Robert L. Bray

Homage to R
Learn the Language of Data Science
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 .......................... 7
    1.2.4 Logicals, Doubles, and Integers .. 13
    1.2.5 Character Strings, Dates, and Factors 17
    1.2.6 Tibbles .......................... 25
  1.3 Lab .................................. 31
  1.4 Commentary ........................... 34
    1.4.1 Saving and Loading Data .......... 34
    1.4.2 Automating Code ................... 35

2 Functions ............................... 37
  2.1 Introduction .......................... 37
  2.2 Lecture .............................. 38
    2.2.1 Calling Functions ................ 38
    2.2.2 Functional Assembly Lines ...... 40
    2.2.3 Defining Functions ............... 47
    2.2.4 mutate() .......................... 50
    2.2.5 summarise() ....................... 53
    2.2.6 group_by() ....................... 56
    2.2.7 filter() ........................... 61
  2.3 Lab .................................. 64
    2.3.1 Overview .......................... 64
    2.3.2 Sample ............................ 65
    2.3.3 Clean ............................. 66
    2.3.4 Analyze ........................... 69
  2.4 Commentary ........................... 72
    2.4.1 Specialized Functions ........... 72
    2.4.2 Nest Functions .................... 74
    2.4.3 Pipe into Bracketed Expressions .. 76
    2.4.4 Define Multivariate Functions .. 80
    2.4.5 Download Packages from CRAN .. 82
    2.4.6 Standardize Objects to Standardize Code 84
# Contents

2.5 Solutions ................................................. 86

3 **Pivots and Plots** .............................................. 95
   3.1 Introduction ........................................... 95
   3.2 Lecture ................................................ 97
      3.2.1 Data .................................................. 97
      3.2.2 Plotting with `ggplot()` ......................... 98
      3.2.3 Pivoting with `pivot_longer()` and `pivot_wider()` ... 111
   3.3 Lab .................................................... 130
      3.3.1 Overview .......................................... 130
      3.3.2 Sample ........................................... 131
      3.3.3 Clean ........................................... 132
      3.3.4 Analyze .......................................... 134
   3.4 Commentary ........................................... 137
      3.4.1 Saving Plots .................................... 137
      3.4.2 Benefits of Long Tibbles ......................... 137
      3.4.3 Aggregating Pivots ............................. 143
      3.4.4 Invert Dependency in Plots ..................... 148
   3.5 Solutions .............................................. 148

4 **Joins** .......................................................... 163
   4.1 Homework ............................................... 163
      4.1.1 Readings .......................................... 163
      4.1.2 Exercises ......................................... 164
   4.2 Lecture ................................................ 166
      4.2.1 Tidy-Select ...................................... 166
      4.2.2 Multivariable Operations ....................... 168
      4.2.3 Mutate with External Data ...................... 171
      4.2.4 Filter with External Data ...................... 176
   4.3 Commentary ........................................... 184
      4.3.1 Standardize Rows ................................ 184
      4.3.2 Lengthening Can Simplify ...................... 186
      4.3.3 Reiterate Groupings .......................... 190
   4.4 Lab .................................................... 191
      4.4.1 Overview .......................................... 191
      4.4.2 Sample ........................................... 193
      4.4.3 Clean ........................................... 194
      4.4.4 Analyze .......................................... 202
   4.5 Solutions .............................................. 205

5 **Generalized Linear Models** .................................. 217
   5.1 Homework ............................................... 217
      5.1.1 Wrangling Techniques ............................ 217
      5.1.2 Limits of Linear Models ....................... 223
   5.2 Lecture ................................................ 225
# Contents

5.2.1 Constraining the Expected Value .............................. 225
5.2.2 Constraining the Distribution ............................... 227
5.2.3 Binary Data ............................................. 230
5.2.4 Counterfactuals ......................................... 233
5.3 Lab ......................................................... 234
5.3.1 Overview ............................................... 234
5.3.2 Sample .................................................. 236
5.3.3 Clean .................................................... 237
5.3.4 Analyze .................................................. 238
5.4 Commentary ................................................ 244
5.4.1 Flexible-Input Functions ................................ 244
5.4.2 Don’t Overburden Metadata ............................... 244
5.4.3 Be Wary of Too-Good-to-be-True Results .................. 245
5.4.4 Use Many Starting Points ................................. 250
5.5 Solutions ................................................... 251

6 Spacial Data .................................................................. 261
6.1 Homework .......................................................... 261
6.1.1 Preparation .................................................. 261
6.1.2 Study Questions ............................................. 262
6.2 Lecture .............................................................. 262
6.2.1 Introduction ............................................... 262
6.2.2 Geometric Objects ....................................... 263
6.2.3 Geometric Tables ........................................... 268
6.2.4 Geometric Joins ............................................ 273
6.3 Lab ................................................................. 278
6.3.1 Overview .................................................. 278
6.3.2 Import ....................................................... 279
6.3.3 Clean ....................................................... 280
6.3.4 Analyze ..................................................... 282
6.4 Solutions .......................................................... 283
1

Data Structures

1.1 Introduction

The R language is comprised of 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 \( \text{str_count()} \) function counts the number of characters in a text string, so that \( \log(2) = 0.6931472 \) and \( \text{str_count("There's time for everything except the things worth doing") = 57}. \)

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 object, 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., \( \text{TRUE} \) and \( \text{FALSE} \)). We will cover R’s primary data types in Sections 1.2.4 and 1.2.5. 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 structure 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, lastly, I’ll conclude with two points in Section 1.4. First, I’ll cover the right way to load and save tibbles and then I’ll explain the importance of writing self-sufficient code, which can be run in full without human intervention.

To prepare for the lecture in the next section, download and install R and RStudio onto your computer. Here’s\(^1\) a good resource that will walk you through

\[^1\]https://learnr-examples.shinyapps.io/ex-setup-r/#section-welcome
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\textsuperscript{2}, and see the additional links provided here\textsuperscript{3}.

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 comprised 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).\textsuperscript{4} The console pane should have a “Console” tab written somewhere along the top. You can submit little snippets of code here and R will execute them immediately. This pane isn’t for proper programming; rather it’s for submitting quick and cheap queries that you don’t want saving. For example, run the following in your console (while connected to the internet):

```
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:

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

\textsuperscript{2}https://rstudio.com/resources/cheatsheets/
\textsuperscript{3}https://education.rstudio.com/learn/beginner/
\textsuperscript{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.
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 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_c(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 Cnd+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 to get the the tidyverse manual, and now enter `?str_which` to see the `str_which()` 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. For example, copy the following into your source pane:

---

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.
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, 288,
    "Estonian Soviet S.R.", 2, 45,
    "Georgian Soviet S.R.", 5, 70,
    "Kazakh Soviet S.R.", 17, 2717,
    "Kirghiz Soviet S.R.", 4, 199,
    "Latvian Soviet S.R.", 3, 65,
    "Lithuanian Soviet S.R.", 4, 65,
    "Moldavian Soviet S.R.", 4, 34,
    "Russian Soviet Federative S.R.", 147, 17075,
    "Tajik Soviet S.R.", 5, 143,
    "Turkmen Soviet S.R.", 4, 488,
    "Ukrainian Soviet S.R.", 52, 604,
    "Uzbek Soviet S.R.", 20, 447
  )

ussr %>%
  mutate(
    pop_density = pop_1989/area_1991,
    socialist_republic =
      str_remove(
        socialist_republic,
        " Soviet" #Remove "Soviet" to simplify plot
      )
  ) %>%
  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)`
1.2 Lecture

so that 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 or not 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). They keyboard shortcut tells R to rerun the bit of code that you most recently executed. This is an essential hotkey—memorize it now, or give up on learning R all together.

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. For example, we can use it evaluate expressions like

\[(2+2) \times (6 + 2)\]

\[\rightarrow [1] 32\]

Or like
\[
\log(4)^2 + \cos(3) / (\exp(4) + \exp(5))
\]

\[> [1] 1.916936\]

We can also give numbers names, so that 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 \texttt{debt.2012}:

\[
\texttt{debt.2012 <- 16066241407385}
\]

Using the R parlance, we say that the code above creates a \textit{variable} called \texttt{debt.2012} that equals the number 16066241407385. The following code analogously defines variables \texttt{debt.2016} and \texttt{debt.2020} as the US national debts on 09/30/2016 and 09/30/2020, respectively:

\[
\texttt{debt.2016 <- 19573444713936}
\]
\[
\texttt{debt.2020 <- 26945391194615}
\]

Henceforth, we can use variables \texttt{debt.2012, debt.2016,} and \texttt{debt.2020} and numbers 16066241407385, 19573444713936, and 26945391194615 interchangeably. For example, the following two expressions are equivalent:

\[
(\texttt{debt.2020 - debt.2016}) / (\texttt{debt.2016 - debt.2012})
\]

\[> [1] 2.181944\]

\[
(\texttt{26945391194615-19573444713936}) / (\texttt{19573444713936-16066241407385})
\]

\[> [1] 2.181944\]

However, the former expression is easier to interpret, because descriptive labels such as \texttt{debt.2016} are more telling than long strings of digits such as \texttt{19573444713936}.

The following defines \texttt{pop.2012, pop.2016,} and \texttt{pop.2020} as the US population in 2012, 2016, and 2020:

\[
\texttt{pop.2012 <- 313.83 * 10^6}
\]
\[
\texttt{pop.2016 <- 322.94 * 10^6}
\]
\[
\texttt{pop.2020 <- 330.66 * 10^6}
\]

And the following redefines variables \texttt{debt.2012, debt.2016,} and \texttt{debt.2020} to equal the per capita debt:
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 both to the left and the 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.** Try anticipating what the following three chunks of code will output and then execute the code in R 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 <- 8.257 # debt measured in trillions of dollars
pop_1950 <- 76.1 # pop measured in millions of people
```
```r
debt_1951 <- 0.255
pop_1951 <- 77.6

debt_1952 <- 0.259
pop_1952 <- 79.2

# ...

debt_2020 <- 26.95
pop_2020 <- 338.7
```

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 we would require the following 142 transformations:

```r
debt_1950 <- debt_1950 * 10^12
pop_1950 <- pop_1950 * 10^6

debt_1951 <- debt_1951 * 10^12
pop_1951 <- pop_1951 * 10^6

# ...

debt_2020 <- debt_2020 * 10^12
pop_2020 <- pop_2020 * 10^6
```

And reexpressing 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 concatenate function `c()`:

```r
debt <- c(
```
The code above defines `debt` as a vector comprising our 71 debt figures and defines `pop` as a vector comprising our 71 population figures. For example, the second element of `debt` is `debt[2] = 0.255` and the fourth element of `pop` is `pop[4] = 160.2`.

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
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
```

In the code above `head()` is a function that returns the first six elements of a vector, and in the output above `e+11` stands for `10^11` (thus, the first number in `debt` is `debt[1] = 2.57e+11 = 257000000000`). And with two more steps we can change the unit of measure of `pop` from millions of people to people, and reexpress the debt in per capita terms:
```r
pop <- pop * 10^6
head(pop)

> [1]  152300000  154900000  157600000  160200000  163000000  165900000

debt <- debt / pop
head(debt)

> [1]  1687.459  1646.223  1643.401  1660.424  1662.577  1651.597


Now since debt encompass all debt values, we can pass this one vector into various R functions to analyze the history of US debt:

```min(debt)```

> [1] 1573.217

```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
Note, in addition to being a vector, \texttt{debt} is also variables. A variable is any named object that’s defined with \texttt{<-}. For example, setting \( x \leftarrow 0.99 \) defines variable \( x \) equal to number \( 0.99 \) and setting \( y \leftarrow c(1, 2, 3) \) defines variable \( y \) equal to vector \( c(1, 2, 3) \). Thus defined, \( x \) and \( y \) are now interchangeable with \( 0.99 \) and \( c(1, 2, 3) \). For example, the following four code chunks yield the same results:

\begin{verbatim}
#Chunk 1
0.99 + c(1, 2, 3)


#Chunk 2
x + c(1, 2, 3)


#Chunk 3
0.99 + y


#Chunk 4
x + y

\end{verbatim}

In addition to concatenation function \texttt{c()} we can create vectors with sequence vector \texttt{seq()}. For example, the following creates a vector comprising the numbers between 1 and 10:

\begin{verbatim}
seq(1, 10)

> [1]  1  2  3  4  5  6  7  8  9 10
\end{verbatim}

The following creates a vector comprising the numbers from 5 to 100, counting by 8:

\begin{verbatim}
seq(5, 100, by = 8)

> [1]  5 13 21 29 37 45 53 61 69 77 85 93
\end{verbatim}

\footnote{The keyboard shortcut for writing the \texttt{-} symbol is “Alt -” with Windows and Linux and “Option -” with Macs.}
And the following creates a vector of five evenly spaced numbers between -.001 and .02

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

> [1] -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:

```r
year <- 1950:2020
year
```


The following exercises provide additional examples.

**Exercise 1.2.** 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 “not applicable” symbol NA indicates an empty slot. Use this function to define `debt_growth` as the vector of year-over-year fractional debt changes. The first element of `debt_growth` should be `debt_growth[1] = NA`, the second element should be `debt_growth[2] = (debt[2] - debt[1]) / debt[1]`, the third element should be `debt_growth[3] = (debt[3] - debt[2]) / debt[2]`, etc.

- Hint: Your solution should be one short line of code.

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

**Exercise 1.3.** Try anticipating what the following three chunks of code will output and then execute the code to confirm your answer.

```r
#Chunk 1
x <- 1:6
x^2

#Chunk 2
x <- 1:6
2^x
```
1.2 Lecture

#Chunk 3
```r
x <- seq(100, 200, by = 25)
y <- c(0, 0, 1, 2, 2)
```

1.2.4 Logicals, Doubles, and Integers

Here’s a weird fact about R: the language has multiple types of number. Specifically, it has three basic number types: integers, doubles, and logics. An integer is a counting number: e.g., 0, 5, 3,000, and -132,432,345 are integers. A double is a decimal number: e.g., 0.5, -10000.00001, and 3.141593 are doubles. 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 as it enables us to apply `sum()` and `mean()` to vectors of logics to get the number of elements that are TRUE and the fraction of elements that are TRUE.

Let me illustrate with a concrete application: If `debt_growth <- (debt - lag(debt)) / lag(debt)` is a vector of year-over-year debt changes (see Exercise 1.2) then `debt_increase <- debt_growth > 0` is a vector of logics that indicate whether the debt increased in a given year (i.e., whether the debt change was positive). For example, compare the first six values of `debt_growth` with the first six values of `debt_increase`:

```r
head(debt_growth)
```

> [1] NA -0.024436501 -0.001714442 0.010358673 0.001296185
> [6] -0.006603809

```r
head(debt_increase)
```

> [1] TRUE TRUE 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:
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 greater-than symbol `>` to convert numeric vector `debt_growth` to logical vector `debt_increase`. We can also create logicals with greater-than-or-equal-to symbol `>=`, exactly-equal-to symbol `==`, less-than-or-equal-to symbol `<=`, less-than symbol `<`, and not-equal-to symbol `!=`. For example:

```r
x <- 1:5
x

> [1] 1 2 3 4 5
x > 3

> [1] FALSE FALSE FALSE TRUE TRUE
x >= 3

> [1] FALSE FALSE TRUE TRUE TRUE
x == 3

> [1] FALSE FALSE TRUE FALSE FALSE
x <= 3

> [1] TRUE TRUE TRUE FALSE FALSE
x < 3
```
1.2 Lecture

> [1] TRUE TRUE FALSE FALSE FALSE

x != 3

> [1] TRUE TRUE FALSE TRUE TRUE

Note, 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.

Also useful are any() and all():

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

> [1] TRUE

any(FALSE, FALSE, FALSE, FALSE, FALSE)

> [1] FALSE

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

> [1] FALSE

all(TRUE, TRUE, TRUE, TRUE, TRUE)

> [1] 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:

as.integer(2.6)

> [1] 2

By default, R sets all numbers to doubles:

is.double(2)

> [1] TRUE
is.integer(2)

> [1] FALSE

To define a number as an integer, use `as.integer()` or write an “L” after it: e.g., `100L`.\(^7\)

Now, 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:

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

Variable `number_killed` should be exactly ten. And it looks like it is:

```r
number_killed

> [1] 10
```

But it’s not:

```r
number_killed == 10

> [1] FALSE
```

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` differs from ten by `0.00000000000001776`:

```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 resolves the problem:

---

\(^7\)This “L” here stand 32-bit long integer.
1.2 Lecture

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

number_killed <= 10

> [1] TRUE
```

This rounding issue is relatively common, so you should use integers when possible, reserving doubles for when you actually need the decimal expansion.

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 stringr package—which was automatically loaded when we called library(tidyverse)—provides many functions for manipulating character strings. All stringr functions start with prefix str_; e.g., str_detect(), str_locate(), str_extract(), str_count(), and str_replace() detect, locate, extract, count, and replace substrings embedded in character strings. And str_replace_na() replaces NA terms—i.e., missing data—in character string vectors:

str_replace_na(c('a', 'b', NA, 'd', 'e', NA, 'g'), '')
1.2 Lecture

> [1] "a" "b" "c" "d" "e" "f" "g"

Note, in the code above "" is an empty string—i.e., a character string that comprises zero characters.

For a more complex example, the following defines logical vector `multiple_battle_name` by identifying the elements of `battle_name` that contain the substring " or ":

```r
multiple_battle_name <-
  str_detect(  
    battle_name,
    " " or ""
  )
```

> [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE

This vector tells us that the last two elements of `battle_name`—"First Battle of Bull Run or First Manassas" and "Battle of Wilson's Creek or Oak Hills"—are the only battles with multiple names. And the following redefines `battle_name` to remove all the instances of "Battle of ":

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

> [1] "Fort Sumter" "Sewell's Point"
> [3] "Aquia Creek" "Philippi (West Virginia)"
> [5] "Big Bethel" "Boonville (Missouri)"
> [7] "Hoke's Run" "Carthage"
> [9] "Rich Mountain" "Blackburn's Ford"
> [11] "First Bull Run or First Manassas" "Wilson's Creek or Oak Hills"

We could save the battle start and end dates as character strings:

```r
battle_start <-
c(  
  "April 12, 1861", "May 18, 1861", "May 29, 1861",
  "June 3, 1861", "June 10, 1861", "June 17, 1861",
```

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

But it would be better if we saved battle_start and battle_end as date
variables, because then we could apply R’s calendar functions to them. We
can convert these calendar string vectors to date vectors with mdy():

battle_start <-
  mdy(battle_start)
battle_end <-
  mdy(battle_end)

Note, we use mdy() because our initial character strings expressed dates with
a month/day/year format; in contrast, we would have used dmy() had they
followed a day/month/year format, 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 battle_start and battle_end. For example, we can difference
these vectors to calculate the battle durations:

battle_end + days(1) - battle_start

> Time differences in days
> [1] 2 2 4 1 1 1 1 1 1 1

We add days(1) to 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 battle_start and battle_end were
character string vectors: e.g., subtracting character string "May 29, 1861" from
character string "June 1, 1861" makes as much sense as subtracting "Donald
Duck" from "Bugs Bunny".

---

8We always refer to date variables with a capital “D”, so that we can refer to dates with
the character "d" and refer to doubles with the character "d".
Functions \texttt{mdy()} and \texttt{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

\begin{verbatim}
today() - battle_end
\end{verbatim}

> Time differences in days
> [1] 58364 58328 58315 58306 58299 58284 58281 58275 58268 58265 58245

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

\begin{verbatim}
floor_date(battle_start, "week")
\end{verbatim}

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

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

\begin{verbatim}
day_of_week <-
  wday(battle_end, label = TRUE)
day_of_week
\end{verbatim}

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

All of these functions are from lubridate.

In the last example above, \texttt{day_of_week} looks like a vector of character strings, but it’s actually a vector of \texttt{factors}. A factor variable is a character variable that’s restricted to a specific set of values, or “levels.” For example, \texttt{day_of_week} has seven valid levels: "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", and "Sat". Constraining this variable to these seven values imposes a useful safeguard on our data. For example, we can’t change the first element of \texttt{day_of_week} to "WWed", which isn’t a valid level, but we can change it to "Wed", which is a valid level:

\begin{verbatim}
day_of_week[1] <- "WWed"
day_of_week
\end{verbatim}

> [1] <NA> Sun Sat Mon Mon Mon Tue Fri Thu Thu Sun Sat
> Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
day_of_week[1] <- "Wed"
day_of_week

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

Hence, the factor structure makes this typo impossible. In contrast, if 
day_of_week were a character string variable, then there’d be nothing stop-
ning us from mistakenly setting one of its values to "\textit{WWed}".

The other benefit of factor variables is that they can have non-alphabetical
orderings. For example, the expression \texttt{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 \texttt{day_of_week} will honor this ordering:

\begin{verbatim}
ggplot() +
  aes(x = day_of_week) +
  geom_bar()
\end{verbatim}

In contrast, if we treat the elements of \texttt{day_of_week} as character strings then \texttt{R} arranges the bars \textit{alphabetically}, from "Fri" to "Wed" (which makes for a pretty lousy plot):

\begin{verbatim}
ggplot() +
  aes(x = as.character(day_of_week)) +
  geom_bar()
\end{verbatim}
While the US, 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".\(^5\)

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

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

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

\(^5\)In this code `Inf` stands for infinity, which is a valid concept in R. For example, \(1/0 = \text{Inf}\) and \(1/\text{Inf} = 0\). 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 nth position—i.e., to the n+1th position. And if we set \(n = \text{Inf}\) we’re telling R to treat “Sun” as the infinityth level, so that no matter how many levels `day_of_week` has “Sun” will be the last one.
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). This package provides a lot of useful factor-specific functions, all of which start with the prefix `fct_`. For example, the following uses `fct_recode()` to lump the "Sun" and "Sat" levels into a common "Weekend" level:

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

day_of_week
```

```
> [1] Wed 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()
```
1.2 Lecture

We’ve now covered most of the important data types. Indeed, we’ll store nearly all our data in vectors that comprise logically, 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:

\[
\text{c(1, 2, 3, "a", "b", "c")}
\]

\[
> \text{[1] "1" "2" "3" "a" "b" "c"}
\]

In this case R downgrades numbers 1, 2, 3 into character strings "1", "2", "3" to make them compatible with the rest of the vector. Hence, even if you try, R will not enable you to create vectors with multiple types.\(^\text{10}\)

1.2.6 Tibbles

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

```r
battle_casualties <-
  c(
    0, 18, 10, 30, 86, 87, 114,
    244, 346, 151, 4698, 2549
  )

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

battle_description <-
  c(
    "Beauregard takes Charleston Federal fort",
    "Union gunboats fight Confederate artillery",
    "Confederate artillery hit by naval bombardment",
  )
```

\(^{10}\text{While we can’t store multiple data types in a vector, we can store multiple data types in a list. We’ll cover this more complex data structure in Chapter \\&\\&list\\&
\&.}
"Small Confederate detachment rooted out of WV",
"Union attack near a church repelled",
"Union forces defeat Missouri State Guard",
"Patterson defeats Jackson, but squanders victory",
"Confederate victory in Missouri",
"Union forces divide Confederate forces; half surrender",
"Irvin McDowell’s recon-in-force defeated at Manassas",
"Thomas Jackson earned the moniker 'Stonewall'",
"First major battle west of the Mississippi"

Since they’re not officially related to one another it’s difficult to cross-reference the information stored in our battle vectors. For example, we can’t tell, simply by looking at the data, in which battle "Thomas Jackson earned the moniker 'Stonewall'", or which battle had the most casualties, or when the "First major battle west of the Mississippi" ended. However, we can easily answer these questions if we stack our vectors side-by-side, 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
  )
```

```r
> # A tibble: 12 x 6
>   start end   casualty victor name desc
>  <date> <date> <dbl> <chr> <chr> <chr>
> 1 1861-04-12 1861-04-13 0 Confederate  Fort Sumter  Beauregard takes Ch-
> 2 1861-05-18 1861-05-19 10 Inconclu- Sewell's Point Union gunboats fight-
> 3 1861-05-29 1861-06-01 10 Inconclu- Aquia Creek Confederate artiller-
> 4 1861-06-03 1861-06-03 30 Union  Philippi (Wes- Small Confederate de-
> 5 1861-06-10 1861-06-10 86 Confederate Big Bethel Union attack near a-
> 6 1861-06-17 1861-06-17 87 Union  Boonville (M1- Union forces defeat Ja-
> 7 1861-07-02 1861-07-02 114 Union  Hoke's Run Patterson defeats Ja-
> 8 1861-07-05 1861-07-05 244 Confederate Carthage Confederate victory -
> 9 1861-07-11 1861-07-11 346 Union  Rich Mountain Union forces divide -
>10 1861-07-18 1861-07-18 151 Confederate Blackburn's F- Irvin McDowell's rec-
>11 1861-07-21 1861-07-21 4690 Confederate First Bull Ru Thomas Jackson earne-
```
1.2 Lecture

> 12 1861-08-10 1861-08-10 2549 Confeder- Wilson's Cree- First major battle w-

The first line of this output describes `battle_data` as a *tibble* with 12 observations (i.e., rows) and 5 variables (i.e., columns). A tibble is a specific sort of data table, invented in 2016. Before that, people saved their R tables as *data frames*. Tibbles and data frames are basically the same, except the former has a nicer design sensibility. Accordingly, we will use tibbles throughout this book. (Although nothing substantive would change if we instead used data frames.)

We can inspect a tibble in multiple ways. For example, we can view the top three rows of `battle_data` with:

```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- Fort Sum- Beauregard takes Charlest-
> 2 1861-05-18 1861-05-19 10 Inconclus- Sewell's- Union gunboats fight Conf-
> 3 1861-05-29 1861-06-01 10 Inconclus- Aquia Cr- Confederate artillery hit-

We can view its bottom four rows with:

```r
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  Rich Mountain  Union forces divide C-
> 2 1861-07-18 1861-07-18 151 Confede- Blackburn's Fo- Irvin McDowell's reco-
> 3 1861-07-21 1861-07-21 4690 Confede- First Bull Run- Thomas Jackson earned-
> 4 1861-08-10 1861-08-10 2549 Confede- Wilson's Creek- First major battle we-

We can view its first two rows when its ordered alphabetically by `desc` with:

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

> # 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- Fort Su- Beauregard takes Charlesto-
1 Data Structures

> 2 1861-05-29 1861-06-01 10 Inconclus- Aquia C- Confederate artillery hit -

We can view its last three rows when its ordered by casualty with:

```r
slice_max(
  battle_data,
  order_by = casualty,
  n = 3
)
```

> # A tibble: 3 x 6
>   start   end casualty victor name desc
>  <date> <date>   <dbl> <chr> <chr> <chr>
> 1 1861-07-21 1861-07-21 4600 Confede- First Bull Run- Thomas Jackson earned-
> 2 1861-08-10 1861-08-10 2549 Confede- Wilson's Creek- First major battle we-
> 3 1861-07-11 1861-07-11 346 Union Rich Mountain Union forces divide C-

We can view a random two rows with:

```r
slice_sample(
  battle_data,
  n = 2
)
```

> # A tibble: 2 x 6
>   start   end casualty victor name desc
>  <date> <date>   <dbl> <chr> <chr> <chr>
> 1 1861-07-18 1861-07-18 151 Confede- Blackburn- Irvin McDowell's recon-in-
> 2 1861-07-05 1861-07-05 244 Confederate- Carthage Confederate victory in Mi-

We can view a compressed summary of its variables with:

```r
glimpse(battle_data)
```

> Rows: 12
> Columns: 6
> $ start <date> 1861-04-12, 1861-05-18, 1861-05-29, 1861-06-03, 1861-06-1...
> $ end <date> 1861-04-13, 1861-05-19, 1861-06-01, 1861-06-03, 1861-06-1...
> $ casualty <dbl> 0, 10, 10, 30, 86, 87, 114, 244, 346, 151, 4690, 2549
> $ victor <chr> "Confederate", "Inconclusive", "Inconclusive", "Union", "C...  
> $ name <chr> "Fort Sumter", "Sewell's Point", "Aquia Creek", "Philippi...  
> $ desc <chr> "Beauregard takes Charleston Federal fort", "Union gunboat...
1.2 Lecture

```r
summary(battle_data)
>
> start     end    casualty    victor
> Min. 1861-04-12 1861-04-13 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
>
>
> Or we can view the tibble in all its glory with view(battle_data).

There are two basic ways to define a tibble: column-wise with tibble() or row-wise with tribble(). For example, the following are two equivalent ways to define the same table:

```r
char <-
c("Tantalus", "Pelops", "Atreus", "Agamemnon",
"Menelaus", "Iphigenia", "Electra", "Orestes"
)
dad <-
c("Zeus", "Tantalus", "Pelops", "Atreus",
"Atreus", "Agamemnon", "Agamemnon", "Agamemnon"
)

mom <-
c("Plouto", "Dione", "Hippodamia", "Aeope", "Aerope",
"Clytemnestra", "Clytemnestra", "Clytemnestra"
)

tribble(
character = char,
father = dad,
mother = mom
)

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

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

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

Between the two, we will more frequently use tibble(), since it constructs tibbles from vectors (e.g., char, dad, and mom) rather than from individual elements (e.g., "Tantalus", "Zeus", "Plutus", etc.), and almost all the data we work with will be prepackaged in a vector.

However, 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 then choose the option that suits your file format (e.g., choose File/Import
1.3 Lab

Dataset/From Text (readr) if your input file is comma separated or tab delimited). After, 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\(^\text{11}\) and load it into your workspace with File/Import Dataset/From Text (readr). Doing so should print something like this to your workspace:

```
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.

And 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 an additional option to the `read_csv()` call:

```
civil_war <-
read_csv(
  "dat/CWSAC_civil_war_data.csv",
  col_types = "cDDcfcI"
)
```

The `col_types` option tells R what data type to make each column: "cDDcfcI" is short for character, Date, Date, character, factor, character, and logical.

---

1.3 Lab

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.

\(^{11}\)\url{https://www.dropbox.com/s/568007ha17fpf5s/CWSAC_civil_war_data.csv?dl=0}
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 click the “New Folder” button in the files pane to create a new folder called raw_data. You will save your .csv files here. Now press Ctrl+Shift+N (Windows or Linux) or Cmd+Shift+N (Mac) to create a new R source file. Write `library(tidyverse)` and `library(lubridate)` at the top of this file. Now press Ctrl+S or Cmd+S to save this R source file as code.R, in the project folder. You will do this entire process for each exercise below.

**Exercise 1.4.** Create an R project called netflix for the lecture in Section 2.2. Save the Netflix data found here\(^{12}\) to the project’s raw_data folder. Within this project’s code.R file, use `read_csv()` to load these data into a tibble called netflix. Then click on File/Close Project.

- **Tip:** You should be able to open “netflix.csv” with `read_csv('raw_data/netflix.csv')`.
- **Tip:** When you close your 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 1.5.** Create an R project called judges for the lab in Section 2.3. Save the courthouse data found here\(^{13}\) to this project’s raw_data folder. Within this project’s code.R file, use `read_csv()` to load these data into a tibble called court_data. Use the `col_types` = “ciccccccccc” option to have R treat the first variable as an integer, the next three as characters, the next three as logicals, and so forth. Finally, close the project.

**Exercise 1.6.** Create an R project called ufc for the lecture in Section 3.2. Save the UFC data found here\(^{14}\) to this project’s raw_data folder. Within this project’s code.R file, use `read_csv()` to load these data into a tibble called ufc. Use the `col_types` = ”i0c1lff ficiddcfd iidddddddiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii” option to assign each variable a data type. Finally, close the project.

**Exercise 1.7.** Create an R project called alibaba for the lab in Section 3.3. Save the track-package records data found here\(^{15}\) to this project’s raw_data folder. Within this project’s code.R file, use `read_csv()` to load these data into a tibble called alibaba_wide. Use the `col_types` = `cols()` option of `read_csv()`

\(^{12}\)https://www.dropbox.com/s/bcz45ouyvd3fc/netflix.csv?dl=0
\(^{13}\)https://www.dropbox.com/sh/glg647yemwkd38c/AAAqJ6yaiPw-7zuvWv6Ef3Pa?dl=0
\(^{14}\)https://www.dropbox.com/s/zh25gfjumjmi/el/alibaba.csv?dl=0
\(^{15}\)https://www.dropbox.com/s/zh25gfjumjmi/el/alibaba.csv?dl=0
to define order, score, item, brand, category, sub_category, and merchant as factors and define quantity as an integer. For example, including `col_types = c(cols(order = col_factor(), quantity = col_integer())` would tell R to treat the order variable as a factor and the quantity variable as an integer. Finally, close the project.

- Note: Although they are numeric, we’ll make most of our primary variables factors because their values, being ID numbers, are really just short-hand labels for pre-defined categories—i.e., labels for factors.
- Note: We’ll specify the data type of the other columns later.
- Check: `alibaba_wide` should have 24808 rows and 68 columns.

**Exercise 1.8.** Create an R project called `chess` for the lecture in Section 4.2. Save the chess match data found here\(^\text{16}\) to this project’s `raw_data` folder. Within this project’s `code.R` file, use `read_csv()` to load these data into a tibble called `chess_games`.

**Exercise 1.9.** Create an R project called `rationing` for the lab in Section 4.4. Save the supermarket data found here\(^\text{17}\) to this project’s `raw_data` folder. Within this project’s `code.R` file, use `read_rds()`—not `read_csv()`—to load

- `order_dates.rds` into a tibble called `order_dates`,
- `category.rds` into a tibble called `category`,
- `order_store.rds` into a tibble called `order_store`,
- `ship_store.rds` into a tibble called `ship_store`,
- `inv_store.rds` into a tibble called `inv_store`, and
- `inv_DC.rds` into a tibble called `inv_DC`.

**Exercise 1.10.** Create an R project called `glm` for the lecture in Section 5.2. Save the shooting star samples found here\(^\text{18}\) to this project’s `raw_data` folder. Within this project’s `code.R` file use `read_csv()` to load `shooting_star_1992.csv` and `shooting_star_1994.csv` into tibbles `star_92` and `star_94`.

**Exercise 1.11.** Create an R project called `radiation` for the lab in Section 5.3. Save the nuclear power data found here\(^\text{19}\) to this project’s `raw_data` folder. Within this project’s `code.R` file use `read_csv()` to load

- `facility.csv` into a tibble called `facility`,
- `power.csv` into a tibble called `power`,
- `shutdown.csv` into a tibble called `shutdown`,
- `ler.csv` into a tibble called `ler`, and
- `radiation.csv` into a tibble called `radiation`.

\(^{16}\)https://www.dropbox.com/sh/ptrf9tx1bz2128v/AAAoxtNy1z8x080hIQG6--Ta?dl=0
\(^{17}\)https://www.dropbox.com/sh/7z3tyf410cp2df2/AAAAXF_N-JOjuq56qH7ZDmQa?dl=0
\(^{18}\)https://www.dropbox.com/sh/fi3rj4pp2awz3a/AAAdOYjRvmWol6x-6TMvM72a?dl=0
\(^{19}\)https://www.dropbox.com/sh/w2bg1lqvlomouj/AAAVg382dF1Dej3CU3MCgCCha?dl=0
1.4 Commentary

1.4.1 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 tibble `save_me` in file "saved_tibble.rds":

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

write_rds(
  save_me,
  "saved_tibble.rds"
)
```

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

```r
loaded_tibble <- read_rds("saved_tibble.rds")
```

```
loaded_tibble
```

```r
> # 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:

```r
save(
  save_me,
  file = "saved_tib.rds"
)
```
#Delete tibble after saving it
rm(save_me)

load("saved_tib.rds")

#Now tibble is back, with original name
save_me

> # 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, but 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 you 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 that the tibble was defined and saved in a different source 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").

1.4.2 Automating Code

I cleaned the code that got generated from the File/Import Dataset/From Text (readr) call at the end of Section 1.2.6 for two reasons. First, I wanted to incorporate the col_types = "cDcfcfcl" 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 the 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 so 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—so that you can cross it off and forget about it—and a job that’s finished all but for one pesky detail that you’ve got to keep lingering tabs on.\(^\text{20}\)

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 simply requires pressing Ctrl+Shift+Enter. In short, if you leave your code in a set-it-and-forget-it state then you can costlessly rerun it any number of times, after any stretch of time.

\(^\text{20}\)My friend Juan Serpa tells me that that I’m too doctrinaire about this point. He reminds me that the paper we wrote together relied on quite a bit of manual data cleaning. So I’m probably being a bit of a hypocrite here.
2

Functions

2.1 Introduction

In this chapter, I will cover the verbs of the R programming language: the functions that act upon data. I’ll begin, in Section 2.2.1, by explaining how to apply functions (although you’ve probably already figured most of this out). I’ll then supercharge this discussion in Section 2.2.2 by showing you how to apply an entire sequence of functions. You’ll 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 2.2.3 I’ll demonstrate how to to turn a functional assembly line into a new stand-alone function. To do so, we’ll 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.)

I’ll end the lecture, in Sections 2.2.4–2.2.7, by covering 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.

Once you’ve mastered the functional quartet 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 mention
some specialized functions that we could have used to streamline our analysis. 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 illustrate how to connect the %>% conveyor belts to non-standard functions. In Section 2.4.4 I explain how to define functions of multiple variables. But we’ll rarely have to create such functions; instead, we usually just download them from the Comprehensive R Archive Network (CRAN). Accordingly, I provide a brief primer on downloading packages from CRAN in Section 2.4.5. And I conclude, in Section 2.4.6, 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()
```

```r
```

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 derives one 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 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"

```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"

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

```r
str_flatten(string = LETTERS)
```

> [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

```r
str_flatten(
  string = LETTERS,
  collapse = ""
)
```

> [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

But, of course, we are free to choose a different `collapse` value:

```r
str_flatten(
  string = LETTERS,
  collapse = " and "
)
```


### 2.2.2 Functional Assembly Lines

So far we’ve applied all our functions in one-off fashion. 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 you wanted to evaluate $\sqrt[3]{\sqrt{3}}$—i.e., the square root of the square root of the square root of the square root of two.$^1$ Here’s one way to do so:

```r
root_of_three <-
sqrt(x = 3)
root_of_root_of_three <-
sqrt(x = root_of_three)
root_of_root_of_root_of_three <-
sqrt(x = root_of_root_of_three)
root_of_root_of_root_of_root_of_three <-
sqrt(x = root_of_root_of_root_of_three)

> [1] 1.871075
```

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 `root_of_three`, `root_of_root_of_three`, `root_of_root_of_root_of_three`, and `root_of_root_of_root_of_root_of_three` will now junk up our working environment until we explicitly delete them (which probably won’t be for some time). And this problem is not only aesthetic: e.g., try storing multiple copies of a 100 Gb dataset! 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 <- sqrt(3)
current_val <- sqrt(x = current_val)
current_val <- sqrt(x = current_val)
current_val <- sqrt(x = current_val)
current_val

> [1] 1.871075
```

In this case, variable `current_val`—which gets redefined from `sqrt(3) = 1.732051` to `sqrt(1.732051) = 1.316074` to `sqrt(1.316074) = 1.147203` to `sqrt(1.147203) = 1.071075`—acts like a wheelbarrow, ferrying data from step to step. Unfortunately, to pass data from one function 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:

$^1$Of course, we could evaluate this with $3^{(1/16)}$, but suppose we didn’t know that.
R turns all functions connected by conveyor belts into a functional assembly line, where the output of one function is used as the input to the next. Hence, the code above passes the number 3 down an assembly line comprised of four `sqrt()` steps. Here’s another example:

```r
> [1] 1.071075
1:10 %>%
  exp %>%
  sum %>%
  log
> [1] 10.45863
```

This code tells R to pass the vector `1:10` down an assembly line comprised of an `exp()` step, a `sum()` step, and a `log()` step. Hence, it’s analogous to

```r
my_value <- 1:10
my_value <- exp(my_value)
my_value <- sum(my_value)
my_value <- log(my_value)
my_value
> [1] 10.45863
```

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

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

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

#Chunk 2
y <- fun_1(x)
```
A functional assembly line could comprise any number of functions. If it comprises just one function then \( x \ggg f \_1 \) is equivalent to \( f \_1(x) \). For example, \( 9 \ggg \text{sqrt} = 3 \). Going forward, we’ll implement most functions this way. And in English we will refer to the expression \( x \ggg f \_1 \) as “\( x \) piped into \( f \_1 \).” For example, if I ask you to “pipe \( 1:100 \) into \( \text{sum}() \)” then you should write \( 1:100 \ggg \text{sum} \).

Note that we can drop the parentheses after our functions when we string them into an assembly line. For example, \( x \ggg f \_1 \ggg \ldots \ggg f \_n \) and \( x \ggg f \_1() \ggg \ldots \ggg f \_n() \) are equivalent. However, we could still use these parentheses to pass in additional variables. For example, the following code chunks yield the same outputs:

```r
#Chunk 1
"Is it all right to boil a lobster alive?" \ggg
str\_replace("lobster", "sentient creature")

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

#Chunk 2
x <- "Is it all right to boil a lobster alive?"
x <- str\_replace(x, "lobster", "sentient creature")
x

> [1] "Is it all right to boil a sentient creature alive?"
```

The `str_replace()` function expects three inputs: primary argument `string`, secondary argument `pattern`, and tertiary argument `replacement`. Unless otherwise specified, R feeds the conveyor belt into the primary argument’s slot. Hence, in the code above R sets `string = "Is it all right to boil a lobster alive?", pattern = "lobster", and replacement = "sentient creature"`. However, you can assign the conveyor belt’s input to another argument with the . symbol. For example, consider the following

```
"nothing besides" \ggg
str\_replace(
  pattern = ",
  "There's time for nothing besides the things worth doing",
```
"everything except"
)

> [1] "There's time for everything except the things worth doing"

In this case, the pattern = . line tells R to route the conveyor belt to the pattern argument. And the other two arguments get filled in order, with "There's time for nothing besides the things worth doing" used for the primary argument and "everything except" for the tertiary argument.

Here's another example:

"Hadley" %>% paste0("", the best human ever", ", is a kiwi")

> [1] "Hadley, the best human ever, is a kiwi"

"Hadley" %>% paste0("The best human ever", ", .. ", is a kiwi")

> [1] "The best human ever, Hadley, is a kiwi"

"Hadley" %>% paste0("The best human ever", ", is a kiwi, ", .)

> [1] "The best human ever is a kiwi, Hadley"

The conveyor belt always feeds into the front unless we explicitly direct it elsewhere. Hence, R interprets the first line as paste0("Hadley", ", the best human ever", ", is a kiwi"). But the second and third lines redirect this input with the . symbol, so that R interprets them as paste0("the best human ever", ", "Hadley", ", is a kiwi") and paste0("the best human ever", ", is a kiwi, ", "Hadley"), respectively.

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

**Exercise 2.1.** Create assembly-line versions of the following two code blocks. In other words, use conveyor belts to replace wheelbarrow variable *my_string*:

```
#Code block 1
my_string <- "comrade Napoleon is always right"
my_string <- str_to_title(my_string)
my_string <- paste0("### ", my_string, "!!!", " ###")
my_string <- str_replace(my_string, "comrade", "Master")
```
2.2 Lecture

```r
my_string <- rep(my_string, 3)
my_string <- str_flatten(my_string)
my_string
```

```r
> [1] "### Master Napoleon Is Always Right!!! #### Master Napoleon Is Always Right!!! #### Master Napoleon Is Always Right!!!

#Code block 2
my_string <-
c("As I stepped out of the front door",
 "a nasty little gust of wind caught",
 "the soapy patch on my neck and made",
 "me suddenly feel that my clothes didn't",
 "fit and that I was sticky all over."
)
my_string <- word(my_string, 2)
my_string <- nth(my_string, 3)
str_count(
c("It was only when I was nearly",
 "dressed and looking for a tie that",
 "I discovered that my neck was still",
 "soapy. It's a rotten thing to have a soapy neck.",
 "It gives you a disgusting sticky feeling",
 "and the queer thing is that, however carefully",
 "you sponge it away, when you've once discovered",
 "that your neck is soapy you feel",
 "sticky for the rest of the day."
),
pattern = my_string
)
```

```r
> [1] 0 0 0 2 0 0 0 1 0
```

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

**Exercise 2.2.** The following code calculates the average number of times each character digit, from "0" to "9", appears in the past ten thousand dates:

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

> head(x)

<table>
<thead>
<tr>
<th>date</th>
<th>digits</th>
<th>digit_count</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>2023-01-01</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>1</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>2</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>3</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>4</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>5</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>6</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>7</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>8</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>2023-01-01</td>
<td>9</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

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:

```r
x <- group_by(x, date, date_char)
x <- summarise(x, digits = as.character(0:9))
x <- mutate(x, digit_count = str_count(date_char, digits))
x <- group_by(x, digits)
x <- summarise(x, m = mean(digit_count))
x <- pivot_wider(x, names_from = digits, values_from = m)
x
```

**Exercise 2.3.** The following code selects the longest name listed in either the director and cast columns of netflix (see Exercise 1.4, Section 1.3).

```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_count(director)
      ),
```
2.2 Lecture

```r
cast =
  last(
    cast,
    order_by = str_count(cast)
  )
) %>%
summarise(
  longest_name =
  elseif(
    str_count(director) > str_count(cast),
    director,
    cast
  )
)
```

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

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

- Note: We will streamline the code above in Section 2.4.6.

2.2.3 Defining Functions

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

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

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

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

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

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

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

f <-
  . %>%
  f_1 %>%
  f_2 %>%
  #...
  f_n

In other words, running this code defines a function f() that satisfies f(x) =
x %>% f_1 %>% f_2 ... %>% f_n.

Exercise 2.4. Use the <- . %>% notation to create an R function called
soft_max() that takes a vector x and applies the exp(), sum(), and log() functions to it, in that order.

• Check: -5:5 %>% soft_max = 5.458658.

Exercise 2.5. Use the <- . %>% notation to create a function called
clean_text(), where clean_text(x) equals the output of the following code:
2.2 Lecture

```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() the following two character string vectors:

```r
defines clean_text() the following two character string vectors:

```r
# Vector 1
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
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 2.6. 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? Figure this out
without R, and then execute the code to confirm your answer.

```r
f_1 <-
2 Functions

2.2.4 `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 define and redefine variables of a tibble. For example, we can use it to clean `netflix`, which we defined in Section &amp;&amp;.

This tibble looks like:

```
netflix %>% glimpse
```

> Rows: 6,223
> Columns: 12
> $ show_id   <dbl> 81145628, 80117401, 70234439, 80058654, 80125979, 8016...
> $ type      <chr> "Movie", "Movie", "TV Show", "TV Show", "Movie", "TV S... 
> $ title     <chr> "Norm of the North: King Sized Adventure", "Jandino: W... 
> $ director  <chr> "Richard Finn, Tim Maltby", NA, NA, "Fernando Lebr... 
> $ cast      <chr> "Alan Marriott, Andrew Toth, Brian Dobson, Cole Howard... 
> $ country   <chr> "United States, India, South Korea, China", "United Ki... 
> $ date_added <chr> "September 9, 2019", "September 9, 2016", "September 8... 
> $ 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", ...
> $ listed_in <chr> "Children & Family Movies, Comedies", "Stand-Up Comedy... 
> $ description <chr> "Before planning an awesome wedding for his grandfathe..."
As you see, the `show_id` and `release_year` variables are tagged with `<dbl>`, which indicates that they’re doubles. But these variables have no decimal component, and should thus be saved as integers (see Section ??). Accordingly, we’ll use `mutate()` to redefine them:

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

#Confirm that these variables are now integers:

```r
netflix %>%
summarise(
  storage.mode(show_id),
  storage.mode(release_year)
)
```

> # A tibble: 1 x 2
> `storage.mode(show_id)` `storage.mode(release_year)`
> <chr>                   <chr>
> 1 integer               integer

Note how we embed `as.integer()` inside `mutate()`. Most of our `mutate()` calls will embed other functions in this fashion. And also note that we use `=` rather than `<-`. Indeed, we’ll basically never use `<-` inside function’s parentheses.

In addition to modifying old variables, `mutate()` enables us to construct new variables. For example, the following code adds a column to `netflix` called `country_primary`, which reports the first country listed in the corresponding country text string:

```r
netflix <-
netflix %>%
mutate(
  country_primary =
    str_remove(country, ",.*")
)
```

#Compare country and country_primary

```r
netflix %>%
select(country, country_primary) %>%
slice(1, 7, 12) #random rows
```
> # A tibble: 3 x 2
> country country_primary
> <chr> <chr>
> 1 United States, India, South Korea, China United States
> 2 Bulgaria, United States, Spain, Canada Bulgaria
> 3 Netherlands, Belgium, United Kingdom, United States Netherlands

In the code above, ",,*" is shorthand for "the"," symbol, and everything after it".\(^2\) Hence, `str_remove()` takes `country` as an input and returns everything before the first comma as an output: e.g., it maps "Indonesia, South Korea, Singapore" to Indonesia. And the `mutate()` operation saves this output as a new tibble variable called `country_primary`.

The following exercises provide additional examples.

**Exercise 2.7.** Use `mutate()` to add a logical variable to `netflix` called `restricted` that is `TRUE` if and only if the show’s `rating` is "R", "TV-MA", or "NC-17".

- Tip: You can use `%in%` to determine whether the elements of one vector belong to another vector. For example: `c("a", "d", "123", "b", "A") %in% c("a", "b", "c") = c(TRUE, FALSE, FALSE, TRUE, FALSE).

**Exercise 2.8.** The duration values of `netflix` are text strings that comprise both a measurement, which is a number, and a metric, which is either "min" or "Season". For example, the first three duration values are "90 min", "94 min", and "1 Season". But storing two values in one variable makes accessing either value difficult. For example, there’s currently no easy way to arrange the sample to put the shows that are measured in minutes on top. And it’s always bad to represent numbers as character strings. For example, there’s currently no easy way to calculate the median show duration, because `median()` only accepts numbers. 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".
  - Hint: `word("fee fi fo fum", 2) = "fi"`.
- Within the same `mutate()` call, use `word()` and `as.integer()` to redefine `duration` as the “first word” of `duration` expressed as an integer.
  - Tip: Conveyor belts are allowed inside `mutate()` calls. So you can arrange `word()` and `as.integer()` into a functional assembly line: e.g., "12 days of Christmas" %>% word(1) %>% as.integer = 12.
  - Check: The first observation should have `duration = 90` and `duration_metric = "min"`.

\(^2\)This is called a *regular expression*. See `vignette("regular-expressions")`.
2.2 Lecture

- Save the updated tibble as `netflix`, overwriting the old version.

**Exercise 2.9.** The lubridate package—which we added to our library in Section 2.2.3—gives us many functions for working with dates. But to exploit this functionality we must express a date as a `Date` variable rather than as a character variable. Use functions `mutate()`, `mdy()`, `wday()`, and `month()` to make the following three changes to `netflix`:

1. change the `date_added` column from a character string to a `Date`,
2. add a new variable called `wday_added` that reports the day of the week the show was added, and
3. add a new variable called `month_added` that reports the month the show was added.

- Tip: You can define multiple variables with one `mutate()` call if you separate their definitions with a comma.
- Tip: Use the `label = TRUE` option of `wday()` and `month()` to express `wday_added` and `month_added` as factors rather than as numbers.

### 2.2.5 `summarise()`

Now we can use `wday_added`, from the last exercise, to gauge the fraction of shows added on a Friday or Saturday. We can calculate this with `summarise()`, our second quartet function:

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

```r
# A tibble: 1 x 1
  `mean(wday_added %in% c("Fri", "Sat"))`
      <dbl>
1    0.483
```

Unfortunately, it’s easy to mix up `summarise()` with `summary()`, which we saw in Section 2.2.4. But the former is far more powerful: whereas `summary()` always returns the same canned report, `summarise()` can return almost any summary statistic you desire (such as the proportion of time that the `wday_added %in% c("Fri", "Sat")` logical is TRUE). Indeed, this function is like a magician’s top hat: you can pull almost anything out of it.

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

```r
netflix %>%
  summarise(mean(day_added),
            median(day_added),
            max(day_added),
            min(day_added))
```

```r
# A tibble: 1 x 4
   mean(day_added) median(day_added) max(day_added) min(day_added)
            <dbl>          <dbl>       <num>       <num>
1            15.3         13.9       123.         0.69
```
mean_age = mean(today() - date_added),
median_age = median(today() - date_added)
)

> # A tibble: 1 x 2
> mean_age  median_age
> <drt> <drt>
> 1 941.3495 days 851 days

In the code above, `today()` is a function that returns the current date. For another example, the following returns the title of the 100th and 1,000th shows, both when the sample is ordered by `date_added` and when it is ordered by `release_year`:

```
netflix %>%
  summarise(
    added_100th = nth(title, n = 100, order_by = date_added),
    added_1000th = nth(title, n = 1000, order_by = date_added),
    release_100th = nth(title, n = 100, order_by = release_year),
    release_1000th = nth(title, n = 1000, order_by = release_year)
  )
```

> # A tibble: 1 x 4
> added_100th added_1000th release_100th release_1000th
> <chr> <chr> <chr> <chr>
> 1 Glee  Liv and Maddie Raging Bull Soul Eater

Note, the function `nth()` 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"`. For a more involved example, the following calculates the fraction of shows filmed in South Korea:

```
netflix %>%
mutate(
in_SK =
  country %>%
    str_detect("South Korea")
```
The `str_detect()` function maps character strings to logicals, based on whether a given substring was detected. So in this case the `mutate()` step creates a new logical variable called `in_SK` that indicates whether or not "South Korea" appears in the corresponding country text string. For example, the first `in_SK` value is `TRUE` because the first country value is "United States, India, South Korea, China". The `summarise()` step then computes the average fraction of time that the `in_SK` logical is `TRUE`. And `na.rm = TRUE` option tells R to ignore instances in which `in_SK` is `NA`, which happens when country is `NA` (i.e., missing).

For explanatory purposes, I divided the above computation into a `mutate()` step and a `summarise()` step. But `summarise()` can actually handle the job on its own:

```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

**Exercise 2.10.** 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 the logical vector into `mean()` (while still within the `summarise()` call).

**Exercise 2.11.** 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.
  - Hint: "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()`.
  - Tip: Piping a vector into `{. + 1}` increases it by one. For example, `1:5` `%>% {. + 1} = 2:6. (See Section 2.4.3.)
• Use `quantile()` to calculate the quartiles of the resulting vector.
  - Tip: Use the `probs = c(0.25, .5, .75)` option to specify the desired quantiles.
  - Tip: Use the `na.rm = TRUE` option to ignore rows with `NA` cast entries.
  - Check: You should get a 0.5 quantile—i.e., median—of 8.

**Exercise 2.12.** We will now calculate the average duration of shows with `duration_metric = c("min", and the average duration of shows with `duration_metric = c("min", "Season[. (See Exercise 2.8 for the definition of `duration_metric.`

• Pipe `netflix` into `summarise()`.
• Within the `summarise()` call, use `weighted.mean()` to calculate the average duration among the programs measured in minutes.
  - Tip: Weight by the `duration_metric = c("min", "Season[. Doing so tells R to give the minute-measured observations a weight of one and the season-measured observations a weight of zero. In other words, it tells R to calculate the mean duration after disregarding the observations with `duration_metric = c("min", "Season[. For example, `weighted.mean(c(90, 94, 1, 2), c("min", "min", "Season", "Season") = 92.
• Within the same `summarise()` call, use `weighted.mean()` to calculate the average duration among the programs measured in seasons.
  - Tip: Do the same as the previous step, except now weight by `duration_metric = c("min", "Season[. For example, `weighted.mean(c(90, 94, 1, 2), c("min", "min", "Season", "Season") = 1.5.

### 2.2.6 `group_by()`

The previous exercise used `weighted.mean()` to calculate the average value of `duration` for both values of `duration_metric`. We could use this technique to calculate the average value of `release_year` for each value of `country_primary`:  

```r
netflix %>%
  summarise(average_Argentina =
    weighted.mean(
      release_year,
```
```r
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 would be ungainly, comprising a `weighted.mean()` call for all 72 values of `country_primary`. A better approach would be 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 15
> # Groups:  type, rating [3]
> show_id type title director cast country date_added release_year rating
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 = mean(str_count(description)),
  length_sd = sd(str_count(description))
) %>%
head(3)
```

> # 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).`

But we can also define more exotic groupings. For example, the following defines grouping scheme `year_added = year(date_added)` within the `group_by()` operation, and uses `summarise()` to calculate the mean `release_year` for each corresponding chunk:
2.2 Lecture

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

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

As you see, Netflix must have added a lot of old shows in 2020, as the average `release_year` of its new offerings decreased to 2008 in that year.

**Exercise 2.13.** We will now calculate the fraction of shows added in each month.

- Start with `netflix` and `group_by()` `month_added` (which we defined in Exercise 2.9).
- Use `summarise()` and `n()` to define `num_shows` as the number of observations in each group.
  - Note: The function `n()` returns the number of observations in a given `group_by()` group.
  - Check: Your "Jan" group should have `num_shows` = 610.
- `ungroup()` the tibble.
- Use `mutate()` to define `frac = num_shows / sum(num_shows)`
  - Check: The value of `frac` should sum to 1 across all groups.
  - Note: The `ungroup()` step makes the `sum(num_shows)` step sum across the entire tibble, rather than across the current `month_added` group.

**Exercise 2.14.** We found in Section 2.2.5 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.
  - Tip: 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".
  - Tip: Calculate the mean of the `type == "TV Show"` logical vector.

```r
```
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 variables from group-level information. For example, the following defines cp_directors as the number of distinct director values among the observations with 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 2.15.** 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.
  - Note: 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 2.16.** 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.

- Note: ifelse(x, y, z) returns y if x is TRUE and return z if x is FALSE. For example, ifelse(-1:2 == 0, "zero", "not zero") = c("not zero", "zero", "not zero", "not zero").

**Exercise 2.17.** 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).
2.2 Lecture

- Tip: Use the `director_count == 1` logical.
- Tip: Use the `mean()` function with `na.rm = TRUE` option.

2.2.7 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 our sample. It takes one or more conditions 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
> <chr>   <chr> <chr> <chr>      <chr>
> Domino Movie R International Movie- Denmark, France, I- 2019
> Everybody K- Movie R Dramas, Internatio- Spain, France, Ita- 2019
In the code above I used `select()` to remove all but the relevant columns. Whereas `filter()` drops rows, `select()` drops columns. For example, we could eliminate everything besides `director`, `title`, and `cast` with

```r
netflix %>%
  select(director, title, cast)
```

Or we could eliminate `director`, `title`, and `cast` with

```r
netflix %>%
  select(-director, -title, -cast)
```

Or, equivalently, with

```r
netflix %>%
  select(-c(director, title, cast))
```

But `filter()` is more important than `select()` because we’ll more frequently need to remove a tibble’s rows than its columns.

**Exercise 2.18.** Use `filter()` and `select()` to list the `title` and `description` of the programs directed by "Bong Joon Ho".

**Exercise 2.19.** We will now create a tibble called `single_director` that comprises all the programs with one person listed as the director.

- Pipe `netflix` into `filter()`.
- Within the `filter()` parentheses derive an inclusion condition that is `TRUE` if the given show has one person listed under `director`, `FALSE` if it has multiple people listed under `director`, and `NA` if `director` is `NA`.
  - Note: `filter()` will drop an observation if its inclusion condition is `NA`. So your `filter()` step will drop all the shows with missing `director` values.
  - Tip: Use `str_detect()` to look for commas. But use the `negate = TRUE` option so that the `director` values without commas correspond to `TRUE`.
  - Check: `single_director` should comprise 3831 rows.

We can combine `group_by()` with `filter()` to create inclusion criteria that are based on group-level information. For example, the following filters out all the programs in `single_director` that do not have a unique `country_primary` value:
2.2 Lecture

```r
single_director %>%
group_by(country_primary) %>%
filter(n() == 1) %>%
select(country_primary, title, type, release_year)
```

> # A tibble: 10 x 4
> # Groups: country_primary [10]
> country_primary title type release_year
> <chr> <chr> <chr> <int>
> 1 Saudi Arabia Barakah Meets Barakah Movie 2016
> 2 Iran A Separation Movie 2011
> 3 Guatemala Septiembre, un llanto en silencio Movie 2017
> 4 Somalia The Pirates of Somalia Movie 2017
> 5 Dominican Republic El testigo Movie 2018
> 6 Slovenia Houston, We Have a Problem! Movie 2016
> 7 Croatia You Carry Me Movie 2015
> 8 Bangladesh Komola Rocket Movie 2018
> 9 Lebanon Very Big Shot Movie 2015
> 10 Georgia The Trader (Sovdagari) Movie 2018

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

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

```r
single_director %>%
group_by(director) %>%
filter(
  str_detect(country, "Mexico", negate = TRUE),
  any(str_detect(country, "Mexico"))
) %>%
select(title, director, country)
```

> # A tibble: 6 x 3
> # Groups: director [5]
> title director country
> <chr> <chr> <chr>
> 1 El Che Matías Gueiburt Venezuela
> 2 The Apostate Federico Veiroj Spain, France, Uruguay
> 3 The Captive Atom Egoyan Canada
> 4 Chloe Atom Egoyan United States, Canada, France
> 5 GoldenEye Martin Campbell United Kingdom, United States
> 6 Hellboy Guillermo del Toro United States
The first filter() condition is a row-level requirement: it stipulates that "Mexico" may not be included in the current program's country list (note the negate = TRUE option). The second filter() condition is a group-level requirement: it stipulates that "Mexico" must be included in the country list of at least one program of the current director group. For example, 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.

The following exercises provides additional examples (and Section 4.2.4 provides even more).

**Exercise 2.20.** We will now select the the programs made by directors that correspond to at least six distinct listed_in values.

- Start with single_director and group_by() director.
- Use filter() and n_distinct() to drop the director chunks with fewer than six different listed_in values.
  - Check: Only two director groups should satisfy this condition.
- select() the director, title, and listed_in columns of the resulting tibble.

**Exercise 2.21.** We will now select the shows whose cast members have all previously worked together.

- Start with single_director and group_by() cast.
- Use filter() to remove all shows whose release_year is not the minimum value for the given cast grouping. That is, keep a show only if there was another show with the same cast released in an earlier year.
- Remove shows with NA cast values.
  - Tip: Do this within the same filter() operation.
- select() the title, release_year, and cast of the resulting tibble.

### 2.3 Lab

#### 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 adopt 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 1.5. It should have 11 variables:\(^4\)

---

\(^3\)We won’t be able to reproduce their results exactly, as our sample differs a bit from their sample.

\(^4\)Cramming all this disparate data into one large tibble is a poor way to store it. For example, the tibble has a lot of redundancy, as it needlessly repeats case-level attributes ptype, dtype, and ctype for each hearing. It would be more efficient to store the case-level variables in one tibble and the hearing-level variables in another tibble. However, we won’t cover joining tibbles until Chapter 4 so we’ll stick to one tibble for this lab.
• 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.\footnote{More technically, jid identifies the judge who serves as the rapporteur for the case. As Bray et al. (2016) explain, “Each case is assigned to a [three-judge] panel for adjudication, and to a rapporteur, a judge on the panel, for supervision. The rapporteur analyzes the testimony, oversees the ruling, writes the opinion, and schedules the hearings. Since panels are stable … we treat a case’s rapporteur as its sole judge.”}
• fdate records the date the case was filed, and bdate 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.\footnote{More 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.} 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 2.22.)
• 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).

2.3.3 Clean

Exercise 2.22. 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)
• Check: court_data should have 304,941 rows after this step.
• Tip: You can get the number of rows in a table with dim() or nrow().
Exercise 2.23. 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.
  - Check: There should be as many rows in file_dates as there are cases in court_data.
  - Note: 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.
  - Check: 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 have it overwrite court_data.
  - Tip: Make sure hid is ordered by date and not by how the rows are arranged in the tibble. For example, consider the following:

```r
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 2.24.** 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. We will thus set `closed = TRUE` for these cases.

- Define `cutoff_date` as the day two years before the latest `date` in our sample.
  - Note: `cutoff_date` should be a standalone Date variable, not a tibble or an element of a tibble.
  - Note: 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.
  - Tip: See what `today() - years(2)` yields.
  - Tip: `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`.
  - Note: Inequalities hold for Date variables. For example, `today() < today() + days(1) = TRUE`.
- `ungroup()` the tibble and save it as the new `court_data`.

**Exercise 2.25.** 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 = 189211` has `closed = FALSE` and so all its hearings have
2.3 Lab

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.

- Tip: You can use & to determine whether two logicals are both TRUE.
  
  For example, TRUE & TRUE = TRUE and TRUE & FALSE = FALSE.

- ungroup() the tibble and use it to overwrite court_data.

**Exercise 2.26.** Use filter() to remove all hearings that were held more than 2,000 days before the January 1, 2011 intervention date.

- Check: court_data should have 310,401 rows after this step.

**Exercise 2.27.** 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 skip these data filters because they have little effect on the overall results.)

2.3.4 Analyze

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

To get a more timely estimate, Bray et al. (2016) conducted a more subtle empirical test. They noted that the average case duration is a function of two factors: the arrival rate of new cases and the number of open cases. 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

---

7 In fact, a classic result from operations management establishes that the average case duration exactly equals the average arrival rate of new cases times the average number of open cases. This result is known as (Little’s law) [https://en.wikipedia.org/wiki/Little%27s_%20law](https://en.wikipedia.org/wiki/Little%27s_%20law).
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 2.28.** 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 = 6` observations from `court_data`.
  - Note: 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).`
  - Check: There should be 226 groups.
  - Note: 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.
  - Tip: 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.
  - Hint: `floor_date(ymd("1865-June-19"), "month") = "1865-06-01`.
  - Tip: `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`.
  - Note: 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.
  - Check: 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)`.
- Note: We won’t use `sigma` until Exercise 2.29.
- `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.
2.3 Lab

- Call the resulting tibble `data_to_plot`.
- Check: `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,
  colour = 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. 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 2.29.** Each point in the plot we made in Exercise 2.28 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 \(X = \frac{\bar{x} \pm z_{\alpha/2}}{\sqrt{N}}\) is `lower_limit` to `upper_limit`, where

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

We will add these 95% confidence intervals to our plot by adding a `geom_errorbar()` layer to our plot blueprints:
hazard_rate_plot <-
hazard_rate_plot +
geom_errorbar(
aes(
  ymin = lower_limit,
  ymax = upper_limit
)
)
}

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

- Use mutate() to add these limit variables to data_to_plot, setting lower_limit = hazard_rate - qnorm(.975) * sigma / sqrt(num_obs) and upper_limit = hazard_rate + qnorm(.975) * sigma / sqrt(num_obs).
  - Note: 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 Specialized Functions

We can use mutate() to transform variables in any way we like and we can use summarise() to define any summary statistic we like. But some variable transformations and some summary statistics are so common that \( R \) has tailor-made functions for them. That said, some of the mutate() and summarise() examples in Section 2.2 are rather contrived, as we would actually perform these tasks with the more specialized functions.

First, in Exercise 2.8 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:
2.4 Commentary

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

Second, in Exercise 2.13 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)
```

> # A tibble: 12 x 2
>   month_added n
>   <ord>      <int>
> 1 Jan        610
> 2 Feb        378
> 3 Mar        551
> 4 Apr        447
> 5 May        428
> 6 Jun        393
> 7 Jul        474
> 8 Aug        509
> 9 Sep        479
>10 Oct        646
>11 Nov        612
>12 Dec        696
```

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

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

> # A tibble: 4 x 3
>  type USSR  n
>   <chr> <lgl> <int>
> 1 Movie FALSE 4259
> 2 Movie TRUE  5
> 3 TV Show FALSE 1947
> 4 TV Show TRUE 12

Third, in Exercise 2.15 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 <-
netflix %>%
  add_count(
    director,
    name = "director_count"
  )
```

### 2.4.2 Nest Functions

In Section 2.2.2 we learned to use conveyor-belt operator %>% to apply a sequence of functions without relying on a wheelbarrow variable. But there’s another way to write such variable-free code: nest the functions inside each other, like Matryoshka dolls. For example, instead of 2 %>% sqrt %>% sqrt %>% sqrt %>% sqrt we could write `sqrt(sqrt(sqrt(sqrt(2))))`, and instead of 1:10 %>% exp %>% sum %>% log we could write `log(sum(exp(1:10)))`. And more generally, x %>% f_1 %>% f_2 %>% f_3 %>% ... %>% f_n is equivalent to `f_n( ... f_3(f_2(f_1(x))) ... )`, so any task that could be performed by a functional assembly line could also be performed by a set of nesting functions.

However, while the nested approach is computationally equivalent to the assembly line approach, it is inferior from a design perspective. For example,
compare the following two solutions to Exercise @red(exr:quartetMillionDates)
(you won’t need to understand the code to follow my argument):

```r
#Assembly-line solution
tibble(date = today() + days(-10^6:1)) %>%
mutable(date_char = str_replace_all(date, "-", "")) %>%
group_by(date, date_char) %>%
summarise(digits = as.character(0:9)) %>%
mutable(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(
  group_by(
    summarise(
      mutuate(
        group_by(
          tibble(date = today() + days(-10^6:1)),
          date_char = str_replace_all(date, "-", "")
        )
      )
    )
  )
)
```

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 = mean(digit_count)`. For another example, see how long it takes for you to determine which function the `digit_count = str_count(date_char, digits)` input belongs to.

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

For these reasons, it’s better to arrange a long sequences 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 Pipe into Bracketed Expressions

We began Section 2.2.2 with the problem of evaluating \( \sqrt{\sqrt{\sqrt{3}}} \). But suppose we wanted to evaluate \( ((3^2)^2)^2 \). This calculation would be straightforward if we allowed ourselves a wheelbarrow variable:

```r
current_val <- 3^2
current_val <- current_val^2
current_val <- current_val^2
current_val <- current_val^2
current_val
```

> [1] 43046721

However, it’s not so clear how to perform this computation with a functional assembly line because \(^\) can’t connect to conveyor belts in the standard fashion, since it’s not a standard function. For example, `3 %>% 2 %>% 2 %>% 2 %>% 2 %>% 2` yields an error because \(^2\) is a nonsensical expression. Here’s the fix:
In the code above, four conveyor belts connect four *bracketed expressions*. A bracketed expression is a chunk of code between brackets, `{ }`. Usually this code contain the . symbol, which inherits the value that the bracketed expression receives from inflowing conveyor belt. Hence, we have

```r
3 %>% { ^2 } %>% { ^2 } %>% { ^2 } %>% { ^2 }
= (3^2) %>% { ^2 } %>% { ^2 } %>% { ^2 }
= 9 %>% { ^2 } %>% { ^2 } %>% { ^2 }
= 9*^2 %>% { ^2 } %>% { ^2 }
= 81 %>% { ^2 } %>% { ^2 }
= 81*^2 %>% { ^2 }
= 6561 %>% { ^2 }
= 6561*^2
= 43046721
```

And here are two additional examples:

```r
# Example 1
c(.25, .75) %>% { - . * log(.) } %>% sum

> [1] 0.5623351

# Example 2
c(c("09/04/1929", "11/13/1929")) %>% mdy %>% {
  interval(
    start = .[1],
    end = .[2]
  )
```
The first example calculates \(-0.25 \cdot \log(0.25) - 0.75 \cdot \log(0.75)\), which is known as the information entropy\(^8\) of vector \(c(0.25, 0.75)\). In this case, \(R\) sets the two symbols in the bracketed expression to \(c(0.25, 0.75)\), which yields \(-c(0.25, 0.75) \cdot \log(c(0.25, 0.75))\) \(\%\%\) sum = \(c(0.3465736, 0.2157616)\) \(\%\%\) sum = \(0.5623351\). The second example calculates the duration of the 1929 stock market crash, which started on September 4th and ended on November 13th. In this case, the bracketed expression receives the vector \(c(mdy("09/04/1929"), mdy("11/13/1929"))\) from the conveyor belt. Hence, \(R\) evaluates this expression as \(\text{interval}(\text{start} = mdy("09/04/1929"), \text{end} = mdy("11/13/1929"))\), which returns a time “Interval” object that spans the duration of the crash. And the last function, \(\text{time_length}()\), then counts the number of days that this time Interval comprises.

Now we can combine the bracketed expression notation, \(\%\%\) \{\}, with the function characterization notation, <- \. \%\%\, to create compact function definitions. For example, the following defines \(\text{na_detector}()\) as a function that returns "NA detected" if the input contains an NA term and returns the input otherwise:

```r
na_detector <-
  . \%\% \{
    if(any(is.na(.))) "NA detected" else .
  }

# Case 1
\(c(\text{"a"}, \text{"b"}, \text{"c"}, \text{NA})\) \%\%
na_detector

> [1] "NA detected"

# Case 2
\(c(\text{"a"}, \text{"b"}, \text{"c"})\) \%\%
na_detector

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

In the first case above the \(\text{.\ values get set to } c(\text{"a"}, \text{"b"}, \text{"c"}, \text{NA}). And since any(is.na(c("a", "b", "c", NA))) = any(c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)) \(\%\%\) sum = \(c(0.3465736, 0.2157616)\) \(\%\%\) sum = \(0.5623351\). The second example calculates the duration of the 1929 stock market crash, which started on September 4th and ended on November 13th. In this case, the bracketed expression receives the vector \(c(mdy("09/04/1929"), mdy("11/13/1929"))\) from the conveyor belt. Hence, \(R\) evaluates this expression as \(\text{interval}(\text{start} = mdy("09/04/1929"), \text{end} = mdy("11/13/1929"))\), which returns a time “Interval” object that spans the duration of the crash. And the last function, \(\text{time_length}()\), then counts the number of days that this time Interval comprises.

Now we can combine the bracketed expression notation, \(\%\%\) \{\}, with the function characterization notation, <- \. \%\%\, to create compact function definitions. For example, the following defines \(\text{na_detector}()\) as a function that returns "NA detected" if the input contains an NA term and returns the input otherwise:

```r
na_detector <-
  . \%\% \{
    if(any(is.na(.))) "NA detected" else .
  }

# Case 1
\(c(\text{"a"}, \text{"b"}, \text{"c"}, \text{NA})\) \%\%
na_detector

> [1] "NA detected"

# Case 2
\(c(\text{"a"}, \text{"b"}, \text{"c"})\) \%\%
na_detector

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

\(^8\)https://en.wikipedia.org/wiki/Entropy_(information_theory)
2.4 Commentary

TRUE)) = TRUE, the if-else statement returns the first expression, which is "NA detected". In the second case the . values get set to c("a", "b", "c"). And since any(is.na(c("a", "b", "c"))) = any(c(FALSE, FALSE, FALSE)) = FALSE, the if-else statement returns the second expression, which is . = c("a", "b", "c").

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

```r
fib <-
  . %>% {
    c(.[2], .[1] + .[2])
  }

c(1, 2) %>% fib
> [1] 2 3

\[ \text{c}(1, 2) \%\% \text{fib} \%\% \text{fib} \]
> [1] 3 5

\[ \text{c}(1, 2) \%\% \text{fib} \%\% \text{fib} \%\% \text{fib} \]
> [1] 5 8

\[ \text{c}(1, 2) \%\% \text{fib} \%\% \text{fib} \%\% \text{fib} \%\% \text{fib} \]
> [1] 8 13

\[ \text{c}(1, 2) \%\% \text{fib} \%\% \text{fib} \%\% \text{fib} \%\% \text{fib} \%\% \text{fib} \]
> [1] 13 21

The second letter of each output enumerates the Fibonacci sequence.\(^9\)

For a third example, the code below defines `santas_calendar()` as a function that takes a date and returns the word "Christmas" if it is December 25th, and otherwise it returns the output of the following day plus the word "Eve".\(^10\)

---

\(^9\)https://en.wikipedia.org/wiki/Fibonacci_number

\(^10\)It may seem strange to define `santas_calendar()` in terms of `santas_calendar()`. But this is actually quite common in computer programming. It’s called recursion.
2.4.4 Define Multivariate Functions

In Section 2.2.2 we learned how to define functions with the <- %>% notation. For example, the following defines `append()` as a function that appends the suffix “ing” to its input:

```r
append <-
  . %>% paste0("ing")
"I love R code" %>% append
```

> [1] "I love R coding"
2.4 Commentary

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 = "ing") {
  string %>% paste0(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 %>% paste0(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 %>% paste0(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:
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 2.2.4 I alluded to the fact that we have “thousands of R functions at our disposal.” And this is indeed the case: because the language is open-source—anyone can contribute packages to the Comprehensive R Archive Network (CRAN)—almost any R function we could want has already been written. We just have to find and download it.

For example, suppose we sought a function that computes the moving average of the past three values, transforming vector \(c(x_1, x_2, x_3, \ldots, x_{20})\) to vector \(c(\text{NA}, \text{NA}, (x_1 + x_2 + 3)/3, (x_2 + x_3 + x4)/3, \ldots, (x_{18} + x_{19} + x_{20})/3)\). To find such a function, google “Moving average tidyverse”. (Searching by “tidyverse” tends to yield more up-to-date solutions than searching by “R!”) Your google results will list several packages that include a moving average function: e.g., forecast, smooth, zoo, and 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 that what you’re looking for. For example, the abjutilis package has an `escape_unicode()` function that “removes all accented characters from the current file, replacing them by their equivalent Unicode-escaped values.” But this package doesn’t specialize in text manipulation, as it provides general “Useful Tools for Jurimetical Analysis Used by the Brazilian Jurimetrics Association.” So `escape_unicode()` is almost certainly not the best accent-removal function we can find.

- **Popularity:** Give preference to more widely adopted packages. These packages are better vetted and have more support—e.g., more blog posts and more stackoverflow 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:

```
library(RcppRoll)

roll_mean(
  1:20,
  n = 3,
  fill = NA,
  align = "right"
)
```

```r
table

> [1] NA NA 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Or you can execute this function without loading the `RcppRoll` package into your workspace with:

---

11 [https://stackoverflow.com/](https://stackoverflow.com/)
12 [https://dplyr.tidyverse.org/articles/window-functions.html](https://dplyr.tidyverse.org/articles/window-functions.html)
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 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.
2.4 Commentary

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 2.3:

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

> # A tibble: 1 x 1
> longest_name
> <chr>


> 1 Gabrielle Susanne Solheim Leithaug

This code violates the “make similar things the same” principle. The objective is to find the longest name listed in either the director or cast columns of *netflix*. But for this query there’s no material difference between directors and cast members, so there’s no reason to distinguish between them. Yet the code does distinguish between them, applying each operation to the director and cast variables separately, which leads to a lot of redundancy.

A more efficient solution starts by lumping directors and cast members into a common variable called `person`, so that, e.g., a show with director = "Nia Dinata" and cast == "Surya Saputra, Cut Mini Theo, Tora Sudiro" would have `person = "Nia Dinata, Surya Saputra, Cut Mini Theo, Tora Sudiro"`. By turning similar things (i.e., directors and cast members) into the same thing (i.e., people), this change enables us to apply each operation only once, which makes for a simpler solution:

```r
netflix %>%
  unite( # Create person variable from director and cast
          person,
          c(director, cast),
          sep = " ",
          na.rm = TRUE
       ) %>%
  mutate(person = str_split(person, " ")) %>%
  unnest(person) %>%
  summarise(person = last(person, order_by=str_count(person)))
```

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

## 2.5 Solutions

### 2.1

```r
# 1
"comrade Napoleon is always right" %>%
  str_to_title %>%
  paste0("### ", ", "!!!", ", ###") %>%
```
2.5 Solutions

```r
str_replace("Comrade", "Master")
rep(3)
str_flatten
```

```r
> [1] "### Master Napoleon Is Always Right!!! ####### Master Napoleon Is Always Right!!! ####### Master Napoleon Is Always Right!!!

# 2
c(
  "As I stepped out of the front door",
  "a nasty little gust of wind caught",
  "the soapy patch on my neck and made",
  "me suddenly feel that my clothes didn't",
  "fit and that I was sticky all over."
) %>%
word(2) %>%
nth(3) %>%
str_count(
c(
  "It was only when I was nearly",
  "dressed and looking for a tie that",
  "I discovered that my neck was still",
  "soapy. It's a rotten thing to have a soapy neck.",
  "It gives you a disgusting sticky feeling",
  "and the queer thing is that, however carefully",
  "you sponge it away, when you've once discovered",
  "that your neck is soapy you feel",
  "sticky for the rest of the day."
),
  pattern = .
)
```

```r
> [1] 0 0 0 2 0 0 0 1 0

2.2
tibble(date = today() + days(-10^4:-1)) %>%
mutate(date_char = str_remove_all(date, ",")) %>%
group_by(date, date_char) %>%
summarise(digits = as.character(0:9)) %>%
mutate(digit_count = str_count(date_char, digits)) %>%
group_by(digits) %>%
summarise(m = mean(digit_count)) %>%
pivot_wider(names_from = digits, values_from = m)
```
```r
> # 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  2.46 1.55 1.47 0.316 0.289 0.292 0.289 0.292 0.292 0.749
```

2.3

```r
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_count(director)
        ),
        cast =
        last(
            cast,
            order_by = str_count(cast)
        )
    )
d <-
    summarise(
        d,
        longest_name =
        ifelse(
            str_count(director) > str_count(cast),
            director,
            cast
        )
    )
d
```
2.5 Solutions

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

2.4

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

2.5

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

2.7

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

2.8

netflix <-
  netflix %>%
  mutate(
    duration_metric = word(duration, 2),
    duration =
      duration %>%
      word(1) %>%
      as.integer
  )

2.9
```r
netflix <-
netflix %>%
mutate(
  date_added = mdy(date_added),
  wday_added = wday(date_added, label = TRUE),
  month_added = month(date_added, label = TRUE)
)

2.10

netflix %>%
summarise(
  director %>%
  is.na %>%
  mean
)

> # A tibble: 1 x 1
> `director %>% is.na %>% mean`
> <dbl>   0.315

2.11

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

> # A tibble: 3 x 1
> quantiles
> <dbl>
> 1  5
> 2  8
> 3 10

2.12
```
2.5 Solutions

```r
netflix %>%
  summarise(
    average_min =
      weighted.mean(
        duration,
        duration_metric == "min"
      ),
    average_season =
      weighted.mean(
        duration,
        duration_metric == "Season"
      )
  )
```

```r
> # A tibble: 1 x 2
> average_min average_season
>       <dbl>       <dbl>
> 1     99.1       1.76
```

2.13

```r
netflix %>%
  group_by(month_added) %>%
  summarise(numShows = n()) %>%
  ungroup %>%
  mutate(frac = numShows / sum(numShows)) %>%
  head(4)
```

```r
> # A tibble: 4 x 3
> month_added numShows     frac
> <ord>         <int>     <dbl>
> 1 Jan         610 0.0980
> 2 Feb         378 0.0607
> 3 Mar         551 0.0885
> 4 Apr         447 0.0718
```

2.14

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

```r
> # A tibble: 2 x 2
> Fri_Sat fraction_tv
```

```r
> <lgl> <dbl>
> 1 FALSE 0.264
> 2 TRUE 0.391
```

### 2.15

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

### 2.16

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

### 2.17

```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]
>   type  Fri_Sat inexperienced_director
>  <chr> <lgl>           <dbl>
> 1 Movie FALSE             0.615
> 2 Movie TRUE              0.690
> 3 TV Show FALSE            0.671
> 4 TV Show TRUE             0.879
2.5 Solutions

2.18

```r
netflix %>%
  filter(director == "Bong Joon Ho") %>%
  select(title, description)
```

> # A tibble: 2 x 2
> title    description
> <chr>     <chr>
> 1 Snowpiercer The Earth's remaining inhabitants are confined to a single train -
> 2 Okja A gentle giant and the girl who raised her are caught in the cros-

2.19

```r
single_director <- netflix %>%
  filter(str_detect(director, ",", negate = TRUE))
```

2.20

```r
single_director %>%
  group_by(director) %>%
  filter(n_distinct(listed_in) >= 6) %>%
  select(director, title, listed_in) %>%
  head(3)
```

> # A tibble: 3 x 3
> # Groups: director [2]
> director title listed_in
> <chr>     <chr> <chr>
> 1 Don Michael Paul Scorpion King 5: Book of S- Action & Adventure, Sci-Fi & Fan-
> 2 Don Michael Paul Death Race: Beyond Anarchy Action & Adventure
> 3 Martin Scorsese Raging Bull Classic Movies, Dramas, Sports M-

2.21

```r
single_director %>%
  group_by(cast) %>%
  filter(
    release_year != min(release_year),
    !is.na(cast)
  ) %>%
  select(title, cast, release_year) %>%
  head(3)
```
> # A tibble: 3 x 3
> # Groups:   cast [3]
> title               cast       release_year
> <chr>               <chr>       <int>
> 1 Marc Maron: Too Real Marc Maron   2017
> 2 Jeff Dunham: Beside Himself Jeff Dunham 2019
> 3 Iliza Shlesinger: Confirmed Kills Iliza Shlesinger 2016
3

Pivots and Plots

3.1 Introduction

In this section you will learn how to make plots and how to 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 meta data stored above the tibble into formal data stored inside the tibble, and `pivot_wider()`, which converts formal data stored inside the tibble into meta data 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, 361.8, 225,
    "Earth", 3958, 499.0, 365,
    "Mars", 2106, 760.1, 687
  )

rocky_planets_1
```

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

```r
rocky_planets_2 <-
  tribble(
    ~planet, ~radius, ~sun_dist, ~year,
    "Mercury", 1516, 193.1, 88,
    "Venus", 3760, 361.8, 225,
    "Earth", 3958, 499.0, 365,
    "Mars", 2106, 760.1, 687
  )
```

```r
> # A tibble: 4 x 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
```
-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. (I’ll cover most, but not every detail of pivoting; for more on this topic see vignette("pivot").)

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 3.4.4 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 `ufc`, a sample of 4111 Ultimate Fighting Championship (UFC) bouts data painstakingly compiled by Matthew Dabbert\(^1\). (See Section 1.3, Exercise 1.6.) Each bout has two combatants, and each combatant is assigned to a corner of the octagonal ring. The favorite is usually assigned to the red corner and the underdog to the blue corner. The variables corresponding to the red-corner fighter begin with "R_" and the variables corresponding to the blue-corner fighter begin with "B_". I will use "X_" to refer to the generic case: e.g., \(X\_\text{fighter}\) mean \(R\_\text{fighter}\) or \(B\_\text{fighter}\). Here are the variables:

- **fight_id**: Distinct bout identification number.
- **date, location**: Date and location of the bout.
- **red_win**: A logical that is `TRUE` if the boxer in the red corner wins and `FALSE` if the boxer in the blue corner wins.
- **title**: Logical indicating whether this was a championship title bout.
- **weight_class**: The weight class of the bout.
- **gender**: Fighter gender, either "MALE" or "FEMALE".
- **rounds**: Maximum number rounds until decision, either three, four, or five.
- **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.
- **finish_details**: The move that finished the fight, if it ended in a knockout or submission.
- **finish_round**: The round the fight ended, a number between one and five.
- **finish_round_time**: Duration of the final round, in seconds.
- **fight_time**: Duration of the fight, in seconds.

\(^1\)https://www.kaggle.com/mdabbert
• audience: Logical indicating whether there was an audience in the arena.
• X_fighter: Fighter name.
• X_odds: The decimal odds that that this fighter will win, from bestfightodds.com. For example, if 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).
• X_stance: Fighting stance.
  – "Orthodox" denotes a left-shoulder-forward stance.
  – "Southpaw" denotes a right-shoulder-forward stance.
  – "Switch" denotes a combination of left-shoulder-forward and right-shoulder-forward stances.
• X_age, X_weight, X_height, X_reach: Fighter age (in years), weight (in pounds), and height and arm span (in centimeters).
• X_wins, X_losses, X_draw, X_rounds_fought: Total number of UFC wins, losses, draws, and rounds fought prior to the current fight.
• X_kd: Number of times this fighter knocked down the other fighter in this bout.
• X_strong_attempt, X_strong_achieve, X_weak_attempt, X_weak_achieve: Number of strong strikes attempted, strong strikes landed, weak attempts, and weak strikes landed by this fighter in this bout.
• X_td_attempt, X_td_achieve: Number of take-downs attempted and achieve by this fighter in this bout.
• X_sub_attempt, X_sub_achieve: Number of submissions attempted and achieve by this fighter in this bout.

3.2.2 Plotting with 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. 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) +
```

\(^2\)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 plot that corresponds to a specific group of data. We define our groups with the `vars()` function, which can parse general expressions. For example, `rows = vars(wday(date, lab = TRUE))` tells R to create a new row of facets for each distinct value of `wday(date, lab = TRUE)` and `cols = vars(gender, str_detect(location, "USA"))` tells R to create 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).
"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 rolls they play. For example, our cast of characters stipulates that R_reach will play the roll of the x-coordinate, B_reach will play the roll of the y-coordinate, and weight_class will play the roll 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.

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
fighter_stance_plot <-
ufc %>%
  mutate(
    spread = B_odds - R_odds,
    R_stance = paste("Red: ", R_stance),
    B_stance = paste("Blue: ", B_stance)
  ) %>%
  ggplot() +
aes(
    x = spread,
    color = R_stance,
    fill = R_stance
  ) +
  geom_histogram(
    bins = 40,
    alpha = .4
  ) +
  facet_wrap(
    vars(B_stance),
    ncol = 1,
    scales = "free"
  )
```
Now let’s walk through the four elements of this plot:

- **Data:** The tibble inputted to `ggplot()` is the tibble outputted by `mutate()`. Hence, the 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), ncol = 1)`. 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. The `ncol = 1` option tells R to stack the three facets on top of one another (i.e., in a single column); if, on the other hand, we wrote `nrow = 1` then R would have positioned the plots side by side (i.e., in a single row). The `scales = “free”` option tells R to give each facet it’s own scaling, so that, e.g., the top facet’s y-axis goes up to 400, whereas the bottom facet’s y-axis maxes out at 25.
- **Cast of characters:** The `aes()` call specifies that the x-coordinate roll is played by `spread` and the color and fill rolls are played by `R_stance`. There is no y-coordinate roll.
- **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 40 bins (i.e., 40 vertical bars). And the `alpha = .4` option tells R to make the bars 60% transparent (so they have a pastel look to them).

Unfortunately, our histogram still looks a little rough. But R plots are perfectly customizable, so we can spruce it up by adding a few extra options:

```r
fighter_stance_plot +
  theme_bw() +
  labs(
    x = "Spread",
    y = "Number of Fights"
  ) +
  theme(
    legend.position = "top",
    legend.title = element_blank(),
    strip.background =
      element_rect(color = "white", fill = "white")
  )
```

---

**3 Pivots and Plots**

---

**Red:** Orthodox, Southpaw, Switch

**Blue:** Orthodox, Southpaw, Switch

**Number of Fights**

Spread
In the code above, the \texttt{theme_bw()} call tells R to change the background from gray to white, the \texttt{labs()} call modifies the labels, and the \texttt{theme()} call moves the legend from the side to the top, removes the legend title, and turns the facet labels from gray to white. There are options like this for basically any change you could want to make to a plot. Rather than memorize them all, I recommend looking them up as needed.

The preceding example illustrate the key feature of \texttt{ggplot}s: their modularity. For example, rather than recreate our histogram from scratch, we simply appended the additional options to R object \texttt{fighter_stance_plot}, which we created to save our work-in-progress histogram. And we could incorporate almost any feature to any \texttt{ggplot} in this fashion. In fact, a \texttt{ggplot} is really just a loose collection of features pasted together with + signs. And since we're free to pick and choose which accessories we add to a \texttt{ggplot}, we can create just about any graph we want with this framework.

The following provides an extreme example:

```r
ufc %>%
  mutate(
    R_weak_achieve = ntile(R_weak_achieve, 3),
    B_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)
  ) %>%
  select(        
    contains("strong.")
  )
```
contains("weak"),
  starts_with("mean")
) %>%
na.omit %>%
ggplot +
aes(
  x = R_strong_achieve,
  y = B_strong_achieve
) +
geom_vline(
  aes(xintercept = mean_R_strong),
  color = "orange"
) +
geom_hline(
  aes(yintercept = mean_R_strong),
  color = "brown"
) +
geom_point(size = 0.1) +
geom_smooth() +
geom_quantile(
  quantiles = c(0.25, 0.5, 0.75),
  color = "red"
) +
facet_grid(
  rows = vars(R_weak_achieve),
  cols = vars(B_weak_achieve)
) +
theme_bw() +
labs(
  x = "Number of Strong Red Punches",
  y = "Number of Strong Blue Punches"
) +
theme(
  strip.background =
    element_rect(color = "white", fill = "white")
)
This plot has an absurd number of features. For example, it has five distinct layers: the orange lines from `geom_vline()`, the brown lines from `geom_hline()`, the dots from `geom_point()`, the blue lines from `geom_smooth()`, and the red lines from `geom_quantile()`. (The layers added later overlay the layers added earlier, so that, e.g., the black dots obscure the yellow lines but not the red lines.) Also, there are three character definitions made by three `aes()` calls. The first is attached with the `+` symbol, which indicates that its x = `R_strong_achieve` and y = `R_strong_achieve` roll assignments apply to all layers. However, the latter two `aes()` calls are embedded inside `geom_vline()` and `geom_hline()`, which indicates that their `xintercept = mean_R_strong` and `yintercept = mean_R_strong` roll assignments apply only within these respective layers.

The last example illustrates the typical `ggplot` syntax, which in general looks like this

```r
initial_data %>%
  transformation_fun_1 %>%
  transformation_fun_2 %>%
  # ...%>%
  transformation_fun_n %>%
  ggplot +
```
plot_fun_1() +
plot_fun_2() +
# ... 
plot_fun_n()

The `transformation_fun_x()` expressions represent functions like `mutate()` and `filter()`, which modify the data, and the `plot_fun_x()` expressions represent functions like `geom_vline()` and `theme_bw()`, which modify the plot. Note that we switch from `%>%` to `+` after the `ggplot()` line. Also, the function parentheses are optional before the `ggplot()` line but mandatory after. For example, in the code for the plot above we exclude the parentheses from `na.omit()` but not from `geom_smooth()`.

Exercise 3.1. If you bet $1/R\text{ odds}$ dollars on the red fighter winning then you will receive $R\text{ odds} \times (1/R\text{ odds}) = 1$ dollar from the bookie if this fighter does win. Similarly, if you bet $1/B\text{ odds}$ dollars on the blue fighter winning then you will win $1$ if this fighter wins. Hence, if you simultaneously bet $1/R\text{ odds}$ dollars on the red fighter winning and $1/B\text{ odds}$ dollars on the blue fighter winning then you’ll be guaranteed to receive $1$ from the bookie after the fight. Now if the market were perfectly efficient then this bet’s cost, $1/R\text{ odds} + 1/B\text{ odds}$, would equal its reward, $1$. But the caseno always takes a cut, so $1/R\text{ odds} + 1/B\text{ odds}$ will always exceed $1$ by some margin, which we will call the house take. We will now plot the house take distribution, by country.

- Pipe `ufc` into `mutate()`.
- Within the `mutate()` call, define `house_take = 1/R\text{ odds} + 1/B\text{ odds} - 1`.
- Within the same `mutate()` call, define `country = str_extract(location, "\\b[^,]+\\$")` as the text that follows the last comma in `location` (i.e., the country of the bout). For example, `location = "Abu Dhabi, Abu Dhabi, United Arab Emirates"` yields `country = "United Arab Emirates"`.
  - Note: The `str_extract(location, "\\b[^,]+\\$")` expression is complex. `str_extract()` is a function that extracts substrings from strings and "\b[^,]+\$" is a regular expression that’s R shorthand for “the text after the last comma.” But now you may wonder: “How would I have know to write that?” But wouldn’t need to know because you could look it up on the fly. For example, I didn’t know how to write this line of code so I just googled “get last phrase in string after comma string” and the answer was on the very first page.
- Pipe the output of `mutate()` into `ggplot()`.
- Use the `+` symbol to attach an `aes()` call that defines characters `x = country` and `y = house_take`.
- Use `+` to add a `geom_violin()` layer and look at the resulting plot.
- You’ll probably get some overlap in the country names that run along the bottom. Use `+` to add a `coord_flip()` option to your figure, and look at the output. You should be able to read the country names more clearly now.
The vertical axis should now run from "Argentina", at the bottom, to "USA", at the top. But it would be more natural if the countries were listed in the opposite order. Within the `mutate()` operation add a step in which you use `fct_relevel()` to reverse the order of the country labels.
- Tip: Enter `?fct_relevel` into your console and consult the examples provided.
- Use `+` to add a `labs()` option that sets the horizontal label to "House Take" and the vertical label to "" (i.e., to remove the vertical label).
- Tip: Since you used `coord_flip()`, the y axis now runs horizontally and the x axis runs vertically.

**Exercise 3.2.** Since it generally takes longer to ensure 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`.
- 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 20 bouts to `paste(result, "Other")`. 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.
  - Check: There should be 14 distinct `finish_details` values after this step.
- Pipe the result into `ggplot()`.
- `ungroup()` the data and pipe it into a second `mutate()` step.
- Within this `mutate()` step use `fct_reorder()` to define `finish_details` as a factor whose levels are ordered by the median value of `finish_round_time`. Use `.desc = TRUE` option so that the first level has the largest median `finish_round_time`.
  - Tip: Look at the examples provided by `?fct_reorder`.
  - Check: The first `finish_details` level should be "Arm Triangle".
- Pipe the result into `ggplot()`.
- Define the cast of characters, letting `finish_round_time` play the x-coordinate roll, `finish_details` play the y-coordinate roll, and `result` playing the fill roll.
- Use `geom_boxplot()` to add a boxplot layer.
  - Note: The order we’ve assigned to the `finish_details` factor levels ensures that the boxes are arranged in an increasing fashion.
3 Pivots and Plots

- Tidy the plot as you see fit and name it `finish_details_plot` (so we can modify it in a later exercise).

**Exercise 3.3.** The following plot depicts the `fight_time` of the rounds = 3 bouts as a function of `abs(B_odds - R_odds)`. As expected, we find that more mismatched fights end sooner. Use `geom_smooth(method = "lm")` to recreate this plot.

- Tip: I have removed the `weight_class = "Catch Weight"` observations, since catch weight isn’t actually a weight class.
- Tip: I have modified the `weight_class` variable to shorten the facet labels.

**Exercise 3.4.** The following plot depicts the number of bouts that took place over time, by weight class. Use `geom_freqpoly(bins = 20)` to recreate this plot.

- Tip: Add a “linetype” roll in your cast of characters.
- Tip: Use `fct_reorder()` to order the weight class levels by `mean(R_weight)`. (See Exercise 3.2.)
Exercise 3.5. We will now create a heat map that depicts the joint distribution of R_weight and B_weight.

- Pipe ufc into ggplot.
- Set R_weight to the x-axis and B_weight to the y-axis.
- Add a `geom_density2d_filled()` layer. You should get a heat map with five distinct bumps.
- These bumps correspond to different weight classes. To illustrate the connection between the bumps and the weight classes, add an additional `geom_point()` layer that depicts the (median(R_weight), median(B_weight)) pair for each weight_class. For example, there should be a dot with coordinates (135, 135), because the Women’s Bantamweight class has median R_weight and B_weight values of 135.
  - Hint: Define variables `R_aggregate_weight = median(R_weight)` and `B_aggregate_weight = median(B_weight)` before the `ggplot()` call and nest `aes(x = R_aggregate_weight, y = B_aggregate_weight)` within the `geom_point()` call.
- Now make the size of the `geom_point()` dots larger for weight classes with more bouts. For example, the Lightweight dot should be largest, since this group has the most observations, and the Women’s Featherweight dot should be the smallest, since this group has the fewest observations.
  - Hint: Calculate the number of fights in each weight_class grouping before the `ggplot()` call and make this variable play the roll of “size.”
• Save the ggplot as `heat_map_plot`, so we can modify it later on.

We usually modify ggplots with the + symbol. For example, `heat_map_plot + scale_fill_grey()` yields a black and white version of the `heat_map_plot` plot we made in Exercise 3.5. But to change a plot’s data we use %>% rather than +. For example, the following removes the `gender = "MALE"` observations from `heat_map_plot` (which we defined in Exercise 3.5):

```r
female_sample <-
  heat_map_plot$data %>%
  filter(gender == "FEMALE")

heat_map_plot %>%
  female_sample
```

![Heat map plot](image)

In the code above, `heat_map_plot$data` is the sample underlying `heat_map_plot` and `female_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`. Also, we can shorten the code above with a bracketed expression (see Section 2.4.3). Specifically, the following code yields the same plot:

```r
heat_map_plot %>% {
  heat_map_plot$data %>%
```
3.2 Lecture

```r
filter(gender == "FEMALE")
```

Specifically, this code tells R to set the plot’s data to the result of the embedded code.

**Exercise 3.6.** Use the `%%` symbol to change the coordinates of the dots in `heat_map_plot` from `(median(R_weight), median(B_weight))` to `(mean(R_weight), mean(B_weight))`.

- Hint: Redefine `R_aggregate_weight` and `B_aggregate_weight`.

**Exercise 3.7.** We will now facet the plot we made in Exercise 3.2 by whether or not there was an upset.

- Start with `heat_map_plot` `%%` `{ }`.
- Within the brackets pipe `finish_details_plot$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 their was an upset.
  - Note: 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`.
- Exit the brackets and use `+` to add the `facet_wrap()`.
  - Tip: You can append this step to the end of the brackets, so your code looks like `heat_map_plot` `%%` `{ } + facet_wrap()`.

### 3.2.3 Pivoting with `pivot_longer()` and `pivot_wider()`

Suppose we wanted to depict the distribution of `R_height` and `B_height` in one facet and the distribution of `R_reach` and `B_reach` in another facet. At first blush this seems impossible: since every facet inherits the same cast of characters, every facet must plot the same variables. Indeed, by definition, facets depict the same variables across different collections of observations. And yet, this plot is very much possible:

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

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

data_to_plot %>%
ggplot +
aes(
  x = R,
  y = B
) +
geom_density_2d_filled() +
facet_wrap(
  vars(stat),
  ncol = 1,
  scales = "free"
) +
labs(
  x = "Red Fighter",
  y = "Blue Fighter"
)
The trick is to redefine our notions of “variable” and “observation.” For example, here’s what our initial sample looks like when we restrict attention to the relevant columns:

```r
ufc %>%
  select(fight_id, R_height, B_height, R_reach, B_reach) %>%
  head(4)
```

```
> # A tibble: 4 x 5
>   fight_id R_height B_height R_reach B_reach
>  <int>   <dbl>     <dbl>  <int>  <dbl>
> 1      1    193.     193.    193.   198.
> 2      2    180.     190.    188.   203.
> 3      3    170.     165.    183.   168.
> 4      4    173.     183.    180.   185.
```

Every `fight_id` corresponds to one observation and four numeric variables: `R_height`, `B_height`, `R_reach`, and `B_reach`. In contrast, here’s the data we pass into `ggplot()`:
3 Pivots and Plots

```r
data_to_plot %>%
  head(5)
```

> # A tibble: 5 x 4
>   fight_id stat   R   B
>   <int> <chr> <dbl> <dbl>
> 1     1    height 193. 193.
> 2     2    reach 193. 198.
> 3     3    height 180. 190.
> 4     2    reach 188. 203.
> 5     3    height 170. 165.

Now every `fight_id` corresponds to two observations—one with `stat = "height"` and the other with `stat = "reach"`—and to two numeric variables: `R` and `B`. Accordingly, we can now define `x = R` and `y = B` in our cast of characters and facet by `stat`. In this case, both facets plot the same variables—namely `R` and `B—but the interpretation of these variables changes with with `stat`: variables `R` and `B` correspond to `R_height` and `B_height` when `stat = "height"` and to `R_weight` and `B_weight` when `stat = "weight"`. Hence, the key to segregating the height variables from the reach variables is to treat them as different observations (with different `stat` values) of the same base variables (`R` and `B`). In other words, the key is to move the words “height” and “reach” from on top of the table to inside the table. This tibble rearranging is called pivoting. There are two basic pivot functions: `pivot_longer()` and `pivot_wider()`. As the names suggest, the former makes tibbles longer, with more rows and fewer columns, and the latter makes them wider, with more columns and fewer rows.

For example, define

```r
wide_tib <-
  ufc %>%
  group_by(R_fighter) %>%
  summarise(
    strong_att = mean(R_strong_attempt, na.rm = TRUE),
    strong_ach = mean(R_strong_achieve, na.rm = TRUE),
    weak_att = mean(R_weak_attempt, na.rm = TRUE),
    weak_ach = mean(R_weak_achieve, na.rm = TRUE)
  ) %>%
  na.omit

wide_tib
```

> # A tibble: 1,001 x 5
3.2 Lecture

```r
> 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

We can lengthen this tibble from 1001 rows to 4004 rows with:

```r
wide_tib %>%
pivot_longer(
  cols = c(strong_att, strong_ach, weak_att, weak_ach),
  names_to = "metric",
  values_to = "measurement"
)
```

```r
> # A tibble: 4,004 x 3
> 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 `pivot_longer()` call tells R to pool the values in `s_att`, `s_ach`, `w_att`, and `w_ach` into one column called "measurement" and to add a new column called "metric" that reports the old column names. But this version is even better:

```r
long_tib <-
  wide_tib %>%
pivot_longer(
  cols = c(strong_att, strong_ach, weak_att, weak_ach),
  names_to = "metric",
  values_to = "measurement"
)
```
cols = ~R_fighter,
names_to = c("strength", "outcome"),
names_sep = "_",
values_to = "measurement"
)

long_tib

> # A tibble: 4,004 × 4
> 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

Whereas the previous pivot_longer() call has cols = c(strong_att, strong_ach, weak_att, weak_ach), which tells R to pull down these four columns, this call has cols = ~R_fighter, which equivalently tells R to pull down all columns besides R_fighter. So in this case ~R_fighter is really just shorthand for c(strong_att, strong_ach, weak_att, weak_ach). Also, whereas the previous pivot_longer() call has names_to = "metric" this call has names_to = c("strength", "outcome"). So in this case we split what would have been the "metric" column into two separate columns, called "strength" and "outcome". And the names_sep = "_" option tells R to split the text strings at the "_" symbol (e.g., to translate metric = "strong_att" into strength = "strong" and outcome ="att").

We can recover wide_tib from long_tib with

long_tib %>%
pivot_wider(
    names_from = c(strength, outcome),
    values_from = measurement
)

> # A tibble: 1,001 × 5
> R_fighter strong_att strong_ach weak_att weak_ach
This `pivot_wider()` call tells R to pool all the measurement values that correspond to a particular `R_fighter` into a single row. In other words, it tells R to rearrange the table so that every observation has a unique `R_fighter` value. Moreover, it tells R to name the new columns according to their previous strength and outcome values.

Since we can undo `pivot_longer()` steps with `pivot_wider()` steps, and vice versa, these functions are basically inverses of one another. For example, whereas the `names_to = c("strength", "outcome")` option of `pivot_longer()` tells R to turn the column names into variables `strength` and `outcome`, the `names_from = c(strength, outcome)` option of `pivot_wider()` tells R to turn variables `strength` and `outcome` into column names. And whereas the `values_to = "measurement"` option of `pivot_longer()` tells R to consolidate multiple columns into `measurement`, the `values_from = measurement` option of `pivot_wider()` tells R to decompose `measurement` variables into multiple columns.

However, these functions are inverses only if they have analogous options. For example, `pivot_wider()` can widen `long_tib` into `wide_tib`, but it can also widen it into several other shapes, depending on the options we give it:

```
#Example 1:
medium_tib_1 <-
  long_tib %>%
  pivot_wider(
    names_from = outcome,
    values_from = measurement
  )

  head(3)

> # A tibble: 3 x 4
```
```r
> R_fighter strength att ach
> <chr> <chr> <dbl> <dbl>
> 1 Aalon Cruz strong 12 2
> 2 Aalon Cruz weak 0 0
> 3 Aaron Phillips strong 54 38

#Example 2:
medium_tib_2 <-
  long_tib %>%
pivot_wider(
    names_from = strength,
    values_from = measurement
  )

medium_tib_2 %>%
  head(3)
```

> # A tibble: 3 x 4
> R_fighter  outcome strong weak
> <chr>       <chr> <dbl> <dbl>
> 1 Aalon Cruz att 12 0
> 2 Aalon Cruz ach 2 0
> 3 Aaron Phillips att 54 225

The first example gives every (R_fighter, strength) pair its own row and every outcome value its own column and the second example gives every (R_fighter, outcome) pair its own row and every strength value its own column.

Note, translating wide_tib to medium_tib_1 or medium_tib_2 would be difficult without first translating it to long_tib—i.e., without first applying pivot_longer(). Hence, you will often apply pivot_wider() immediately after applying pivot_longer(). For example, taking the transpose of wide_tib (i.e., converting its columns to rows and rows to columns) requires both functions:

```r
#Give every R_fighter their own column
wide_tib %>%
pivot_longer(
  cols = -R_fighter,
  names_to = c("metric"),
  values_to = "measurement"
) %>%
pivot_wider(
  names_from = R_fighter,
  values_from = measurement
)
3.2 Lecture

```r
) %>%
  select(1:5) # Select first 5 columns
```

```r
> # A tibble: 4 x 5
> # Groups: metric [5]
> metric      strong_att strong_ach weak_att weak_ach
> <chr>        <dbl>   <dbl>    <dbl>    <dbl>
> 1 'Aaron Cruz'   12      54     138      99
> 2 'Aaron Phillips'   2      38      36      59
> 3 'Aaron Riley'      0     225      40     212
> 4 'Aaron Rosa'      0     192      39     197
```

By combining `pivot_longer()` steps with `pivot_wider()` steps we can transform a tibble from any shape to any shape. And this reshaping is crucial, as the different shapes permit different plots. For example, only `long_tib` can yield this plot:

```r
long_tib %>%
  ggplot() +
  aes(x = measurement) +
  geom_density() +
  facet_grid(      
    rows = vars(strength),
    cols = vars(outcome)
  )
```

![Graph](attachment:image.png)

Only `wide_tib` can yield this plot:

```r
wide_tib %>%
  mutate(weak_ratio = weak_ach / weak_att) %>%
  ggplot() +
  aes(
```

![Graph](attachment:image.png)
x = strong_att,
y = strong_ach,
colour = 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:
Exercise 3.8. 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.
- Use count() to count the number of times each name appears.
- ggplot() the result with a geom_histogram() layer.

Exercise 3.9. We will now create a plot that demonstrates that the average house take has generally increased from around 0.02 to around 0.037.

- 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.
- Use group_by() and mutate() to redefine date as the median date value across the given date_bucket.
  - Check: The observations with date_bucket = 1 should have date = 2010-03-21.
• group_by() `date` and `weight_bucket` and use summarise() to define `house_take_mean = mean(house_take)`.
• Use `pivot_wider()` to transform the tibble from one with columns `date`, `weight_bucket`, and `house_take_mean` to one with columns `date`, `heavy`, and `light`.
  – Check: The first heavy value should be 0.0382 and the first light value should be `NA` (since there were no lightweight fights in this time bucket).
• Pipe the result into `ggplot()`.
• Set the cast of characters to `aes(x = light, y = heavy, color = date)` and add a `geom_path()` layer.

**Exercise 3.10.** 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.
  – Hint: Set `cols = -c(fight_id, red_win)`.
  – Check: 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`.
  – Check: 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)`.
• Tweak the plot so that it resembles the template.
  – Tip: 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 3.11. 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?

- Hint: 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().

Exercise 3.12. Without looking at the code, try to recreate the plot presented at the beginning of this section.

We applied both pivot_longer() and pivot_wider() to wide_tib to create medium_tib_l. But we can actually derive this tibble with pivot_longer() alone by adding ".value" to its names_to input:

```r
wide_tib %>%
pivot_longer(

".
```
```r
-R_fighter,
names_to = c("strength", ".value"),
names_sep = "-")
```

> # A tibble: 2,002 x 4
> 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 98.7 48.7
> 10 Aaron Simpson weak 37.7 32
> # ... with 1,992 more rows

The ".value" text string is a special flag that tells R to lengthen the tibble only partially. Specifically, replacing `c("strength", ",outcome")` with `c("strength", ",.value")` replaces the outcome column—which takes values in "att" and "ach"—with distinct att and ach columns. In other words, this change widens the tibble along the outcome axis, which correspondingly shortens it: the output of `pivot_longer()` has 4004 rows with `names_to = c("strength", ",outcome")`, but only 2002 rows with `names_to = c("strength", ",.value")`.

We can use the same technique to derive `medium_tib_2`:

```r
wide_tib %>%
 pivot_longer(
 -R_fighter,
 names_to = c(".value", ",outcome"),
 names_sep = "-")
```

> # A tibble: 2,002 x 4
> 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
Replacing `c("strength", "outcome") with `c("value", "outcome") tells R to use the "strength" portion of the old column names as new column names. Or, put another way, it tells R to lengthen the tibble along the outcome axis but not along the strength axis.

Partially lengthening a tibble, in this manner, is roughly equivalent to fully lengthening it with `pivot_longer()` and then selectively widening it with `pivot_wider()`. For example, the following defines a tibble that stores all the numeric variables that begin with `R_` or `B_`:

```r
numeric_stats <-
  ufc %>%
  select(
    fight_id,
    where(is.numeric) & starts_with(c("R_", "B_"))
  )
numeric_stats %>%
  head(3)
```

```r
table:

<table>
<thead>
<tr>
<th></th>
<th>fight_id</th>
<th>R_odds</th>
<th>R_age</th>
<th>R_weight</th>
<th>R_height</th>
<th>R_reach</th>
<th>R_wins</th>
<th>R_losses</th>
<th>R_draw</th>
<th>B_odds</th>
<th>B_age</th>
<th>B_weight</th>
<th>B_height</th>
<th>B_reach</th>
<th>B_wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3.25</td>
<td>32</td>
<td>205</td>
<td>193.1</td>
<td>193.</td>
<td>10</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3.3</td>
<td>30</td>
<td>125</td>
<td>170</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>38</td>
<td>170</td>
<td>180.</td>
<td>188.</td>
<td>19</td>
<td>13</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>30</td>
<td>125</td>
<td>183.</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3.3</td>
<td>30</td>
<td>125</td>
<td>170.</td>
<td>183.</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3.3</td>
<td>30</td>
<td>125</td>
<td>170</td>
</tr>
</tbody>
</table>
```

> # A tibble: 3 x 37
> fight_id R_odds R_age R_weight R_height R_reach R_wins R_losses R_draw
> <int> <dbl> <int> <dbl> <dbl> <int> <int> <int> <int>
> 1 1 3.25 32 205 193. 193. 10 7 0
> 2 2 3 38 170 180. 188. 19 13 0
> 3 3 3.3 30 125 170. 183. 3 2 0
> # ... with 28 more variables: 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_odds <dbl>, 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>

The goal is to lengthen `numeric_stats` from a tibble with `fight_id`, `R_odds`, ..., `R_sub_achieve`, `B_odds`, ..., `B_sub_achieve` to a tibble with variables `fight_id`,...
color, odds, ..., sub_achieve, where color is either "R" or "B". We can do this with a semi-lengthening pivot.3

\[
\text{numeric_stats} \%\% \\
\text{pivot_longer(} \\
\quad \text{-fight_id,} \\
\quad \text{names_to = c("color", ".value"),} \\
\quad \text{names_pattern = "(.\_\_\_\_)"} \\
\) \%\%
\]

> # A tibble: 4 x 20
> # ... 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>

Or we can do this with a full lengthening pivot followed by a widening pivot:

\[
\text{numeric_stats} \%\% \\
\text{pivot_longer(} \\
\quad \text{-fight_id,} \\
\quad \text{names_to = c("color", "stat"),} \\
\quad \text{names_pattern = "(.\_\_\_)",} \\
\quad \text{values_to = "measure"} \\
\) \%\%
\]

\[
\text{pivot_wider(} \\
\quad \text{names_from = stat,}
\]

---

3We've now replaced the names_sep = "." option with the names_pattern = "(.\_\_\_)" option because we don't want to split along the second underscore. For example, want to cleave off the # from R strongly Attempt, but we don't want to further divide strongly Attempt into strong and attempt (unlike in previous exercises). And whereas names_sep = "." tells R to cut at every underscore, names_pattern = "(.\_\_\_)" tells R to cut only at the first underscore. More technically, "(.\_\_\_)" 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 \_.
3.2 Lecture

values_from = measure
) %>%
head(4)

> # A tibble: 4 x 20
> fight_id color odds age weight height reach wins losses draw
> int <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1 R 1 R 3.25 32 205 193. 193. 10 7 0
> 2 B 1 1.36 28 205 193. 198. 4 1 0
> 3 R 2 R 3 38 170 180. 188. 19 13 0
> 4 B 2 B 1.4 33 170 190. 203. 16 6 0
> # ... with 10 more variables: rounds_fought <dbl>, kd <dbl>,
> # strong_attempt <dbl>, strong_achieve <dbl>, weak_attempt <dbl>,
> # weak_achieve <dbl>, td_attempt <dbl>, td_achieve <dbl>, sub_attempt <dbl>,
> # sub_achieve <dbl>

Unfortunately, we can’t always replace a semi-lengthening pivot with traditional pivot_longer() and pivot_wider() steps. For example, the following defines a tibble that stores all variables that begin with R_ or B_:

all_stats <-
ufc %>%
select(
  fight_id,
  starts_with(c("R_", "B_"))
)

numeric_stats %>%
head(3)

> # A tibble: 3 x 37
> fight_id R_odds R_age R_weight R_height R_reach R_wins R_losses R_draw
> int <dbl> <int> <int> <dbl> <dbl> <int> <int> <int>
> 1 1 3.25 32 205 193. 193. 10 7 0
> 2 2 3 38 170 180. 188. 19 13 0
> 3 3 3.3 30 125 170. 183. 3 2 0
> # ... with 28 more variables: 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_odds <dbl>, 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>
> # B_sub_attempt <int>, B_sub_achieve <int>

Now we can semi-lengthen all_stats, as we did before:

```r
all_stats %>%
  pivot_longer(
    -fight_id,
    names_to = c("color", "value"),
    names_pattern = "(.\_)\(.\)"
  )
  head(4)
```

> # A tibble: 4 x 22
> fight_id color_fighter odds stance age weight height reach wins losses
> <int> <chr> <chr> <dbl> <fct> <int> <int> <dbl> <dbl> <int> <int>
> 1 1 R Anthon 3.25 Ortho- 32 205 193. 193. 10 7
> 2 1 B Aleksa 1.36 Ortho- 28 205 193. 198. 4 1
> 3 2 R Robbie 3 South- 38 170 180. 188. 19 13
> 4 2 B Neil M 1.4 Ortho- 33 170 190. 203. 16 6
> # ... with 11 more variables: draw <int>, 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>

But we can’t fully lengthen it:

```r
all_stats %>%
  pivot_longer(
    -fight_id,
    names_to = c("color", "stat"),
    names_pattern = "(.\_)\(.\)",
    values_to = "measure"
  )
```

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

The problem is that this pivot asks R to house the R_fighter values and the R_odds values in the same column. But this isn’t allowed, since a column can’t store both character strings and numbers. Hence, semi-lengthening pivots are especially useful when your variables are incompatible with one another.

**Exercise 3.13.** Define the following tibble:

```r
wide_punch <-
  ufc %>%
```
3.2 Lecture

```r
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>
> 1 75 151 94 125 66
> 2 76 272 137 268 102
> 3 77 309 194 157 83
> 4 78 265 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 102 9
> 9 77 R attempt 309 67
> 10 77 R achieve 194 64
> # ... with 11,066 more rows

Recreate this output without using a pivot_wider() call.

**Exercise 3.14.** Modify your solution to Exercise 3.13 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.

---

### 3.3 Lab

#### 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 packagewend its way from Handan to 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
3.3 Lab

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\(^4\), and you can find summaries of it here\(^5\) and here\(^6\).

3.3.2 Sample

We will study alibaba\_wide, which you created in Section 1.3, Exercise 1.7. 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 = 45017 has NA values for action.8-action.15 because it had only seven posted actions. (Orders with more than 15 posted actions have been removed from the sample.) Besides NA, variables action.1-action.15 can take seven other values:
  - “ORDER”: The customer places the order.
  - “CONSIGN”: The warehouse sends the package to the shipper.
  - “GOT”: The shipper receives the package.
  - “DEPARTURE”: The package departs a facility.
  - “ARRIVAL”: The package arrives at a facility.
  - “SENT\_SCAN”: The package is scanned for final delivery.

\(^5\)https://insight.kellogg.northwestern.edu/article/improve-delivery-ratings
\(^6\)https://www.wsj.com/articles/with-online-package-delivery-alerts-later-in-the-process-is-better-11585352291
- "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. In other words, all timestamps record the time that 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, however, 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 == 14314.)

### 3.3.3 Clean

We will now follow the methodology of Bray (2020) to clean our sample.

**Exercise 3.15.** Run `alibaba_wide %>% pull(score) %>% levels` 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".

- **Hint**: Consult the examples under the “Relevel with a function” heading of the `?fct_relevel` documentation.

**Exercise 3.16.** Use `pivot_longer()` 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.

- Reference the columns to pivot with `contains(".")`, since they are the only ones with a period in their names.
- 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 a character.
- Similarly use the values_transform option to define the action, facility, and shipper variables as factors.

**Exercise 3.17.** 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.
    * Note: We’ll use this variable to distinguish the packages that arrived after 1 day, those that arrived after 2 days, etc.
  - Divide time by max(time).
    * Check: 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).

Finally, we will filter out the extreme shipments to standardize our sample. All of the data filters we impose in the following exercise are from Bray (2020).

**Exercise 3.18.** We will now subject our sample to 10 data filters.

- Again start with alibaba_long and group_by() order.
  - Note: We wouldn’t need this group_by() operation if we didn’t ungroup() the data at the end of the previous exercise. However, you should commit to always “saving” your data in an ungrouped state because otherwise you’re constantly having to check whether a given tibble is grouped or not. Consistently leaving your named tibbles ungrouped helps you standardize your workflow. Also this group_by() confirms to the reader that the following filter() applies group-wise. (For more on this topic see Section @ref(wrangle_groupings).)
With a single `filter()` operation, do the following:

- Remove the orders with a "FAILURE" action. For example, you should remove all the order = 6935945 observations since this order’s 10th action was a "FAILURE".
  * Tip: Use `all()` or `any()`.
- 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".
  * Hint: Since the last action occurs at `time` = 1 you can you can ensure that no action happened after the "SIGNED" action by imposing `any(time == 1 & action == "SIGNED")`.
- Remove orders that correspond to multiple `shipper` values.
  * Tip: Use `n_distinct()`.
- Remove orders with `day_count > 8`.
- Remove orders with more than 10 or fewer than 5 posted actions.
  * Note: 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).
- Remove observations with "ORDER" and "SIGNED" actions because their `time` values are degenerate (mechanically being either 0 or 1).
  * Note: This is the only observation-level data filter. The other data filters all apply at the group level.

- `ungroup()` the filtered tibble and save it as `alibaba_long`.
  - Check: `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 tend to yield higher scores—i.e., that `time` and `score` are positively correlated. Our empirical strategy is simple: we will plot the distribution of `time` for each value of `score`. These plots will demonstrate that actions cluster more toward the `time` = 0 pole when the `score` is low and cluster more toward the `time` = 1 pole when the `score` is high. We will create six plots of this nature in the next six exercises.

**Exercise 3.19.** Use `geom_density()` to plot the distribution of `time` conditional on `score`. Give each of the five distributions a different color. Save the plot as an R object called `density_plot`, so that we can modify it in later exercises.

- Check: 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 <= 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.

Exercise 3.20. Facet density_plot by day_count.

• Tip: Do not recreate density_plot from scratch. Just append the facet_wrap() to it with +.
• Tip: Instead of vars(day_count) use vars(paste("Day Count: ", day_count)).

Exercise 3.21. Facet density_plot by the number of actions in a given order.

• Tip: Use %>% to modify the data underlying density_plot. Specifically, add a variable to this tibble called action_count that records the number of actions (i.e., the number of observations) corresponding to a given order value.
• Tip: Use add_count() to derive action_count.
• Check: 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 3.22. 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".
  – Note: This change will make our facet labels look a little nice.
• Use fct_reorder() to order the action labels by their corresponding mean(time).
  – Tip: See the ?fct_reorder documentation.
  – Note: This change will arrange the box plots in increasing order.
  – Check: The first label should be "Consign".
• Pipe the result into ggplot().
• Set x = score and y = time in your cast of characters.
• Add a geom_boxplot() layer.
  – Tip: The plot looks better if you use the outlier.shape = NA option.
• Add a coord_flip() and facet_wrap() by action.
  – Tip: Use the strip.position = "right" and ncol=1 options.
• Call this plot box_plot.

Exercise 3.23. 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 update the data underlying box_plot. Specifically, keep only the (shipper, category) pairs that have at least 1000 observations.
\- Check: You should have 23 distinct (shipper, category) groups after this step.
- facet_grid() by shipper and category.
  \- Tip: Rather than rows = vars(shipper) and cols = vars(category), try using rows = vars(paste("Ship: ", shipper)) and cols = vars(paste("Cat: ", category)).
  \- Tip: Add theme(axis.text.x = element_text(angle = -90)) to your plot specification, to buy more room on the horizontal axis.

**Exercise 3.24.** So far we’ve illustated 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.
  \- Tip: The n() function returns the size of a group_by() group.
  \- Check: The order = 1117792, score = 5, and action = "DEPARTURE" row should have n = 3 and time = 0.344.
- 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.
  \- Check: Your tibble should have 17895 rows after this step.
- Use filter() to impose n_ARRIVAL %in% 1:2 and n_DEPARTURE %in% 1:2. 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 paste("Arrivals: ", n_ARRIVAL) and paste("Departures: ", n_DEPARTURE) in the columns and paste("Score: ", score) in the rows.
  \- Note: 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. If we didn’t control for these variables our results would likely be biased, since the total number of "ARRIVAL" or "DEPARTURE" actions likely influences both score and time.
  \- Check: 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")
```

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 Benefits of Long Tibbles

Long tibbles are generally more practicable than wide tibbles. There are a few reasons for this. First, it’s usually easier to work with data stored inside a tibble than data stored above a tibble. 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,
```
Unfortunately, merging `unit_of_measure` and `rocky_planets_1` would be difficult, since the latter stores `radius`, `sun_dist`, and `year` as metadata—i.e., as column names. However, if we convert these metadata into formal data 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

> # 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

rocky_planets_cleaned <- rocky_planets_merged %>%
  mutate(unit = paste0("", unit, ",")) %>%
  unite(stat, c("stat", "unit"), sep = " ")
### 3.4 Commentary

```r
rocky_planets_cleaned

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

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 3.17 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,
3.4 Commentary

```r
time.13 = time.13/time_max,
time.14 = time.14/time_max,
time.15 = time.15/time_max
)

select(-time_min, -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 its 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_1

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

wide_example_2

long_example <-
  alibaba_long %>%
  group_by(action, action_num) %>%
  summarise(
    mean_time = mean(time),
    .groups = "drop"
  )
```
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
3.4 Commentary

action = "ARRIVAL" mean_time values, for each value of action_num. We could
derive this from long_example without widening the tibble:

```r
long_example %>%
group_by(action_num) %>%
summarise(
  time_diff =
    weighted.mean(mean_time, action == "DEPARTURE") -
    weighted.mean(mean_time, action == "ARRIVAL")
)
```

However, the solution is more elegant if we use pivot_wider():

```r
long_example %>%
pivot_wider(
  names_from = action,
  values_from = mean_time
) %>%
mutate(time_diff = DEPARTURE - ARRIVAL)
```

3.4.3 Aggregating Pivots

Sometimes a pivot_wider() operation shortens a tibble faster than it widens it so that the reconfigured table doesn’t have enough cells to comfortably store all the information. For example, consider the following tibble:

```r
R_from_july_2011 <-
ufc %>%
filter(
  floor_date(date, "month") == "2011-07-01"
) %>%
select(
  R_fighter,
  red_win,
  result
)

R_from_july_2011
```

> # A tibble: 11 x 3
>    R_fighter red_win result
>    <chr>     <lgl>    <fct>
And now let’s pivot this tibble so that the values of `red_win` run along the rows and the values of `result` run along the columns:

```r
R_from_july_2011 %>%
pivot_wider(
  names_from = result,
  values_from = R_fighter
)
```

As you see, we now get the warning that “Values are not uniquely identified; output will contain list-cols.” 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 x 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 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:
### 3.4 Commentary

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

> # A tibble: 2 x 4
> red_win 'U-DEC' 'KO/TKO' SUB
> <lgl> <int> <int> <int>
> 1 TRUE 5 2 1
> 2 FALSE 1 2 NA

For example, the first value in the second column is now `length(c("Dominick Cruz", "Dennis Siver", "Brian Bowles", "Anthony Njokuani", "Jeff Hougland")) = 5`. The bottom-right value is NA because there's no vector of observations with `red_win = FALSE` and `result = "SUB"` that R can take the `length()` of. And when a cell has no corresponding vector, R assigns it the default value of NA. However, we can change this default value to zero with the `values_fill = 0` option:

```r
R_from_july_2011 %>%
pivot_wider(
    names_from = result,
    values_from = R_fighter,
    values_fn = length,
    values_fill = 0
  )
```

> # A tibble: 2 x 4
> red_win 'U-DEC' 'KO/TKO' SUB
> <lgl> <int> <int> <int>
> 1 TRUE 5 2 1
> 2 FALSE 1 2 0

Now instead of `length()`, we can pass in any aggregating function we like into the `values_fn` option. For example, the following tells R to use `str_flatten()` to compress the multiple text strings assigned to a cell into one long text string:

```r
R_from_july_2011 %>%
pivot_wider(
    names_from = result,
    values_from = R_fighter,
    values_fn = str_flatten
  )
```
values_fn = str_flatten,
values_fill = "NO DATA"
)

> # A tibble: 2 x 4
> red_win 'U-DEC' 'KO/TKO' SUB
> <lgl> <chr> <chr>
> 1 TRUE Dominick Cruz Dennis Siver Brian Bowles Carlos Condit Melvin G-Tito O-
> 2 FALSE Brad Tavares Wanderlei Silva George 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 Cruz Dennis Siver Brian Bowles Anthony Njokuani Jeff Hougland". And the values_fill = "NO DATA" option tells R to make a cell store "NO DATA" if it has no other corresponding text strings.

And we can even pass in home-made functions (see Section 2.2.3):
3.4 Commentary

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
> <lg> <chr> <chr> <chr>
> 1 FALSE Tavares Silva, Sotiropoulos <NA>
> 2 TRUE Cruz, Siver, Bowles, Njokuani, Houglund Condit, Guillard Ortiz

Exercise 3.25. The following reports the average time in the round that each
finishing move took place, by finishing_round and gender:

ufc %>%
filter(
  !is.na(finish_round_time),
  !is.na(finish_details)
) %>%
group_by(finish_details, finish_round, gender) %>%
summarise(
  mean_time = mean(finish_round_time),
  .groups = "drop"
) %>%
pivot_wider(
  names_from = c(finish_round, gender),
  values_from = mean_time
)

> # A tibble: 27 x 10
> finish_details '1_MALE' '2_MALE' '3_MALE' '1_FEMALE' '2_FEMALE' '4_MALE'
> <chr> <chr> <chr> <chr> <chr> <chr>
> 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 Ch-  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_FEMALE’ <dbl>

Recreate this tibble without the group_by() and summarise() steps. (Your rows and columns may be ordered differently.)

- Tip: 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 3.26.** 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.
  - Check: 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.
  - Check: Your final tibble should have 145 rows and 14 columns.

### 3.4.4 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
3.5 Solutions

ufc %>%
mutable(
  house_take = 1/R_odds + 1/B_odds - 1,
  country = str_extract(location, "\\b[^,]+\$", 1)
)
)

```
ggplot +
aes(  
x = country,
  y = house_take
)
+
geom_violin() +
coord_flip() +
labs(  
x = "",
  y = "House Take"
)
```

---

3.2
```r
finish_details_plot <-
ufc %>%
filter(
  result != "DQ",
  !is.na(finish_details),
  !is.na(finish_round_time)
) %>%
group_by(finish_details) %>%
mutate(
  finish_details =
    ifelse(  
      n() < 18,
      paste(result, "Other"),
      finish_details
    )
  ) %>%
ungroup %>%
mutate(  
  finish_details =
    fct_reorder(  
      finish_details,
      finish_round_time,
      median,
      .desc = TRUE
    )
  ) %>%
ggplot() +
aes(  
  x = finish_round_time,
  y = finish_details,
  fill = result
  ) %>%
geom_boxplot() +
theme_bw() +
labs(  
  x = "Time in Round",
  y = "Finishing Move"
  ) +
theme(  
  legend.position = "top",
  legend.title = element_blank()
)
)

finish_details_plot
```
3.3

```r
ufc %>%
  filter(rounds == 3) %>%
  ggplot() +
  aes(
    x = abs(B_odds - R_odds),
    y = fight_time
  ) +
  geom_point(size = .2) +
  geom_smooth(method = "lm") +
  facet_wrap(
    vars(weight_class),
    scales = "free",
    ncol = 2
  )
```

3.4

```r
ufc %>%
  mutate(
    weight_class =
    fct_reorder(
```
weight_class,
  R_weight,
  mean
)
) %>%
  ggplot +
  aes(
    x = date,
    colour = weight_class,
    linetype = weight_class
  ) +
  geom_freqpoly(bins = 20) +
  facet_wrap(
    vars(gender),
    ncol = 1
  ) +
  scale_color_grey() +
  theme_bw()

3.5

heat_map_plot <-
  ufc %>%
  group_by(weight_class) %>%
  mutate(
    R_aggregate_weight = median(R_weight),
    B_aggregate_weight = median(B_weight),
    num_bouts = n()
  ) %>%
  ggplot +
  geom_density2d_filled(
    aes(
      x = R_weight,
      y = B_weight
    )
  ) +
  geom_point(
    aes(
      x = R_aggregate_weight,
      y = B_aggregate_weight,
      size = num_bouts
    )
  ) +
3.5 Solutions

```r
theme(legend.position = "none")

heat_map_plot
```

3.6

```r
heat_map_plot %>% {
  heat_map_plot$data %>%
    group_by(weight_class) %>%
    mutate(
      R_aggregate_weight = mean(R_weight),
      B_aggregate_weight = mean(B_weight)
    )
}
```

3.7

```r
finish_details_plot %>% {
  finish_details_plot$data %>%
  mutate(
    upset =
    ifelse(
      red_win,
      R_odds > B_odds,
    
      B_odds > R_odds,
      red_win == B_odds,
    
      FALSE
    )
  )
```
3 Pivots and Plots

```r
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")
)
```

3.8

```r
ufc %>%
pivot_longer(c(R_fighter, B_fighter)) %>%
count(value) %>%
ggplot +
aes(x = n) +
geom_histogram()
```
3.5 Solutions

3.9

```r
ufc %>%
  mutate(
    house_take = 1/R.odds + 1/B.odds - 1,
    date_bucket = ntile(date, 200),
    weight_bucket = ntile(R_weight + B_weight, 2),
    weight_bucket =
      ifelse(
        weight_bucket == 1,
        "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"

3.10

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(~c(fight_id, red_win),
  names_to = c("fighter", "move", "outcome"),
names_sep = "_"
) %>%
na.omit %>%
pivot_wider(
  names_from = fighter,
  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,
  colour = red_win
) %>%
geom_point() +
facet_wrap(
  vars(outcome, move),
  scales = "free",
  nrow = 2
) +
theme_bw() +
theme(
  legend.position = "top",
  legend.title = element_blank(),
  strip.background =
    element_rect(color = "white", fill = "white")
) +
labs(
  x = "Red Fighter",
  y = "Blue Fighter"
)
3.11

#The takedown facets are poorly scaled under facet_grid:
red_blue_scatterplot +
  facet_grid(
    rows = vars(outcome),
    cols = vars(move),
    scales = "free"
  )

Red Win  Blue Win

#Switching the rows and columns doesn’t help:
red_blue_scatterplot +
  facet_grid(
    rows = vars(move),
    cols = vars(outcome),
    scales = "free"
  )
3.13

```r
# wide_punch %>%
# pivot_longer(
#   ~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 1 75 R attempt 151 190
> 2 2 75 R achieve 94 169
> 3 3 75 B attempt 125 32
> 4 4 75 B achieve 66 22
> 5 5 76 R attempt 272 3
> 6 6 76 R achieve 137 2
> 7 7 76 B attempt 268 9
> 8 8 76 B achieve 102 9
> 9 9 77 R attempt 399 67
> 10 10 77 R achieve 194 64
> # ... with 11,066 more rows
```

3.14

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

> # 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

```

```r
c # wide_punch %>%
pivot_longer(
  -fight_id,
  names_to = c("\value", "strength", "\outcome"),
  names_sep = ":"
)

> # A tibble: 11,076 x 5
> fight_id strength outcome attempt achieve
> <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
> 8  76 weak  achieve  2       9
> 9  77 strong attempt 309     157
>10  77 strong achieve 194     83
> # ... with 11,066 more rows
```
3.5 Solutions

3.25

```r
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
> finish_details '2_FEMALE' '1_MALE' '1_FEMALE' '2_MALE' '3_MALE' '5_MALE'
> <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
> 1 Rear Naked Ch- 145. 172. 192. 171. 177. NA
> 2 Armbar        182. 183. 186. 185 210. 239
> 3 Guillotine Ch- NA 161. 211. 166. 161. NA
> 4 Punch         188. 149. 148 161. 151. 166.
> 5 Knee          NA 143. 146 201. 179. NA
> 6 Triangle Choke 220 169. 176 160. 220. 190
> 7 Heel Hook     NA 107  NA  NA  NA  NA
> 8 Kneebars      NA 137. 122 181. 143 NA
> 9 Kick          34 150. 115 134. 136. 140
> 10 D'Arce Choke NA 156. 223 158. 147. NA
> # ... with 17 more rows, and 3 more variables: '3_FEMALE' <dbl>,
> # '4_MALE' <dbl>, '5_FEMALE' <dbl>

3.26

```r
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
```r
> location 'Light Heavywei- Welterweight 'Women's Flywei- Featherweight
> <chr> <dbl> <dbl> <dbl>
> 1 Las Veg- 31 30 31 28
> 2 Abu Dha- 31.5 33 26.5 30.5
> 3 Jackson- 32.5 31.5 33 31
> 4 Brasili- 27.5 32.5 28 29
> 5 Norfolk- 28 29.5 NA 29
> 6 Auckland- 27.5 28 30.5 28.5
> 7 Minneap- 31 30.5 NA 29
> 8 Greenvi- NA 29 27 30
> 9 Rio Ran- 31.5 34 29 NA
> 10 Sacrame- 28 27.5 NA 30.5
> # ... 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>
```
4

Joins

&&Define data wrangling, as opposed to the tidyverse &&Rather than teach you every wrangling trick, I’ll teach you what you need to know to progress through the book. And I’ll teach you the other wrangling tricks as you go. That said, for additional wrangling tricks, see Sections ??glmNumber2Vector), ...

4.1 Homework

&&Make commentary follow lab. &&Turn Homework into intro &&Suppose you want to allocate a random number to each group. You can do this with group_by and mutate, but this will require calling sample() individually for each group, which is way slow! It’s faster to call sample once offline, allocating each group its random number, and then join it in. &&The ggplot lab does all its filters with group_by() steps. This is doable, but it’s super slow. So explain how it’s after to take the calculation offline.

4.1.1 Readings

- Sections 4.2.1 and 4.2.2 cover the “tidy-select” protocol for specifying variables. You may find vignette("colwise") and vignette("programming") helpful when working through these sections. Rather than memorize all the tidy-select options, I recommend that you look them up as needed. (At least that’s what I do.)
- Sections 4.2.3 and 4.2.4 pertain to mutating joins and filtering joins. For more on these joins see vignette("two-table").
4.1.2 Exercises

Download the chess data posted here\(^1\), open it with \texttt{read_csv()}, and load it into a tibble called \texttt{chess_games}. This tibble describes 19,113 chess games played on \texttt{lichess.org}. We’ll now work with this sample a bit to prepare us and it for the lecture in Section 4.2.

**Exercise 4.1.** Plot a histogram of the number of \texttt{turns} the games comprise. Use the \texttt{binwidth = 1} option of \texttt{geom_histogram()} to give each number of moves its own bar. And \texttt{facet_wrap()} by \texttt{victory_status}, with \texttt{scales = "free_y"} and \texttt{ncol = 1} options. These latter options will help us see how \texttt{victory_status} influences the distribution of \texttt{turns}.

As expected, we find that games played to a checkmate last longer than those cut short by a resignation, but not as long as those that muddle along until a draw or until someone runs out of time. Also, we find resignation and time violation spikes at \texttt{turn <= 2}. There are two reasons for these spikes: (i) players sometimes immediately quit games they don’t want to play, and (ii) players sometimes aren’t at their computer when they’re assigned to a game. We will remove these degenerate games in the next exercise.

**Exercise 4.2.**

- \texttt{filter()} out the games in \texttt{chess_games} that have fewer than eight \texttt{turns}: a game that’s aborted before the eighth move isn’t a proper game.
- \texttt{filter()} out the games with more than a 250-point rating differential between the players (i.e., between \texttt{white_rating} differs from \texttt{black_rating})—these games are too lopsided to take seriously.
  - Tip: You can impose this restriction and the previous restriction with one \texttt{filter()} call. (Just separate the conditions with a comma.)
- Call your tibble \texttt{chess_panel}.
  - Check: \texttt{chess_panel} tibble should now comprise 14,322 games.

The games are timed, so the clock plays a central role. Actually, there are two clocks, to keep track of the two players’ remaining time. The white clock counts down when it’s the white player’s turn to move and the black clock runs down when it’s the black player’s turn to move. If one player runs out of time, the other player wins. \texttt{increment_code} specifies the amount of time allotted to each player. The \texttt{increment_code} values have the form \(x+y\), where \(x\) is the number of minutes initially put on the clock and \(y\) is the number of seconds added to the clock after each move. For example, in a game with \texttt{increment_code == "15+0"} the players both have a 15-minute budget to make

\(^1\text{https://www.dropbox.com/s/0fy8gfo4k48tlak/chess_games.csv?dl=0}\)
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. The next exercise confirms that a generous time allowances decreases the probability of the clock being the decisive factor.

Exercise 4.3.

• Use separate() to divide the increment_code of chess_panel into two variables called initial_time and extra_time, such that if the increment_code were $x + y$ then initial_time would be $x$ and extra_time would be $y$.
  – Tip: Use the convert = TRUE option to make increment_code and extra_time integer variables, rather than text string variables.
• Define logical variable time_out that indicates whether a player ran out of time.
  – Hint: Consult the victory_status variable.
• Disregard all variables besides game, initial_time, extra_time, time_out.
  – Note: This step is actually unnecessary—the code would produce the same output without it. But while it doesn’t affect the computer, disregarding unused variables can make it easier for us humans to follow along. This is especially true in this case, since the next step is a pivot_longer() operation that will dramatically alter the configuration of the table: disregarding the superfluous variables will make the pivot_longer() step more intelligible.
• Use pivot_longer() to transform the table into one with names game, time_out, value, and name, where the values of the last variable are either "initial_time" or "extra_time".
  – Check: The tibble should have 28,644 rows after this step.
• Use pmin() to cap the value of value at 50. That is, replace any values in excess of 50 with 50.
  – Note: We perform this step for purely aesthetic reasons. Truncating value in this manner will “zooms in” the plot, making it easier to read.
• Boxplot the distribution of value by name and time_out.
  – Note: There are two ways to make these plots: with aes(x=name, y=value, colour=time_out) or with aes(x=name, y=value, colour=time_out). Choose the specification that better enables us to compare the time_out == TRUE games with the time_out == FALSE games.

As you see, the games that timed out tended to have lower initial_time and extra_time values. Hence, the clock is more salient in speed chess (naturally).
4.2 Lecture

We will now apply some more advanced data wrangling techniques to chess_panel, a tibble we constructed in Exercise 4.2.

4.2.1 Tidy-Select

The moves variable of chess_panel records the games’ moves in chess notation. For example, game == "msIqfn6q" has moves == "e3 e5 Bc4 Qf6 Qf3 Qg6 Qd5 Nh6", which indicates that white first moved a 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. But storing all the moves in one variable is inconvenient. For example, there’s no easy way for to select, say, the fifth move. Structuring the data in a different way, the following tibble gives each of the first 200 moves its own variable:

```r
first_200_moves <-
  chess_panel %>%
  select(game, moves) %>%
  separate(
    moves,
    paste0("move_", 1:200),
    sep = " ")
```

For example, game == "iyPuQJxx" lasted 15 turns, so it has chess moves in variables move_1-move_15 and NA terms in variables move_16-move_200.

Since white always goes first, it makes all the odd moves and black all the even moves. Suppose we wanted to select all the white moves. We could do so by listing all 100 odd move variables in a select() operation:

```r
first_200_moves %>%
  select(
    move_1,
    move_3,
    move_5,
    #...
```
4.2 Lecture

```r
move_199
```

But that would be ridiculous. 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. This tidy-select standard is a collection of functions that 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(everything())` picks all columns,
- `select(contains("20"))` picks `move_20`, `move_120`, and `move_200`,
- `select(where(is.character))` picks the character variables, or
- `select(where(~any(str_detect(.x, "Nxf6"), 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 to each column. This test 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 pre-defined R function, such as `is.character`, or some code of our own creating, such as `~any(str_detect(.x, "Nxf6"), na.rm=TRUE)`.

In the latter case, we always begin with a tilde, `~`, which indicates that we're using our own chunk of code, and then we include our code chunk inside a set of curly brackets, `{ }`. And within these brackets, `.x` serves as a stand-in for the current variable. Thus, the `select()` operation picks `game` if `any(str_detect(game, "Nxf6"), na.rm=TRUE) = TRUE`, picks `move_1` if `any(str_detect(move_1, "Nxf6"), na.rm=TRUE) = TRUE`, etc.

The following exercises provide additional tidy-select examples.

**Exercise 4.4.** Use `where()` and `is.numeric()` to select the numeric columns of `chess_panel`.

**Exercise 4.5.** Use `&` and `ends_with()` to select all the black moves between the 20th and 60th turns (inclusive).

**Exercise 4.6.** When pawns make it to the other side of the board—i.e., when

---

2 You can actually forgo the brackets if your snippet of code is only one line long.
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 they are the only moves whose notation incorporates an equals sign. For example, the 62nd move of `game = "jw1DvLmW"` 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.
- Write `~()` inside the `where()` to indicate that we’ll be using a custom-tailored column selection criterion (rather than some pre-defined R function).
- Within the brackets, use `str_detect()` to write a snippet of code that maps the current variable, denoted by `.x`, to `TRUE` if at least 10 of its elements contain an equals sign, `"="`, and returns `FALSE` otherwise.
  - Tip: Use the `na.rm=TRUE` option of `sum()`.
  - Tip: You don’t have to break the pipe to evaluate whether a number is as large as 10. For example, `11 %>% { . >= 10 } = TRUE` and `9 %>% { . >= 10 } = FALSE`.

The tidy-select convention isn’t limited to `select()`. In fact, it’s been rolled out to nearly all tidyverse functions. For example, I can pivot by all numeric variables with:

```r
chess_panel %>%
  pivot_longer(where(is.numeric)) %>%
  select(game, name, value)
```

Or I can `group_by()` every variable with

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

The tidy-select protocol is especially powerful when combined with `mutate()` and and `summarise()`, as we’ll see in the next section.

### 4.2.2 Multivariable Operations

`across()` enables you tidy-select the variables you wish to `mutate()` or `summarise()`. For example, suppose I wanted to flag all white moves with "_w" and all black moves with ".b”. Rather than spell out 200 distinct `paste()` transformations, I could use one `across()` step to append ".w" to the odd moves and another `across()` step to append ".b" to the even moves:
4.2 Lecture

\[
\text{first}_200\text{.moves } \%\% \\
\text{mutate(} \\
\text{\hspace{1em} \text{across(}} \\
\text{\hspace{2em} \text{ends_with(c("1", "3", "5", "7", "9")),}} \\
\text{\hspace{2em} \text{paste,}} \\
\text{\hspace{2em} "w",} \\
\text{\hspace{2em} \text{sep = "_"}}} \\
\text{\hspace{1em} \text{across(}} \\
\text{\hspace{2em} \text{ends_with(c("0", "2", "4", "6", "8")),}} \\
\text{\hspace{2em} \text{paste,}} \\
\text{\hspace{2em} "b",} \\
\text{\hspace{2em} \text{sep = "_"}}} \\
\text{\hspace{1em})} \\
\text{\)}}
\]

\textit{across()} must receive a set of variables—specified via tidy-select—and a function, and can optionally receive an auxiliary set of function parameters. It then performs the function on the variables (under the specified parameters, if provided). For example, the first \textit{across()} call above applies \textit{paste()}, with additional parameters "w" and \textit{sep = "_"}, to the set of variables that end in an odd number.

The function you pass into \textit{across()} could be something pre-defined, like \textit{paste()}, or it could be a chunk of code you create especially for the occasion. Thus, you can use \textit{across()} to implement any calculation of your choosing, not just just those with pre-defined R functions. For example, the following strings two \textit{str_replace()} steps together to replaces checkmate symbol "#" with "_checkmate" and check symbol "+" with "_check" (note, for technical reasons R requires us to refer to the plus sign with "\+" rather than "*+"):

\[
\text{first}_200\text{.moves } \%\% \\
\text{mutate(} \\
\text{\hspace{1em} \text{across(}} \\
\text{\hspace{2em} \text{\text{-game,}}}} \\
\text{\hspace{2em} \{}} \\
\text{\hspace{3em} \text{\text{x } \%\%}} \\
\text{\hspace{3em} \text{\text{str_replace("#", "_checkmate") } \%\%}} \\
\text{\hspace{3em} \text{str_replace("\+", "_check")}}} \\
\text{\hspace{2em} \}}} \\
\text{\)}}
\]

The first line in \textit{across()} specifies that we want to transform all variables
besides game. And the rest follows the notation we saw in Section 4.2.1 with where(). It begins with a tilde, ~, to tells R that we’re going to provide our own code snippet, which we put inside a set of curly brackets, { }. Our snippet of code can refer to the variable we’re currently transforming with .x. That is, the across() operation will first set move_1 to the snippet of code evaluated under .x = move_1, and then set move_2 to the snippet of code evaluated under .x = move_2, and so forth.

The following Exercise provides another example of across().

**Exercise 4.7.** 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 "0-0"—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 "0-0-0"—the king moves from square e1 to c1 if white and from e8 to c8 if black. Rather than this inconsistent "0-0" and "0-0-0" notation, we’ll express these castle moves with the same piece-location notation used for the other moves. That is, we will change "0-0" to "Kg1" if the piece is white and "Kg8" if black, and change "0-0-0" 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.
  - Tip: Recall how we appended ".w" to white moves and ".b" to black moves.
- In the white across() change "0-0-0" to "Kc1" and "0-0" to "Kg1".
  - Tip: Recall how we changed ".#" to "._checkmate" and ".*" to "._check".
  - Tip: Change "0-0-0" before "0-0". If you do it the other way around then your first step will will change "0-0-0" to "Kg1-0".
- In the black across() change "0-0-0" to "Kc8" and "0-0" to "Kg8".

across() also works with summarise() operations, as the following example illustrates.

**Exercise 4.8.** Pawn moves are those that do not begin with "R", "N", "B", "Q", "K", or "O". We will now plot the fraction of moves made by pawns, as a function of the turn number.

- Pipe first_200_moves into a summarise() call that contains a nested across() call.
- Make all variables besides game subject to the across() operation.
- Use ~() to pass your own code snippet into the across().
- Start your code snippet by piping .x into str_sub() to select the first letter of each move.
4.2 Lecture

- Tip: Use the end = . option.
- Use str_detect() to identify the pawn moves.
  - Hint: You can identify specific characters by listing them inside brackets, "[]". For example, 
    ```r
    c("a", "x", "b") %>% str_detect("[abc]")
    ```
    - Tip: Use the negate=TRUE option.
- Use mean() to calculate the fraction of moves that correspond to pawns. Let this be the last line of your nested code snippet.
  - Tip: Use the na.rm=TRUE option.
- Pipe the output of your summary() function into pivot_longer() to give it the long format ggplot() expects. Call the variable saving the variable names name (the default value) and the variable storing the pawn fractions frac.
  - Tip: Select everything().
  - Tip: Use the values_to = . option.
- Now the name variable should have values "move_1", "move_2", ... "move_280". Use separate() to turn the number in each of these strings into an integer variable called move_num.
  - Tip: Use sep = "_" to divide the strings by the underscore.
  - Tip: Use into = c(NA, "move_num"). The NA tells R to disregard the part before the underscore, which is always "move", and the "move_num" gives the integer part its name.
  - Tip: Use convert = TRUE to make move_num is an integer.
- Use ggplot() to scatterplot frac by move_num.

4.2.3 Mutate with External Data

So far our mutate() steps have used only the information at hand, in the given row of data. For example, in Exercise 4.3 we derived time_out from victory_status, a variable that was standing at the ready just a few columns over. However, we will sometimes want to construct variables with information stored in different rows of the table. We’ll now learn how to do this.

The simplest case is when the statistic is constant across some group_by() grouping that the observation belongs to. In this case, we just need to group_by() before we mutate(). We saw some examples of this in Section 2.2.6. For another example, consider the following, which adds a variable that records the average white win rate among the games with the same opening_eco (a variable that specifies the chess opening) and opening_ply (a variable that specifies the number of opening moves).

```
chess_panel %>%
group_by(opening_eco, opening_ply) %>%
mutate(white_win_frac = mean(winner == "white"))
```
In this case, the value returned by `mean()` is assigned to all the observations corresponding to a given `(opening_eco, opening_ply)` pair. For example, the 239 games with `opening_eco == "C41"` and `opening_ply == 5` each receive `white_win_frac = 0.577`. The following exercise provides another example.

**Exercise 4.9.** Use `group_by()` and `n()` to add a variable to `chess_panel` that specifies the total number of times that the current white player has played as white.

The examples above uses *summary* functions `mean()` and `n()`, which map *n* input values into one output value. But the technique also works with *window* functions, which map *n* input values into *n* output values. Examples of window functions are `cumsum()`, `rank()`, and `lag()`, all of which return vectors that are as long as those passed into them. For example, the following adds in a logical variable that specifies whether white won their last game.

```
chess_panel %>%
group_by(white_id) %>%
mutate(white_won_last_game = lag(winner == "white"))
```

Note that the `lag()` operation accesses information stored outside of the current row but inside the current `group_by()` chunk. The following exercise provides another example in which we use a window function to extract information about other games.

**Exercise 4.10.** Exercise 4.1 illustrates a clear relationship between `turns` and `victory_status`. However, while `turns` influences *whether* someone wins, does it influence *who* wins? To determine this, we will will plot the fraction of games white wins as a function of `turns`.

- Start with `chess_panel` and `filter()` out games that ended in a draw.
  - Check: You should have 13,610 games after this step.
- Use `mutate()` and `ntile()` to create a variable called `turn_decile` that specifies the decile of the `turns` variable. That is, you should have `turn_decile = 1` when `turns` is below the 10th percentile, `turn_decile = 2` when `turns` is between the 10th and 20th percentiles, `turn_decile = 3` when `turns` is between the 20th and 30th percentiles, etc.
  - Check: `ntile()` is the window function that uses information from other rows. We don’t use `group_by()` because the relevant group is the entire sample.
  - Check: Use `count()` to confirm that each value of `turn_decile` has 1,361 observations.
- `group_by()` `turn_decile`.
- Use `summarise()` to calculate the fraction of observations in which `winner == "white"` (for each value of `turn_decile`). Call this variable `white_win_prob.`
Tip: You can calculate `white_win_prob` with the `mean()` function. When you apply `mean()` to logical statements, such as `winner == "white"`, it treats the `TRUE` values as ones and the `FALSE` values as zeros. For example, `mean(3 < 1:10)` returns 0.7, because 7 out of the 10 numbers in 1:10 are larger than 3.

- Scatter plot `turn_decile` and `white_win_prob`.

There seems to be a negative relationship between `turn_decile` and `white_win_prob`. And this makes sense: the initial advantage white enjoys by moving first wears off over the course of a long game.

It’s more difficult to add in information that’s not confined to a `group_by()` chunk that encompasses the current observation. For example, in Exercise 4.9 we added in a variable that counts the total number of times that the current white player played as white. But now suppose we wanted to add a variable that specifies the total number of time that the current white player played as black. Our previous `group_by()` trick wouldn’t work because the current observation doesn’t reside in the relevant group. For example, `game == "epDNgi0"` has `white_id == "yokocito"` and `black_id == "luis77"`; thus it belongs to the `black_id == "luis77"` group, but it requires information from the `black_id == "yokocito"` group. So to add this new variable we must be able to derive statistics that pertain to foreign groups. To do so, we must use `joins`, such as `left_join()` or `inner_join()`. For example, we can `left_join()` in the number of times that the current white player played as black:

```r
black_count <-
count(
  chess_panel,
  black_id,
  name = "times_white_played_black"
)

chess_panel %>%
  left_join(
    black_count,
    by = c("white_id" = "black_id")
  ) %>%
  replace_na(list(times_white_played_black = 0))
```

In the code above, I first calculate and store the desired statistics in an ancillary tibble called `black_count`. I then `left_join()` this tibble in with `chess_panel`, matching `black_id` in the source table (`black_count`) with `white_id` in the target table (`chess_panel`). The `left_join()` step adds `times_white_played_black` to `chess_panel`, which is what I wanted. But not all
rows of `chess_panel` receive a value for `times_white_played_black`. Specifically, `white_played_black_count` is `NA` immediately after the `left_join()` whenever the player specified by `white_id` played no black games. For example, `game == "mIIcQgHh"` has `white_id == "ischia"`, but we never observed this player play as black and thus have no record of them in `black_count`, which means that its `times_white_played_black` value is left as `NA`. However, these `NA` values should should be zero—since they indicate that zero black games were played—and we make this substitution with the final `replace_na()` step.

The solution above has an annoying feature: it creates disposable tibble `black_count`, which will linger in your environment, cluttering up your R session long after its single use. Fortunately, we can avoid creating this computational detritus by *nesting the query within the join*. To do so, we simply replace the `black_count` term in the `inner_join()` with the code that defines it:

```r
chess_panel %>%
  left_join(
    count(      # ..ά+)'
      chess_panel,
      black_id,
      name = "times_white_played_black"
    ),
    by = c("white_id" = "black_id")
  ) %>%
  replace_na(list(times_white_played_black = 0))
```

R solves nested queries like this inside out: first it performs the nested `count()` step and then it uses the output of that to implement the `left_join()`.

Now we’ve cut out the junk table, but we can still simplify our code a bit further to the following:

```r
chess_panel %>%
  left_join(
    count(
      chess_panel, # ..ά+)'
      black_id,
      name = "times_white_played_black"
    ),
    by = c("white_id" = "black_id")
  ) %>%
  replace_na(list(times_white_played_black = 0))
```

What we’ve done is replace the nested reference to `chess_panel` with a dot. In
general, whenever your code is inside the parentheses of a function, you can access the data that gets piped into that function with \(.\). Hence, the \(\cdot\) term gets set to whatever was piped into the outer `left_join()` function, which in this case was `chess_panel`. For another example, consider the following:

```r
1:5 %>%
tibble(
  x = ., 
  y = mean(.), 
  z = .^2
)
```

> # A tibble: 5 x 3
> x   y    z
> <int> <dbl> <dbl>
> 1    1    3
> 2    2    4
> 3    3    9
> 4    4   16
> 5    5   25

In this case, all the nested \(\cdot\) terms refer to the piped in vector, 1:5.

You’ll find this \(\cdot\) notation surprisingly useful, but it has one minor caveat: it’s incompatible with pipes, inasmuch as you can’t write \(\%\%\%\). For example, I couldn’t replace \(y = \text{mean}(.)\) with \(y = . \%\%\% \text{mean}\) in the code above. But I could replace \(y = \text{mean}(.)\) with \(y = \text{mean}(.) \%\%\% \text{as.character}\), because the \(\text{mean}(.)\) term can still be connected to a pipe.

I will conclude this section with an ambitious example.

**Exercise 4.11.** 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 away the captured count mean and then divide by the captured count standard deviation (by `opening_name` groupings). But there’s an extra catch: we will want to exclude outliers when calculating these means and standard deviations. So we will have to calculate these statistics “offline” and join their values back into our original tibble.

- The moves that capture pieces are flagged with an “\(x\)” . For example, `game == "JcGqobhw"` has `moves == "e4 e5 Ne3 Nc6 Bb5 a6 Bxc6 dxc6 Nxe5 Bd6",` three of which capture pieces: `Bxc6, dxc6,` and `Nxe5`. Accordingly, the number of times the character "\(x\)" appears in the `moves` text string equals the number of pieces that were captured in a game. With this, use `str_count()` to construct a variable called `captured` that records the total number of pieces captured in a game.
4 Joins

- Check: 15.02 pieces are captured in the average game.
- Ultimately, we will want to define z-scores $\text{captured}_z = (\text{captured} - \text{captured_mean}) / \text{captured_sd}$, where $\text{captured_mean}$ and $\text{captured_sd}$ are the mean and standard deviation of captured across opening_name groupings. However, we don’t want our mean and standard deviation calculations to be influenced by outliers. To flag these outliers, $\text{group_by() opening_name}$ and create a logical random variable called $\text{outlier}$ that indicates that the current captured value is in the top three or bottom three values.
  - Check: For opening_name $= "$\text{Alekhine Defense}$", the $\text{outlier} == \text{FALSE}$ observations should have captured values between 7 and 23.
  - Tip: Use $\text{row_number(captured)}$ and $\text{row_number(-captured)}$ to define outlier.
- Now we only want to use the $\text{outlier} == \text{FALSE}$ observations to calculate $\text{captured_mean}$ and $\text{captured_sd}$. But we don’t want to remove the $\text{outlier} == \text{TRUE}$ observations from our sample. Hence, we’ll have to join in these statistics. To do so, pipe your tibble into an $\text{inner_join()}$ operation.
- Within the $\text{inner_join()}$, (i) $\text{filter()}$ out the $\text{outlier} == \text{TRUE}$ observations, (ii) $\text{group_by()}$ opening_name, and (iii) define $\text{captured_mean}$ and $\text{captured_sd}$ as the mean and standard deviation of the remaining captured values.
  - Tip: Use the . convention to refer to the tibble that got piped in.
  - Check: Your tibble should have 14,322 rows and 16 columns before the $\text{inner_join()}$ and 12,192 rows and 18 columns after.
- After merging $\text{captured_mean}$ and $\text{captured_sd}$ into your master panel, define $\text{captured}_z = (\text{captured} - \text{captured_mean}) / \text{captured_sd}$.
- Remove observations with $\text{abs(captured}_z) >= 6$. (These z-scores are too extreme to take seriously.)
- Scatter plot $\text{captured}_z$ by turns.
  - Tip: Set size = .2 to make the dots smaller.

In addition to widening the tibble from 16 to 18 columns, the $\text{inner_join()}$ above shortens it from 14,322 to 12,192 rows. Hence, the join acts both like a $\text{mutate()}$ step—adding variables—and a $\text{filter()}$ step—omitting observations. Specifically, it removes the opening_name values with fewer than seven games, and thus no $\text{outlier} == \text{FALSE}$ games. We’ll flesh out the filtering potential of joins in the following section. The techniques will be nearly the same, except we’ll use them to remove rows rather than add columns.

4.2.4 Filter with External Data

In Section 4.2.3 we created new variables with data stored in other rows. In this section we will create new filters with data stored in other rows.

By itself, the $\text{filter()}$ function enables you to filter your sample based on
information stored in the current row. For example, the following removes games without a queen-side castle (a move that’s denoted by "0-0-0"):

```r
c # chess_panel %>%
filter(str_detect(moves, "0-0-0"))
```

Adding a `group_by()` step enables you to base filters on group-level statistics (as we saw in Section 2.2.7). For example, the following removes games whose openings never lead to a queen-side castle:

```r
c # chess_panel %>%
group_by(opening_name) %>%
filter(any(str_detect(moves, "0-0-0")))
```

In the code above, the `str_detect()` operation returns a vector of logicals that indicates which games corresponding to a given `opening_name` had a queen-side castle, the `any()` operation compresses this vector to a single logical that indicates whether any of these games had a queen-side castle, and the `filter()` operation uses this single logical to determine whether to keep this specific `opening_name`. For example, there are four games with `opening_name` == "Philidor Defense: Lopez Countergambit", only the second of which has a queen-side castle. So for this group the `str_detect()` step returns `c(FALSE, TRUE, FALSE, FALSE) and the `filter()` step accordingly receives the value of

```r
any(c(FALSE, TRUE, FALSE, FALSE)) = TRUE, and thus keeps this group of games.
```

**Exercise 4.12.** Use `n_distinct()` to disregard the games with openings that are played by fewer than a dozen white players. That is, remove the `opening_name` values that correspond to fewer than 12 `white_id` values.

The examples above use `summary` functions `any()` and `n_distinct()` to assign a TRUE or FALSE value to each group of games and then use `filter()` to pick out the TRUE groups. But we can also use a `window` function to assign a TRUE or FALSE value to each game within each group and then use `filter()` to pick out the TRUE games of each group. For example, the following uses window function `cumany()` to remove a game if the corresponding `opening_name` hasn’t yet had a queen-side castle.

```r
c # chess_panel %>%
group_by(opening_name) %>%
filter(cumany(str_detect(moves, "0-0-0")))
```

For example, for `opening_name` == "Philidor Defense: Lopez Countergambit"
the `str_detect()` step returns `c(FALSE, TRUE, FALSE, FALSE)` and thus the `filter()` function receives `cumany(c(FALSE, TRUE, FALSE, FALSE)) = c(FALSE, TRUE, TRUE, TRUE)`, and thus keeps the last three games of this group.

**Exercise 4.13.** Remove the games that comprise fewer than the median number of turns relative to the other games with the same `opening_name`.

Now suppose the information we need for our data filter does not neatly reside in a `group_by()` group that our current observation belongs to. In this case, we’ll need to implement the filter with a join operation, similar to how we used joins in Section 4.2.3 to define new variables. For example, the following selects games whose opening is played in at least 100 rated games:

```r
common_rated_openings <-
  chess_panel %>%
  filter(rated) %>%
  count(opening_name) %>%
  filter(n >= 100) %>%
  select(opening_name)

chess_panel %>%
  inner_join(common_rated_openings)
```

The code above first uses `count()` and `filter()` to identify the openings with at least 100 rated games. It then saves this information in an auxiliary tibble called `common_rated_openings`, which has only one variable, `opening_name`. Finally, it takes the `inner_join()` of `chess_panel` and `common_rated_openings`, matching the tibbles by their one common variable, `opening_name`. 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_name` does not appear in `common_rated_openings`. In other words, `common_rated_openings` contains a list of permissible `opening_name` values and the `inner_join()` step removes the games whose opening 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 `common_rated_openings`, `opening_name`, already exists in `chess_panel`.

The following exercise provides another example.

**Exercise 4.14.** Suppose you want to study the openings of the player “wall-of-shields” to prepare for an upcoming face-off with them. To derive some study materials, you wish to select all the games that share an `opening_name` with one of this player’s games.

- `filter()` out the games from `chess_panel` that do not have "wall-of-shields" as either its white or black player.
4.2 Lecture

- Save the distinct(opening_name) values in the resulting rows to a
tibble called WoS_openings. This tibble should have only one variable,
opening_name.
- `inner_join()` WoS_openings with chess_panel.

When filtering via `inner_join()` our source tibble (e.g., WoS_openings) should
be minimal both row-wise and column-wise. Row-wise minimality means it
doesn’t have repeat rows. This is crucial because the `inner_join()` matches
every row in the target table with every possible row in the source table. So
repeats in the source table will lead to repeats in your final answer. This
is why we used `distinct(opening_name)` rather than `select(opening_name)` in
our solution to Exercise 4.14: "wall-of-shields" played opening_name == "Rat
Defense: Balogh Defense" three times, so our final tibble would have three
copies of each game with this opening if we used `select(opening_name)`. (Try
this out for yourself.)

Column-wise minimality means that our source tibble has only the variables
necessary to perform the filter. Any “excess baggage” columns in our source
table will be copied over to our final answer, so we want to remove its stray
variables. Usually, this means keeping only the variables we’re matching on.
For example, we gave WoS_openings only one column because the only informa-
tion we needed to perform the filter were the relevant opening_name values.

However, sometimes we want our join operation to both filter the sample (in
the fashion of Exercise 4.14) and to add new variables (e.g., see Exercise 4.11).
In this case our source tibble will not be column-wise minimal because it will
incorporate both the variables necessary to perform the filter and the extra
variables we wish to add to our table. The following provides an example.

**Exercise 4.15.** Now suppose that in addition to selecting all the games
that share an opening_name with a "wall-of-shields" game we also wanted
to add two new columns to our tibble: `num_played_by_WoS`, which
specifies the number of "wall-of-shields" games with this opening_name,
and `fraction_won_by_WoS`, which specifies the fraction of time that
"wall-of-shields" won a game with this opening_name.

- `filter()` out the games from chess_panel that do not have
"wall-of-shields" as either its white or black player.
- Define a logical variable called `WoS_victory` that indicates whether
"wall-of-shields" won the game.
- `group_by()` opening_name, calculate statistics `num_played_by_WoS` and
`fraction_won_by_WoS`, and `ungroup()` the tibble.
- Save the resulting to `WoS_data`.
  - Check: `WoS_data` should now have three rows and three columns.
- `inner_join()` `WoS_data` with `chess_panel`. 
As you see, `inner_join()` is a powerful function. It enables us to merge in columns, filter out rows, or simultaneously merge in columns and filter out rows. However, the combined merge-and-filter joins are relatively rare. Most of your joins will be either to merge in columns or filter out rows, not both. And if you only want to add new columns then you should use `left_join()`. This function incorporates the merging functionality but not the filtering functionality: whereas `inner_join()` removes rows from the target table (e.g., `chess_panel`) that do not match any observations in the source table (e.g., `WoS_openings`), a `left_join()` always keeps the target table’s rows intact. So for adding variables `left_join()` is both safer—you’ll never accidental drop rows—and clearer—you’ll signal to reader that the filter is additive rather than subtractive.

In contrast, if you only want to remove rows then you should use `semi_join()`. This function incorporates the filtering functionality, but not the merging functionality: whereas `inner_join()` adds columns to the target table (e.g., `chess_panel`) that exist only in the source table (e.g., `WoS_openings`), a `semi_join()` always keeps the target table’s columns intact. So for removing rows `semi_join()` is both safer—you’ll never accidentally add columns—and clearer—you’ll signal to the reader that the filter is subtractive rather than additive.

In other words, `semi_join()` is like `inner_join()`, except it forces the source tibble to be row-wise and column-wise minimal. For example, `semi_join(chess_panel, WoS_openings)` and `semi_join(chess_panel, WoS_data)` would return the same thing, because the `semi_join()` function would strip away the extra columns in `WoS_data`, since they are not in `chess_panel`. And if we used `semi_join()` in Exercise 4.14 then it wouldn’t matter if we used `select(opening_name)` or `distinct(opening_name)`, because the `semi_join()` function would be careful to remove any redundant rows in `WoS_openings`. In this fashion, the minimalization safeguards built into `semi_join()` makes it a more dependable filtering join.

To recap: Use `semi_join()` if you want to remove rows without adding columns, use `left_join()` if you want to add columns without removing rows, and use `inner_join()` if you want to remove rows and add columns. For example, define:

```r
mini_1 <-
tibble(
  x = c("a", "a", "b", "b", "c"),
  y = 1:5
)
mini_2 <-
tibble(
  x = c("a", "c"),
```
\begin{verbatim}
4.2 Lecture

z = c("Hello", "World")

We could remove the x == "b" rows from mini_1 with a semi_join():

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

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

We could add the z column to mini_1 with a left_join():

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

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

And we could remove the x == "b" rows and add the z column with an inner_join():

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

\begin{verbatim}
> # A tibble: 3 x 3
> x   y   z
> <chr> <chr> <chr>
> 1 a    1     Hello
> 2 a    2     Hello
> 3 c    5     World
\end{verbatim}
\end{verbatim}
Now there’s one more important join I want to teach you: `anti_join()`.³ This join is like the evil twin of `semi_join()`—indeed, using it always gives me a little devilish delight. Whereas `semi_join()` returns the rows in the target table that correspond to rows in the source table, `anti_join()` returns the rows in the target table that do not correspond to rows in the source table. In other words, the source table acts like an inclusion list under `semi_join()` and an exclusion list under `anti_join()`. For example, we have

```
mini_1 %>%
  anti_join(mini_2)
```

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

since the `x == "b"` rows are the only ones that have no analog in `mini_2`.

The following provides an `anti_join()` example.

**Exercise 4.16.** We want to identify the most advanced openings. To do so, we will select the openings that (i) were played by at least 10 white players whose score exceeds 1900 and (ii) were played by no white players whose score does not exceed 1200.

First, create a tibble called `advanced_openings`:

- `filter()` out the games from `chess_panel` whose `white_rating` does not exceed 1900.
- `group_by()` `opening_name`.
- Use `n_distinct()` to `filter()` out the groups with fewer than 10 distinct `white_id` values.
- Select the `distinct()` `opening_name` values.
- `ungroup()` the data and save it to a tibble called `advanced_openings`.

Second, create a tibble called `novice_openings`:

- `filter()` out the games from `chess_panel` whose `white_rating` exceeds 1200.
- Select the `distinct()` `opening_name` values.
- Save these openings to a tibble called `novice_openings`.

Finally, use `anti_join()` to select the `opening_name` values in `advanced_openings` but not `novice_openings`.

---

³There are also `right_join()`, `full_join()`, and `nest_join()`, but they aren’t particularly useful.
All our join solutions, thus far, have a drawback: they construct auxiliary tables that junk up our R session (e.g., `advanced_openings` and `novice_openings` in Exercise 4.16). We faced this issue in Section 4.2.3 and identified an elegant solution: nest the query. This solution applies here as well. Indeed, rather than include a source tibble inside the parentheses of our join function, we can include the code that defines this tibble. For example, we could solve Exercise 4.16 without defining any tibbles with

```r
chess_panel %>%
  filter(white_rating > 1900) %>%
  group_by(opening_name) %>%
  filter(n_distinct(white_id) >= 10) %>%
  distinct(opening_name) %>%
  anti_join(%
    chess_panel %>%
      filter(white_rating <= 1200) %>%
      distinct(opening_name)
  )
```

Or, alternatively, we could write

```r
chess_panel %>%
  anti_join(%
    filter(., white_rating > 1900) %>%
      group_by(opening_name) %>%
      filter(n_distinct(white_id) >= 10) %>%
      distinct(opening_name) %>%
      ungroup,
    filter(., white_rating <= 1200) %>%
      distinct(opening_name)
  )
```

I prefer the former solution, since it places less code inside the parentheses (although the latter solution makes good use of the . convention).

**Exercise 4.17.** Rewrite your solution to Exercise 4.14 so that it does not define helper table `WoS_openings`. 
4.3 Commentary

&&&&&& Discuss add_count() function &&&&&&

4.3.1 Standardize Rows

We created the first_200_moves tibble in Section 4.2.1 to enable us to analyze individual moves. But this was a bad solution. First, there’s no simple way to save move-specific information: e.g., we can’t easily flag the white moves. Second, over two-thirds of the tibble’s values are NA:

```r
first_200_moves %>%
  unlist %>%
  is.na %>%
  mean
```

> [1] 0.6792702

And third, the tibble discards information, storing only the first 200 moves of each game. To save all the information, we’d need as many columns as there are moves in the longest game. But in this case a single long game could dramatically inflate the dimensions—and thus the memory requirements—of our tibble.

A better solution is to express the tibble in the “long” format, where each move of each game has its own row. Such a long tibble could efficiently store all the information without any NA terms. It would look something like this:

```r
first_200_moves %>%
  pivot_longer(-game)
```

> # A tibble: 2,864,400 x 3
> game  name    value     <chr>  <chr>  <chr>
> 1 \nNXvwaE move_1 d4
> 2 \nNXvwaE move_2 Nc6
> 3 \nNXvwaE move_3 e4
> 4 \nNXvwaE move_4 e5
> 5 \nNXvwaE move_5 f4
> 6 \nNXvwaE move_6 f6
> 7 \nNXvwaE move_7 dxe5
> 8 \nNXvwaE move_8 fxe5
But thisibble still only incorporates the first 200 moves of each game. And
deriving the full longibble with pivot_longer() would be awkward, as it’d
require us to first express the data in the ridiculously wide format in which
every move of the longest game has its own column. To avoid constructing
this memory-strainingibble, we will pursue a different approach. Rather than
store a game’s moves across hundreds of columns, we will store it in a list that
is housed in a single column of ouribble. And we will then unnest() these
lists to express their elements rowwise.

**Exercise 4.18.**

- Start with chess_panel and drop all variables besides game and moves.
- Use str_split() to transform the moves variable from a collection of char-
  acter strings to a collection of lists of character vectors.
  - Tip: Use the pattern = '' option, since the various moves are separated
    by spaces.
  - Check: game == 'uUTHWT8w' should have moves == 'e4 b6 d4 Bb7 Bd3
e6 Nf3 c5' before this step and moves == list(c('e4', 'b6', 'd4',
'Bb7', 'Bd3', 'e6', 'Nf3', 'c5')) after this step.
  - Note: R treats character vector c('e4', 'b6', 'd4', 'Bb7', 'Bd3',
e6', 'Nf3', 'c5') as eight distinct objects: length(c('e4', 'b6',
d4', 'Bb7', 'Bd3', 'e6', 'Nf3', 'c5')) == 8. But it treats
list(c('e4', 'b6', 'd4', 'Bb7', 'Bd3', 'e6', 'Nf3', 'c5')) as a single object:
length(list(c('e4', 'b6', 'd4', 'Bb7', 'Bd3',
e6', 'Nf3', 'c5'))) == 1. Thus, we can set a given row’s moves value
to list(c('e4', 'b6', 'd4', 'Bb7', 'Bd3', 'e6', 'Nf3', 'c5'))
but not to c('e4', 'b6', 'd4', 'Bb7', 'Bd3', 'e6', 'Nf3', 'c5'),
because eachibble cell can store only one object.
- Use unnest() to unpack the lists in moves.
  - Check: Youribble should have 909,432 rows and two columns after
this step.
- group_by() game and use row_number() to define a variable called move_number
that specifies the move number. For example, the first move always has
move_number == 1 and so forth.
- Define a variable called player that is white if move_number is odd and black
if even.
  - Tip: x %% y yields the remainder you get when dividing x by y. Hence,
x %% 2 is zero if x is even and one if x is odd.
- ungroup() the data and call the resultingibble moves_tibble.
  - Note: Technically you don’t need this ungroup() step, but you should
always leave your named tables in their natural, ungrouped state. Other-
wise, you’re liable to forget their groupings, which could lead to bugs.
Look how much nicer `moves_tibble` is than `first_200_moves`: it flags the player, it contains no NA terms, and it incorporates all the moves.

There are a few lessons here. First, you want to structure your data so that each observation has the same number of measurements. For example, every observation of `moves_tibble` reports the same four variables: the game, the move_number, the moves, and the player. In contrast, the observations in `first_200_moves` report a widely varying number of variables: e.g., the `game == '01Jfm49Q'` row has measurements `moves_1-moves_11` whereas the `game == '0k2Hjm8Q'` row has measurements `moves_1-moves_196`. This “ragged right edge” leads to a proliferation of NA terms. Thus, when the number of measurements varies from row to row, you should reexpress the data in long mode.

Second, define the “observation” as the thing you’re trying to describe. For example, we can’t easily classify moves as black or white in `first_200_moves` because it’s observations correspond to games, not moves. But we can easily add such move-level attributes to `moves_tibble`, because it casts the move as the essence of an observation.

Finally, it’s usually best to split your data across multiple tibbles. Just as you generally wouldn’t stuff all your data in a single Excel worksheet, you generally shouldn’t stuff all your data in a single R tibble. For example, in our chess setting, there are two fundamental things that we would like to append attributes to: games and moves. Hence, we should maintain two distinct tibbles, one for the game-level attributes and another for the move-level attributes. Cramming all the features in a single tibble would lead either to a lot of redundancy—e.g., repeating `opening_name == 'Italian Game: Two Knights Defense | Knight Attack'` for each move—or a lot of NA terms—e.g., specifying the player with a set of mostly empty columns `player_move_1-player_move_200`.

### 4.3.2 Lengthening Can Simplify

Unfortunately, I would consider our solution to Exercise 4.8 an abuse of the `across()` functionality. It’s the best solution that works with the “wide” data, but for this problem it’s easier to manipulate the tibble into its “long” format upfront, which obviates the `across()` steps. (See Section ?? for the “long mode” of a tibble.) Indeed, there are generally two ways to perform any multivariable operation: the “wide” approach, which uses `across()` to apply the operation columnwise, or the “long” approach, which uses `pivot_longer()` to stack the relevant variables into a single column that receives the operation. For example, define
```r
mini_tib <-
  first_200_moves %>%
  slice(1:3) %>%
  select(1:4)
mini_tib

> # A tibble: 3 x 4
>   game  move_1 move_2 move_3
>  <chr>  <chr>  <chr> <chr>
> 1 l1NXvwaE d4   Nc6   e4
> 2 mIICvQHh e4   e5    d3
> 3 kWKvqYL d4   d5    Nf3
```

Now suppose we wanted to express the moves in upper case. The “wide” approach uses `across()` to apply `str_to_upper()` column-by-column:

```r
mini_tib %>%
  mutate(  
    across(  
      -game,  
      str_to_upper  
    )
  )

> # A tibble: 3 x 4
>   game  move_1 move_2 move_3
>  <chr>  <chr>  <chr> <chr>
> 1 L1NXVWA d4   NC6   E4
> 2 MIICVQHH E4   E5    D3
> 3 KWKVQYL D4   D5    NF3
```

The long approach, in contrast, uses `pivot_longer()` to stack all the moves in a single column, called `value`, and applies `str_to_upper()` to that:

```r
mini_tib %>%
  pivot_longer(-game) %>%
  mutate(value = str_to_upper(value))

> # A tibble: 9 x 3
>   game  name  value
>  <chr> <chr> <chr>
> 1 l1NXvwaE move_1 D4
> 2 l1NXvwaE move_2 NC6
```
The following exercise illustrates that the long approach yields a shorter solution to Exercise 4.8.

**Exercise 4.19.** We will now perform Exercise 4.8’s calculation with the long approach.

- Pipe `first_200_moves` into `pivot_longer()`. Use the `cols` argument to specify all variables besides `game`, so that your output tibble has three columns: `game`, `name`, and `value`.
- `group_by()` `name`.
- Use `summarise()` to calculate the fraction of the `value` text strings that correspond to pawn moves. Call this variable `frac`.
  - Tip: Use the code you wrote for Exercise 4.8.
- Now your tibble should match the tibble outputted by the `pivot_longer()` step of your previous solution (albeit with rearranged rows). Copy the rest of your previous solution—the `separate()` and `ggplot()` calls—to finish the job.

For this problem, the long approach is simpler than the wide approach because the `ggplot()` forces you to `pivot_longer()` the tibble anyway and moving this lengthening step upfront gives you more opportunity to exploit the simpler “long” format. But more generally, here’s a good rule of thumb for determining which method to adopt: if your transformation applies to nearly all columns use the long approach, otherwise use the wide approach. For example, define

```r
code_snippet <-
mini_tib <-
  chess_panel %>%
  slice(1:4) %>%
  select(game, white_id, black_id, moves) %>%
  separate(
    moves,
    paste0("move_", 1:3),
    sep = " ",
  )
```

mini_tib
4.3 Commentary

Now suppose you wanted to express moves in capital letters. The wide approach would leave the table in its nice wide format:

```r
mini_tib %>%
mutate(
    across(
        starts_with("move"),
        str_to_upper
    )
)
```

But the long approach would add a lot of redundancy to the table. For example, it would repeat the white_id and black_id for each move:

```r
mini_tib %>%
pivot_longer(starts_with("move")) %>%
mutate(value = str_to_upper(value))
```

---

> # A tibble: 4 x 6
> game white_id black_id move_1 move_2 move_3
> <chr>  <chr>  <chr>  <chr>  <chr>  <chr>
> 1 11NXwaE a-00  skinnerua  d4  Nc6  e4
> 2 mIIcQHh ischia  a-00  e4  e5  d3
> 3 kWKvqYL daniamurashov adivanov2009 d4  d5  Nf3
> 4 9txo1AUZ nik221107 adivanov2009 e4  e5  Nf3
```
Now in this mini-problem it’s not so bad, but in theory there could be hundreds to columns repeating over scores of moves, which would be terribly inefficient. The problem is that pivot_long() pulls down only the variables that receive the treatment; all other variables are copied as they stand. Accordingly, we don’t want to use pivot_long() to modify a relatively small fraction of variables.

### 4.3.3 Reiterate Groupings

My solution to Exercise 4.11 includes a redundant group_by(opening_name) step inside the inner_join(). This group_by() is superfluous because the prior group_by() still applies: the tibble represented by , inherits the opening_name groupings defined above. But I include this extra grouping step to make the nested query self-sufficient, which makes the code more resilient (i.e., more robust to future modifications).

For example, suppose later on I 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 squeezing group_by(increment_code) %>% filter(n() > 250) between the mutate() and inner_join() operations. But without the backup group_by(opening_name) step I would instead need to squeeze group_by(increment_code) %>% filter(n() > 250) %>% group_by(opening_name) between these operations. And it’d be easy to overlook this last group_by(opening_name) step because it wouldn’t be obvious that the code inside the inner_join() relies on a group_by() step outside the inner_join(). Hence, the dependence of the internal inner_join() code on an external group_by() represents a liability in the code base. Thus, it’s safer to give the nested query its own group_by(opening_name) step, to make it independent of the code outside of the inner_join().

This principle applies more broadly: you want to make the separate sections of your code as independent as possible, so that mucking around with one chunk of code doesn’t break another chunk. Hence, you should reestablish your group_by() groupings liberally, to help decouple the different sections of your code. Moreover, reasserting the grouping structure makes your code easier to read: e.g., the nested group_by(opening_name) makes it clear that captured_mean and captured_sd vary with opening_name.

Finally, there’s an even larger lesson here: code is a “living document,” revised over many versions. So you’ll almost certainly have to modify any program you write in the future after you’ve forgotten its key details. Hence, it’s not
enough for your program to be correct—it should also be idiot-proof, because a less informed version of yourself will be tinkering with it in the future (unless you have a perfect memory for code). When programming, you should keep this future version of yourself in mind: you should make your code resilient so that it withstands the clumsy alterations you’ll make, and you should make your code flexible so that these alterations need not be too invasive.

4.4 Lab

4.4.1 Overview

The COVID-19 pandemic incited panic purchasing that nearly crippled our supply chains. Stores faced runs on inventory as customers scrambled to stock up their home pantries and freezers with basic essentials. Customers took photos of bare Walmart and Target shelves—a spectacle they had never before seen—and posted them to social media, reinforcing the fear that initiated the hoarding. An evil cycle began: we stockpiled because others stockpiled. And once it got going, the cycle sustained itself; we kept hoarding even after learning that the virus wouldn’t dramatically undermine our production capacity, because we feared that even if our supply chains could withstand the virus, they couldn’t withstand our hoarding. Hence, hoarding begat hoarding. Leonard (2020) describes these precarious few months:

Suddenly every consumer wants copious amounts of toilet paper, hand sanitizer and shelf-stable food. At the same time, the healthcare system needs ventilators and masks.

The spike in demand is evident in empty grocery store shelves and state politicians’ pleas for medical supplies.

When the retail or end-user node of the supply chain sees even a slight variation in demand, it quickly ripples and grows larger as it reaches suppliers and manufacturers. In the world of supply chain, this is known as the bullwhip effect—and we’re seeing it play out in real-time a 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, ...
The 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. (2019) 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. (2019) 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 not contrived; it is precisely the data cleaning methodology that Bray et al. (2019) employed.

### 4.4.2 Sample

We will use `order_dates`, `category`, `order_store`, `ship_store`, `inv_store`, and `inv_dc` from Section 1.3, Exercise 1.9. 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. (2019) 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.
  - Note: An `order_code` generally corresponds to multiple `item` values, since stores usually include multiple products in an order.
  - Note: 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`).
  - Note: `order` measures inventories in physical units. We will always mea-
sure 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.
  - Note: `inv` denotes the DC’s inventories whereas `inv` denotes the stores’ inventories.

### 4.4.3 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
> <int> <int> <date> <int>
> 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
> [1] 16490877
```
4.4 Lab

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

> [1] 16498877
```

Since there are as many distinct \( \text{store, item, date} \) 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 \( \text{item, date} \) pairs than observations:

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

> [1] 342782
```

Hence, some \( \text{item, date} \) pairs must repeat. And the same is true for the \( \text{store, date} \) and \( \text{store, item} \) pairs. Thus, \( \text{store, item, date} \) 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 “meta data” necessary to keep track of these recording. The key is usually the collection of the identification variables. For example, identification variables \text{store, item, and date} identify what measurement variable \text{inv} actually measures.

The \textit{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 \texttt{inv.store}. The following exercise provides another example.

**Exercise 4.20.**

- Determine the identification variables of \texttt{inv_df}, and use this information to guess what this table’s key must be.
- Use \texttt{nrow()} and \texttt{distinct()} to confirm that your proposed key uniquely identifies the observations.
- Use \texttt{nrow()} and \texttt{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 4.21.
- Load the `item`, `date`, and `inv` columns of `inv_store` pertaining to `store == 7511` into a tibble called `inv_7511`.
- 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`.
  - Check: `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.
  - Check: 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`.
  - Note: By construction, `order_7511` has key `(item, date)`.
  - Check: `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.
  - Check: 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 amply 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—its 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
> mean(n > 1)
> 1 0.007780183
```

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 4.22.**

- 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.
  - Bonus: Use `across()` to avoid writing `sum()` more than once.
  - Tip: Use the `.groups = “drop”` option of `summarise()` to ungroup the data after you `sum()` them.
  - Check: The observation with `store == 1015, item == 11488, and date == ymd("2014-02-19")` should have `order` and `ship` values of 12.
  - Note: `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`.
  - Check: `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 4.23.** 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`.
  - Tip: 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`.
  - Bonus: 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 4.23 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. (2019) 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.
4.4 Lab

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. (2019) 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. (2019), 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 4.24.

- Use arrange() to order inv_store by store, item, and date.
  - Check: 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 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.
  – Check: The first observation should have `product == "1083_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 4.25.**

• 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.
  – Check: 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.
  – Check: 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 4.26.**

• 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`.
  – Check: `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.
  – Check: `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 4.27 will motivate this filtering step and Exercise 4.28 will execute it.

**Exercise 4.27.**

• 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.
  – Check: 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 ???. 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 4.28.**

• Use group\_by() and summarise() to calculate for each product in master\_panel the fraction of observations in which order > 0.
  – Tip: Use the na.rm = TRUE option of mean().
• inner\_join() these distinct product values with master\_panel to filter your master panel accordingly.
  – Bonus: Nest the query of the preceding two steps inside the inner\_join() parentheses so that the filter can be performed without breaking the %>% pipe.
  – Check: master\_panel should have 8,663 distinct product values after this filter.
4.4.4 Analyze

We will now use master_panel to illustrate the inventory runs endemic to our supermarket supply chains. Bray et al. (2019) 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. (2019) 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. (2019) explained how these two graphs indicate ration gaming:

Ration gaming has two aspects: rationing—the DC curtailing shipments when its supply runs short—and gaming—the stores selfishly manipulating the inventory allocation scheme. Figure 2 depicts both phenomena.

First, the DC rations inventory. It fulfills 95% of orders when its inventory level is above the first decile, but only 36% when its inventory level is below the first decile (overall, 13% of orders go unfulfilled). And a stint of rationing can last awhile: if we define a “rationing spell” as a span of time when the estimated order fulfillment probability is less than one half, then 50% of rationing spells last at least 5 days, 10% last at least 14 days, and 1% last at least 35 days. Sample-wide, these rationing spells comprise 10% of our sample and account for 74% of unfulfilled orders. Moreover, these rationing spells are predictable because the DC inventory declines at a steady pace. ...

Second, the stores game the inventory rationing scheme. They
4.4 Lab

![Graphs showing Rationing and Gaming](image)

**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.

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 4.25, whereas Bray et al. (2019) limited theirs to products with at least 500 observations.

**Exercise 4.29.**

- 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`.
  - Check: `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.
  - Tip: Use the `na.rm = TRUE` option of `sum()`.
  - Check: 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 tibble 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 `colour = category`, and `facet_wrap()` by `name`.
- Label the plot as Bray et al. (2019) 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.
4.5 Solutions

4.1

```r
chess_games %>%
ggplot() +
aes(x = turns) +
geom_histogram(binwidth = 1) +
facet_wrap(  
  vars(victory_status),
  scales = "free_y",
  ncol = 1
) +
theme_bw()
```

4.2

```r
chess_panel <-
chess_games %>%
filter(  
  turns >= 8,
  abs(white_rating - black_rating) <= 250
)
```
4.3

chess_panel %>%
  separate(
    increment_code,
    c("initial_time", "extra_time"),
    convert = TRUE
  ) %>%
  mutate(time_out = victory_status == "outoftime") %>%
  select(game, initial_time, extra_time, time_out) %>%
  pivot_longer(c(initial_time, extra_time)) %>%
  mutate(value = pmin(value, 50)) %>%
  ggplot() +
  aes(
    x = name,
    y = value,
    colour = time_out
  ) +
  geom_boxplot() +
  theme_bw()

4.4

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

> # A tibble: 14,322 x 4
> turns white_rating black_rating opening_ply
> <dbl> <dbl> <dbl> <dbl>
> 1 16 1322 1261 4
> 2 61 1496 1500 3
> 3 61 1439 1454 3
> 4 95 1523 1469 5
> 5 33 1520 1423 10
> 6 66 1439 1392 6
> 7 119 1381 1209 4
> 8 39 1381 1272 1
> 9 31 1381 1607 2
> 10 31 1094 1141 8
> # ... with 14,312 more rows

4.5

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

> # A tibble: 14,322 x 21
> move_20 move_22 move_24 move_26 move_28 move_30 move_32 move_34 move_36
> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
> 2 a3 Rx a3 c4 d5 Qxd5 Be6 Ke7 Kf6 Kg6
> 3 Nb4 Kb8 Bxc2 Kc8 Qc6 Kd7 bxc6 Kxd8 e5
> 4 Bxf6 Qd7 Nxd3 c6 cxd5 Qe6 Qg4 Bd7 Rc8
> 5 Bxf3 Qd4+ Qg4 Rd8 Nd7 e6 Ke7 <NA> <NA>
> 6 c6 b5 Qa5 Bb7 c5 d6 bxc4 Rac8 Bxd5
> 7 Bxg2 Qxf7 Kxf7 Bf3 Bh5 g6 Nf6 e6 Rc8
> 8 Bc6 0–0 Nd5 Qxd5 Qd7 Qb7 Re8 Rxe7 Re8
> 9 Bg7 Nh5 Nf4 Nxd3+ Bxd4 Nc5 <NA> <NA> <NA>
> 10 Re8 d6 Re7 Be6 Rxe6 Kh8 <NA> <NA> <NA>
> # ... with 14,312 more rows, and 12 more variables: move_38 <chr>,
> # move_40 <chr>, move_42 <chr>, move_44 <chr>, move_46 <chr>, move_48 <chr>,
> # move_50 <chr>, move_52 <chr>, move_54 <chr>, move_56 <chr>, move_58 <chr>,
> # move_60 <chr>

4.6

```r
first_200_moves %>%
  select(
```

---

The extracted text includes R code snippets and data frames, indicating an analysis of various moves in chess. The code is used to filter and select specific columns from a data frame named `first_200_moves`. The code is executed in a typical R environment, possibly in a Sweave document or R Markdown, as indicated by the use of `%>%` operator for chaining operations in the code. The data appears to be related to chess moves and ratings, possibly for an analysis or visualization of game data.
where(~{
  .x %>%
    str_detect("") %>%
    sum(na.rm=TRUE) %>% {
      . >= 10
    }
})

4.7

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"
      )
  }
)

4.8
4.5 Solutions

```r
first_200_moves %>%
  summarise(
    across(
      -game,
      ~{
        .x %>%
          str_sub(end = 1) %>%
          str_detect(
            "^[RNBQK0]",
            negate=TRUE
          ) %>%
          mean(na.rm=TRUE)
      }) %>%
    pivot_longer(
      everything(),
      values_to = "frac"
    ) %>%
    separate(
      name,
      c(NA, "move_num"),
      sep = " ",
      convert = TRUE
    ) %>%
    ggplot() +
    aes(
      x = move_num,
      y = frac
    ) +
    geom_point() +
    labs(
      x = "Move Number",
      y = "Fraction of Moves Made by Pawns"
    ) +
    theme_bw()
```
4.9

chess_panel %>%
  group_by(white_id) %>%
  mutate(times_white_played_white = n())

4.10

chess_panel %>%
  filter(winner != "draw") %>%
  mutate(turn_decile = ntile(turns, 10)) %>%
  group_by(turn_decile) %>%
  summarise(white_win_prob = mean(winner == "white")) %>%
  ggplot() +
  aes(
    x = turn_decile,
    y = white_win_prob
  ) +
  geom_point() +
  theme_bw()
4.5 Solutions

4.11

```r
chess_panel %>%
muate(captured = str_count(moves, "x")) %>%
group_by(opening_name) %>%
muate(
   outlier =
   row_number(captured) <= 3 |
   row_number(~captured) <= 3
)
)
inner_join(
   filter(., !outlier) %>%
group_by(opening_name) %>%
s summarise(
   captured_mean = mean(captured),
   captured_sd = sd(captured)
)
)
mutate(
   captured_z =
   (captured - captured_mean) / captured_sd
)
filter(abs(captured_z) < 6) %>%
ggplot() +
aes(
   x = turns,
   y = white_win_prob)
```
4.12

```r
captured_z <- captured_z %>%
  geom_point(aes(x = turns, y = captured_z), size = 0.1) +
  theme_bw()
```

4.13

```r
chess_panel %>%
  group_by(opening_name) %>%
  filter(n_distinct(white_id) > 12)
```

4.14

```r
WoS_openings <-
  chess_panel %>%
  filter(
    white_id == "wall-of-shields" |
    black_id == "wall-of-shields"
  )
```
4.5 Solutions

) %>%
distinct(opening_name)

chess_panel %>%
inner_join(WoS_openings)

4.15

WoS_data <-
chess_panel %>%
filter(
  white_id == "wall-of-shields" |
  black_id == "wall-of-shields"
) %>%
mutate(
  WoS_victory =
    (winner == "white" & white_id == "wall-of-shields") |
    (winner == "black" & black_id == "wall-of-shields")
) %>%
group_by(opening_name) %>%
summarise(
  num_played_by_WoS = n(),
  fraction_won_by_WoS = mean(WoS_victory)
) %>%
ungroup

chess_panel %>%
inner_join(WoS_data)

4.16

advanced_openings <-
chess_panel %>%
filter(white_rating > 1900) %>%
group_by(opening_name) %>%
filter(n_distinct(white_id) >= 10) %>%
distinct(opening_name) %>%
ungroup

novice_openings <-
chess_panel %>%
filter(white_rating <= 1200) %>%
distinct(opening_name)
advanced_openings %>%
  anti_join(novice_openings)

> # A tibble: 4 x 1
> opening_name  
> <chr>          
> 1 Sicilian Defense: Old Sicilian
> 2 French Defense: Exchange Variation
> 3 Sicilian Defense: Najdorf Variation
> 4 Dutch Defense

4.17

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

> # A tibble: 133 x 14
> # game rated turns victory_status winner increment_code white_id white_rating
> <chr> <gl> <dbl> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
> 1 btrH- TRUE 85 mate white 15+15 rougeth- 1316
> 2 LCB4- TRUE 70 resign black 10+0 doom123- 1571
> 3 PRsV- TRUE 60 resign black 5+5 fischer- 1692
> 4 DQe- TRUE 49 resign white 10+5 ducksan- 1832
> 5 yP8C- TRUE 42 mate black 10+5 wall-of- 1704
> 6 NLpj- TRUE 32 resign black 10+5 ducksan- 1841
> 7 cNGs- TRUE 70 resign black 10+5 wall-of- 1695
> 8 WHRd- TRUE 52 resign black 10+5 ducksan- 1851
> 9 43et- TRUE 30 mate black 10+0 vishalv- 1577
> 10 CUpT- TRUE 86 mate black 10+0 vishalv- 1592
> # ... with 123 more rows, and 6 more variables: black_id <chr>,
> # black_rating <dbl>, moves <chr>, opening_eco <chr>, opening_name <chr>,
> # opening_ply <dbl>

4.18
4.5 Solutions

```r
moves_tibble <-
  chess_panel %>%
  select(game, moves) %>%
  mutate(moves = str_split(moves, pattern = " ")) %>%
  unnest(moves) %>%
  group_by(game) %>%
  mutate(
    move_number = row_number(),
    player = ifelse(move_number %% 2, "white", "black")
  ) %>%
  ungroup
```

4.19

```r
first_200_moves %>%
  pivot_longer(-game) %>%
  group_by(name) %>%
  summarise(
    frac =
      value %>%
      str_sub(end = 1) %>%
      str_detect(
        "[RNBQKO]",
        negate=TRUE
      ) %>%
      mean(na.rm=TRUE)
  ) %>%
  separate(
    name,
    c(NA, "move_num"),
    sep = " ",
    convert = TRUE
  ) %>%
  ggplot() +
  aes(
    x = move_num,
    y = frac
  ) +
  geom_point() +
  labs(
    x = "Move Number",
    y = "Fraction of Moves Made by Pawns"
```
\texttt{theme_bw()}
5

Generalized Linear Models

5.1 Homework

5.1.1 Wrangling Techniques

I will now teach you two wrangling tricks that will help you solve Exercises 5.5 and 5.11, later in this chapter.

First, I'll cover the technique that you'll use in Exercise 5.5. Suppose I have a tibble that characterizes some normal distributions:

```r
normal_tib <-
tibble(
  mean = 11:14,
  sd = 1:4
)
```

I could calculate each distribution's 0.05 and 0.95 quantiles with

```r
normal_tib %>%
  mutate(
    quantile_5 = qnorm(.05, mean, sd),
    quantile_95 = qnorm(.95, mean, sd)
  )
```

> # A tibble: 4 x 4
> mean sd quantile_5 quantile_95
> <int> <int>    <dbl>     <dbl>
> 1  11   1     9.36      12.6
> 2  12   2     8.71      15.3
> 3  13   3     8.07      17.9
> 4  14   4     7.42      20.6

But suppose that instead of the 0.05 and 0.95 quantiles, I wanted the $q$ quan-
tiles for all $q \in \{0.001, 0.002, \ldots, 0.999\}$. Of course I could solve this with 999 distinct `qnorm()` calls:

```r
normal_tib %>%
  mutate(
    quantile_001 = qnorm(.001, mean, sd),
    quantile_002 = qnorm(.002, mean, sd),
    # ...
    quantile_999 = qnorm(.999, mean, sd)
  )
```

But that would be cockamamie. The following is a more reasonable solution:

```r
normal_tib %>%
  crossing(tibble(q = seq(.001, .999, .001)) %>%
    mutate(quantile = qnorm(q, mean, sd)))
```

> # A tibble: 3,996 x 4
>   mean   sd  q quantile
>  <int> <int> <dbl> <dbl>
>  1  11  1 0.001  7.91
>  2  11  1 0.002  8.12
>  3  11  1 0.003  8.25
>  4  11  1 0.004  8.35
>  5  11  1 0.005  8.42
>  6  11  1 0.006  8.49
>  7  11  1 0.007  8.54
>  8  11  1 0.008  8.59
>  9  11  1 0.009  8.63
> 10  11  1 0.01  8.67
> # ... with 3,986 more rows

The `crossing()` function joins every row of `normal_tib` with every row of `tibble(q = seq(.001, .999, .001))` to create a tibble in which every q value is repeated for every (mean, sd) pair. The `qnorm()` function then maps the q, mean, and sd vectors into corresponding quantile vectors.

But this solution is a heavy-handed because it creates and joins in an entirely new tibble. And, if possible, we’d rather not construct new R objects, as doing so creates more moving parts and thus more complexity. Hence, the following solution, which applies basic `group_by()` and `summarise()` transformations to our one tibble, is preferable.
5.1 Homework

```r
normal_tib %>%
  group_by(across(everything())) %>%
  summarise(
    q = seq(.001, .999, .001),
    quantile = qnorm(q, mean, sd)
  )
```

> # A tibble: 3,996 x 4
> # Groups:   mean, sd [4]
> mean sd  q quantile
> <int> <int> <dbl>     <dbl>
>  1  11 1  0.001  7.91
>  2  11 1  0.002  8.12
>  3  11 1  0.003  8.25
>  4  11 1  0.004  8.35
>  5  11 1  0.005  8.42
>  6  11 1  0.006  8.49
>  7  11 1  0.007  8.54
>  8  11 1  0.008  8.59
>  9  11 1  0.009  8.63
> 10  11 1  0.010  8.67
> # ... with 3,986 more rows

The `group_by()` step tells R to treat each (mean, sd) pair distinctly. Accordingly, the `summarise()` step constructs `q` and `quantile` separately for each normal distribution. And since these vectors each have 999 elements this step expands each `group_by()` chunk from one row of two variables (mean and sd) to 999 rows of four variables (mean, sd, q and quantile).

The key is that `summarise()`, unlike `mutate()`, allows you to change the number of rows spanned by your `group_by()` groupings. So you should use it whenever you want to apply a function whose output differs in length from its input. Usually this means mapping vectors to individual numbers (e.g., transforming variable x to summary statistic mean(x)). But it can also mean mapping individual numbers to vectors (e.g., transforming the individual mean and sd parameters to the vector quantile). The following problem provides another example.

**Exercise 5.1.** The following defines a tibble that lists the inauguration dates of the US presidents, going back to Lyndon Johnson:

```r
inauguration_dates <-
  tribble(   
    ~ prez, ~ date,
```
"lbj", ymd("1963-11-22"),
"nixon", ymd("1969-1-20"),
"ford", ymd("1974-08-09"),
"carter", ymd("1977-01-20"),
"reagan", ymd("1981-01-20"),
"hw", ymd("1989-01-20"),
"clinton", ymd("1993-01-20"),
"w", ymd("2001-01-20"),
"obama", ymd("2009-01-20"),
"trump", ymd("2017-01-20"),
"biden", ymd("2021-01-20")
)

Use inauguration_dates and summarise() to create a tibble that lists the first 100 days of each presidency.

- Check: Your new tibble should have 1,100 rows.
- Hint: See what today() + days(0:9) gives you.
- Tip: Use the .groups = "drop" option of summarise() to return an ungrouped tibble.

Now I’ll cover the technique that you’ll use in Exercise 5.11. First, define

```r

tib_1 <-
tibble(
  id = c("x", "y", "z", "y", "z"),
  var = c("a", "a", "b", "b", "c"),
  val = 1:5
)

tib_1
```

> # A tibble: 5 x 3
> id  var val
> <chr> <chr> <int>
> 1 x     a     1
> 2 y     a     2
> 3 z     b     3
> 4 y     b     4
> 5 z     c     5

And define
tib_2 <-
  tib_1 %>%
  bind_rows(tibble(id = "x", var = "a", val = 100))

tib_2

> # A tibble: 6 x 3
>     id  var val
>  <chr> <chr> <dbl>
> 1     x   a    1
> 2  y     a    3
> 3  z     b    4
> 4  y     c    5
> 5  z     c    6
> 6  x     a  100

When we pipe tib_1 into pivot_wider() we get a typical wide-mode tibble:

```r
  tib_1 %>%
  pivot_wider(
    id_cols = id,
    names_from = var,
    values_from = val
  )
```

> # A tibble: 3 x 4
>     id  a  b  c
>  <chr> <int> <int> <int>
> 1     x   1 NA NA
> 2  y    2  NA NA
> 3  z  NA   3  5

But when we try the same for tib_2 we get a weird output:

```r
  tib_2 %>%
  pivot_wider(
    id_cols = id,
    names_from = var,
    values_from = val
  )
```

> # A tibble: 3 x 4
>     id  a  b  c
>  <chr> <list> <list> <list>
The problem is that `tib_2` has two observations with `id == 1` and `var == "a"`, and `pivot_wider()` stuffs both of their `val` values into the same cell of the wide table. But the only way to store two numbers in one tibble cell is to wrap them in a list (see Exercise 4.18). So the `pivot_wider()` operation turns all variables into lists (although this is only really necessary for the `a` column).

For example, the cell in row 1, column 2 has the value `list(c(1, 100))` (which is different from `c(1, 100)` and `list(1, 100)`), and the cell in row 3, column 3 has the value `list(5)` (which is different from 5). These lists are a hassle, so we usually tell `pivot_wider()` to apply some function to each of them with the `values_fn` = `.` option. This option allows you to pass a function into your `pivot_wider()` call, which gets applied to the lists that would otherwise be returned. The function outputs then get saved in the cells of the wide table. For example, the following returns the means of the lists created above:

```
tib_2 %>%
pivot_wider(
  id_cols = id,
  names_from = var,
  values_from = val,
  values_fn = mean
)
```

> # A tibble: 3 x 4
> id  a   b   c
> <chr> <dbl> <dbl> <dbl>
> 1 x   50.5 NA  NA
> 2 y   2   4   NA
> 3 z   NA  3   5

And the following returns the lengths of the lists created above:

```
tib_2 %>%
pivot_wider(
  id_cols = id,
  names_from = var,
  values_from = val,
  values_fn = length,
  values_fill = 0
)
```

> # A tibble: 3 x 4
5.1 Homework

> id  a  b  c
> <chr> <int> <int> <int>
> 1 x  2  0  0
> 2 y  1  1  0
> 3 z  0  1  1

Note, in this case we additionally used the values_fill = 0 option to assign zero to cells with no corresponding lists. (Without this option, we would have NA terms where the zeros currently are.)

Exercise 5.2. Define tib_3 <- tib_2 %>% select(-val) and use pivot_wider() to transform tib_3 into a tibble with names id, a, b, and c, where the latter three variables are logicals that indicate a lack of data. Hence, the a column should read c(FALSE, FALSE, TRUE), the b column should read c(TRUE, FALSE, FALSE), and the c column should read c(TRUE, TRUE, FALSE). You may not use any functions other than pivot_wider().

- Hint: Without val it’s now not clear what to set the values_from = . option to. Fortunately, there are only a few values this option could take.
- Hint: Look into is.null().

5.1.2 Limits of Linear Models

Download the data here\(^1\) and load Excel file shooting_star_1992.csv into a tibble called star_92. The tibble should look like this:

> # A tibble: 6 x 2
> sky_glow star_count
> <dbl>   <dbl>
> 1 0.00821   6
> 2 0.0904    5
> 3 0.0442    4
> 4 0.0583    6
> 5 0.0103    5
> 6 0.105     5

The data set describes the number of shooting stars observed in a set of 4-hour measurements held across Eastern Oregon in 1992. Variable sky_glow records the amount of measured light pollution and variable star_count records the number of shooting stars. We want to gauge how light pollution effects our ability to see shooting stars. We’ll start by plotting the data.

Exercise 5.3. We will plot the distribution of star_count for each decile of sky_glow to see how the latter variable influences the former.

\(^1\)https://www.dropbox.com/sh/bf13r84pp2awz3a/AADX0YjRvmWt6o6x-GTMvY2Za?dl=0
5 Generalized Linear Models

- Use ntile() to divide star_92 into 10 equally sized subsamples, based on the deciles of sky_glow. More specifically, define a variable called sky_glow_decile that equals one when sky_glow is below the 10th percentile, a value of two when sky_glow is between the 10th and 20th percentiles, etc.
  - Check: The average sky_glow is 0.005 in the sky_glow_decile == 1 subsample and 0.346 in the sky_glow_decile == 10 subsample.
- Append the string "Decile: " to the beginning of each sky_glow_decile. This will make the plot easier to read.
  - Check: The first observation should now have sky_glow_decile = "Decile: 1".
- Use geom_histogram() to plot the distribution of star_count, and facet_wrap() by sky_glow_decile.
  - Tip: Turn sky_glow_decile into a factor to ensure that the facets are ordered properly. You can use `fct_relabel(. , ~ paste("Decile: " , .))` to append the "Decile: " labels to the factors without changing their order.

There are two key take-aways from this plot. First, star_count consistently decreases with sky_glow: every increase in sky_glow_decile decreases star_count. Second, star_count is not normally distributed conditional sky_glow. For example, the sky_glow_decile == 10 observations exhibit a clear rightward skew.

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:

```r
star_92 %>%
  lm(
    star_count ~ sky_glow,
    data = .
  ) %>%
  summary
```

Call:
  `lm(formula = star_count ~ sky_glow, data = .)`

Residuals:
  Min 1Q Median 3Q Max
  -6.6080 -1.6521 -0.2154 1.4598 13.3920

Coefficients:
  Estimate Std. Error t value Pr(>|t|)
  (Intercept) 6.82481  0.04684 145.71 <2e-16 ***
  sky_glow  -14.90555  0.31770  -46.92 <2e-16 ***
```
> ---
> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> Residual standard error: 2.359 on 4998 degrees of freedom
> Multiple R-squared:  0.3058, Adjusted R-squared:  0.3056
> F-statistic: 2201 on 1 and 4998 DF,  p-value: < 2.2e-16

The significantly negative `sky_glow` coefficient makes sense. But this basic linear model is quite bad, as the following exercise illustrates.

**Exercise 5.4.** We will now overlay our regression line on a scatterplot of `star_count` and `sky_glow` to illustrate how poorly our model fits the data. Rather than use the regression coefficients from our `lm()` call, we will plot the regression line with `geom_smooth()` (which has `lm()` built in).

- Pipe `star_92` 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 0.45. For example, our model predicts that we’ll see an average of -2.12 shooting stars when `sky_glow = 0.6`. This nonsense prediction illustrates that our linear specification is overly simplistic. So we’ll develop a more realistic regression framework in the next section that models `star_count` as decreasing and **non-negative**.

---

## 5.2 Lecture

### 5.2.1 Constraining the Expected Value

The generalized linear model (GLM) unifies many statistical models, some of which you may have seen. It incorporates linear regression, Poisson regression, logistic regression, and multinomial regression as special cases. A GLM describes the distribution of a dependent variable, $y$, in terms of a weighted sum of independent variables, $w(x) = \beta_1 x_1 + \cdots + \beta_n x_n$. In this last expression, $x_1, \ldots, x_n$ are a collection of observed independent variables and $\beta_1, \ldots, \beta_n$ are a collection of unobserved coefficients, which we seek to estimate.

Now suppose that response variable $y$ has a normal distribution with mean $w(x)$ and standard deviation $\sigma$. In this case, OLS yields the maximum likelihood estimates, which are maximally efficient. But this isn’t the case for
star_count, which isn’t normally distributed (see Exercise 5.3) and doesn’t follow a straight line (see Exercise 5.4). Indeed, a linear model can’t properly describe this strictly positive variable because any non-flat line will be negative somewhere.

A proper model of star_count would yield positive predictions for all sky_glow values. The simplest way to prevent the expected value of star_count from being negative is to set it to \( \exp(w(x)) \) rather than \( w(x) \). We use the exponential function because (i) it has a nice interpretation (specifying geometric decay rather than arithmetic loss) and (ii) it is the simplest non-negative, increasing function. We can change the expected value from \( w(x) \) to \( \exp(w(x)) \) by switching from \( \text{lm()} \) to \( \text{glm()} \):

```
glm_model <-
star_92 %>%
glm(
  star_count ~ sky_glow,
  data = .,
  family = gaussian(link = "log"),
  start = c(0, 0)
)
glm_model %>%
  summary
```

```
> Call:
glm(formula = star_count ~ sky_glow, family = gaussian(link = "log"),
     data = ., start = c(0, 0))

> Deviance Residuals:
    Min       1Q   Median       3Q      Max
  -7.0010  -1.5801  -0.1692   1.4762  12.9989

> Coefficients:
  Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.003039    0.008282 241.86  <2e-16 ***
  sky_glow   -3.917036    0.103386  -37.89  <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> (Dispersion parameter for gaussian family taken to be 5.380844)
> Null deviance: 40088  on 4999  degrees of freedom
> Residual deviance: 26893  on 4998  degrees of freedom
> AIC: 22688
> Number of Fisher Scoring iterations: 9

The `start = c(0, 0)` line is a pesky technical condition. Basically, `glm()` estimates the parameters with an optimization routine, and the optimizer needs to be initialized with some starting value. So by specifying `start = c(0, 0)`, we’re telling to optimizer to start its search from the origin.

The family = `gaussian(link = "log")` line is more important. First, the `link = "log"` term tells the computer that we’re setting the expected value of `star_count` to `exp(w(x))`. The GLM specification allows us to set the expected value to `f(w(x))` for any function `f()`, so we must specify that we want `f()` to be the exponential function. If life were simple, we’d denote this with something like `link = "exp"`. Unfortunately, for historical reasons that predate the R programming language, we never directly refer to function `f()`. Instead we refer to its inverse, `f^{-1}()`, which we call the “link function.” So by setting `link = "log"`, we’re implicitly setting `f() = exp()` because the inverse of the exponential function is the logistic function.

The following exercise will depict our model’s fitted values. We will use the `augment()` function from the broom package, which is compatible with `glm()` models.

**Exercise 5.5.** We will now create a `glm()` analog of the plot we created in Exercise 5.4.

- Pipe `glm_model` into `augment(. , type.predict = "response")` to get our regression model’s fitted values.
  - Note: The `type.predict = "response"` option sets `.fitted` to `exp(w(x))` rather than `w(x)`.
- Pipe the resulting tibble into `ggplot()`.
- Create a scatter plot of `star_count` against `sky_glow` (as you did in Exercise 5.4).
- Create a second scatter plot of `.fitted` against `sky_glow`. Paint these dots red so that the red dots denote the expected number of shooting stars and the black dots denote the realized number of shooting stars.

### 5.2.2 Constraining the Distribution

Exercise 5.5 illustrates that our GLM model provides a reasonable characterization of the expected value of `star_count` given `sky_glow`. However, it provides an unreasonable characterization the distribution of `star_count` given `sky_glow`. Specifically, the `gaussian(link = "log")` term specifies that `star_count` follows a normal distribution with mean `exp(w(x))`. (Indeed, “Gaussian distribution” is just a fancy way of saying “normal distribution.”)
And since normal random variables can take any number from negative infinity to positive infinity, the computer thinks that its just a coincidence that all of the star_count values happen to be non-negative integers. This is preposterous, as the following exercise illustrates.

**Exercise 5.6.** We will now calculate the likelihood of none of the star_count values being negative, according to our model.

- Pipe `glm_model` into `augment(. , type.predict = "response")` to get a tibble with eight columns, two of which are named `.fitted` and `.sigma`.
- Our `glm` model treats a given `star_count` as normally distributed with a mean given by `.fitted` and a standard deviation given by `.sigma`. Use `pnorm()` to define a variable called `prob_nonnegative` that specifies the probability, under this distribution, of `star_count` being non-negative.
  - Check: The median `prob_nonnegative` is 0.9924.
- Use `pred()` to calculate the probability that our model assigns to all `star_count` values being non-negative.
  - Hint: The joint probability is the product of the individual probabilities.

Our model refutes our data. We clearly need a better model.

A proper empirical model will recognize that `star_count` is a non-negative integer. Hence, we must bake the fact that our dependent variable is a natural number into our GLM regression. We can do so by replacing `gaussian(link = "log")` with `poisson(link = "log")`:

```r
glm_model_2 <-
star_92 %>%
  glm(
    star_count ~ sky_glow, 
    data = ., 
    family = poisson(link = "log"), 
    start = c(0, 0)
  )

summary(glm_model_2 %>%
  Deviance Residuals:
  Min 1Q Median 3Q Max
```
5.2 Lecture

> -3.7472 -0.7717 -0.0753 0.6233 3.9894
>
> Coefficients:
> Estimate Std. Error z value Pr(>|z|)
> (Intercept) 2.006722 0.008588 233.7 <2e-16 ***
> sky_glow -3.975105 0.082813 -48.0 <2e-16 ***
> ---
> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
> (Dispersion parameter for poisson family taken to be 1)
>
> Null deviance: 8228.0 on 4999 degrees of freedom
> Residual deviance: 5335.7 on 4998 degrees of freedom
> AIC: 21998
>
> Number of Fisher Scoring iterations: 9

Under this specification, the computer treats star_count as a Poisson random variable with mean \( \exp(w(x)) \). And while a normal random variable can be any real number, a Poisson random variable is restricted to the non-negative integers. In fact, the Poisson distribution is the most common distribution over natural numbers. (In fact, the number of shooting stars observed in a given time frame is generally understood to follow a Poisson distribution.)

Our Poisson GLM model gives each value of sky_glow its own star_count distribution. We will depict this family of distributions in the following exercise.

**Exercise 5.7.** We will now plot the distributions that glm_model_2 assigns to star_count. 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.

- Pipe glm_model_2 into augment(., type.predict = "response") and keep the distinct() values of the resulting tibble.
  - Note: The type.predict = "response" option sets the .fitted column to the mean value of the Poisson random variable, \( \exp(w(x)) \).
- 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 evaluate this probability for all num_stars = 0, 1, ..., 9 for each value of .fitted. Call this new probability variable prob_count.
- Tip: Use group_by() to coerce R into evaluating this expression separately for each value of .fitted (so that dpois() receives a single number, rather than a vector, in its second argument). See Section 5.1.1.
- Check: Your tibble should have 50,000 rows after this step: 5,000 .fitted values times 10 num_stars values.
• Pipe the output into ggplot(). Scatter plot it with `x = sky_glow`, `y = prob_count`, and `colour = as.factor(num_stars)`.
• Check: Your plot should trace out 10 different lines, each painted a different color.

5.2.3 Binary Data

Load shooting_star_1994.csv into a tibble called star_94. The tibble should look like this:

```r
# A tibble: 6 x 3
sky_glow visibility star_obs
<dbl> <dbl> <lgl>
1 0.0377 0.818 TRUE
2 0.0641 0.663 TRUE
3 0.0126 0.0194 FALSE
4 0.0448 0.802 TRUE
5 0.0699 0.972 TRUE
6 0.131  0.105 FALSE
```

This data set describes whether any shooting stars were observed in a set of 15-minute measurements held across Eastern Oregon in 1994. Its independent variables are `sky_glow`, which we’ve seen before, and `visibility`, which measures air clarity. The dependent variable is logical `star_obs` which is TRUE if a shooting star was observed and is FALSE otherwise. But since regressions relate to numbers, our glm() call will treat each TRUE value as 1 and each FALSE value as 0 (it’ll do this automatically).

This binary data set is even more difficult to work with. Before we had to constrain the expectation of $y$ to a number greater than zero; now we must constrain it to a number greater than zero and less than one. And before we had to constrain the realization of $y$ to an integer no smaller than zero; now we must constrain it to an integer no smaller than zero and no larger than one. The simplest way to impose this binary structure is to specify that $y$ has probability $p$ of being 1 (i.e., TRUE) and probability $1-p$ of being 0 (i.e., FALSE), where probability $p = f(w(x))$ equals linear weighted sum $w(x)$ transformed by logistic function $f(x) = \frac{\exp(x)}{1+\exp(x)}$. This logistic function looks a mess, but it’s actually the simplest increasing function that’s guaranteed to return a number between 0 and 1. (I defy you to find a simpler one.)

Unfortunately, defining $f()$ doesn’t suffice, because the annoying historical convention I explained earlier compels us to characterize $f()$ via link function $f^{-1}()$. The inverse of the logistic function is the logit function, which has form $f^{-1}(p) = \log \left( \frac{p}{1-p} \right)$. We adopt the logit link function—and hence the logistic transformation function—by replacing `link = "log"` with `link = "logit"`. And
we force \( y \) to be binary by replacing `poisson()` with `binomial()`. Our `glm()` call thus becomes the following.

```r
glm_model_3 <-
  star_94 %>%
  glm(
    star_obs ~ sky_glow + visibility,
    data = .,
    family = binomial(link = "logit")
  )
```

```r
glm_model_3 %>%
  summary
```

> Call:
> `glm(formula = star_obs ~ sky_glow + visibility, family = binomial(link = "logit"),
>     data = .)`
>
> Deviance Residuals:
>    Min      1Q  Median      3Q     Max
> -1.0921 -0.9747 -0.8298  1.3492  2.2400
>
> Coefficients:
>                       Estimate Std. Error z value Pr(>|z|)
> (Intercept)          -0.35896   0.03872  -9.270  <2e-16 ***
> sky_glow           -3.49379   0.24552 -14.230  <2e-16 ***
> visibility          0.15656   0.06446   2.429   0.0151 *
> ---
> Signif. codes:  
> 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
> (Dispersion parameter for binomial family taken to be 1)
>
> Null deviance: 12900 on 9999 degrees of freedom
> Residual deviance: 12661 on 9997 degrees of freedom
> AIC: 12667
>
> Number of Fisher Scoring iterations: 4

This is the classic logistic regression. To repeat, it specifies that \( \text{star}_\text{obs} = \text{TRUE} \) with probability \( \frac{\exp(w(x))}{1 + \exp(w(x))} \) where \( w(x) = \beta_1 + \beta_2 \text{sky}_\text{glow} + \beta_3 \text{visibility} \), for estimated coefficients \( \beta_1, \beta_2, \) and \( \beta_3 \).

The following exercise will illustrate the power of our model in predicting \( \text{star}_\text{obs} \).
Exercise 5.8. To gauge our model’s statistical strength, we will plot its fitted values, which specify the probability \( p \) that \( \text{star}_\text{obs} = \text{TRUE} \). Specifically, we will compare the fitted values in the subsample with \( \text{star}_\text{obs} = \text{TRUE} \) against the fitted values in the subsample with \( \text{star}_\text{obs} = \text{FALSE} \). If our model has good predictive power then the fitted values should be high in the former sample and low in the latter sample. That is, our model should predict a high probability that \( \text{star}_\text{obs} = \text{TRUE} \) when this variable did turn out to be \( \text{TRUE} \) and, vice versa, a low probability that \( \text{star}_\text{obs} = \text{TRUE} \) when this variable did not turn out to be \( \text{TRUE} \).

- Pipe `glm_model_3` into `augment(., type.predict = "response")`.
  - Note: The `type.predict = "response"` option sets the `.fitted` column to the the probability that \( \text{star}_\text{obs} = \text{TRUE} \), \( p = \frac{\exp(w(x))}{1+\exp(w(x))} \).
- Pipe the resulting tibble into `ggplot()` and use `geom_density()` to plot the distribution of `.fitted`. Use `fill = \text{star}_\text{obs}` to give the \( \text{star}_\text{obs} = \text{TRUE} \) and \( \text{star}_\text{obs} = \text{FALSE} \) subsamples their own plots.
  - Tip: Include the option `alpha = .3` in your `geom_density()` call, so you can see both distributions, even when they overlap.
- Save this `ggplot()` as an R object called `density_plot` so we can later modify it.

Unfortunately, our model isn’t particularly powerful: e.g., the correlation between `.fitted` and \( \text{star}_\text{obs} \) is only 0.148. We’ll try to strength our model in the following exercise.

Exercise 5.9. Their may be an interaction between \text{sky\_glow} and \text{visiblity}. For example, a high \text{sky\_glow} may undermine the benefits of a high \text{visiblity}. To explore such a possibility, we will try adding a multiplicative interaction term into our linear model, setting \( w(x) = \beta_1 + \beta_2 \text{sky\_glow} \times \text{visiblity} + \beta_3 \text{sky\_glow} \times \text{visiblity} \).

- Change the regression formula from additive form `\text{star}_\text{obs} \sim \text{sky\_glow} + \text{visiblity}` to multiplicative form `\text{star}_\text{obs} \sim \text{sky\_glow} \times \text{visiblity}` to indicate that we want \text{sky\_glow}, \text{visiblity}, and \text{sky\_glow times visiblity} as independent variables.
- Rerun the `glm()` model with this new specification. Call the result `glm_model_4`.
- Recreate the density plots you made in Exercise 5.8 with `glm_model_4`.
  - Tip: Rather than redefine the plots, use the `%>%` convention to update the `density_plot` data.

The model is a bit better. The \text{sky\_glow}\text{visiblity} coefficient is statistically significant and the correlation between `.fitted` and \( \text{star}_\text{obs} \) increased from 0.148 to 0.152.
5.2.4 Counterfactuals

Copy this language into linear model chapter.

The counterfactual analysis is the most important use of a statistical model. Such an analysis enables us to anticipate how a hypothetical change in the independent variables would influence the dependent variable. To run a counterfactual analysis, we first estimate a model on a sample of data and then use the coefficient estimates to calculate the dependent variable’s expected value under an alternative set of independent variables.

Exercise 5.10. In this exercise we will estimate how the probability of seeing a shooting star changes as `sky_glow` changes by -50%, -40%, -30%, -20%, -10%, 0%, 10%, 20%, 30%, 40%, and 50%.

- Divide `star_94` `rowwise()`, so that each row represents its own `group_by()` group.
- Use `summarise()` to define vectors `change_percent = seq(-50, 50, 10)` and `sky_glow = sky_glow * (1 + change_percent / 100)` for each row of your tibble. Also have `summarise()` return the (unchanged) visibility values, so we don’t lose them.
  - Tip: See Section 5.1.1.
  - Check: This step should increase your tibble from 10,000 rows to 110,000 rows. The length
- Call the output tibble `hypothetical_indep_vars`.
  - Note: This tibble comprises 11 copies of your original sample, except the `sky_glow` variable is scaled by a different amount in each copy.
- Pipe `glm_model_4` into `augment(.)`, type.predict = "response"), but this time also use the option `newdata = hypothetical_indep_vars`, which tells R to compute the fitted values that correspond to the independent variables stored in `hypothetical_indep_vars`. This step creates our out-of-sample predictions.
- Use `geom_density()` to plot the distribution of `.fitted`, with `colour = as.factor(change_percent)`.
  - Note: Each line in your graph plots the distribution of `.fitted`—i.e., the distribution of the probability of seeing a shooting star—under a different hypothetical scenario.
5.3 Lab

5.3.1 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 ap-
propriate, 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 fol-
lowed 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 lim-
ited 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 indi-
vidual’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.

5.3.2 Sample

We will use facility, power, shutdown, ler, and radiation from Section 1.3, Exercise 1.11.

- facility: The pwr_bwr variable specifies whether the reactor is a pressurized water reactor (PWR) or a boiling water reactor (BWR). The pra_month variable specifies when the facility performed its PRA.
- Load power.csv into a tibble called power. The generation variable specifies the electricity output, in megawatt hours. This variable can be negative because the plants, themselves, use energy. The capacity_factor variable 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 = "voluntary" 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 = "involuntary" 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.” And 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: The ler variable specifies the total number of Licensee Event Reports filed. The actuation_ler, techspec_ler, degrade_ler, inoperable_ler, and safety_equi_ler variables specify the number of LERs that are flagged with the “actuation”, “technical specifications”, “degradation”, “inoperable”, and “prevent safety equipment” Code of Federal Regulations (CFR) codes. Note, a given LER can be assigned zero or multiple CRF codes.

- radiation: This table lists the employee radiation exposure levels that Blanco et al. (2019) got from Hausman (2014). The exposure_sum variable denotes the radiation exposure of the entire workforce collectively and the exposure_average variable denotes the radiation exposure of the average worker. These variables are measured Roentgen equivalent man (rem) units. The NRC defines a rem as

One of the two standard units used to measure the dose equivalent (or effective dose), which combines the amount of energy (from any type of ionizing radiation that is deposited in human tissue), along with the medical effects of the given type of radiation. For beta and gamma radiation, the dose equivalent is the same as the absorbed dose. By contrast, the dose equivalent is larger than the absorbed dose for alpha and neutron radiation, because these types of radiation are more damaging to the human body. Thus, the dose equivalent (in rems) is equal to the absorbed dose (in rads) multiplied by the quality factor of the type of radiation.

5.3.3 Clean

We will combine these tibbles into a master panel in Exercise 5.12. But first we must streamline our shutdown tibble.

Exercise 5.11. Use pivot_wider() to reshape shutdown into a tibble with variables reactor_name, start, shut_voluntary, shut_involuntary, 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 observation with reactor_name == "Browns Ferry 2" and month == ymd("1984-09-01") should have shut_voluntary == TRUE, shut_involuntary == TRUE, and shut_refuel == TRUE, because this reactor had stop_type
== "voluntary" and stop_type == "refuel" shutdowns with start == ymd("1984-09-01").

- Tip: See Exercise 5.2.
- Tip: Use the names_prefix = "shut" option of pivot_wider().
- Tip: The function that negates a logical has a strange name: it’s called `!`.
  For example, TRUE %% `! == FALSE and `!((FALSE) == TRUE. Hence, you can
  negate all logicals with mutate(across(where(is.logical), `!)).

We’re now ready to create our master panel.

**Exercise 5.12.**

- Create a variable in `power` called `year` that specifies the year of the given
date. (This variable will give the radiation tibble something to join to.)
- Create a list containing `facility`, `power`, `ler`, `radiation`, and `shutdown` in
  that order, and pipe it into reduce(left_join).
  - Note: This reduce() operation joins the elements of the list one
table at a time. In other words, it implements facility %>%
    `left_join(power)` %>% `left_join(ler)` %>% `left_join(radiation)` %>%
    `left_join(shutdown)`.
- The `shut_voluntary`, `shut_involuntary`, and `shut_refuel` variables will be
  NA unless the plant shut down in the specified month (the `shutdown` tibble
does not include records for months without shutdowns). 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_voluntary`, `shut_involuntary`,
  and `shut_refuel` with their cumulative sums (restarting the counts for each
  reactor_name).
  - Tip: Make sure a reactor’s observations are ordered by `month` before
    implementing cumsum().
- Bonus: 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.”
- Save these data in a tibble called `dat`.

**5.3.4 Analyze**

We are now ready to replicate the three Poisson regressions that Blanco et al.
(2019) reported in Table 2 of their manuscript. First we must characterize
the regression formulas. Unfortunately, the authors never clearly explained
what regressions they ran. Specifically, Table 2 specifies the cryptic condition “Reactor-level trends = Yes.” I emailed Christian Blanco asking him what this condition meant, and answered by explaining that “each reactor has its own slope. I think this might have been at the request of the reviewer [smiley emoji] and we interacted reactor dummy variables with year trends.” In other words, the authors gave each reactor its own intercept—which describes the reactor’s year-zero LER rate—and its own slope—which describes how the reactor’s LER rate changes from one year to the next. We can incorporate these reactor-specific intercepts and slopes by adding `as.factor(reactor_name) * year(month)` to the right-hand side of our regression formulas. This expression indicates that we want to multiply the reactor dummies generated by `as.factor(reactor_name)` with the year integers generated by `year(month)`. With this, our regression will incorporate 102 reactor-level intercepts that look like `as.factor(reactor_name)Arkansas Nuclear 2` and will incorporate 102 reactor-level slopes that look like `as.factor(reactor_name)Arkansas Nuclear 2:year(month)`. The former variables fully account for reactor fixed effects and the latter variables fully account for time trends, so this specification also satisfies Table 2’s “Reactor fixed effects = Yes” and “Time trends = Yes” conditions. So, all that said, we get the following regression formulas:

```r
formulas <-
paste(  
  "ler - pra",
  "+ as.factor(reactor_name) * year(month) + shut_refuel",
  c(  
    "",
    "+ shut_voluntary",
    "+ shut_voluntary + shut_involuntary"
  )
)
map(as.formula)

formulas

> [[1]]
> ler - pra + as.factor(reactor_name) * year(month) + shut_refuel
> <environment: 0x7f8d177b3d90>
>
> [[2]]
> ler - pra + as.factor(reactor_name) * year(month) + shut_refuel +
> shut_voluntary
> <environment: 0x7f8d177b3d90>
>
> [[3]]
> ler - pra + as.factor(reactor_name) * year(month) + shut_refuel +
> shut_voluntary + shut_involuntary
> <environment: 0x7f8d177b3d90>

The first formula corresponds to column (1) of Table 2, the second formula to column (2), and the third formula to column (3). We will run these regressions in the following exercise.

**Exercise 5.13.**

- Use `map()` to run the three Poisson regressions specified by `formulas` with the data in `dat`. Save the three `glm()` objects in a list called `estimated_models`.
  - Tip: Try removing the `start = .` option from your `glm()` call. (The optimizer can sometimes make due without.)
- Use `map_df()` to `tidy()` each model in `estimated_models`. And, while still under the `map_df()` umbrella, `mutate()` the output of `tidy()` to add the regression formula as a column. This variable, which you can call `specification`, will allow us to associate coefficient estimates with regression formulas.
  - Note: Since the regression models inherit the `.` inside the `map()` call, you can access the corresponding regression formula with `formula(.x) %>% deparse`. The `formula(.x)` call returns a formula object, which tibbles cannot store, and the `deparse()` call translates this formula object into a character object, which tibbles can store.
  - Check: The resulting tibble should have 627 rows and 6 columns.
- To focus on our primary coefficient estimates, `filter()` out the observations with `term := "prTRUE"`. Remove the `term` variable and call the resulting tibble `coef_est3`.

Our first regression, with `specification = "ler ~ pra + as.factor(reactor_name) * year(month) + shut_refuel"`, suggests that implementing PRAs reduced the number of LERs by \(1 - \exp(-0.164) = 15.1\%\). Adding `shut_voluntary` as an additional covariate weakens this to \(1 - \exp(-0.162) = 15.0\%\), and additionally adding `shut_involuntary` weakens this further to \(1 - \exp(-0.149) = 13.8\%\). Nevertheless, the result is strongly significant in all three cases.

The following exercise will more clearly illustrate the effect of PRAs on LERs.

**Exercise 5.14.** We will now conduct a counterfactual analysis that will gauge what the LER rate would have been, ceteris paribus, without the PRAs.

- Create a tibble called `current_dat` by starting with `dat` and then filtering to keep only the observations with `pra = TRUE`.
- Create a tibble called `counterfactual_dat` by taking `current_dat` and setting `pra = FALSE`. These observations represent the counterfactual scenario in which there were no PRAs.
- Create a tibble called `hypothetical_dat` by taking the `bind_rows()` combination of `current_dat` and `counterfactual_dat`. Note, this tibble repeats the same data with both `pra = TRUE` and `pra = FALSE`.


5.3 Lab

- Use `map_df()` to apply `augment(.x, newdata = hypothetical_dat, type.predict = "response")` to the three models in `estimated_models`. This `augment()` call will output the `hypothetical_dat` tibble, but with an extra column: `.fitted`, which reports the expected number of LERs given the variables in the current row. And while still under the `map_df()` umbrella, append to the output of your `augment()` call a variable called `specification` that reports the regression formula, in the fashion of Exercise 5.13. Call the tibble that `map_df()` outputs `out_of_sample_predictions`.
  - Check: `out_of_sample_predictions` should have 44,784 rows and 25 columns.
- Use `str_remove()` simplify the `specification` variable of `out_of_sample_predictions` so that it takes three values: "refuel", "voluntary", and "involuntary".
  - Hint: See what "text to remove shut_text to keep" ```str_remove(".*shut_.")` yields. The ".*shut_" regular expression tells R to select the longest possible substring that ends in "shut_". For more, see vignette("regular-expressions")
- Use `bq` to turn `specification` into a factor variable with levels "refuel", "voluntary", and "involuntary", in that order. This change will ensure that our plots are arranged according to the ordering of formulas (rather than arranged alphabetically).
- Use `geom_density()` to plot the distribution of `.fitted`. Set `fill = pra` to distinguish between the current and counterfactual scenarios and `facet_wrap()` by `specification` to distinguish between regression models. Save your plot as an R object called `plot_object` so you can later modify it.
  - Tip: Tune the alpha parameter of `geom_density()`.

To more comprehensively characterize the effect of implementing PRAs, we will now repeat the above analysis with some other specifications. However, we'll first create a few functions to avoid copy-and-pasting the code multiple times.

**Exercise 5.15.** We will now create functions to automate our implementations of `glm()`, `tidy()`, and `augment()`.

- Create a function called `run(glm)` that transforms formulas to estimated models. Specifically, this function should comprise the `map()` call that turned formulas into `estimated_models`.
- Create a function called `get_coefs()` that transforms estimated models to coefficient estimates. Specifically, this function should comprise the `map_df()`, `filter()`, and `select()` calls that turned `estimated_models` into `coef_est`s.
- Create a function called `get_prediction()` that transforms estimated models to out of sample predictions. Specifically, this function should comprise the `map_df()` call that turned `estimated_model` to `out_of_sample_predictions`.  


We’ll now use these functions to try out other specifications. First, we’ll explore whether the PRAs influenced different types of LERs differently with a streamlined version of the regressions Blanco et al. (2019) reported in Table 8.

Exercise 5.16.

- Define `formulas_2` as a list of five formulas that comprise five different dependent variables—`actuation_ler`, `techspec_ler`, `degrade_ler`, `inoperable_ler`, and `safety_equipment_ler`—and one common set of independent variables—pra + as.factor(reactor_name) + year(month) + shut_refuel + shut_voluntary + shut_involuntary.
  - Note: The set of independent variables is the same as the third regression above, except with as.factor(reactor_name) + year(month) rather than as.factor(reactor_name) * year(month). This specification satisfies the “Reactor fixed effects = Yes” and “Time trends = Yes” conditions of Table 8, but not the “Reactor-level trends = Yes” condition. Thus, our regressions won’t have as many control variables as Blanco et al.’s regressions, but will yield nearly the same results.
- Pipe `formulas_2` into `run_glm` to get a new set of estimate models, which you can call `estimated_model_2`.
- Pipe `estimated_model_2` into `get_coefs` to see the corresponding "praTRUE" coefficient estimates.
- Pipe `estimated_model_2` into `get_prediction` to get the analog of `out_of_sample_predictions` for these regressions.
- Use `str_remove()` to simplify this tibble’s specification variable to one that takes values in "actuation", "techspec", "degrade", "inoperable", and "safety_equipment".
  - Hint: See what this yields: "text to keep_ler text to remove" %>% str_remove("_ler.*").
- Save the resulting tibble to an R object called `data_to_plot`.
- Use `ggplot_object %>% data_to_plot` to recreate your plot with these new data. Add the `xlim(0, 1.5)` option to zoom in the plot.

Our results appear even stronger than Blanco et al.’s: whereas they report four significantly negative estimates, we report five significantly negative estimates. Our coefficient estimates are more pronounced because our regression specification is less conservative (i.e., has fewer control variables). However, the plots don’t appear as convincing because disaggregating the data disperses the effect across too many panels.

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 are exposed to. We will replicate our analysis with this alternative dependent variable in the following exercise.
Exercise 5.17. We will now rerun our analysis with `exposure_average` as the dependent variable.

- Define `formulas_3` as `formulas`, except with `exposure_average` as the dependent variable.
- Since `exposure_average` takes non-integer values, it cannot be modeled as a Poisson random variable, but it can be modeled as a gamma random variable. Accordingly, you will want to change your Poisson regression specification to a Gamma regression specification by replacing `family = poisson(link = "log")` with `family = Gamma(link = "log")` in your `glm()` call. Use `map()` to run this Gamma regression with each formula in `formulas_3` and call the output `estimated_models_3`.
  - Note: We can’t use `run_glmm()` here because we’re not running a Poisson regression.
- Pipe `estimated_models_3` into `get_coefs()` to get the new coefficient estimates.
  - Check: Your estimates should suggest that the radiation exposure levels are only around $\exp(-0.021) = 97.9\%$ as high after the PRAs were implemented.
- Now try piping `estimated_models_3` into `get_prediction()`. You’ll get an error. The problem is that some reactors don’t report `exposure_average` values. For example, this variable is `NA` for all `reactor_name == "Browns Ferry 2"` observations, so our regression doesn’t estimate a fixed effect for this reactor. And since it doesn’t have a fixed effect, `augment()` can’t derive `.fitted` values for this reactor. Fortunately, the fix is simple: remove from `hypothetical_dat` the observations in which `exposure_average` is `NA`. Now you can pipe `estimated_models_3` into `get_prediction()` to get the analog of `out_of_sample_predictions` for these regressions.
- Simplify this tibble’s specification variable so that it is a factor variable with levels "refuel", "voluntary", and "involuntary", in that order.
- Update your `plot_object` plot with these new data.

The effect is a lot weaker, but it’s still significant at the $p = 0.05$ level. And I believe this is a new result: I don’t think anyone has ever before noted the significant correlation between PRA adoption and radiation exposure levels.\footnote{Unfortunately, Section 5.4.3 calls this result into question.}
5.4 Commentary

5.4.1 Flexible-Input Functions

In Section 5.1.1 we used three completely different versions of \texttt{qnorm}():

- When defining \texttt{quantile.5 = qnorm(.05, mean, sd)}, we used a number for the first argument and vectors for the latter two arguments.
- In the \texttt{mutate(quantile = qnorm(q, mean, sd))} call, we used vectors for all three arguments.
- And in the \texttt{summarise(q = seq(.000, .999, .001), quantile = qnorm(q, mean, sd))} call, we used a vector for the first argument and numbers for the latter two (\texttt{mean} and \texttt{sd} represented numbers in this case because the data were grouped by these variables).

This illustrates a great feature of R: the permissiveness of its functions. Indeed, you can usually just pass what you have into a function and trust that R will do the right thing with it. For example, all of the following are valid R expressions:

\begin{verbatim}
dpois(x = 5, lambda = 5)
dpois(x = 1:3, lambda = 5)
dpois(x = 5, lambda = 1:3)
dpois(x = 1:3, lambda = 1:3)
dpois(x = 1:3, lambda = 1:8)
dpois(x = 1:8, lambda = 1:3)
\end{verbatim}

The first three calls are self-explanatory. The fourth tells R to evaluate \texttt{dpois()} under \texttt{x = 1} and \texttt{lambda = 1}, and then under \texttt{x = 2} and \texttt{lambda = 2}, and then under \texttt{x = 3} and \texttt{lambda = 3}. And the fifth and sixth calls invoke a rarely used convention: if a function receives multiple vectors inputs with varying lengths then it repeats the shorter vectors until they have the same length as the longest vector. So, for example, \texttt{dpois(x = 1:3, lambda = 1:8)} is really just \texttt{dpois(x = c(1, 2, 3, 1, 2, 3, 1, 2), lambda = 1:8)}.

5.4.2 Don’t Overburden Metadata

In Section 5.1.1, I switched from “wide mode” to “long mode” when I switched from calculating two quantiles to calculating 999 quantiles. That is, when I only needed the 0.05 and 0.95 quantiles I saved these data in distinct columns \texttt{quantile.5} and \texttt{quantile.95}, but when I needed all the .001, .002, ..., and .999 quantiles I did not save these data in distinct columns \texttt{quantile.001},
quantile_002, ..., and quantile_999. Instead, I pivoted longwise to lump the
data in a pooled column called quantile. The benefit of this pivot is that it
moves the numbers .001, ..., .999 from the column names to a variable, q. In
other words, it transforms metadata—the column labels—to formal data—a
column in the tibble. And formal data are always easier to work with than
metadata. This is why you should never have 1,000 columns: something that
takes 1,000 distinct values warrants being treated as “data”, saved inside the
tibble (in a column) rather than outside of it (in the column labels). And, per
the same principle, you should never rely on row names.

5.4.3 Be Wary of Too-Good-to-be-True Results

We didn’t get a single statistically insignificant coefficient estimate in Section
5.3. This is suspicious—empirical analyses rarely provide such clean results.
Indeed, the consistent strength of our regression estimates suggests that our
empirical specification may be too quick to output “statistically significant”
results. Accordingly, we will test whether our model outputs insignificant re-
results when it ought to by running our regressions with a dependent variable
that we’re fairly confident isn’t correlated with \( p_a \). If our model is properly
calibrated these regressions should yield statistically insignificant coefficient
estimates. This check is called a placebo test: like a sugar pill, our new depen-
dent variable should yield no significant results.

First, we must pick our placebo variable. The only real candidate in \( \text{dat} \) is
generation: it’s the one numeric variable that \( p_a \) shouldn’t meaningfully in-
fluence. But our Poisson specification requires the dependent variable to be a
non-negative integer, so we’ll derive our placebo variable from generation by
subtracting away its smallest value—which makes the vector non-negative—
and then rounding it to the nearest integer:

```r
dat <-
dat %>%
mutate(
  placebo = generation - min(generation),
  placebo = round(placebo)
)
```

Now let’s define our regression formulas as those used in Exercise 5.13, but
with `placebo` on the left-hand side:

```r<formulas_placebo <-
paste(
  "placebo - p_a",
```
"+ as.factor(reactor_name) * year(month) + shut_refuel",
  c(
    "",
    "+ shut_voluntary",
    "+ shut_voluntary + shut_involuntary"
  )
) %>%
map(as.formula)

We can now run our placebo test.

formulas_placebo %>%
run_glm %>%
get_coefs

> # A tibble: 3 x 5
>   estimate std.error statistic p.value specification
>    <dbl>     <dbl>     <dbl>    <chr> <chr>
> 1  1 -0.0138  0.0000453  -305. 0.    placebo - pra + as.factor(reactor_name) -
> 2  2 -0.00542  0.0000454   -119. 0.    placebo - pra + as.factor(reactor_name) -
> 3  3  0.00341  0.0000453    75.1 0.    placebo - pra + as.factor(reactor_name) -

These estimates should be insignificant, but they’re far from it: the results appear so strong that R rounds the p-value down to zero (i.e., it considers them significant at the \( p = 0.0000001 \) level).

Something must be wrong with our analysis. Let’s try replacing `glm()` with `lm()`:

run_lm <-
. %>%
map(~{
  lm(
    formula = .x,
    data = dat
  )
})

formulas_placebo %>%
run_lm %>%
get_coefs

> # A tibble: 3 x 5
>   estimate std.error statistic p.value specification
>    <dbl>     <dbl>     <dbl>    <chr> <chr>
Ah, now we get the insignificant estimates that we expected. \texttt{lm()} returning the expected results suggests that \texttt{glm()} must be the source of the problem. And, unfortunately, this will sometimes be the case, as \texttt{glm()} doesn’t always produce reasonable estimates: whereas a computer can always find the best estimates of a linear model, it can have difficulty finding sensible estimates—let alone the best estimates—of a generalized linear model. And if the computer can’t find a good way to fit a \texttt{glm()} model to the data, it will output an error message\(^3\) or bogus estimates (like those reported above).

Our failed placebo test calls into question our other \texttt{glm()} results. Let’s try replicating them with OLS:

\begin{verbatim}
formulas %>%
run_lm %>%
get_coefs

> # A tibble: 3 x 5
> estimate std.error statistic p.value specification
> <dbl>   <dbl>      <dbl>     <dbl>     <chr>
> 1 -0.150 0.0482   -3.10      0.00192 ler ~ pra + as.factor(reactor_name) * ye-
> 2 -0.144 0.0480   -3.00      0.00267 ler ~ pra + as.factor(reactor_name) * ye-
> 3 -0.136 0.0482   -2.83      0.00473 ler ~ pra + as.factor(reactor_name) * ye-

formulas_2 %>%
run_lm %>%
get_coefs

> # A tibble: 5 x 5
> estimate std.error statistic  p.value specification
> <dbl>   <dbl>      <dbl>   <dbl>     <chr>
> 1 -0.116 0.0265   -4.38  1.19e-5 actuation_ler ~ pra + as.factor(reacto-
> 2 -0.145 0.0287   -5.05 4.44e-7 techspec_ler ~ pra + as.factor(reacto-
> 3 -0.0596 0.0146  -4.09 4.36e-5 degrade_ler ~ pra + as.factor(reacto-
> 4 -0.0212 0.00685 -3.09 2.02e-3 inoperable_ler ~ pra + as.factor(react-
> 5 -0.0316 0.0128  -2.48 1.33e-2 safety equip_ler ~ pra + as.factor(rea-
\end{verbatim}

\(^3\)For example, we get this message when running the regressions of Exercise 5.16 with \texttt{as.factor(reactor_name)} \* \texttt{year(month)} on the right-hand side. That’s why we used \texttt{as.factor(reactor_name) + year(month)} for this problem.
```r
formulas_3 %>%
  run_lm %>%
  get_coefs
```

> # A tibble: 3 x 5
> estimate std.error statistic p.value specification
> <dbl> <dbl> <dbl> <chr>
> 1 0.00445 0.00262 1.70 0.0887 exposure_average ~ pra + as.factor(react-
> 2 0.00421 0.00261 1.62 0.106 exposure_average ~ pra + as.factor(react-
> 3 0.00331 0.00261 1.27 0.205 exposure_average ~ pra + as.factor(react-

Our `lm` results still hold significantly, but our `exposure_average` results don’t.

However, we didn’t use the best linear model for `exposure_average`. Our `glm()` model of `exposure_average` specified `link = "log"`, which implies that our independent variables linearly influence `log(exposure_average)` rather than `exposure_average`. So we should use `log(exposure_average)` as our dependent variable. And with this, we recover our statistical significance:

```r
formulas_4 <-
paste(
  "I(log(exposure_average)) ~ pra",
  "+ as.factor(reactor_name) * year(month) + shut_refuel",
  c(
    "",
    "+ shut_voluntary",
    "+ shut_voluntary + shut_involuntary"
  )
)
%>%
map(as.formula)

formulas_4 %>%
  run_lm %>%
  get_coefs
```

> # A tibble: 3 x 5
> estimate std.error statistic p.value specification
> <dbl> <dbl> <dbl> <chr>
> 1 -0.0267 0.0106 -2.52 0.0116 I(log(exposure_average)) ~ pra + as_fact-
> 2 -0.0275 0.0106 -2.60 0.00932 I(log(exposure_average)) ~ pra + as_fact-
> 3 -0.0295 0.0106 -2.79 0.00534 I(log(exposure_average)) ~ pra + as_fact-

Our `lm` Poisson regressions also specify `link = "log"`, so we should ideally use their logged values in their `lm()` regressions. Unfortunately, it’s not so clean in this case because our `lm` variables are sometimes zero and we can’t
take the logarithm of zero. So the best we can do is use \( \log(\text{ler} + 1) \) as our
dependent variable (although adding one is completely arbitrary). The results
appear even stronger with this logged specification:

```r
formulas_5 <-
paste(  
  "I(log(ler + 1)) ~ pra",
  "+ as.factor(reactor_name) * year(month) + shut_refuel",
  c(    
    "",
    "+ shut_voluntary",
    "+ shut_voluntary + shut_involuntary"
  ),
  "- pra",
  "+ as.factor(reactor_name) + year(month)",
  "+ shut_refuel + shut_voluntary + shut_involuntary"
)
%
map(as.formula)

formulas_6 <-
paste(  
  c(    
    "I(log(activation_ler + 1))",
    "I(log(techspec_ler + 1))",
    "I(log(degrade_ler + 1))",
    "I(log(inoperable_ler + 1))",
    "I(log(safety_equip_ler + 1))"
  ),
  "- pra",
  "+ as.factor(reactor_name) + year(month)",
  "+ shut_refuel + shut_voluntary + shut_involuntary"
)
%
map(as.formula)

formulas_5 %>%
run_lm %>%
get_coeffs

> # A tibble: 3 x 5
>   estimate std.error statistic  p.value specification
>   <dbl>    <dbl>     <dbl>     <dbl> <chr>
> 1 -0.0813 0.0171    -4.75     2.04e-6 I(log(ler + 1)) ~ pra + as.factor(react-
> 2 -0.0796 0.0171    -4.67     3.09e-6 I(log(ler + 1)) ~ pra + as.factor(react-
> 3 -0.0763 0.0171    -4.45     8.47e-6 I(log(ler + 1)) ~ pra + as.factor(react-

formulas_6 %>%
```
5 Generalized Linear Models

```r
run_lm %>%
get_coefs
```

> # A tibble: 5 x 5
>    estimate std.error statistic p.value specification
>    <dbl>    <dbl>     <dbl>    <dbl>          <chr>
> 1  0.0780  0.0127     6.14 8.66e-10 I(log(actuation_1er + 1)) ~ pra + as.fac-
> 2  0.0784  0.0140     5.62 1.97e- 8 I(log(techspec_1er + 1)) ~ pra + as.fact-
> 3  0.0323  0.00825    3.92 8.90e- 5 I(log(degrade_1er + 1)) ~ pra + as.fac-
> 4  0.0138  0.00444    2.93 3.34e- 3 I(log(inoperable_1er + 1)) ~ pra + as.f-
> 5  0.0205  0.00777    2.63 8.47e- 3 I(log(safety_equip_1er + 1)) ~ pra + as-

There are a few lessons here:

- An empirical model can go haywire and start labeling everything as “significant.” You should check for this—especially if everything seems to be going your way—with a placebo test that confirms that your specification does classify insignificant relationships as insignificant.
- `glm()` can be unreliable, especially when there are a lot of variables. So you should always compare your `glm()` estimates to to the corresponding `lm()` estimates. If switching from `glm()` to `lm()` flips your key estimate’s sign or undermines its statistical significance then you should be wary, because a linear model—even though it may not honor the non-negativity or integrality of your dependent variable—will always yield something pretty close to the right answer. In contrast, `glm()` can sometimes output nonsensical estimates. So if there’s a drastic discrepancy between your `glm()` and `lm()` outputs, the latter is probably closer to the truth.
- If possible, apply the link function, \( f^{-1}(\cdot) \), to your dependent variable to get the `lm()` model that’s most similar to your `glm()` model.

### 5.4.4 Use Many Starting Points

In Section 5.4.3 we saw that a `glm()` model can fail miserably. When asked to estimate a `glm()` model the computer searches frantically for set of estimates that explain the data. But sometimes it comes up empty handed and returns nonsense (or an error). However, we have recourse when this happens: try again with a different `start` value. The `start = .` option of `glm()` tells the computer where to start looking for a good set of estimates. Thus, a different `start` value leads the computer down a different search path.

If your model is complex then you should run `glm()` with a few dozen—if not a few hundred—`start` values.\(^4\) You can then pick the best fitting model by selecting the one with the lowest deviance, which is a goodness-of-fit statistic.

\(^4\)You can run these `glm` calls in parallel if your computer has multiple cores.
5.5 Solutions

5.1

```r
inauguration_dates %>%
group_by(prez) %>%
summarise(
    first_100_days = date + days(0:99),
    .groups = "drop"
)
```

> # A tibble: 1,100 x 2
>   prez  first_100_days
>   <chr>       <date>
> 1 biden 2021-01-20
> 2 biden 2021-01-21
> 3 biden 2021-01-22
> 4 biden 2021-01-23
> 5 biden 2021-01-24
> 6 biden 2021-01-25
> 7 biden 2021-01-26
> 8 biden 2021-01-27
> 9 biden 2021-01-28
>10 biden 2021-01-29
> # ... with 1,090 more rows

5.2

```r
tib_3 %>%
pivot_wider(
    id_cols = id,
    names_from = var,
```

5.5 Solutions

analogous to the sum of squared errors. Moreover, using multiple start values can help you gauge how good your estimates are. For example, if every start value leads to the same estimates, you’ve probably found the true best estimates. But if every start value leads to a different set of estimates, then you probably haven’t yet found the true best estimates.

---

5.5 Solutions

5.1

```r
inauguration_dates %>%
group_by(prez) %>%
summarise(
    first_100_days = date + days(0:99),
    .groups = "drop"
)
```

> # A tibble: 1,100 x 2
>   prez  first_100_days
>   <chr>       <date>
> 1 biden 2021-01-20
> 2 biden 2021-01-21
> 3 biden 2021-01-22
> 4 biden 2021-01-23
> 5 biden 2021-01-24
> 6 biden 2021-01-25
> 7 biden 2021-01-26
> 8 biden 2021-01-27
> 9 biden 2021-01-28
>10 biden 2021-01-29
> # ... with 1,090 more rows

5.2

```r
tib_3 %>%
pivot_wider(
    id_cols = id,
    names_from = var,
```

---

For example, the deviance of the model we estimated in Exercise 5.9 is `glm_model_4$deviance = 1.2645 \times 10^4`. 
```r
values_from = var,
values_fn = is.null,
values_fill = TRUE
)

> # A tibble: 3 x 4
> id   a   b   c
> <chr> <lgl> <lgl> <lgl>
> 1 x   FALSE TRUE TRUE
> 2 y   FALSE FALSE TRUE
> 3 z   TRUE FALSE FALSE

5.3

star_92 %>%
  mutate(
    sky_glow_decile = ntile(sky_glow, 10),
    sky_glow_decile = as.factor(sky_glow_decile),
    sky_glow_decile =
      fct_relabel(
        sky_glow_decile,
        ~ paste("Decile: ", .)
      )
  )
  ggplot() +
  aes(x = star_count) +
  geom_histogram() +
  facet_wrap(
    vars(sky_glow_decile),
    nrow = 2
  ) +
  labs(y = "Number of Observations") +
  theme_bw() +
  theme(strip.background = element_rect(fill="white", colour="white"))
```
5.5 Solutions

5.4

```r
star_92 %>%
ggplot() +
aes(
  x = sky_glow,
  y = star_count
) +
geom_point() +
geom_smooth(method = "lm") +
theme_bw()
```
```r
glm_model %>%
  augment(type.predict = "response") %>%
ggplot() +
aes(x = sky_glow) +
geom_point(aes(y = star_count)) +
geom_point(
  aes(y = .fitted),
  colour = "red"
) +
labs(
  x = "x (i.e., sky_glow)",
  y = "exp(w(x)) (i.e., expected number of shooting stars)"
) +
theme_bw()
```
5.6

```r
glm_model %>%
augment(type.predict = "response") %>%
mutate(prob_nonnegative = 1 - pnorm(0, .fitted, .sigma)) %>%
summarise(prod(prob_nonnegative))
```

> # A tibble: 1 x 1
> `prod(prob_nonnegative)`
> <dbl>  
> 1 1.0e-83

5.7

```r
glm_model_2 %>%
augment(type.predict = "response") %>%
distinct %>%
group_by(across(everything())) %>%
summarise(
  num_stars = 0:9,
  prob_count = dpois(num_stars, .fitted)
) %>%
ggplot() +
aes(
  x = sky_glow,
  y = prob_count,
```
5.8

density_plot <-
glm_model_3 %>%
augment(type.predict = "response") %>%
geomplot() +
aes(
  x = .fitted,
  fill = star_obs
) +
geom_density(alpha = .3) +
labs(
  x = "Probability of Shooting Star",
  y = "Distribution Density",
  fill = "Shooting Star Observed"
) +
5.5 Solutions

```r
theme_bw() +
theme(legend.position = "top")

plot(density_plot)
```

5.9

```r
glm_model_4 <-
  star_94 %>%
  glm(
    star_obs ~ sky_glow * visibility,
    data = .,
    family = binomial(link = "logit")
  )

density_plot %>% {
  glm_model_4 %>%
  augment(type.predict = "response")
}
```
5.10

```r
hypothetical_indep_vars <-
  star_94 %>%
  rowwise %>%
  summarise(
    visibility,
    change_percent = seq(-50, 50, 10),
    sky_glow = sky_glow * (1 + change_percent / 100),
    .groups = "drop"
  )

glm_model_4 %>%
  augment(
    type.predict = "response",
    newdata = hypothetical_indep_vars
  ) %>%
  ggplot() +
  aes(
    x = .fitted,
    colour = as.factor(change_percent)
  ) +
  geom_density() +
  labs(
    x = "Probability of Shooting Star",
    y = "Distribution Density",
  )
```
5.5 Solutions

colour = "Change in sky_glow"
+
theme_bw()
6

Spatial Data

6.1 Homework

6.1.1 Preparation

- Download and skim the following reference materials, ignoring the parts related to the sp package:
  - Geocomputation with R\(^1\)
  - Using Spatial Data with R\(^2\)
  - Drawing beautiful maps programmatically with R, sf and ggplot2\(^3\)
  - Spatial Manipulation with sf: Cheat Sheet\(^4\)
  - The vignettes returned by `browseVignettes("sf")`.
- Install the sf and spData packages and load them into your library.
- Loading spData should give you access to several new tables of data. Test this by entering `seine` into your console. You should get the following:

```
seine
```

```
Simple feature collection with 3 features and 1 field
geometry type: MULTILINESTRING
dimension: XY
bbox:   xmin: 518344.7 ymin: 6660431 xmax: 879955.3 ymax: 6938864
projected CRS: RGF93 / Lambert-93
name  geometry
1 Marne MULTILINESTRING ((879955.3 ...)
2 Seine MULTILINESTRING ((828893.6 ...)
3 Yonne MULTILINESTRING ((773482.1 ...)
```

\(^1\)https://geocompr.robinlovelace.net/
\(^2\)https://cengel.github.io/R-spatial/
\(^3\)https://www.r-spatial.org/r/2018/10/25/ggplot2-sf.html
\(^4\)https://github.com/rstudio/cheatsheets/blob/master/sf.pdf
6.1.2 Study Questions

Suppose you wanted to store all the information stored in the following map in a collection of Excel spreadsheets.

How would you represent the data? For example,

- How would you specify the location of New Madrid, Missouri?
- How would you describe the Arkansas River?
- Or how would you save the dimensions of Lake Michigan?

6.2 Lecture

6.2.1 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 \texttt{sp} and \texttt{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 study this more modern package.

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:

- sf 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 sf 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. The sf package can be a bit confusing at first. But it gives you tremendous power once you get the hang of it. Indeed, it comprises dozens of functions—see this cheat sheet—so it will have 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.”

### 6.2.2 Geometric Objects

In the sf package, the fundamental geometric unit is the simple feature geometry, which has class sf. These sf 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(28) %>%
  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()`:

---

[1](https://github.com/rstudio/cheatsheets/blob/master/sf.pdf)
A **MULTILINESTRING** is a `sf` 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" "sf"
We can analogously create POINT, MULTIPoint, POLYGON, and MULTIPOLYGON sfg objects with `st_point()`, `st_multipoint()`, `st_polygon()`, and `st_multipolygon()` options. As expected, we define a POINT object with a single (x, y) coordinate pair (e.g., `st_point(c(0, 1))`) and a MULTIPOLYGON object with a matrix of coordinate pairs (e.g., `st_multipoint(matrix(runif(8), 4))`). However, the POLYGON and MULTIPOLYGON objects are more difficult to define. For example, the following constructs and plots a single POLYGON object:

```r
base_triangle <-
  rbind(  
    c(0, 0),
    c(1/2, 1/2),
    c(1, 0),
  )
```
c(0, 0)
)

base_square <-
  rbind(
    c(0, 0),
    c(1, 0),
    c(1, 1),
    c(0, 1),
    c(0, 0)
  )

shape_list <-
  list(
    2 * base_square,
    base_square/2 + 1/4,
    base_triangle/6 + 1/2,
    base_square/16 + 1/3,
    2 * base_triangle + 1
  )

complex_polygon <-
  shape_list %>%
  st_polygon

complex_polygon %>%
  ggplot() +
  geom_sf() +
  theme_bw()
As you see, a POLYGON object is not a single polygon, but rather a collection of polygons layered on top of one another. This collection of polygons partitions the plane into an “interior” region and an “exterior” region, where the inside points are those that reside in an odd number of polygons and the outside points those that reside in an even number. We construct this object by giving `st_polygon()` a list of matrices (e.g., `shape_list`), where each matrix stores the coordinates of a polygon’s corners. Note, the first and last row of each matrix must be the same, to ensure that the polygons close off (e.g., `base_triangle` both starts and ends at point `c(0, 0)`).

Whereas a POLYGON object stems from a list of matrices, a MULTIPOLYGON object stems from a list of a list of matrices (or from a list of POLYGON objects). A MULTIPOLYGON object stores a collection of spacial 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))
) %>%
```
6.2.3 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
> random_data   dots
> <int> <MULTIPOINT>
> 1 1 ((0.1855241 0.328261), (0.9228731 0.705254), (0.6556705 0.5909134~
> 2 2 ((0.7683677 0.4158775), (0.6824085 0.1293008), (0.7033164 0.03682~
> 3 3 ((0.9790432 0.7630251), (0.7899535 0.4607254), (0.258433 0.659334~

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
> Simple feature collection with 3 features and 1 field
> geometry type: MULTIPOLY
> dimension: XY
> bbox: xmin: 0.1691795 ymin: 0.03615434 xmax: 0.9914432 ymax: 0.9875046
> CRS: NA
> # A tibble: 3 x 2
>   random_data dots
>   <int>     <MULTIPOINT>
> 1 1 (0.1855241 0.328261), (0.9228731 0.705254), (0.6556705 0.5909134-
> 2 2 (0.7683677 0.4158775), (0.6824008 0.1293008), (0.7033164 0.03682-
> 3 3 (0.9790432 0.7630251), (0.7899535 0.4607254), (0.258433 0.659334-

Changing `mini_tibble` to `mini_sf` gives the table extra meta data attributes: a bounding box that encompasses the geometric objects, a MULTIPOLY 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

```
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 6.1.** 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`.

---

6EPGS stands for the European Petroleum Survey Group, a defunct organization.
6.2 Lecture

- Check: The mean lifeExp is 58.11 years in the LE_group == 1 subsample.
- Check: This step compresses the the sf table from 167 rows and 12 columns to five rows and three columns.
- Round lifeExp to the nearest hundredth of a year and save it as a factor variable.
- ggplot() the map with the fill = lifeExp option.
- Note: The map is much clearer when lifeExp is a factor than when it’s a number.

Exercise 6.1 illustrates one of the key differences between tibbles and sf tables. If world were a tibble then the summarise() operation would discard the un referenced 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_zoom ||= 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_zoom ||=
  group_by(continent) %>%
  summarise(area = sum(area_km2))
```

> Simple feature collection with 8 features and 2 fields
> geometry type: MULTIPOLYGON
> dimension: XY
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 6.2. 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.
  - Check: nz_sex should have two columns and two rows.
  - Note: 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.
- ggpplot() nz_sex with the fill = mostly_men option.

Exercise 6.3. Partition New Zealand, in the fashion of Exercise 6.2, 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 6.4. Add country borders to the map you made in Exercise 6.1. 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.

- Hint: Change one word of your former solution.
6.2.4 Geometric Joins

The intersection of two shapes is the area that lies in both (e.g., the red-orange portion of the Mastercard logo) and the difference of two shapes is the area that lies in the first one and not the second one (e.g., the red portion of the Mastercard logo). The intersection operation is roughly analogous to an inner_join() and the difference operation to an anti_join().

st_intersection() implements geometric intersections. It takes two sf tables and returns a third sf table that reports the geometric intersections. Specifically, if x and y are sf tables then st_intersection(x, y) joins every row of x with every row of y for which the x and y geometric objects intersect, and sets the new geometric variable to the corresponding overlapping region. For example, the following depicts the st_intersection() of nz_sex and nz_wealth (from Exercises 6.2 and 6.3):

```r
st_intersection(
  nz_sex,
  nz_wealth
) %>%
  mutate(
    mostly_men = paste("men:", mostly_men),
    richest_part = paste("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).⁷

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 6.5.** We will now calculate the number of peaks that reside in each district.

- Take the st_intersection() of nz and nz_height.
  - Note: The joined table has the same number of rows as nz_height because each peak intersects only one district.
  - Note: 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.
  - Note: This count() operation doesn’t drop the geometry column, as

---

⁷You can remove this border line with st_collection_extract("POLYGON"), which discards all geometries besides enclosed polygons.
6.2 Lecture

it would with a standard tibble. Instead, it aggregates the mountain coordinates at the district level, so that the districts with multiple peaks have MULTIPOLY geometries.

Exercise 6.6. We will now plot the region of New Zealand that’s within 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.
  - Note: This step turns the POINT geometries to POLYGON geometries.
  - Note: `st_crs(nz_height)$units` confirms that lengths are measured in meters (under this CRS).
- Intersect the result with `nz`.
  - Check: The average `ts@fid` circle intersects 317/101 = 3.14 districts.
- `group_by(Name)` and `summarise()`.
  - Note: This step flatten the various circles into cohesive regions, which makes the map look cleaner.
- `ggplot()` the result, with `fill = Name`.

The other key spatial join is the geometric difference, which we implement with `st_difference()`. It takes two sf tables and returns a third sf table that reports the geometric differences. Specifically, if `x` and `y` are sf tables then `st_difference(x, y)` joins every row of `x` with every row of `y` for which the `y` geometric object does not subsume the `x` geometric object, and sets the new geometric variable to the region in the `x` object but not the `y` object. For example, the following depicts the `st_difference()` of `nz_sex` and `nz_wealth`:

```r
st_difference(
  nz_sex,
  nz_wealth
) %>%
  mutate(
    mostly_men = paste("men:", mostly_men),
    richest_part = paste("rich:", richest_part)
  ) %>%
  ggplot() +
  geom_sf() +
  facet_grid(
    rows = vars(richest_part),
    cols = vars(mostly_men)
  ) +
  theme_bw()
```
Note, the output of `st_difference()` does not have a `mostly_men == TRUE, richest_part == FALSE` row, because there’s nothing left when we subtract the `richest_part == FALSE` zone from the `mostly_men == TRUE`.

Or for another example, here’s the part of New Zealand that’s not within 100 Km of any of its 101 tallest peaks:

```r
nz %>%
  st_difference(
    nz_height %>%
      st_buffer(100000) %>%
      summarise
  ) %>%
  ggplot() +
  aes(fill = Name) +
  geom_sf() +
  theme_bw() +
  theme(legend.position = "none")
```
Note, the summarise() step after st_buffer() is crucial! It takes the union of the 101 circles outputted by st_buffer() to create a single MULTIPOLYGON that comprises all points within 100 Km of a peak. Taking the st_difference() with respect to this MULTIPOLYGON removes all the points within 100 Km of any peak. The result is an sf table with 16 rows—one for each district—where a given row describes the region of a given district that’s more than 100 Km from all peaks.

In contrast, if we didn’t include the summarise() step then the st_difference() operation would be performed peak by peak. The result would be an sf table with 1,616 rows—one for each (peak, district) combination—where a given row would describe the region of a given district that’s more than 100 Km from a given peak.

To reiterate, if x and y are sf tables then st_difference(x, y) returns the geometric difference of every combination of x and y elements. But we don’t usually want to take geometric differences with respect to individual elements of y. Instead, we usually want to take geometric differences with respect to the entirety of y—i.e., to remove the area in x that belongs to any element of y. And to do this, we must first flatten y with summarise() and before implementing the st_difference().

The following exercises provide additional examples.
Exercise 6.7. 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.

- Tip: Don’t forget to flatten the hull polygons before differencing them out.

Exercise 6.8. Use `st_convex_hull()` and `st_difference()` to plot the region of the world that does not lie between Ecuador and Japan. That is, remove from our map the diagonal strip that lies between these two countries.

- Tip: Be conscious of the order in which you implement the `st_convex_hull()` and `summarise()` steps.

6.3 Lab

6.3.1 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.

6.3.2 Import

Access the project data here\textsuperscript{8}. The following exercises will help you load these tables into your R environment.

Exercise 6.9. 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\textsuperscript{9}. Use \texttt{map_df()} to load these polygon maps into an \texttt{sfb} table called RFMO.

- **Tip:** Although its name suggests that it’ll output a data frame, \texttt{map_df()} will indeed output an \texttt{sfb} table.
- **Tip:** Try using \texttt{dir()} with the \texttt{full.names = TRUE} option.
- **Check:** Your \texttt{sfb} table should have 32 rows and 12 columns.

Exercise 6.10. Use \texttt{read_rds()} to load polygons/eoz/eoz.rds into an \texttt{sfb} table called Eoz. 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.

\textsuperscript{8}https://www.dropbox.com/sh/yh4elc9tyl2nyqf/AAAyjR81cWd_oValubU2ZTna?dl=0
\textsuperscript{9}http://www.fao.org/figis/geoserver/factsheets/rfbs.html
Exercise 6.11. Use read_csv() to load transshipment.csv into a tibble called all_vessles and load authorized.csv into a tibble called authorized_vessles. The former file describes all vessels that are capable of transshipping and the latter describes all vessels that are legally authorized to transship.

- Tip: Use the col_types = "iccciiTT" option when loading transshipment.csv, to ensure that mmsi andimo are treated as integers rather than doubles (i.e., fractional numbers).

Exercise 6.12. 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 instances “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.”

6.3.3 Clean

We’ll clean our data with the following exercises.

Exercise 6.13. We’ll first derive an sf table called incidents from loiter and encounter.

- In loiter, (i) define lat = (starting_latitude + ending_latitude) / 2 and long = (starting_longitude + ending_longitude) / 2, (ii) rename transshipment_mmsi as mmsi and starting_timestamp as time, and (iii) drop all variables besides mmsi, lat, long, and time.
- Modify encounter to look like loiter (i.e., with columns mmsi, lat, long, and time).
- Use bind_rows() to combine loiter and encounter into a unified tibble called incidents. This tibble should have a new variable called activity, which is either "loiter" or "encounter", based on where the observation originated.
  - Tip: Before combining them, give loiter an activity = "loiter" variable and give encounter an activity = "encounter".
- Use st_as_sf(. , coords = c(x = "long", y = "lat")) to turn incidents into an sf table.
- Use st_crs(incidents) <- st_crs(RFMO) to give incidents the coordinate reference system used by RFMO (which is the World Geodetic System 1984).

Exercise 6.14. We’ll now remove from incidents the vessels that are authorized to transship.

- Use anti_join() to remove the ships in authorized_vessles from all_vessles. Call the resulting tibble unauthorized_vessles
  - Tip: Join the tibbles by IMO number.
• Use `inner_join()` to drop the observations in incidents that do not correspond to a ship in `unauthorized_vessels`.
  - Tip: Join the tibbles by MMSI number.

**Exercise 6.15.** Our `sf` tables are cumbersomely large. For example, calling `plot(EEZ)` will probably crash your R session. Use `st_simplify( , dTolerance = .5)` to streamline RFMO and `EEZ`. This operation will replace the complex borders with simpler approximations that are accurate to within a half angular degree of precision.

• Note: Ignore the warning message you get on this step.

**Exercise 6.16.** Ewell et al. (2017) report that the RFMO zones that ban transshipments (either fully or partially) are SEAFO, IATTC, ICCAT, IOTC, GFPCM, 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 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.
  - Note: 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.
  - Tip: Define the `ban` variable in `no_ban_region` and `ban_region` before combining these tables.
  - Check: `RFMO` should have two columns and two rows.

**Exercise 6.17.** 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.
  – Note: Ignore the warning message you get on this step.
  – Check: RFMO should be two-by-two before and after this step.

**Exercise 6.18.** 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.
• Note: This step will probably take your computer a few minutes.
• Check: 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.

### 6.3.4 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 6.19.** 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.
• 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.
  – Note: 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 colour = "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 6.20.** 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.
  - Note: 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**.
  - Check: 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.

### 6.4 Solutions

6.1

```r
world %>%
  filter(!is.na(lifeExp)) %>%
  mutate(LE_group = ntile(lifeExp, 5)) %>%
  group_by(LE_group) %>%
  summarise(
    lifeExp = mean(lifeExp),
```
6.2

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()
6.3

```r
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()
```
6.4
Change `summarise()` to `mutate()` in your solution to Exercise 6.1.

6.5

```r
st_intersection(
  nz,
  nz_height
) %>%
  count(Name)
```

> Simple feature collection with 7 features and 2 fields
> geometry type: GEOMETRY
> dimension: XY
> bbox: 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...
6.4 Solutions

> 5 Southland 1 POINT (1204143 5849971)
> 6 Waikato 3 MULTIPOLYGON ((1820860 564948...)
> 7 West Coast 22 MULTIPOLYGON ((1259702 507657...

6.6

nz_height %>%
  st_buffer(100000) %>%
  st_intersection(nz) %>%
  group_by(Name) %>%
  summarise() %>%
  ggplot() +
  aes(fill = Name) +
  geom_sf() +
  theme_bu() +
  theme(legend.title = element_blank())

![Map of New Zealand showing regions with different colors for each region.](image)

6.7

world %>%
  st_difference(
    filter(. , continent == 'Asia') %>%
    st_convex_hull %>%
    summarise
```r
world %>%
  st_difference(
    filter(., name.long %in% c("Ecuador", "Japan")) %>%
    summarise %>%
    st_convex_hull
  ) %>%
  ggplot() +
  geom_sf() +
  theme_bw()
```
Bibliography


