal distribution whose mean scales linearly with the independent variables, \(\text{x\_1}\) and \(\text{x\_2}\). (In contrast, the \texttt{lm()} specification assumes that the mean dependent variable scales linearly with the independent variables, but \textit{does not} assume that the dependent variable is normally distributed.)

We can get the fitted values of our GLM with \texttt{augment()}:

```
> # A tibble: 100,000 x 12
>    x_1  x_2  mean_val var_gamma var_normal var_poisson .fitted  .resid
>    <dbl> <dbl>    <dbl>      <dbl>      <dbl>        <int>    <dbl>    <dbl>
> 1  0.897  0.123   0.795        1.92        3.04            0  0.787   2.25
> 2  0.266  0.701   1.63         1.29        0.743            1  1.65   -0.904
> 3  0.372  0.702   1.69         2.08        4.43            2  1.70    2.73
> 4  0.573  0.313   1.01         0.546        2.79            1  1.01    1.77
> 5  0.908  0.552   1.66         1.17        1.21            3  1.65   -0.447
> 6  0.202  0.872   1.95         3.18       -1.42            3  1.96   -3.39
> 7  0.898  0.681   1.91         2.15        2.29            0  1.91    0.384
> 8  0.945  0.888   2.35         3.26        3.20            1  2.35    0.858
```
Note the similarity between the `mean_val` and `.fitted` values. This isn’t a coincidence: the `.fitted` values are our model’s best guess for the mean of the distribution that `var_normal` is drawn from—i.e., its best guess for `mean_val`. Hence, `.fitted` is constructed the same way as `mean_val`, except with the estimated parameters—\( \hat{\beta}_0 \), \( \hat{\beta}_1 \), and \( \hat{\beta}_2 \)—rather than with the actual parameters—\( \beta_0 \), \( \beta_1 \), and \( \beta_2 \). For example, the first observation has `.fitted = 0.787` and `mean_val = 0.795`, which indicates that the GLM thinks the first `var_normal` value was drawn from a normal distribution with mean 0.787, when it was actually drawn from a normal distribution with mean 0.795. Moreover, the GLM thinks the normal distribution has standard deviation `.sigma = 2.01`, when it actually has `sd = 2`. Hence, our GLM pegs the probability of the first `var_normal` value being no greater than 3 at `pnorm(3, mean = 0.787, sd = 2.01) = 0.8646`, when it’s actually `pnorm(3, mean = 0.795, sd = 3) = 0.8649`.

We can change the regression formula to `var_poisson ~ x_1 + x_2`, so long as we also change the `family` input to `poisson(link = "identity")`, which indicates that the dependent variable is drawn from a Poisson distribution, conditional on `x_1` and `x_2`:

```r
glm_poisson <-
  tib_varying_means %>%
  glm(
    var_poisson ~ x_1 + x_2,
    family = poisson(link = "identity"),
    start = c(1, 1, 1),
    data = .
  )

glm_poisson %>% tidy
```

The `start = c(1, 1, 1)` option tells R to begin its search for \( \hat{\beta}_0 \), \( \hat{\beta}_1 \), and \( \hat{\beta}_2 \) from the point `c(1, 1, 1)`. (I include this option because R returns an error without it: the `glm()` function sometimes fails to identify reasonable estimates, in which case it’ll ask you to try again from a different starting point.)
Again, the .fitted values closely track the mean_val values:

```r
glm_poisson %>%
augment(data = tib_varying_means)
```

> # A tibble: 100,000 x 12
> x_1 x_2 mean_val var_gamma var_normal var_poisson .fitted .resid
> <dbl> <dbl> <dbl> <dbl> <dbl> <int> <dbl> <dbl>
> 1 0.897 0.123 0.795 1.92 3.04 0 0.793 -1.26
> 2 0.266 0.701 1.63 1.29 0.743 1 1.63 -0.532
> 3 0.372 0.702 1.69 2.08 4.43 2 1.69 0.235
> 4 0.573 0.313 1.01 0.546 2.79 1 1.01 -0.0102
> 5 0.908 0.552 1.66 1.17 1.21 3 1.66 0.937
> 6 0.202 0.872 1.95 3.18 -1.42 3 1.94 0.703
> 7 0.898 0.681 1.91 2.15 2.29 0 1.91 -1.95
> 8 0.945 0.888 2.35 3.26 3.20 1 2.34 -0.991
> 9 0.661 0.747 1.92 0.415 1.86 5 1.92 1.85
> 10 0.629 0.0351 0.485 0.307 4.21 0 0.484 -0.984
> # ... with 99,990 more rows, and 4 more variables: .std.resid <dbl>,
> # .hat <dbl>, .sigma <dbl>, .cooksd <dbl>

For example, the first observation has .fitted = 0.793 and mean_val = 0.795, which indicates that the GLM thinks the first var_poisson value was drawn from a Poisson distribution with mean 0.793, when it was actually drawn from a Poisson distribution with mean 0.795. Hence, our GLM pegs the probability of the first var_poisson value being no greater than 3 at ppois(3, 0.793) = 0.9912, when it’s actually ppois(3, 0.795) = 0.9911.

**Exercise 6.5.** Run a GLM of var_gamma on x_1 and x_2. Save the model as glm_gamma.

- Use family = Gamma(link = "identity") to specify that the dependent variable has a gamma distribution.

---

So far we’ve extended the standard linear model by assigning a distribution to the dependent variable. (To repeat: we did not specify a distribution for the dependent variable when we used lm().) For example, here’s the distribution of var_gamma for various x_1 and x_2 values:

```r
glm_gamma %>%
augment(newdata =
```
```r
expand_grid(
  x_1 = c(.25, 1, 4),
  x_2 = c(.25, 1, 4)
)
)

mutate(
  gamma_shape =
  glm_gamma %>%
  summary %>%
 pluck(., "dispersion"),
  gamma_scale = .fitted / gamma_shape
)

%>%

group_by(x_1, x_2) %>%

summarise(
  value = seq(0, 30, .01),
  density =
  dgamma(
    value,
    shape = gamma_shape, scale = gamma_scale
  ),
  .groups = "drop"
)

%>%

mutate(
  x_1 = str_c("x_1 = ", x_1),
  x_2 = str_c("x_2 = ", x_2)
)

%>%

ggplot() +

aes(  
  x = value,
  y = density
)

%>%

geom_area() +

facet_grid(
  rows = vars(x_1),
  cols = vars(x_2),
  scales = "free"
)
)
These distributions can be useful. For example, suppose schools close when the number of inches of snowfall, as measured by \( \text{var}_\gamma \), exceeds 12. In this case, rather than the average \( \text{var}_\gamma \) we would care about the probability of \( \text{var}_\gamma \) exceeding 12. And this is a statistic we couldn’t derive from \( \text{lrm}() \) but can derive from \( \text{glm}() \), since its output characterizes the full distribution of the dependent variable.

**Exercise 6.6.** We will now plot the probability of school closure—i.e., the probability of \( \text{var}_\gamma \) exceeding 12—for various values of \( x_1 \) and \( x_2 \).

- Use `augment()` to calculate the .fitted value of \( \text{glm}_\gamma \) for all values in `expand_grid(x_1 = seq(0, 100, .01), x_2 = 0:10)`.
- Use `mutate()` to define `gamma_shape` and `gamma_scale`, as in the example above.
  - The gamma distribution is parameterized by its “shape” and “scale,” rather than by its mean and standard deviation.
- Within the `mutate()` call, use `gamma_shape` and `gamma_scale` to define `prob_closure` as the probability of a school closure.
  - \( \text{pgamma}(x, \text{shape} = y, \text{scale} = z, \text{lower.tail} = \text{FALSE}) \) denotes the probability that a random variable drawn from a gamma distribution with shape \( y \) and scale \( z \) exceeds \( x \).
- Plot `prob_closure` as a function of \( x_1 \), while coloring by \( x_2 \).
There are two properties that distinguish GLMs from linear models. First, as we have discussed, GLMs specify the distribution of the dependent variable, but linear models do not. Second, whereas a linear model sets the expected value of the dependent variable to a linear combination of the independent variables—i.e., to \( \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n \), for some coefficients \( \beta_0, \ldots, \beta_n \)—a GLM sets the expected value of the dependent variable to a function of a linear combination of independent variables—i.e., to \( f(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \), for some coefficients \( \beta_0, \ldots, \beta_n \) and function \( f() \).

For example, the code below defines \( \text{var} \_\text{gamma} \) with \( f(x) = 1/x \), defines \( \text{var} \_\text{normal} \) with \( f(x) = x^2 \), and defines \( \text{var} \_\text{poisson} \) with \( f(x) = \exp(x) \):

```r
set.seed(0)  # fix random seed
tib_f_fn <-
tibble(
  x_1 = runif(N),
  x_2 = runif(N),
  linear_combo = 0.1 + 0.5 * x_1 + 2 * x_2,
  mean_gamma = 1/linear_combo,
  mean_normal = linear_combo^2,
  mean_poisson = exp(linear_combo),
  var_gamma = rgamma(N, scale = mean_gamma/2, shape = 2),
  var_normal = rnorm(N, mean = mean_normal, sd = 2),
  var_poisson = rpois(N, lambda = mean_poisson)
)
tib_f_fn %>%
  head
```

> # A tibble: 6 x 9
>   x_1  x_2 linear_combo mean_gamma mean_normal mean_poisson var_gamma
>  <dbl> <dbl> <dbl>       <dbl>       <dbl>       <dbl>    <dbl>
> 1  0.897 0.123        0.795        1.26       0.631       2.21   3.04
> 2  0.266 0.701        1.63         0.612      2.67       5.12   0.483
> 3  0.372 0.702        1.69         0.592      2.85       5.42   0.730
> 4  0.573 0.313        1.01         0.988      1.02       2.75   0.533
> 5  0.908 0.552        1.66         0.603      2.75       5.25   0.425
> 6  0.202 0.872        1.95         0.514      3.79       7.00   0.840
> # ... with 2 more variables: var_normal <dbl>, var_poisson <int>

In the example above, \( \text{linear} \_\text{combo} \) is a linear combination of the independent variables, \( x_1 \) and \( x_2 \). And \( \text{mean} \_\text{gamma} \), \( \text{mean} \_\text{normal} \), and \( \text{mean} \_\text{poisson} \) are functions of \( \text{linear} \_\text{combo} \) that serve as the mean values of \( \text{var} \_\text{gamma} \), \( \text{var} \_\text{normal} \), and \( \text{var} \_\text{poisson} \). For example, the first row has \( \text{linear} \_\text{combo} = 0.795 \), and thus has \( \text{mean} \_\text{gamma} = 1 / 0.795 = 1.26 \),
mean\_normal = 0.795^2 = 0.631, and mean\_poisson = \exp(0.795) = 2.21, which indicates that this row’s \var\_gamma is drawn from a gamma distribution with mean 1.26, its \var\_normal is drawn from a normal distribution with mean 0.631, and its \var\_poisson is drawn from a Poisson distribution with mean 2.21.

Let \( f^{-1}(x) \) denote the inverse of function \( f() \), such that \( f^{-1}(f(x)) = x \) for all \( x \). For example:

- if \( f(x) = 1/x \) then \( f^{-1}(x) = 1/x \) since \( \frac{1}{1/x} = x \),
- if \( f(x) = x^2 \) then \( f^{-1}(x) = \sqrt{x} \) since \( \sqrt{x^2} = x \), and
- if \( f(x) = \exp(x) \) then \( f^{-1}(x) = \log(x) \) since \( \log(\exp(x)) = x \).

The \( f^{-1}(x) \) inverse functions, or “link functions,” allow us to derive the \( \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n \) linear combination from the variable means. For example, we can use these link functions to recover the first row’s linear\_combo value from its mean\_gamma, mean\_normal, and mean\_poisson values:

\[
\frac{1}{1.26} = 0.795, \quad \sqrt{0.631} = 0.795, \quad \log(2.21) = 0.795.
\]

To accommodate the \( f() \) transformations, we pass the \( f^{-1()} \) link functions into our GLM calls:

```r
glm\_gamma\_2 <-
tib\_f\_fn %>%
glm(
  var\_gamma ~ x\_1 + x\_2,
  family = Gamma(link = "inverse"),
  data = .
)
glm\_gamma\_2 %>% tidy
```

```r
> # A tibble: 3 x 5
> term  estimate std.error statistic  p.value
> <chr>   <dbl>     <dbl>     <dbl>     <dbl>
> 1 (Intercept)  0.102     0.00305    33.5 1.66e-244
> 2 x\_1        0.495     0.00660    75.0  0.
> 3 x\_2        2.01      0.00850    236.  0.
```

```r
glm\_normal\_2 <-
tib\_f\_fn %>%
glm(
  var\_normal ~ x\_1 + x\_2,
  family = gaussian(link = "sqrt"),
```

360
```
start = c(1, 1, 1),
data = .
```

```
# A tibble: 3 x 5
term  estimate std.error statistic p.value
<chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)  0.115   0.00842   13.7  1.89e-42
2 x_1         0.487   0.00754   64.6      0.
3 x_2         1.99    0.00954   209.       0.
```

```
glm_poisson_2 <- tib_f_fn %>%
  glm(
    var_poisson ~ x_1 + x_2,
    family = poisson(link = "log"),
    start = c(1, 1, 1),
data = .
  )
```

```
# A tibble: 3 x 5
term  estimate std.error statistic p.value
<chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)  0.0998  0.00487   20.5  1.82e-93
2 x_1         0.504   0.00515   97.9      0.
3 x_2         2.00    0.00563  354.       0.
```

For the regressions above we’ve replaced the `link = "identity"` expressions with `link = "inverse"`, `link = "sqrt"`, and `link = "log"`, which respectively denote \( f^{-1}(x) = 1/x \), \( f^{-1}(x) = \sqrt{x} \), and \( f^{-1}(x) = \log(x) \). In contrast, `link = "identity"` specifies \( f^{-1}(x) = x \) (i.e., no transformation).

The \( f() \) transformations allow us to constrain the expected value of the dependent variable. For example, since \( \exp(x) > 0 \), setting `mean_poisson = \exp(linear_combo)` ensures that the mean of `var_poisson` is positive, which is crucial since a Poisson random variable must have a positive mean.\(^4\) Indeed, the following plots the mean of `var_poisson` for various values of `x_1` and `x_2`:

\(^4\)Setting `mean_poisson = linear_combo^2` also guarantees that `var_poisson` has a positive mean, but this specification is more awkward, since \( f(x) = x^2 \) is neither strictly increasing nor strictly decreasing in \( x \).
Note, we now include `type.predict = "response"` in our `augment()` call. Without this option `augment()` returns the untransformed linear combination of independent variable—i.e., $\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_n x_n$—but with this option the function returns the mean of $\text{var_poisson}$—i.e., $f(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_n x_n)$. 
6.2.3 Application: Shooting Stars

We are now properly equipped to model star_count.

**Exercise 6.7.** The number of shooting stars observed in a given span of time is generally understood to follow a Poisson distribution\(^5\). Use this fact to create a GLM of star_count as a function of sky_glow. Call the model glm_star.

- Use transformation function \(f(x) = \exp(x)\) to ensure that the Poisson’s mean is positive.

**Exercise 6.8.** We will now recreate the plot we made in exercise 6.2 with our GLM model’s fitted value.

- Use augment() to get the .fitted values of glm_star.
  - Use the `type.predict = "response"` option to account for the \(f()\) transformation function.
- Plot the star_count and .fitted values as a function of sky_glow. Use a geom_point() layer for the former and a geom_line() layer for the latter. That is, scatter plot the star_count values and overlay a line that specifies their corresponding .fitted values.

**Exercise 6.9.** Our GLM gives each value of sky_glow its own star_count distribution. We will now depict this family of distributions. Specifically, for each value of sky_glow in our sample, we will calculate the probability that star_count = num_stars for num_stars = 0, 1, ..., 9. We will then plot these various probabilities as a function of sky_glow.

- Use `augment(type.predict = "response")` to get the .fitted values of glm_star.
- Since a Poisson random variable is fully characterized by its mean, the distribution of star_count is fully characterized by .fitted. Specifically, the probability that star_count = n is given by dpois(n, .fitted), according to our model. Use `summarise()` to define num_stars = 0:9 and prob_count = dpois(num_stars, .fitted) for each .fitted value.
  - The result should be a tibble with columns sky_glow, .fitted, num_stars, and prob_count.
- Scatter plot prob_count by sky_glow, with color = as.factor(num_stars).
  - Your plot should trace out 10 lines, each painted a different color.

---

We can extend the regression formula to factor visibility in our model:

```r
shooting_star %>%
  glm(
    star_count ~ sky_glow + visibility,
    family = poisson(link = "log"),
    data = .
  ) %>%
  tidy
```

> # A tibble: 3 x 5
> term     estimate std.error statistic p.value
> <chr>    <dbl>    <dbl>     <dbl>    <dbl>
> 1 (Intercept)  1.73      0.014  123.000     0
> 2 sky_glow    -0.183     0.0024 -78.139     0
> 3 visibility  0.861      0.0555 15.51523 3.06e-54

As you see, `star_count` increases significantly with `visibility`. Now let’s add a term that factors the *interaction* between `sky_glow` and `visibility`:

```r
shooting_star %>%
  mutate(glow_times_vis = sky_glow * visibility) %>%
  glm(
    star_count ~ sky_glow + visibility + glow_times_vis,
    family = poisson(link = "log"),
    data = .
  ) %>%
  tidy
```

> # A tibble: 4 x 5
> term                estimate std.error statistic p.value
> <chr>                <dbl>    <dbl>     <dbl>    <dbl>
> 1 (Intercept)        1.79      0.0171  104.975     0
> 2 sky_glow           -0.198    0.00329 -60.30738     0
> 3 visibility         0.264      0.105   2.51939   1.21e-02
> 4 glow_times_vis     0.142      0.0207  6.86085  6.95e-12

The `glow_times_vis` term is significantly positive, which indicates that the effect of `visibility` is larger when `sky_glow` is high. For example:

- if `sky_glow = 0` and `visibility = 0` then \( \text{.fitted} = \exp(1.79 - 0.198 \times 0 + 0.264 \times 0 + 0.142 \times 0 \times 0) = 6.00 \).
• if sky_glow = 0 and visibility = 0.1 then \( \text{fitted} = \exp(1.79 - 0.198 \times 0 + 0.264 \times 0.1 + 0.142 \times 0 \times 0.1) = 6.16, \)
• if sky_glow = 1 and visibility = 0 then \( \text{fitted} = \exp(1.79 - 0.198 \times 1 + 0.264 \times 0 + 0.142 \times 1 \times 0) = 4.92, \) and
• if sky_glow = 1 and visibility = 0.1 then \( \text{fitted} = \exp(1.79 - 0.198 \times 1 + 0.264 \times 0.1 + 0.142 \times 1 \times 0.1) = 5.12. \)

Hence, increasing visibility from 0 to 0.1 increases the expected star_count by 6.16 - 6.00 = 0.16 when sky_glow = 0, and increases the expected star_count by 5.12 - 4.92 = 0.20 when sky_glow = 1. And this makes sense: kicking dust into the air is more blinding when there’s more upward-pointing light to reflect off of the particulate matter.

Here’s a simpler way to run the regression above:

```r
shooting_star %>%
  glm(
    star_count ~ sky_glow * visibility,
    family = poisson(link = "log"),
    data = .
  ) %>%
  tidy
```

> # A tibble: 4 x 5
> term              estimate std.error statistic  p.value
> <chr>            <dbl>     <dbl>      <dbl>    <dbl>
> 1 (Intercept)     1.79      0.0171     105.       0.
> 2 sky_glow        -0.198     0.00329   -60.3       0.
> 3 visibility      0.264      0.105      2.51      1.21e-2
> 4 sky_glow:visibility 0.142     0.0207     6.86     6.95e-12

This output is the same as the output above—except with the "glow_times_vis" label changed to "sky_glow:visibility"—because R interprets sky_glow * visibility as sky_glow + visibility + glow_times_vis.

### 6.2.4 Logistic Regression

Suppose that instead of shooting_star we received shooting_star_binary, defined as follows:
shooting_star_binary <-
  shooting_star %>%
  mutate(any_stars_observed = star_count > 0) %>%
  select(-star_count)

shooting_star_binary %>% head

> # A tibble: 6 x 3
> visibility sky_glow any_stars_observed
> <dbl> <dbl> <lgl>
> 1 0.0184 8.88 TRUE
> 2 0.0146 3.62 TRUE
> 3 0.0140 3.69 TRUE
> 4 0.0436 9.62 FALSE
> 5 0.289 0.729 TRUE
> 6 0.123 2.98 TRUE

This is a more impoverished dataset: it indicates whether any shooting stars were observed, but not how many were observed. Nevertheless, we can still capture the influence of sky_glow and visibility on any_stars_observed with a GLM. But we can’t assign any_stars_observed a gamma, normal, or Poisson distribution, since 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 any_stars_observed is TRUE with probability \( p \) and FALSE with probability \( 1 - p \), where the value of \( p \) varies with sky_glow and visibility.

In this case, the mean value of any_stars_observed is TRUE \( \cdot p \) + FALSE \( \cdot (1 - p) = 1 \cdot p + 0 \cdot (1 - p) = p \). Thus, any_stars_observed’s mean equals its probability of being TRUE. And since this probability must be between 0 and 1, the mean value of any_stars_observed must be between 0 and 1, and thus the output of \( f(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \) must be between 0 and 1. Accordingly, we must choose a transformation function \( f() \) that always outputs a number between 0 and 1. The simplest such function is the logistic function\(^6\): \( f(x) = \frac{e^x}{1 + e^x} \). 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:

logistic_regression_model <-
  shooting_star_binary %>%
  glm(  
    any_stars_observed ~ sky_glow * visibility,  
    family = binomial(link = "logit"),

---

\(^6\)https://en.wikipedia.org/wiki/Logistic_function

\(^7\)https://en.wikipedia.org/wiki/Logit
data =.

logistic_regression_model %>% tidy

> # A tibble: 4 x 5
> term estimate std.error statistic p.value
> <chr> <dbl> <dbl> <dbl> <dbl>
> 1 (Intercept) 3.90 0.105 37.1 6.99e-301
> 2 sky_glow -0.358 0.0117 -30.7 2.18e-206
> 3 visibility 0.778 0.795 0.979 3.28e- 1
> 4 sky_glow:visibility 0.244 0.0874 2.79 5.29e- 3

The `binomial()` term specifies that `any_stars_observed` has a binary distribution over TRUE and FALSE. The `.fitted` values below report probability, $p$, of the dependent variable being TRUE:

logistic_regression_model %>%
  augment(type.predict = "response")

> # A tibble: 10,231 x 9
> any_stars_observed sky_glow visibility .fitted .resid .std.resid .hat .sigma
> <lgl>          <dbl>        <dbl>    <dbl>   <dbl>    <dbl>  <dbl>     <dbl>
> 1 TRUE          8.88        0.0184 0.685 0.870  3.11e-4  0.858
> 2 TRUE          3.62        0.0146 0.933 0.373  2.51e-4  0.858
> 3 TRUE          3.69        0.0140 0.931 0.378  2.54e-4  0.858
> 4 FALSE         9.62        0.0436 0.644 -1.44 -2.83e-4  0.858
> 5 TRUE          0.729       0.289  0.980 0.199  4.92e-4  0.858
> 6 TRUE          2.98        0.123  0.953 0.309  1.37e-4  0.858
> 7 TRUE          4.00        0.0540 0.928 0.385  1.64e-4  0.858
> 8 FALSE         2.82        0.0957 0.954 -2.48 -1.27e-4  0.858
> 9 FALSE         14.5        0.0147 0.225 -0.715 -0.715  0.858
> 10 TRUE         7.61        0.139  0.824 0.623  1.77e-4  0.858
> # ... with 10,221 more rows, and 1 more variable: .cooksd <dbl>

For example, since $3.90 - 0.358 \cdot 8.88 + 0.778 \cdot 0.0184 + 0.244 \cdot 8.88 \cdot 0.0184 = 0.776$, the first `.fitted` value is $\exp(0.776)/(1 + \exp(0.776)) = 0.685$.

**Exercise 6.10.** To gauge the statistical strength of our GLM, we will compare the `.fitted` values in the `any_stars_observed = TRUE` subsample against those in the `any_stars_observed = FALSE` subsample. If our model has good predictive power then the fitted values should be high in the former subsample and low in the latter subsample.
Pipe logistic_regression_model into augment(type.predict = "response").
Plot the distribution of .fitted with geom_density(). Use fill = any_stars_observed to distinguish between the any_stars_observed = TRUE and any_stars_observed = FALSE subsamples.
- Include the option alpha = .3 in your geom_density() call, so you can see both distributions, even when they overlap.

Exercise 6.11. We will now use our logistic regression model to anticipate how much harder it’d be to see shooting stars if with twice as much light pollution.

- Pipe shooting_star_binary into mutate() and define double_glow = 2 * sky_glow.
- rename() sky_glow to current_glow.
- Pull down the current_glow and double_glow columns with pivot_longer(values_to = "sky_glow").
- Save the result in wheelbarrow variable tib_hypothetical.
- Use augment() to calculated the .fitted values corresponding to the sky_glow and visibility values in tib_hypothetical.
- Use geom_density() to plot the distribution of .fitted under both the "current_glow" and "double_glow" scenarios.

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 Sample

We will use facility, power, shutdown, ler, and radiation from section 1.3, exercise 1.23. 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 equip_ler
variables specify the number of LERs that are flagged with the “actuation”,
“technical specifications”, “degradation”, “inoperable”, and “prevent safety
- 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 man\(^8\) (rem) units.

### 6.3.3 Clean

We will combine these tibbles into a master panel in exercise 6.13. But first we must
streamline our shutdown tibble.

**Exercise 6.12.** Use `pivot_wider()` to reshape `shutdown` into a tibble with 2458 rows and
five columns: `reactor_name`, `start`, `shut_vol`, `shut_invol`, and `shut_refuel`, where the latter
three variables are logicals that indicate whether a shutdown of the given type was initiated
in the given month. For example, the row with `reactor_name = "Browns Ferry 2"` and `month =
ymd("1984-09-01")` should have `shut_vol = TRUE`, `shut_invol = FALSE`, and `shut_refuel =
TRUE`, because this reactor had `stop_type = "vol"` and `stop_type = "refuel"` shutdowns with
`start = ymd("1984-09-01")`, but did not have a `stop_type = "invol"` shutdown with `start =
ymd("1984-09-01")`. Your tibble does not need to include rows for reactor-months without
shutdowns. For example, it doesn’t need an observation with `reactor_name = "Browns Ferry
2"` and `month = ymd("1984-10-01")`, since Browns Ferry 2 did not shut down in October 1984.

- Model your `pivot_wider()` call on the following example:

```r
mini_tib <-
tibble(
  x = c(1, 2, 3, 1),
  y = c("a", "b", "a", "a")
)
mini_tib
```

\(^8\)https://www.nrc.gov/reading-rm/basic-ref/glossary/rem-roentgen-equivalent-man.html
We're now ready to create our master panel.

**Exercise 6.13.**

- Create a list containing facility, power %>% mutate(year = year(month)), ler, radiation, and shutdown %>% rename(month = start) in that order.
- Pipe this list into reduce(left_join). This reduce() operation joins the elements of the list one tibble at a time. In other words, it implements the following:
• The shut_vol, shut_invol, and shut_refuel variables will be NA unless the plant shut down in the specified month. Use replace_na() to replace these NA values with zeros, since there were zero shutdowns in these months.

• Use group_by() and cumsum() to replace shut_vol, shut_invol, and shut_refuel with their cumulative sums (restarting the counts for each reactor_name).
  – Make sure a reactor’s observations are ordered by month before implementing cumsum().
  – Use across() to avoid writing cumsum() multiple times.

• Let logical variable pra indicate whether the facility has performed its PRA by the given month.

• Remove 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.”

• Define \( \text{day} = \text{as.integer(month - min(month))} \) as the number of days since the start of the sample (i.e., since mdy("01-01-1985")).
  – ungroup() the tibble before this step.

• Save these data in a tibble called tib_nuclear.

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 tib_nuclear to run a linear model (not a GLM) with formula ler ~ pra + day + reactor_name. Use tidy() 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 "reactor_nameArkansas Nuclear 2", ... , "reactor_nameWolf Creek 1" estimates benchmark against the reactor_name = "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_name to ler ~ pra + day * reactor_name:
```r
lm_nuclear <-
  tib_nuclear %>%
  lm(
    ler ~ pra + day * reactor_name,
    data = .
  )

# top six estimates:
lm_nuclear %>% tidy %>% head

> # A tibble: 6 x 5
> term              estimate  std.error statistic  p.value
> <chr>                     <dbl>      <dbl>     <dbl>    <dbl>
> 1 (Intercept)             1.69       0.241       7.01    2.42e-12
> 2 praTRUE                 -0.169     0.0479      -3.54   4.03e- 4
> 3 day                      -0.000196 0.0000828   -2.37   1.79e- 2
> 4 reactor_nameArkansas Nuclear 2      0.582      0.339       1.71   8.65e- 2
> 5 reactor_nameBeaver Valley 1        -0.443     0.339      -1.31   1.91e- 1
> 6 reactor_nameBeaver Valley 2        -0.426     0.446      -0.955  3.40e- 1

# bottom six estimates:
lm_nuclear %>% tidy %>% tail

> # A tibble: 6 x 5
> term              estimate  std.error statistic  p.value
> <chr>                     <dbl>      <dbl>     <dbl>    <dbl>
> 1 day:reactor_nameVermont Yankee 0.000376    0.000116    3.25   0.00116
> 2 day:reactor_nameVogtle 1       -0.000645    0.000136   -4.74  0.00000213
> 3 day:reactor_nameVogtle 2       -0.0008744    0.000164   -5.453  0.651
> 4 day:reactor_nameWaterford 3    -0.0000669    0.000120   -0.558  0.577
> 5 day:reactor_nameWatts Bar 1   -0.000750     0.000987   -0.759  0.448
> 6 day:reactor_nameWolf Creek 1  -0.000602     0.000120   -5.01  0.000000536
```

Now in addition to the estimates we had previously—"(Intercept)", "praTRUE", "day", "reactor_nameArkansas Nuclear 2", ..., "reactor_nameWolf Creek 1"—we have 102 reactor-level time trends: "day:reactor_nameArkansas Nuclear 2", ..., "day:reactor_nameWolf Creek 1". These estimates report the rate of improvement relative to the reactor_name = "Arkansas Nuclear 1" baseline. For example, Watts Bar 1 decreased its LER rate by 0.000750 reports per day more than Arkansas Nuclear 1 did. Put differently, 365 days the expected LER rate is 1.69−0.000196⋅365 = 1.62 at Arkansas Nuclear 1 and 1.69−0.000196⋅365−0.000750⋅365 = 1.34 at Watts Bar 1 (assuming that neither facility implemented a PRA in the first year).
Exercise 6.15. Use `augment()` to get the expected number LERs, according to `lm_nuclear`, filed at the Turkey Point 4 reactor in December, 1998. In other words, calculate this observation's `.fitted` value.

According to our model, the expected number of LERs reported at Turkey Point 4 in December, 1998 is `negative`. But, of course, this is nonsense: engineers can’t file a negative number of reports. So to resolve this issue we will now switch to the GLM framework.

Exercise 6.16. Define `glm_nuclear` as the GLM analog of `lm_nuclear`. Use a Poisson distribution with the \( f(x) = \exp(x) \) transformation function. Pipe `glm_nuclear` into `tidy()` to view its estimates.

The "praTRUE" estimate is -0.153, which suggests that the expected number of LERs should only be \( \exp(-0.153) = 85.8\% \) as high after implementing a PRA. For example, the "(Intercept)", "day", "reactor_nameKewaunee", "day:reactor_nameKewaunee" estimates are 0.585, -.000183, -0.237, and 0.000124, respectively. Thus, the expected LER rate corresponding to `reactor_name = "Kewaunee"`, `day = 100`, and `pra = FALSE` is

\[
.fitted = \exp(0.585 - 0.000183 \cdot 100 - 0.237 + 0.000124 \cdot 100) = 1.41
\]

In contrast, the expected LER rate corresponding to `reactor_name = "Kewaunee"`, `day = 100`, and `pra = TRUE` is

\[
.fitted = \exp(0.585 - 0.153 - 0.000183 \cdot 100 - 0.237 + 0.000124 \cdot 100) \\
= \exp(-0.153) \exp(0.585 - 0.000183 \cdot 100 - 0.237 + 0.000124 \cdot 100) \\
= 0.858 \cdot 1.41 \\
= 1.21
\]

Note, the \( \exp(x + y) = \exp(x) \exp(y) \) property of the exponential function enables us to isolate \( \exp(-0.153) \).

We’re now ready to run the regressions reported in Table 2. These regressions have the following formulas:

```r
glm_formulas <-
c(
  ler ~ pra + reactor_name * day + shut_refuel,
  ler ~ pra + reactor_name * day + shut_refuel + shut_vol,
  ler ~ pra + reactor_name * day + shut_refuel + shut_vol + shut_invol
)
```
The first formula corresponds to column (1) of Table 2, the second formula to column (2), and the third formula to column (3).

**Exercise 6.17.** Use `map()` to run a GLM regression with each formula in `glm_formulas`. As before, use a Poisson distribution with the $f(x) = \exp(x)$ transformation function. The output of your `map()` call should be a list of GLMs. Save this list as `glm_list`.

**Exercise 6.18.** Use `map_df()` to apply `tidy()` to each GLM in `glm_list` and compress the results into a tibble called `glm_est`. Within the `map_df()` call, incorporate a `mutate()` step that defines a character string column called `specification` that stores the GLM’s regression formula. This column will report which GLM a given estimate corresponds.

- Piping a GLM into functional assembly line . `%.>% formula %>% deparse1` yields its formula as a character string. For example, `glm_nuclear %.>% formula %>% deparse1 = "ler ~ pra + day * reactor_name"`.
- `glm_est` should have 627 rows and six columns: `term`, `estimate`, `std.error`, `statistic`, `p.value`, and `specification`.

The row of `glm_est` with `term = "praTRUE"` and `specification = "ler ~ pra + reactor_name * day + shut_refuel"` should have `estimate = -0.150`. Thus, our first GLM indicates that implementing a PRA reduces the LER rate by $1 - \exp(-0.150) = 13.9\%$. Our other GLMs yield similar results.9

**Exercise 6.19.** We will now run a streamlined version of the regressions that Blanco et al. [2019] reported in Table 8.

- Define the following regression formulas:10
- Use `tib_nuclear` and `map()` to create a GLM model with each formula, in the fashion of exercise 6.17.
- Use `map_df()` to apply `tidy()` to each GLM model, in fashion of exercise 6.18.
  - As before, save the regression formula in a column called `specification`.
- Use `filter()` to isolate the "praTRUE" coefficient estimates.
  - All the estimates should be significantly negative.

**Exercise 6.20.** We will now conduct a counterfactual analysis to gauge what the LER rate would have been without the PRAs. We will use the third GLM in `glm_list`.

9Our results differ from those presented in Table 2 of Blanco et al. [2019] because they “report the marginal effects of the results,” whereas we report the $\hat{\beta}$ coefficient estimates.

10We’ll use `day + reactor_name` rather than `day * reactor_name`, to streamline the regressions.
• Start with `tib_nuclear` and filter out the `pra = FALSE` observations.
• Use `mutate()` to define `no_pra = FALSE`.
• Use `pivot_longer()` to pull down the `pra` and `no_pra` columns.
  – Use options `values_to = "pra"` and `names_to = "scenario"`.
• Call the resulting tibble `tib_cf` (since it stores the counterfactual data).
• `pluck()` the third element of `glm_list` and pipe the result into
  `augment(newdata = tib_cf, type.predict = "response")`.
• Plot the distribution of `.fitted` with `geom_density(alpha = .35)`.
  – Include `fill = scenario` in the `aes()` call, to distinguish between "pra" and
    "no_pra" distributions.

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 6.21.** 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 `tidy()` to view the coefficient estimates.

The gamma regression you ran in exercise 6.21 should have a statistically insignificant
"praTRUE" coefficient estimate. So we fail to reject the null hypothesis that PRAs had no
influence on employee radiation exposure levels.

### 6.4 Solutions

6.1

```r
shooting_star %>%
  mutate(
    sky_glow_decile = ntile(sky_glow, 10),
    sky_glow_decile = as.factor(sky_glow_decile),
    sky_glow_decile = fct_relabel(
        sky_glow_decile,
        ~ str_c("Decile: ", .)
    )
)```

377
```r
) %>%
  ggplot() +
  aes(x = star_count) +
  geom_histogram() +
  facet_wrap(  
    vars(sky_glow_decile),
    nrow = 2
  ) +
  labs(y = "Number of Observations")
```

![Histograms for different deciles of sky glow](image)

6.2

```r
shooting_star %>%
  ggplot() +
  aes(
    x = sky_glow,
    y = star_count
  ) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_bw()
```
6.3

tib_base %>%
  pivot_longer(-mean_val) %>%
  ggplot() +
  aes(x = value) +
  geom_histogram(bins = 100) +
  facet_wrap(
    vars(mean_val, name),
    scales = "free",
    ncol = 3
  )
6.4

tib_varying_means %>%
mutate(mean_decile = ntile(mean_val, 10)) %>%
```r
pivot_longer(starts_with("var")) %>%
ggplot() +
aes(x = value) +
geom_histogram(bins = 50) +
facets_grid(
  rows = vars(mean_decile),
  cols = vars(name),
  scales = "free"
)
```

<table>
<thead>
<tr>
<th>var</th>
<th>var</th>
<th>var</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>gamma</td>
<td>normal</td>
<td>poisson</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

value
count
glm_gamma <-
  tib_varying_means %>%
  glm(
    var_gamma ~ x_1 + x_2,
    family = Gamma(link = "identity"),
    data = .
  )

6.6

glm_gamma %>%
  augment(
    newdata =
      expand.grid(
        x_1 = seq(0, 100, .01),
        x_2 = 0:10
      )
  ) %>%
  mutate(
    gamma_shape =
      glm_gamma %>%
        summary %>%
        {1/pluck(., "dispersion")},
    gamma_scale = .fitted / gamma_shape,
    prob_closure =
      pgamma(
        12,
        shape = gamma_shape,
        scale = gamma_scale,
        lower.tail = FALSE
      )
  ) %>%
ggplot() +
  aes(
    x = x_1,
    y = prob_closure,
    color = as.factor(x_2)
  ) %>%
geom_line() +
  theme_bw() +
  labs(
    y = "Probability of School Closure",
  )
color = "x_2"

6.7

glm_star <-
  shooting_star %>%
  glm(
    star_count ~ sky_glow,
    family = poisson(link = "log"),
    data = .
  )

6.8

glm_star %>%
  augment(type.predict = "response") %>%
  ggplot() +
  aes(x = sky_glow) +
  geom_point(aes(y = star_count)) +
  geom_line(
    aes(y = .fitted),
  )
```r
color = "red"
+
labs(y = "Number of shooting stars") +
theme_bw()
```

6.9

```r
glm_star %>%
augment(type.predict = "response") %>%
group_by(sky_glow, .fitted) %>%
summarise(
  num_stars = 0:9,
  prob_count = dpois(num_stars, .fitted)
) %>%
ggplot() +
aes(
  x = sky_glow,
  y = prob_count,
  color = as.factor(num_stars)
) +
geom_point() +
labs(
  y = "Probability of X Shooting Stars",
  color = "X"
) +
theme_bw()
```
6.10

```r
logistic_regression_model %>%
  augment(type.predict = "response") %>%
  ggplot() +
  aes(
    x = .fitted,
    fill = any_stars_observed
  ) +
  geom_density(alpha = .3) +
  labs(
    x = "Prob. of Shooting Star",
    y = "Density",
    fill = "Shooting Star Observed:"
  ) +
  theme_bw() +
  theme(legend.position = "top")
```
6.11

tib_hypothetical <-
  shooting_star_binary %>%
  mutate(double_glow = 2 * sky_glow) %>%
  rename(current_glow = sky_glow) %>%
  pivot_longer(
    c(current_glow, double_glow),
    values_to = "sky_glow"
  )

logistic_regression_model %>%
  augment(
    type.predict = "response",
    newdata = tib_hypothetical
  ) %>%
  ggplot() +
  aes(
    x = .fitted,
    color = name
  ) +
  geom_density() +
  labs(
    x = "Probability of Seeing any Shooting Stars",
    y = "Distribution Density",
    color = ""
  ) +
Probability of Seeing any Shooting Stars

Distribution Density

current_glow
double_glow
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 sheet\(^1\) 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.

\(^1\)https://github.com/rstudio/cheatsheets/blob/master/sf.pdf
\(^2\)Other useful reference materials are Geocomputation with R\(^3\), Using Spatial Data with R\(^4\), Drawing beautiful maps programmatically with R, sf and ggplot2\(^5\), Spatial Manipulation with sf: Cheat Sheet\(^6\), 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.

### 7.2 Lecture

#### 7.2.1 Geometric Objects

Download and load the sf and spData packages with the following:

```r
install.packages(sf)
install.packages(spData)
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()`:
A MULTILINESTRING is a sfg object that describes a collection of squiggly lines:

```r
multi_crooked_lines <- list(
  runif(20) %>%
    matrix(ncol = 2),
  runif(40, min = 1.5, max = 4) %>%
    matrix(ncol = 2)
) %>%
st_multilinestring

class(multi_crooked_lines)
```

> [1] "XY" "MULTILINESTRING" "sfg"
We can analogously create POINT, MULTIPOINT, POLYGON, and MULTIPOLYGON sf objects with \texttt{st\_point()}, \texttt{st\_multipoint()}, \texttt{st\_polygon()}, and \texttt{st\_multipolygon()} options. As expected, we define a POINT object with a single (x, y) coordinate pair (e.g., \texttt{st\_point(c(0, 1))}) and a MULTIPOINT object with a matrix of coordinate pairs (e.g., \texttt{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:
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))
)
```
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:

mini_tibble <-
tibble(
  random_data = 1:3,
  dots = point_var
)

mini_tibble

> # A tibble: 3 x 2
given
> random_data dots
>   <int> <MULTIPOINT>
> 1 1 ((0.1716879 0.5914912), (0.06471741 0.6297856), (0.4871732 0.5387~
> 2 2 ((0.8771212 0.9254937), (0.9677966 0.8335996), (0.8786393 0.80876~
> 3 3 ((0.5981177 0.9425327), (0.6431626 0.04683751), (0.6578744 0.3354~

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():

mini_sf <-
  mini_tibble %>%
  st_as_sf

mini_sf
Simple feature collection with 3 features and 1 field
Geometry type: MULTIPOINT
Dimension: XY
Bounding box: xmin: 0.009352818 ymin: 0.01195526 xmax: 0.9745359 ymax: 0.9910483
CRS: NA
# A tibble: 3 x 2
random_data  dots
<int> <MULTIPOINT>
1 ((0.1716879 0.5914912), (0.06471741 0.6297856), (0.4871732 0.5387~
2 ((0.8771212 0.9254937), (0.9677966 0.8335996), (0.8786393 0.80876~
3 ((0.5981177 0.9425327), (0.6431626 0.04683751), (0.6578744 0.3354~

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. 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.

---

7EPSG stands for the European Petroleum Survey Group, a defunct organization.
• `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))
```

```
# 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.
```

```r
# sf summarise()
world %>%
  group_by(continent) %>%
  summarise(area = sum(area_km2))
```

```
Simple feature collection with 8 features and 2 fields
  Geometry type: MULTIPOLYGON
  Dimension:      XY
  Bounding box:  xmin: -180 ymin: -90 xmax: 180 ymax: 83.64513
  Geodetic CRS:  WGS 84
  # A tibble: 8 x 3
```
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 \( \text{st_intersection}(x, y) \) 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 \( \text{st_intersection()} \) of \( \text{nz\_sex} \) and \( \text{nz\_wealth} \) (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() +
  facet_grid(
    rows = vars(richest_part),
    cols = vars(mostly_men)
  ) +
  theme_bw()
```
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.

---

8You can remove this border line with st_collection_extract("POLYGON"), which discards all geometries besides enclosed polygons.
Exercise 7.6. We will now plot the region of New Zealand that’s within 100 Km of any of its 101 tallest peaks.

- 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 `t50_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
)
```
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 \texttt{st_convex_hull()} and \texttt{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 \texttt{st_convex_hull()} and \texttt{summarize()} 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 Import

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/AAAvjR01cB8Wd_oVaUuUZH7na?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 \text{transshipment\_mmsi} as \text{mmsi} and \text{starting\_timestamp} as \text{time}, and (iii) drop all variables besides \text{mmsi}, \text{lat}, \text{long}, and \text{time}.

• Modify encounter to look like loiter (i.e., with columns \text{mmsi}, \text{lat}, \text{long}, and \text{time}).

• Use \text{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 \text{st\_as\_sf(., coords = c(x = "long", y = "lat"))} to turn incidents into an sf table.

• Use \text{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 \text{anti\_join()} to remove the ships in \text{authorized\_vessels} from \text{all\_vessels}. Call the resulting tibble \text{unauthorized\_vessels}
  
  – Join the tibbles by IMO number.

• Use \text{inner\_join()} to drop the observations in incidents that do not correspond to a ship in \text{unauthorized\_vessels}.
  
  – Join the tibbles by MMSI number.

**Exercise 7.15.** Our sf tables are cumbersomely large. For example, calling \text{plot(EEZ)} will probably crash your R session. Use \text{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.
- Use `summarise()` to flatten the zones that do not ban transshipments into an sf table with one row and one column called `no_ban_region`.
- Use `st_difference()` to create an analogous sf table called `ban_region` that describes the area in RFMO that’s not in `no_ban_region`. This object should also have one row and one column.
  - Ignore the warning message you get on this step.
- Use `bind_rows()` to combine `no_ban_region` and `ban_region` into a single sf table, which inherits the name RFMO. Give this table a logical variable called `ban` that indicates whether transshipments are banned in this zone.
  - Define the `ban` variable in `no_ban_region` and `ban_region` before combining these tables.
  - RFMO should have two columns and two rows.

**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.

- Use `summarise()` to flatten EEZ to a single row.
- Use `st_difference()` to remove the area in RFMO that belongs to the flattened EEZ.
  - Ignore the warning message you get on this step.
  - RFMO should be two-by-two before and after this step.

**Exercise 7.18.** Use `st_intersection()` to add the `ban` column from RFMO to `incidents`. This variable indicates whether an incident occurred in a region where transshipments are banned.

- This step will probably take your computer a few minutes.
- This step should reduce the size of `incidents` from 32,084 rows to 12,310 rows, since most incidents occur in region not described by RFMO. For example, the `st_intersection()` drops all incidents that occur in EEZs, which we have removed from RFMO.

### 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.

**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 `geom_sf()` calls. The first layer will illustrate the `ban = TRUE` and `ban = FALSE` polygons of RFMO. And the second layer will illustrate the event points of `incidents`.

408
• 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.
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

```r
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_intersection <- st_intersection(nz, nz_height)

> Simple feature collection with 7 features and 2 fields
> Geometry type: GEOMETRY
> Dimension: XY
```
> Bounding box: xmin: 1204143 ymin: 5048309 xmax: 1822492 ymax: 5650492
> Projected CRS: NZGD2000 / New Zealand Transverse Mercator 2000
> Name n geom
> 1 Canterbury 70 MULTIPOINT ((1365809 516894...)
> 2 Manawatu-Wanganui 2 MULTIPOINT ((1820643 564833...)
> 3 Marlborough 1 POINT (1654820 5351665)
> 4 Otago 2 MULTIPOINT ((1234725 504830...)
> 5 Southland 1 POINT (1204143 5049971)
> 6 Waikato 3 MULTIPOINT ((1820660 564948...)
> 7 West Coast 22 MULTIPOINT ((1259702 507657...)

7.6

```r
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

```r
world %>%
  st_difference(
    filter(., continent == 'Asia') %>%
    st_convex_hull %>%
    summarise
  ) %>%
ggplot() +
  geom_sf() +
  theme_bw()
```

7.8

```r
world %>%
  st_difference(
    filter(., name_long %in% c('Ecuador', 'Japan')) %>%
    summarise %>%
    st_convex_hull
  ) %>%
  ggplot() +
  geom_sf() +
  theme_bw()
```
Bibliography


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 Bray11. 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/
Supplemental Materials

Data Files

Here’s a list of the data sets we will study:

- Civil war data\textsuperscript{12} used in Section 1.2.6.
- Netflix data\textsuperscript{13} used in Section 2.2.
- Courthouse data\textsuperscript{14} used in Section 2.3.
- UFC data\textsuperscript{15} used in Section 2.3.
- Package-delivery data\textsuperscript{16} used in Section 3.2.
- Chess data\textsuperscript{17} used in Section 4.2.
- Supermarket data used in Section 4.3:
  - product category data\textsuperscript{18},
  - order date data\textsuperscript{19},
  - DC inventory data\textsuperscript{20},
  - store inventory data\textsuperscript{21},
  - order data\textsuperscript{22}, and
  - shipment data\textsuperscript{23}
- Wine data\textsuperscript{24} used in Section 5.2. Click on the “Download” button and use the winemag-data-130k-v2.json file.
- Auto data\textsuperscript{25} used in Section 5.3.
- Shooting star data\textsuperscript{26} used in Section 6.2.
- Nuclear power data used in Section 6.3:
  - facility data\textsuperscript{27},

\textsuperscript{12}https://www.dropbox.com/s/56067hal7fpfps5/CWSAC_civil_war_data.csv?dl=1
\textsuperscript{13}https://www.dropbox.com/s/brcz45ouugd3fc/netflix.csv?dl=1
\textsuperscript{14}https://www.dropbox.com/s/6nbtr8q4tk12lt/court_data.csv?dl=1
\textsuperscript{15}https://www.dropbox.com/s/le5qs3ty5vk5hs/ufc.csv?dl=1
\textsuperscript{16}https://www.dropbox.com/s/fh25g7fumjmief/aliibaba.csv?dl=1
\textsuperscript{17}https://www.dropbox.com/s/0fy8gfokd48tlak/chess_games.csv?dl=1
\textsuperscript{18}https://www.dropbox.com/s/p2hkg2z8w2h3evid/category.csv?dl=1
\textsuperscript{19}https://www.dropbox.com/s/vprgd8ddo3e4bnf/order_dates.csv?dl=1
\textsuperscript{20}https://www.dropbox.com/s/uuwsp5v7qd66ng/inv_DC.csv?dl=1
\textsuperscript{21}https://www.dropbox.com/s/zncvi0f0y3hhwrtex/inv_store.csv?dl=1
\textsuperscript{22}https://www.dropbox.com/s/crlqxr2y4nqm7pa/order_store.csv?dl=1
\textsuperscript{23}https://www.dropbox.com/s/s1zjrjm5jlgf83db/ship_store.csv?dl=1
\textsuperscript{24}https://www.kaggle.com/znycide/wine-reviews
\textsuperscript{25}https://www.dropbox.com/s/6jvy7njb41lgxgf/car_data.csv?dl=1
\textsuperscript{26}https://www.dropbox.com/s/umbc3wnbh0maou/shooting_star.csv?dl=1
\textsuperscript{27}https://www.dropbox.com/s/2fw57ayxtrgja20/facility.csv?dl=1
• event data\textsuperscript{28},
• power data\textsuperscript{29},
• radiation data\textsuperscript{30}, and
• shutdown data\textsuperscript{31},

• Fishing data\textsuperscript{32} used in Section 7.3.

Notes and Videos

The following videos will help you set up your R environment:

• This video\textsuperscript{33} illustrates how to download R, RStudio, and the tidyverse. It covers everything you need to open your first Rmarkdown file.

• This video\textsuperscript{34} walks you through the basics of RStudio.

The following Rmarkdown notes cover the lecture materials in an interactive fashion. The first video above explains how to open these .Rmd files.

• Notes\textsuperscript{35} for chapter 1.

\begin{footnotesize}
\textsuperscript{28}https://www.dropbox.com/s/f07kbtpu4gk2cdc/ler.csv?dl=1
\textsuperscript{29}https://www.dropbox.com/s/zgsyt7zgl0dplc/power.csv?dl=1
\textsuperscript{30}https://www.dropbox.com/s/4s8shd41b4e228c/radiation.csv?dl=1
\textsuperscript{31}https://www.dropbox.com/s/7j5yu6h3tehjm7z/shutdown.csv?dl=1
\textsuperscript{32}https://www.dropbox.com/sh/hh4e1c8ty12nyqf/AAAavjR01c8Wd_eVaUuO3ZH7na?dl=0
\textsuperscript{33}https://www.dropbox.com/s/ikw8vbga7r8g59/open_first_rmarkdown.mov?dl=0
\textsuperscript{34}https://www.dropbox.com/s/6lnsd98me6q3g0j/rstudio_tour.mov?dl=0
\textsuperscript{35}https://www.dropbox.com/s/ff1o4xnu60ncsey/slides_data_structures.Rmd?dl=0
\end{footnotesize}