

Crime by the Numbers

Jacob Kaplan

2020-07-31

Contents

1	Introduction to R and RStudio	17
1.1	Using RStudio	17
1.1.1	Opening an R Script	19
1.1.2	Setting the working directory	20
1.1.3	Changing RStudio	22
1.1.4	Helpful Cheat Sheets	26
1.2	Reading data into R	27
1.2.1	Loading data	27
1.3	First steps to exploring data	28
1.4	Finding help about functions	31
I	Clean	33
2	Subsetting: Making big things small	35
2.1	Select specific values	35
2.2	Assigning values to objects (Making “things”)	37
2.3	Vectors (collections of “things”)	39
2.4	Logical values and operations	39
2.4.1	Matching a single value	40
2.4.2	Matching multiple values	41
2.4.3	Does not match	42
2.4.4	Greater than or less than	42
2.4.5	Combining conditional statements - or, and	43
2.5	Subsetting a data.frame	43
2.5.1	Select specific columns	47
2.5.2	Select specific rows	48

2.5.3	Subset Colorado data	54
3	Exploratory data analysis	57
3.1	Summary and Table	59
3.2	Graphing	64
3.3	Aggregating (summaries of groups)	67
4	Regular Expressions	73
4.1	Finding patterns in text with <code>grep()</code>	75
4.2	Finding and replacing patterns in text with <code>gsub()</code>	79
4.3	Useful special characters	83
4.3.1	Multiple characters []	83
4.3.2	n-many of previous character {n}	85
4.3.3	n-many to m-many of previous character {n,m}	86
4.3.4	Start of string	88
4.3.5	End of string \$	89
4.3.6	Anything	89
4.3.7	One or more of previous +	89
4.3.8	Zero or more of previous *	90
4.3.9	Multiple patterns 	92
4.3.10	Parentheses ()	92
4.3.11	Optional text ?	93
4.4	Changing capitalization	93
5	Reading and Writing Data	97
5.1	Reading Data into R	97
5.1.1	R	97
5.1.2	Excel	98
5.1.3	Stata	99
5.1.4	SAS	100
5.1.5	SPSS	100
5.2	Writing Data	100
5.2.1	R	101
5.2.2	Excel	101
5.2.3	Stata	101
5.2.4	SAS	101
5.2.5	SPSS	101

CONTENTS	5
----------	---

II Visualize	103
6 Graphing with ggplot2	105
6.1 What does the data look like?	106
6.2 Graphing data	108
6.3 Time-Series Plots	109
6.4 Scatter Plots	119
6.5 Color blindness	120
7 More graphing with ggplot2	123
7.1 Exploring Data	124
7.2 Graphing a Single Numeric Variable	133
7.2.1 Histogram	134
7.2.2 Density plot	138
7.2.3 Count Graph	139
7.2.4 Graphing a Categorical Variable	140
7.3 Bar graph	140
7.4 Graphing Data Over Time	146
7.5 Pretty Graphs	150
7.5.1 Themes	150
8 Hotspot maps	155
8.1 A simple map	157
8.2 What really are maps?	164
8.3 Making a hotspot map	164
8.3.1 Colors	168
9 Choropleth maps	171
9.1 Spatial joins	178
9.2 Making choropleth maps	185
10 Interactive maps	191
10.1 Why do interactive graphs matter?	192
10.1.1 Understanding your data	192
10.1.2 Police departments use them	192
10.2 Making the interactive map	193
10.3 Adding popup information	198
10.4 Dealing with too many markers	202

10.5 Interactive choropleth maps	203
11 R Markdown	213
11.1 Code	218
11.1.1 Hiding code in the output	220
11.2 Inline Code	220
11.3 Tables	221
11.4 Footnotes	222
11.5 Citation	222
11.6 Spell check	227
11.7 Making the output file	228
III Collect	229
12 Webscraping with rvest	231
12.1 Scraping one page	233
12.2 Cleaning the webscraped data	237
13 Functions	239
13.1 A simple function	240
13.2 Adding parameters	241
13.3 Making a function to scrape recipes	242
14 For loops	245
14.1 Basic for loops	245
14.2 Scraping multiple recipes	248
15 Scraping tables from PDFs	251
15.1 Scraping the first table	255
15.2 Making a function	262
16 More scraping tables from PDFs	267
16.1 Pregnant Women Incarcerated	279
16.2 Making PDF-scraped data available to others	288
17 Geocoding	293
17.1 Geocoding a single address	293
17.2 Making a function	298

17.3 Geocoding San Francisco marijuana dispensary locations	300
IV Project Management	307
18 <i>Mise en place</i>	309
18.1 Starting with a pencil and paper	309
18.1.1 Tables and graphs	311
18.2 R Projects	314
18.2.1 Folders	320
18.3 Modular R scripts	322
18.4 Modular code	324
18.4.1 Section Labels	325
18.4.2 Helper R Scripts	326
19 Collaboration	329
19.1 Code review	329
19.1.1 Style guidelines	331
19.2 Documentation	331
19.2.1 Comments	332
19.2.2 Vignettes	333
20 Tests	335
20.1 Why test your code?	335
20.2 Unit tests	338
20.2.1 Modular test scripts	340
20.2.2 How to write unit tests	343
20.2.3 What to test	347
20.3 Test-driven development (TDD)	349
21 Git	351
21.1 What is Git and why do I need it?	351
21.2 Git basics	353
21.3 When to commit	355
21.4 Code review	355

V Data	357
22 Introduction	359
23 Uniform Crime Report (UCR) - Offenses Known and Clearances by Arrest	361
23.1 Exploring the UCR data	362
23.2 ORIs - Unique agency identifiers	366
23.3 Hierarchy Rule	366
23.4 Which crimes are included?	367
23.4.1 Index Crimes	367
23.4.2 The problem with using index crimes	369
23.4.3 Rape definition change	370
23.5 Actual offenses, clearances, and unfounded offenses	370
23.5.1 Actual	370
23.5.2 Total Cleared	370
23.5.3 Cleared Where All Offenders Are Under 18	371
23.5.4 Unfounded	371
23.6 Number of months reported	371
Appendix	373
A Useful resources	373
A.1 Learning R and coding issues	373
A.2 Data	373

Welcome

This book introduces the programming language R and is meant for undergrads or graduate students studying criminology. R is a programming language that is well-suited to the type of work frequently done in criminology - taking messy data and turning it into useful information. While R is a useful tool for many fields of study, this book focuses on the skills criminologists should know and uses crime data for the example data sets.

For this book you should have the latest version of [R](#) installed and be running it through [RStudio Desktop \(The free version\)](#). We'll get into what R and RStudio are soon but please have them installed to be able to follow along with each chapter. I highly recommend following along with the code for each lesson and then try to use the lessons learned on a data set you are interested in. To download the data used in this book please see [here](#).

Why learn to program?

With the exception of some more advanced techniques like scraping data from websites or from PDFs, nearly everything we do here can be done through Excel, a software you're probably more familiar with. The basic steps for research projects are generally:

1. Open up a data set - which frequently comes as an Excel file!
2. Change some values - misspellings or too specific categories for our purposes are very common in crime data
3. Delete some values - such as states you won't be studying
4. Make some graphs
5. Calculate some values - such as number of crimes per year
6. Sometimes do a statistical analysis depending on the type of project

7. Write up what you find

R can do all of this but why should you want (or have) to learn an entirely new skill just to do something you can already do? R is useful for two main reasons: scale and reproducibility.

Scale

If you do a one-off project in your career such as downloading some data and making a graph out of it, it makes sense to stick with software like Excel. The cost (in time and effort) of learning R is certainly not worth it for a single (or even several) project - even one perfectly suited for using R. R (and many programming languages more generally, such as Python) has its strength in doing something fairly simple many times. For example, it may be quicker to download one file yourself than it is to write the code in R to download that file. But when it comes to downloading hundreds of files, writing the R code becomes very quickly the better option than doing it by hand.

For most tasks you do in criminology when dealing with data you will end up them doing many times (including doing the same task in future projects). So R offers the trade-off of spending time upfront by learning the code with the benefit of that code being able to do work at a large scale with little extra work from you. Please keep in mind this trade-off - you need to front-load the costs of learning R for the rewards of making your life easier when dealing with data - when feeling discouraged about the small returns you get early in learning R.

Reproducibility

The second major benefit of using R over something like Excel is that R is reproducible. Every action you take is written down. This is useful when collaborating with others (including your future self) as they can look at your code and follow along what you did without you having to show them every click you made as you frequently would on Excel. Your collaborator can look at your code to help you figure out a bug in the code or to add their own code to yours.

In the research context specifically, you want to have code to give to people to ensure that your research was done correctly and there aren't bugs in the

code. Additionally, if you build a tool to, for example, interpret raw crime data from an agency and turn it into a map, being able to share the code so others can modify it for their own city saves these people a lot of time and effort.

What you will learn

For many of the lessons we will be working through real research questions and working from start to finish as you would on your own project. This involves thinking about what you want to accomplish from the data you have and what steps you need to take to reach that goal. This involves more than just knowing what code to write - it includes figuring out what your data has, whether it can answer the question you're asking, and planning out (without writing any code yet) what you need to do when you start coding.

Skills

There is a large range of skills in criminological research - far too large to cover in a single book. Here we will attempt to teach fundamental skills to build a solid foundation for future work. We'll be focusing on the following skills and trying to reinforce our skills with each lesson.

- Subsetting - Taking only certain rows or columns from a data set
- Graphing
- Regular expressions - Essentially R's "Find and Replace" function for text
- Getting data from websites (webscraping)
- Getting data from PDFs
- Mapping
- Writing documents through R

Data

Criminology has a large - and growing - number of data sets publicly available for us to use. In this book we will focus on a few prominent ones including the following:

- Uniform Crime Report (UCR) - A FBI data set with agency-level crime data for nearly every agency in the United States

We'll also cover a number of other data sets such as local police data and government data on alcohol consumption in the United States.

What you won't learn

This book is not a statistics book so we will not be covering any statistical techniques. Though some data sets we handle are fairly large, this book does not discuss how to deal with Big Data. While the lessons you learn in this book can apply to larger data sets, Big Data (which I tend to define loosely as data that are too large for my computer to handle) requires special skills that are outside the realm of this book. If you do intend to deal with huge data sets I recommend you look at the R package [data.table](#) which is an excellent resource for it. While we briefly cover mapping, this book will not cover working with geographic data in detail. For a comprehensive look at geographic data please see [this book](#).

Simple vs Easy

In the course of this book we will cover things that are very simple. For example, we'll take a data set (think of it like an Excel file) with crime for nearly every agency in the United States and keep only data from Colorado for a small number of years. We'll then find out how many murders happened in Colorado each year. This is a fairly simple task - it can be expressed in two sentences. You'll find that most of what you do is simple like this - it is quick to talk about what you are doing and the concepts are not complicated. What it isn't is easy. To actually write the R code to do this takes knowing a number of interrelated concepts in R and several lines of code to implement each step.

While this distinction may seem minor, I think it is important for newer programmers to understand that what they are doing may be simple to talk about but hard to implement. It is easy to feel like a bad programmer because something that can be articulated in 10 seconds may take hours to do. So during times when you are working with R try to keep in mind that even

though a project may be simple to articulate, it may be hard to code and that there is often very little correlation between the two.

How to Contribute

If you have any questions, suggestions, or find any issues please email me at jacob@crimedatatool.com. If this book has helped you, also email me so I can try to measure the book's impact and who is using it.

About the author

Jacob Kaplan is a Postdoctoral Fellow at the University of Pennsylvania. He holds a PhD and a master's degree in criminology from the University of Pennsylvania and a bachelor's degree in criminal justice from California State University, Sacramento. His research focuses on Crime Prevention Through Environmental Design (CPTED), specifically on the effect of outdoor lighting on crime. He is the author of several R packages that make it easier to work with data, including [fastDummies](#) and [asciiSetupReader](#). His [website](#) allows easy analysis of crime-related data and he has released over a [dozen crime data sets](#) (primarily FBI UCR data) on openICPSR that he has compiled, cleaned, and made available to the public. He is currently on the job market.

For a list of papers he has written (including working papers), please see [here](#).

For a list of data sets he has cleaned, aggregated, and made public, please see [here](#).

Chapter 1

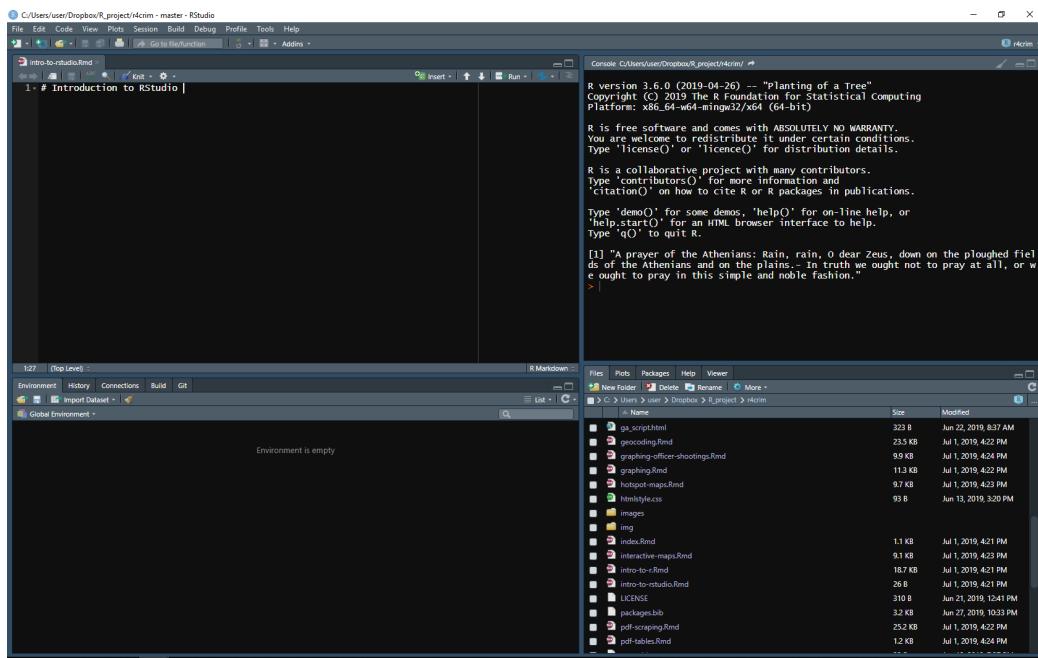
Introduction to R and RStudio

1.1 Using RStudio

In this lesson we'll start by looking at RStudio then write some brief code to load in some crime data and start exploring it. This lesson will cover code that you won't understand completely yet. That is fine, we'll cover everything in more detail as the lessons progress.

RStudio is the interface we use to work with R. It has a number of features to make it easier for us to work with R - while not strictly necessary to use, most people who use R do so through RStudio. We'll spend some time right now looking at RStudio and the options you can change to make it easier to use (and to suit your personal preferences with appearance) as this will make all of the work that we do in this book easier.

When you open up RStudio you'll see four panels, each of which plays an important role in RStudio. Your RStudio may not look like the setup I have in the image below - that is fine, we'll learn how to change the appearance of RStudio soon.



At the top right is the Console panel. Here you can write code, hit enter/return, and R will run that code. If you write `2+2` it will return (in this case that just mean it will print an answer) 4. This is useful for doing something simple like using R as a calculator or quickly looking at data. In most cases during research this is where you'd do something that you don't care to keep. This is because when you restart R it won't save anything written in the Console. To do reproducible research or to be able to collaborate with others you need a way to keep the code you've written.

The way to keep the code you've written in a file that you can open later or share with someone else is by writing code in an R Script (if you're familiar with Stata, an R Script is just like a .do file). An R Script is essentially a text file (similar to a Word document) where you write code. To run code in an R Script just click on a line of code or highlight several lines and hit enter/return or click the “Run” button on the top right of the Source panel. You'll see the lines of code run in the Console and any output (if your code has an output) will be shown there too (making a plot will be shown in a different panel as we'll see soon).

For code that you don't want to run, called comments, start the line with a pound sign `#` and that line will not be run (it will still print in the console

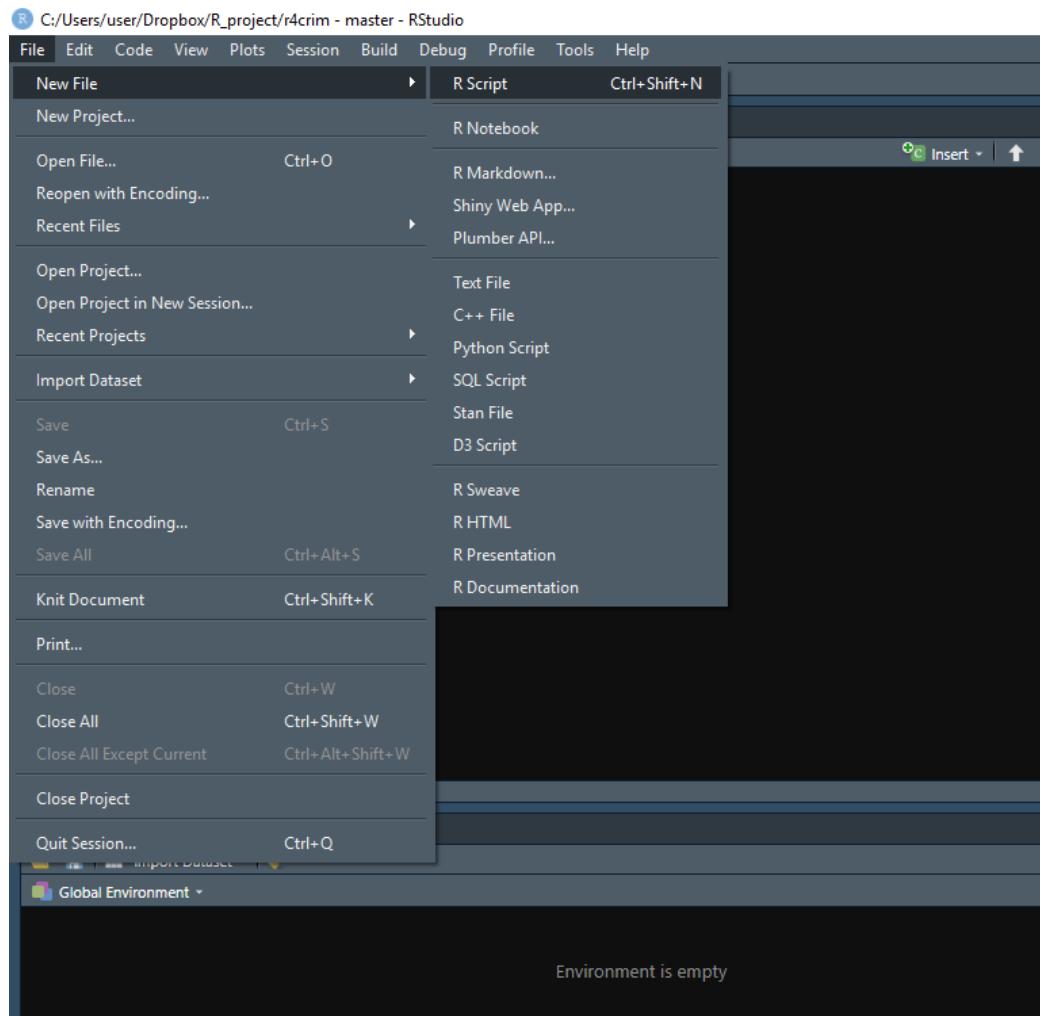
if you run it but it won't do anything). These comments should explain the code you wrote (if not otherwise obvious).

The Source panel is where the R Scripts will be and is located at the top left on the image below. It is good practice to do all of your code writing in an R Script - even if you delete some lines of code later - as it eliminates the possibility of losing code or forgetting what you wrote. Having all the code in front of you in a text file also makes it easier to understand the flow of code from start to finish to a task - an issue we'll discuss more in later lessons.

While the Source and Console panels are the ones that are of most use, there are two other panels worth discussing. As these two panels let you interchange which tabs are available in them, we'll return to them shortly in the discussion of the options RStudio has to customize it.

1.1.1 Opening an R Script

When you want to open up a new R Script you can click File on the very top left, then R Script. It will open up the script in a new tab inside of the Source panel. There are also a number of other file options available: R Presentation which can make PowerPoints, R Markdown which can make Word Documents or PDFs that incorporate R code used to make tables or graphs (and which we'll cover in Chapter 11), and Shiny Web App to make websites using R. There is too much to cover for an introductory book such as this but keep in mind the wide capabilities of R if you have another task to do.

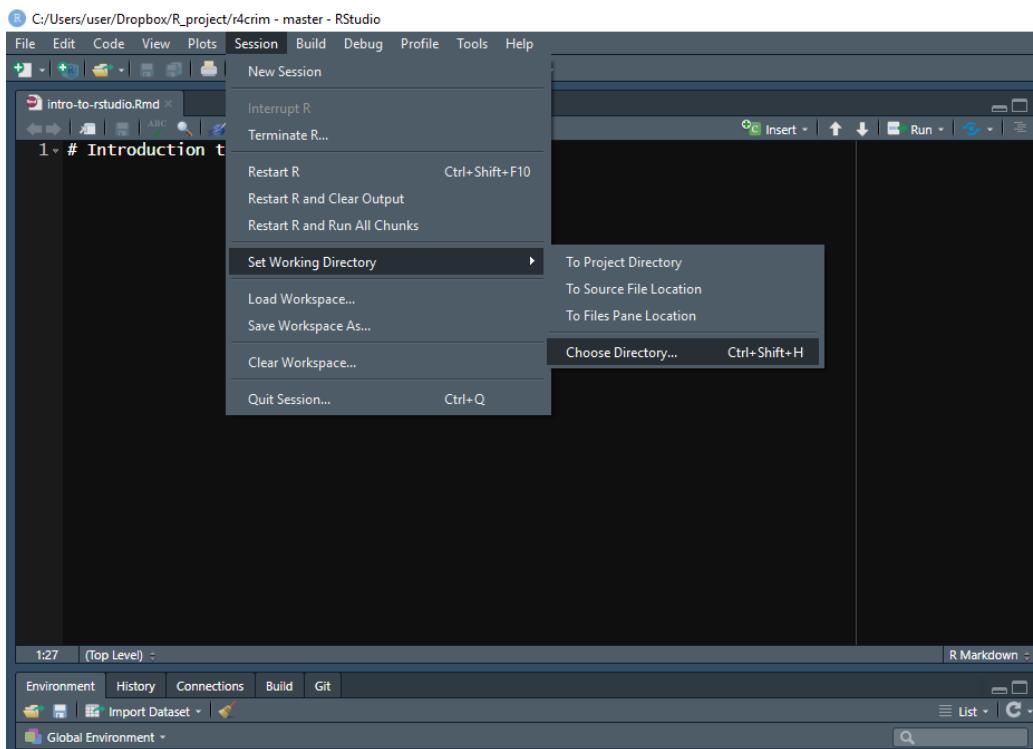


1.1.2 Setting the working directory

Many research projects incorporate data that someone else (such as the FBI or a local police agency) has put together. In these cases, we need to load the data into R to be able to use it. In a little bit we'll load a data set into R and start working on it but let's take a step back now and think about how to even load data. First, we'll need to get the data onto our computer somehow, probably by downloading it from an agency's website. Let's be specific - we don't download it to our computer, we download it to a specific folder on our computer (usually defaulted to the Downloads

folder on a Windows machine). So let's say you wanted to load a file called "data" into R. If you have a file called "data" in both your Desktop and your Downloads folder, R wouldn't know which one you wanted. And unless your data was in the folder R searches by default (which may not be where the file is downloaded by default), R won't know which file to load.

We need to tell R explicitly which folder has the data to load. We do this by setting the "Working Directory" (or the "Folders where I want you, R, to look for my data" in more simple terms). To set a working directory in R click the Session tab on the top menu, scroll to Set Working Directory, then click Choose Directory. This will open a window where you can navigate to the folder you want.

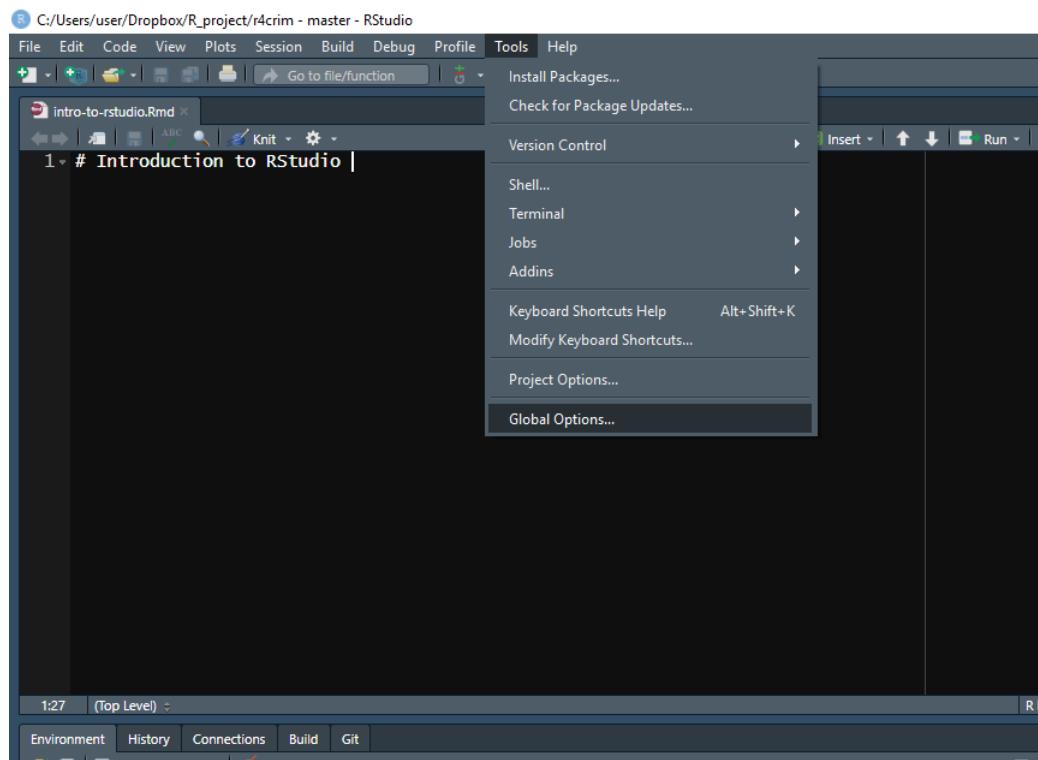


After clicking Open in that window you'll see a new line of code in the Console starting with `setwd()` and inside of the parentheses is the route your computer takes to get to the folder you selected. And now R knows which folder to look in for the data you want. It is good form to start your R Script with `setwd()` to make sure you can load the data. Copy the line of

code that says `setwd()` (which stands for “set working directory”), including everything in the parentheses, to your R Script when you start working.

1.1.3 Changing RStudio

Your RStudio looks different than my RStudio because I changed a number of settings to suit my preferences. To do so yourself click the Tools tab on the top menu and then click Global Options.



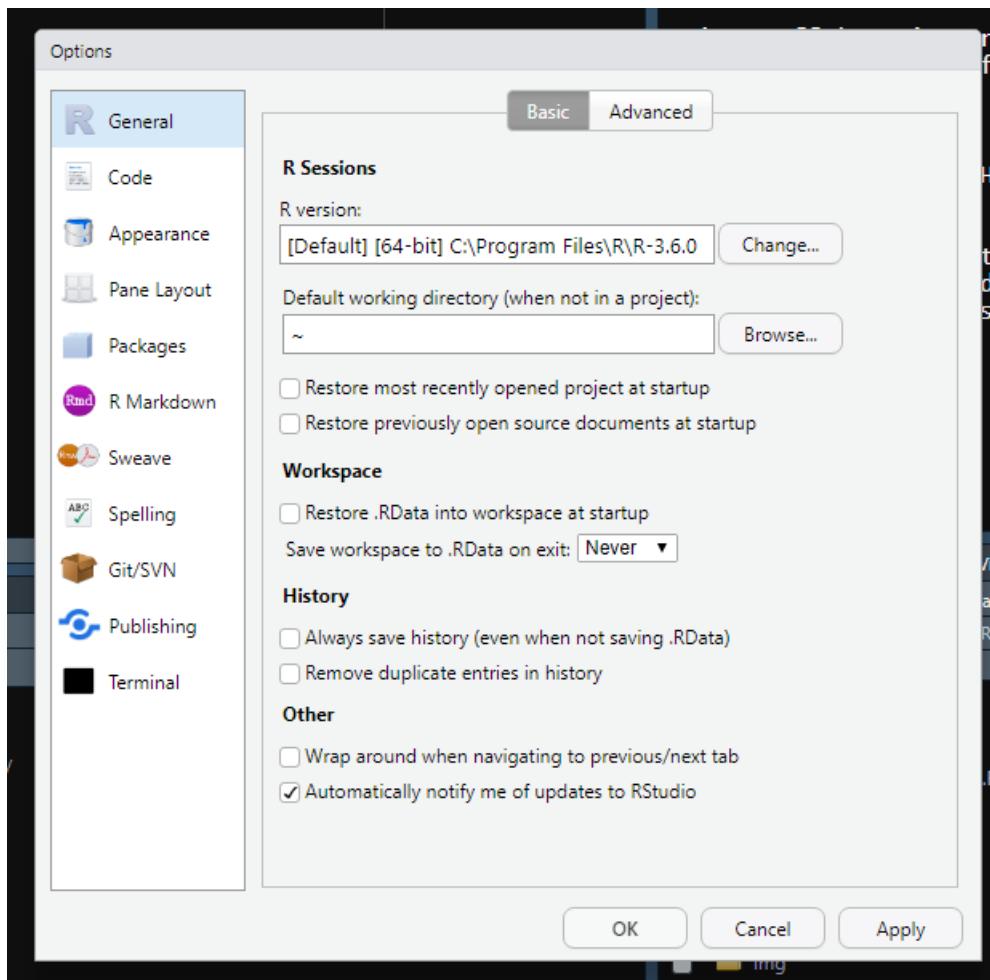
This opens up a window with a number of different tabs to change how R behaves and how it looks.

1.1.3.1 General

Under Workspace in the General tab make sure to **uncheck** the “Restore .RData into workspace at startup” and to set “Save workspace to .RData on exit:” to Never. What this does is make sure that every time you open R it starts fresh with no objects (essentially data loaded into R or made in R)

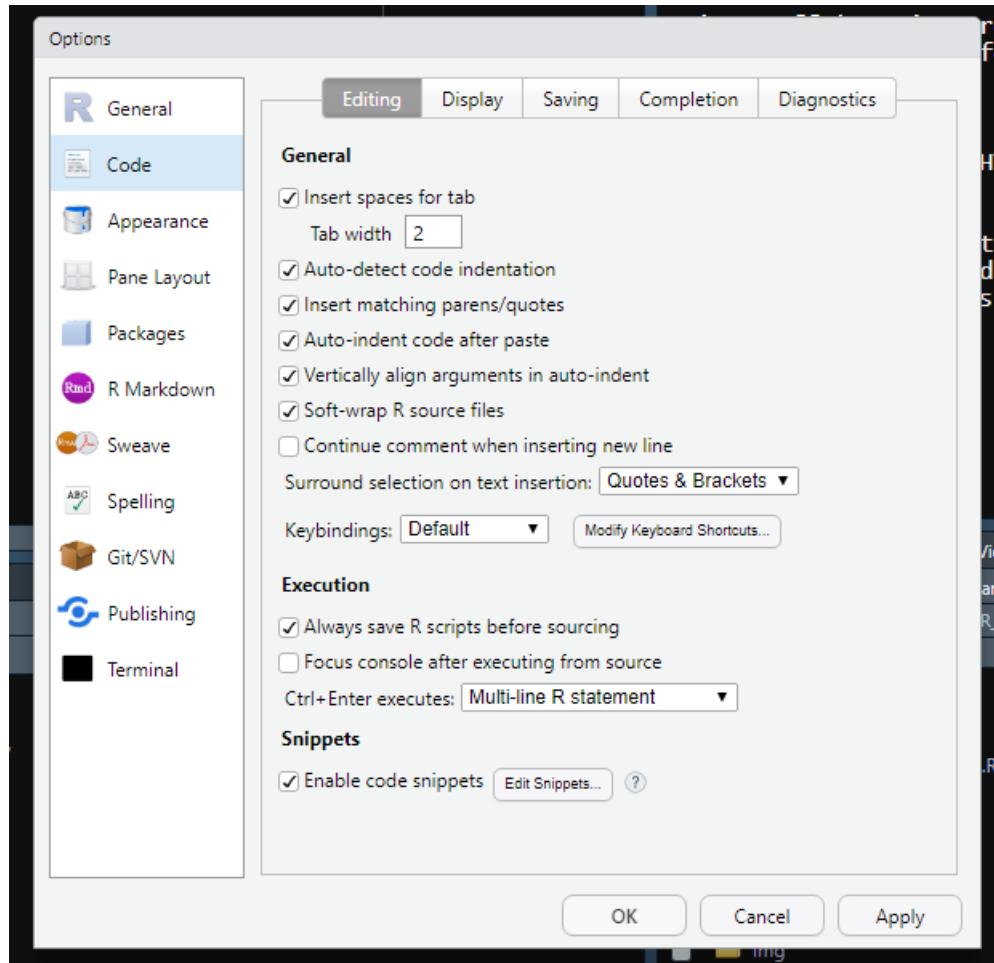
from previous sessions. This may be annoying at times, especially when it comes to loading large files, but the benefits far outweigh the costs.

You want your code to run from start to finish without any errors. Something I've seen many students do is write some code in the Console (or in their R Script but out of order of how it should be run) to fix an issue with the data. This means their data is how it should be but when the R session restarts (such as if the computer restarts) they won't be able to get back to that point. Making sure your code handles everything from start to finish is well-worth the avoided headache of trying to remember what code you did to fix the issue previously.



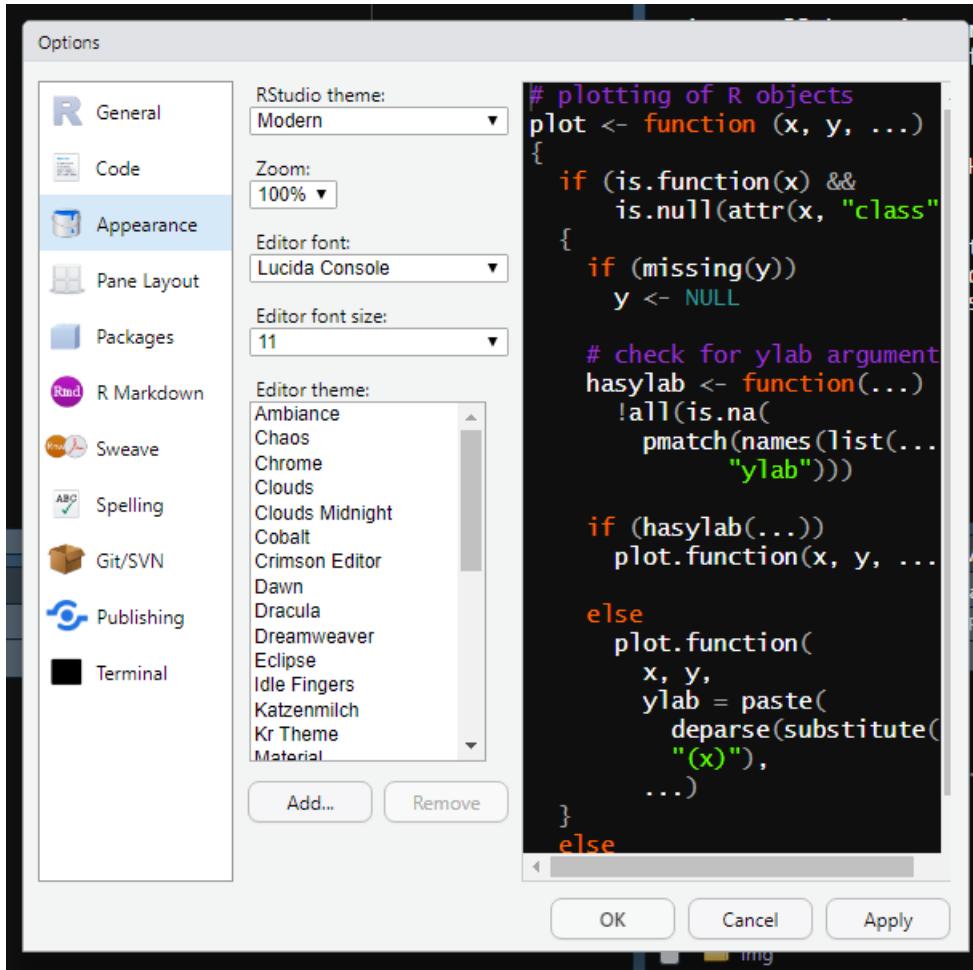
1.1.3.2 Code

The Code tab lets you specify how you want the code to be displayed. The important section for us is to make sure to check the “Soft-wrap R source files” check-box. If you write a very long line of code it gets too big to view all at once and you must scroll to the right to read it all. That can be annoying as you won’t be able to see all the code at once. Setting “Soft-wrap” makes it so if a line is too long it will just be shown on multiple lines which solves that issue. In practice it is best to avoid long lines of codes as it makes it hard to read but that isn’t always possible.



1.1.3.3 Appearance

The Appearance tab lets you change the background, color, and size of text. Change it to your preferences.

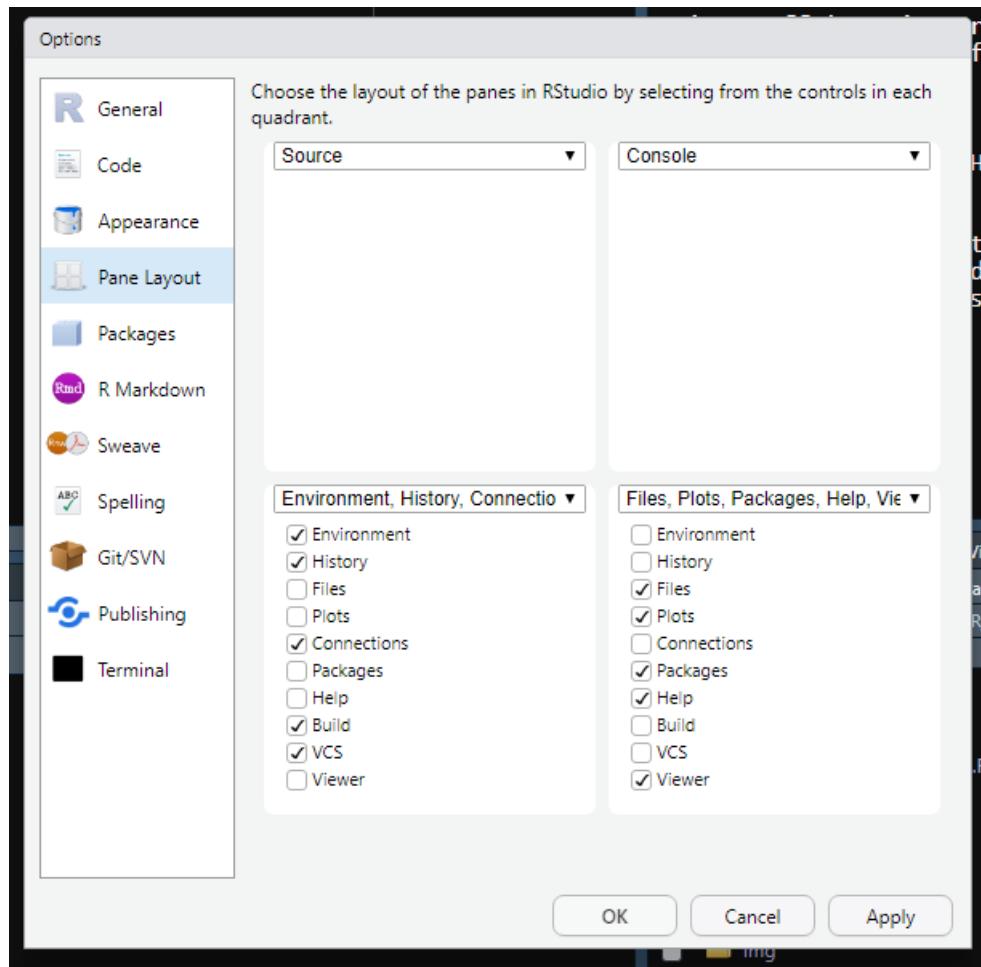


1.1.3.4 Pane Layout

The final tab we'll look at is Pane Layout. This lets you move around the Source, Console, and the other two panels. There are a number of different tabs to select for the panels (unchecked one just moves it to the other panel, it doesn't remove it from RStudio) and we'll talk about three of them. The Environment tab shows every object you load into R or make in R. So if you load a file called "data" you can check the Environment tab. If it is there,

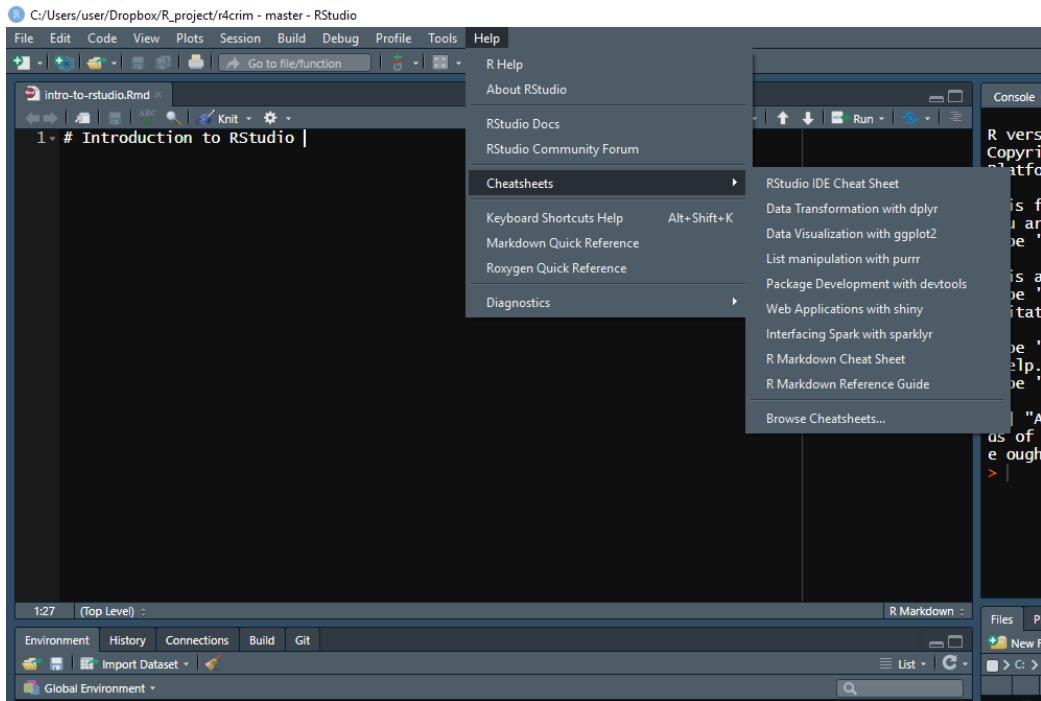
you have loaded the file correctly.

As we'll discuss more in Section 1.4, the Help tab will open up to show you a help page for a function you want more information on. The Plots tab will display any plot you make. It also keeps all plots you've made (until restarting R) so you can scroll through the plots.



1.1.4 Helpful Cheat Sheets

RStudio also includes a number of links to helpful cheat sheets for a few important topics. To get to it click Help, then Cheatsheets and click on whichever one you need.



1.2 Reading data into R

For many research projects you'll have data produced by some outside group (FBI, local police agencies) and you want to take that data and put it inside R to work on it. We call that reading data into R. R is capable of reading a number of different formats of data which we will discuss in more detail in Chapter 5. Here, we will talk about the standard R data file only.

1.2.1 Loading data

As we learned above in Section 1.1.2, we need to set our working directory to the folder where the data is. For my own setup, R is already defaulted to the folder with this data so I do not need to set a working directory. For those following along on your own computer, make sure to set your working directory now.

The `load()` function lets us load data already in the R format. These files will end in the extension “.rda” or sometimes “.Rda” or “.RData”. Since we

are telling R to load a specific file, we need to have that file name in quotes and include the file extension “.rda”. With R data, the object inside the data already has a name so we don’t need to assign (something we will discuss in detail in Section 2.2) a name to the data. With other forms of data such as .csv files we will need to do that as we’ll see in Chapter 5.

```
load("data/ucr2017.rda")
```

1.3 First steps to exploring data

The object we loaded is called `ucr2017`. We’ll explore this data more thoroughly in the Chapter 3 but for now let’s use four simple (and important) functions to get a sense of what the data holds. For each of these functions write the name of the data set (without quotes since we don’t need quotes for an object already made in R) inside the `()`.

- `head()`
- `summary()`
- `plot()`
- `View()`

Note that the first three functions are lowercase while `View()` is capitalized. That is simply because older functions in R were often capitalized while newer ones use all lowercase letters. R is case sensitive so using `view()` will not work.

The `head()` function prints the first 6 rows of each column of the data to the console. This is useful to get a quick glance at the data but has some important drawbacks. When using data with a large number of columns it can be quickly overwhelming by printing too much. There may also be differences in the first 6 rows with other rows. For example, if the rows are ordered chronologically (as is the case with most crime data) the first 6 rows will be the most recent. If data collection methods or the quality of collection changed over time, these 6 rows won’t be representative of the data.

```
head(ucr2017)
#>      ori year agency_name state population actual_murder actual_rape_to
```

#>	ori	year	agency_name	state	population	actual_murder	actual_rape_to
#> 1	AK00101	2017	anchorage	alaska	296188		27
#> 2	AK00102	2017	fairbanks	alaska	32937		10

```
#> 3 AK00103 2017      juneau alaska      32344      1      50
#> 4 AK00104 2017      ketchikan alaska    8230       1      19
#> 5 AK00105 2017      kodiak alaska     6198       0      15
#> 6 AK00106 2017      nome alaska      3829       0       7
#>   actual_robbery_total actual_assault_aggravated
#> 1                      778                  2368
#> 2                      40                   131
#> 3                      46                   206
#> 4                      0                    14
#> 5                      4                    41
#> 6                      0                   52
```

The `summary()` function gives a six number summary of each numeric or Date column in the data. For other types of data, such as “character” types (which are just columns with words rather than numbers or dates), it’ll say what type of data it is.

The six values it returns for numeric and Date columns are

- The minimum value
- The value at the 1st quartile
- The median value
- The mean value
- The value at the 3rd quartile
- The max value
- In cases where there are NAs, it will say how many NAs there are. An NA value is a missing value. Think of it like an empty cell in an Excel file. NA values will cause issues when doing math such as finding the mean of a column as R doesn’t know how to handle a NA value in these situations. We’ll learn how to deal with this later.

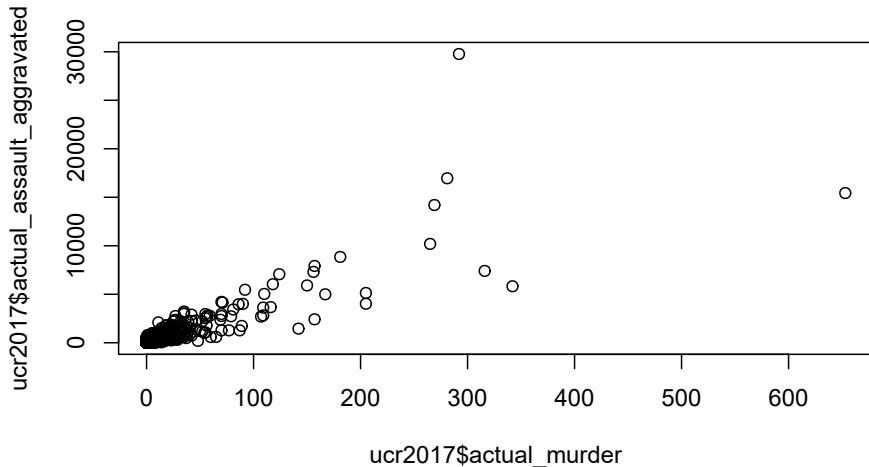
```
summary(ucr2017)
#>      ori            year      agency_name      state
#> Length:15764      Min.   :2017  Length:15764      Length:15764
#> Class :character  1st Qu.:2017  Class :character  Class :character
#> Mode  :character  Median :2017  Mode  :character  Mode  :character
#>                  Mean   :2017
#>                  3rd Qu.:2017
#>                  Max.   :2017
```

```
#>   population      actual_murder    actual_rape_total actual_robbery_to
#> Min. :     0  Min. : 0.000  Min. : -2.000  Min. : -1.00
#> 1st Qu.:  914  1st Qu.: 0.000  1st Qu.: 0.000  1st Qu.: 0.00
#> Median : 4460  Median : 0.000  Median : 1.000  Median : 0.00
#> Mean   : 19872  Mean   : 1.069  Mean   : 8.262  Mean   : 19.85
#> 3rd Qu.: 15390  3rd Qu.: 0.000  3rd Qu.: 5.000  3rd Qu.: 4.00
#> Max.  :8616333  Max.  :653.000  Max.  :2455.000  Max.  :13995.00
#>   actual_assault_aggravated
#> Min. : -1.00
#> 1st Qu.: 1.00
#> Median : 5.00
#> Mean   : 49.98
#> 3rd Qu.: 21.00
#> Max.  :29771.00
```

The `plot()` function allows us to graph our data. For criminology research we generally want to make scatterplots to show the relationship between two numeric variables, time-series graphs to see how a variable (or variables) change over time, or barplots comparing categorical variables. Here we'll make a scatterplot seeing the relationship between a city's number of murders and their number of aggravated assaults (assault with a weapon or that causes serious bodily injury).

To do so we must specify which column is displayed on the x-axis and which one is displayed on the y-axis. In Section 2.5.1 we'll talk explicitly about how to select specific columns from our data. For now, all you need to know is to select a column you write the data set name followed by dollar sign \$ followed by the column name. Do not include any quotations or spaces (technically spaces can be included but make it a bit harder to read and are against conventional style when writing R code so we'll exclude them). Inside of `plot()` we say that "x = ucr2017\$actual_murder" so that column goes on the x-axis and "y = ucr2017\$actual_assault_aggravated" so aggravated assault goes on the y-axis. And that's all it takes to make a simple graph.

```
plot(x = ucr2017$actual_murder, y = ucr2017$actual_assault_aggravated)
```



Finally, `View()` opens essentially an Excel file of the data set you put inside the `()`. This allows you to look at the data as if it were in Excel and is a good way to start to understand the data.

```
View(ucr2017)
```

1.4 Finding help about functions

If you are having trouble understanding what a function does or how to use it, you can ask R for help and it will open up a page explaining what the function does, what options it has, and examples of how to use it. To do so we write `help(function)` or `?function` in the console and it will open up that function's help page.

If we wrote `help(plot)` to figure out what the `plot()` function does, it will open up this page. For finding the help page of a function the parentheses (e.g. `plot()`) are optional.

The screenshot shows the R Documentation Viewer interface. The top navigation bar includes 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the bar, there are icons for back, forward, search, and refresh. The main content area has a title 'R: Generic X-Y Plotting' and a subtitle 'plot {graphics}'. A 'Find in Topic' search bar is present. The page content starts with a 'Description' section, followed by 'Usage' (with code example `plot(x, y, ...)`) and 'Arguments' sections. The 'Arguments' section details the 'x', 'y', and '...' parameters, and provides extensive information about the 'type' argument, including a list of valid types: "p", "l", "b", "c", "o", "n", "h", "s", and "S". It also notes that other types like "punkte" give a warning or error. Below this, there are descriptions for 'main', 'sub', 'xlab', and 'ylab' arguments.

```

plot {graphics}
Generic X-Y Plotting

Description
Generic function for plotting of R objects. For more details about the graphical parameter arguments, see par.
For simple scatter plots, plot.default will be used. However, there are plot methods for many R objects, including functions, data.frames, density objects, etc. Use methods\(plot\) and the documentation for these.

Usage
plot(x, y, ...)

Arguments
x      the coordinates of points in the plot. Alternatively, a single plotting structure, function or any R object with a plot method can be provided.
y      the y coordinates of points in the plot, optional if x is an appropriate structure.
...    Arguments to be passed to methods, such as graphical parameters (see par). Many methods will accept the following arguments:
      type
          what type of plot should be drawn. Possible types are
          • "p" for points,
          • "l" for lines,
          • "b" for both,
          • "c" for the lines part alone of "b",
          • "o" for both 'overplotted',
          • "n" for 'histogram' like (or 'high-density') vertical lines,
          • "s" for stair steps,
          • "S" for other steps, see 'Details' below,
          • "h" for no plotting.

          All other types give a warning or an error; using, e.g., type = "punkte" being equivalent to type = "p" for S compatibility. Note that some methods, e.g. plot.factor, do not accept this.

      main
          an overall title for the plot: see title.
      sub
          a sub title for the plot: see title.
      xlab
          a title for the x axis: see title.
      ylab
          a title for the y axis: see title.

```

Part I

Clean

Chapter 2

Subsetting: Making big things small

Subsetting data is a way to take a large data set and reduce it to a smaller one that is better suited for answering a specific question. This is useful when you have a lot of data in the data set that isn't relevant to your research - for example, if you are studying crime in Colorado and have every state in your data, you'd subset it to keep only the Colorado data. Reducing it to a smaller data set makes it easier to manage, both in understanding your data and avoiding have a huge file that could slow down R.

2.1 Select specific values

```
animals <- c("cat", "dog", "gorilla", "buffalo", "lion", "snake")  
animals  
#> [1] "cat"      "dog"      "gorilla"   "buffalo"  "lion"     "snake"
```

Here we have made an object called *animals* with a number of different animals in it (we'll explain what it really means to "make an object" soon). In R, we will use square brackets [] to select specific values in that object, something called "indexing". Put a number (or numbers) in the square bracket and it will return the value at that "index". The index is just the place number where each value is. "cat" is the first value in *animals* so it is at

the first index, “dog” is the second value so it is the second index or index 2. “snake” is our last value and is the 6th value in *animals* so it is index 6 (some languages use “zero indexing” which means the first index is index 0, the second is index 1. So in our example “cat” would be index 0. R does not do that and the first value is index 1, the second is index 2 and so on.).

The syntax (how the code is written) goes

```
object[index]
```

First, we have the object and then we put the square bracket []. We need both the object and the [] for subsetting to work. Let’s say we wanted to choose just the “snake” from our *animals* object. In normal language we say “I want the 6th value from *animals*. We say where we’re looking and which value we want.

```
animals[6]
#> [1] "snake"
```

Now let’s get the third value.

```
animals[3]
#> [1] "gorilla"
```

If we want multiple values, we can enter multiple numbers. If you have multiple values, you need to make a vector using `c()` and put the numbers inside the parentheses separated by a comma. We’ll learn more about vectors and using `c()` in Section 2.3 shortly. If we wanted values 1-3, we could use `c(1, 2, 3)`, with each number separated by a comma.

```
animals[c(1, 2, 3)]
#> [1] "cat"      "dog"      "gorilla"
```

When making a vector of sequential integers, instead of writing them all out manually we can use `first_number:last_number` like so

```
1:3
#> [1] 1 2 3
```

To use it in subsetting we can treat `1:3` as if we wrote `c(1, 2, 3)`.

```
animals[1:3]
#> [1] "cat"      "dog"      "gorilla"
```

The order we enter the numbers determines the order of the values it returns. Let’s get the third index, the fourth index, and the first index, in that order.

```
animals[c(3, 4, 1)]
#> [1] "gorilla" "buffalo" "cat"
```

Putting a negative number inside the [] will return all values **except** for that index, essentially deleting it. Let’s remove “cat” from *animals*. Since it is the 1st item in *animals*, we can remove it like this

```
animals[-1]
#> [1] "dog"      "gorilla"   "buffalo"   "lion"      "snake"
```

Now let’s remove multiple values, the first 3.

```
animals[-c(1, 2, 3)]
#> [1] "buffalo" "lion"     "snake"
```

2.2 Assigning values to objects (Making “things”)

Earlier we wrote `animals <- c("cat", "dog", "gorilla", "buffalo", "lion", "snake")` to make the object *animals* with the value of each of the different animals we wrote.

We say `<-` as “gets”. So above “*animals* gets the values cat, dog, etc.”. This is read from left to right as thing on left (the name of the object) “gets” the value of the thing on the right of the `<-`. The proper terminology is that the “thing” on the left is an “object”. So if we had `x <- 5` the object *x* gets the value 5. We could also say “five was assigned to *x*”.

The terminology is “object gets value” or “value assigned to object”, both work.

You can use the `=` instead of `<-`. Again, the thing on the left gets the value of the thing on the right even when using `=`.

```
x = 2
```

x now has the value of the number 2.

```
x
#> [1] 2
```

It is the convention in R to use `<-` instead of `=` and in some cases the `=` will not work properly. For those reasons we will use `<-` for this class.

Earlier I said we can remove values with using a negative number and that index will be removed from the object. For example, `animals[-1]` prints every value in *animals* except for the first value.

```
animals[-1]
#> [1] "dog"      "gorilla"   "buffalo"   "lion"      "snake"
```

However, it doesn't actually remove anything from *animals*. Let's print *animals* and see which values it returns.

```
animals
#> [1] "cat"      "dog"       "gorilla"   "buffalo"   "lion"      "snake"
```

Now the first value, “cats”, is back. Why? To make changes in R you need to tell R very explicitly that you are making the change. If you don't save the result of your code (by assigning an object to it), R will run that code and simply print the results in the console panel without making any changes.

This is an important point that a lot of students struggle with. R doesn't know when you want to save (in this context I am referring to creating or updating an object that is entirely in R, not saving a file to your computer) a value or update an object. If *x* is an object with a value of 2, and you write `x + 2`, it would print out 4 because $2 + 2 = 4$. But that won't change the value of *x*. *x* will remain as 2 until you explicitly tell R to change its value. If you want to update *x* you need to run `x <- somevalue` where “somevalue” is whatever you want to change *x* to.

So to return to our *animals* example, if we wanted to delete the first value and keep it removed, we'd need to write `animals <- animals[-1]`. Which is essentially making a new object, also called *animals* (to avoid having many, slightly different objects that are hard to keep track of we'll reuse the name) with the same values as the original *animals* except this time excluding the first value, “cats”.

2.3 Vectors (collections of “things”)

When we made `x`, we wrote `x <- 2` while when we made `animals`, we wrote `animals <- c("cat", "dog", "gorilla", "buffalo", "lion", "snake")`. The important difference is that when assigning multiple values to an object we must use the function `c()` which combines them together. With multiple values we follow the same pattern of `object <- value` but put the value inside of `c()` and separate each value by a comma.

```
x <- c(1, 2, 3)
```

The result of the `c()` is called a vector and you can think of it as a collection of values.

Note that vectors take values that are the same type, so all values included must be the same type such as a number or a string (a character type such as words or values with letters. In R they are put in quotes.). If they aren’t the same type R will automatically convert it.

```
c("cat", "dog", 2)
#> [1] "cat" "dog" "2"
```

Above we made a vector with the values “cat”, “dog” and 2 (without quotes) and it added quotes to the 2. Since everything must be the same type, R automatically converted the 2 to a string of “2”.

2.4 Logical values and operations

We also frequently want to conditionally select certain values. Earlier we selected values indexing specific numbers, but that requires us to know exactly which values we want. We can conditionally select values by having some conditional statement (e.g. “this value is lower than the number 100”) and keeping only values where that condition is true. When we talk about logical values, we mean TRUE and FALSE - in R you must spell it in all capital letters.

First, we will discuss conditionals abstractly and then we will use a real example using data from the FBI to make a data set tailored to answer a specific question.

We can use these TRUE and FALSE values to index and it will return every element which we say is TRUE.

```
animals[c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE)]
#> [1] "cat" "dog"
```

This is the basis of conditional subsetting. If we have a large data set and only want a small chunk based on some condition (data in a single state (or multiple states), at a certain time, at least a certain population) we need to make a conditional statement that returns TRUE if it matches what we want and FALSE if it doesn't. There are a number of different ways to make conditional statements. First let's go through some special characters involved and then show examples of each one.

For each case you are asking: does the thing on the left of the conditional statement return TRUE or FALSE compared to the thing on the right.

- == Equals (compared to a single value)
- %in% Equals (one value match out of multiple comparisons)
- != Does not equal
- < Less than
- > Greater than
- <= Less than or equal to
- >= Greater than or equal to

Since many conditionals involve numbers (especially in criminology), let's make a new object called *numbers* with the numbers 1-10.

```
numbers <- 1:10
```

2.4.1 Matching a single value

The conditional == asks if the thing on the left equals the thing on the right. Note that it uses two equal signs. If we used only one equal sign it would assign the thing on the left the value of the thing on the right (as if we did <-).

```
2 == 2
#> [1] TRUE
```

This gives TRUE as we know that 2 does equal 2. If we change either value,

it would give us FALSE.

```
2 == 3
#> [1] FALSE
```

And it works when we have multiple numbers on the left side, such as our object called *numbers*.

```
numbers == 2
#> [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

This also works with characters such as the animals in the object we made earlier. “gorilla” is the third animal in our object, so if we check *animals* == “gorilla” we expect the third value to be TRUE and all others to be FALSE. Make sure that the match is spelled correctly (including capitalization) and is in quotes.

```
animals == "gorilla"
#> [1] FALSE FALSE TRUE FALSE FALSE FALSE
```

The == only works when there is one thing on the right hand side. In criminology we often want to know if there is a match for multiple things - is the crime one of the following crimes..., did the crime happen in one of these months..., is the victim a member of these demographic groups...? So we need a way to check if a value is one of many values.

2.4.2 Matching multiple values

The R operator %in% asks each value on the left whether or not it is a member of the set on the right. It asks, is the single value on the left hand side (even when there are multiple values such as our *animals* object, it goes through them one at a time) a match with any of the values on the right hand side? It only has to match with one of the right hand side values to be a match.

```
2 %in% c(1, 2, 3)
#> [1] TRUE
```

For our *animals* object, if we check if they are in the vector `c("cat", "dog", "gorilla")`, now all three of those animals will return TRUE.

```
animals %in% c("cat", "dog", "gorilla")
#> [1] TRUE TRUE TRUE FALSE FALSE FALSE
```

2.4.3 Does not match

Sometimes it is easier to ask what is not a match. For example, if you wanted to get every month except January, instead of writing the other 11 months, you just ask for any month that does not equal “January”.

We can use `!=`, which means “not equal”. When we wanted an exact match, we used `==`, if we want a not match, we can use `!=` (this time it is only a single equals sign).

```
2 != 3
#> [1] TRUE

"cat" != "gorilla"
#> [1] TRUE
```

Note that for matching multiple values with `%in%`, we cannot write `!%in%` but have to put the `!` before the values on the left.

```
!animals %in% c("cat", "dog", "gorilla")
#> [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

2.4.4 Greater than or less than

We can use R to compare values using greater than or less than symbols. We can also express “greater than or equal to” or “less than or equal to.”

```
6 > 5
#> [1] TRUE

6 < 5
#> [1] FALSE

6 >= 5
#> [1] TRUE

5 <= 5
#> [1] TRUE
```

When used on our object `numbers` it will return 10 values (since `numbers` is 10 elements long) with a `TRUE` if the condition is true for the element and `FALSE` otherwise. Let’s run `numbers > 3`. We expect the first 3 values to be

FALSE as 1, 2, and 3 are not larger than 3.

```
numbers > 3
#> [1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

2.4.5 Combining conditional statements - or, and

In many cases when you are subsetting you will want to subset based on more than one condition. For example, let's say you have crime data from every state between 1960 and 2017. Your research question is “did Colorado’s marijuana legalization affect crime in the state?” In that case you want only data from Colorado. Since legalization began in January 2014, you wouldn’t need every year, only years some period of time before and after legalization to be able to measure its effect. So you would need to subset based on the state and the year.

To make conditional statements with multiple conditions we use | for “or” and & for “and”.

Condition 1 | Condition 2

```
2 == 3 | 2 > 1
#> [1] TRUE
```

As it sounds, when using | as long as at least one condition is true (we can include as many conditions as we like) it will return TRUE.

Condition 1 & Condition 2

```
2 == 3 & 2 > 1
#> [1] FALSE
```

For &, all of the conditions must be true. If even one condition is not true it will return FALSE.

2.5 Subsetting a data.frame

Earlier we were using a simple vector (collection of values). In this class - and in your own work - you will usually work on an entire data set. These generally come in the form called a “data.frame” which you can imagine as being like an Excel file with multiple rows and columns.

Let's load in data from the Uniform Crime Report (UCR), an FBI data set that we'll work on in a later lesson. This data has crime data every year from 1960-2017 and for nearly every agency in the country.

```
load("data/offenses_known_yearly_1960_2017.rda")
```

Let's peak at the first 6 rows and 6 columns using the square bracket notation [] for data.frames which we'll explain more below.

```
offenses_known_yearly_1960_2017[1:6, 1:6]
#>      ori      ori9 agency_name state state_abb year
#> 1 AK00101 AK0010100 anchorage alaska          AK 2017
#> 2 AK00101 AK0010100 anchorage alaska          AK 2016
#> 3 AK00101 AK0010100 anchorage alaska          AK 2015
#> 4 AK00101 AK0010100 anchorage alaska          AK 2014
#> 5 AK00101 AK0010100 anchorage alaska          AK 2013
#> 6 AK00101 AK0010100 anchorage alaska          AK 2012
```

The first 6 rows appear to be agency identification info for Anchorage, Alaska from 2017-2012. For good measure let's check how many rows and columns are in this data. This will give us some guidance on subsetting which we'll see below. `nrow()` gives us the number of rows and `ncol()` gives us the number of columns.

```
nrow(offenses_known_yearly_1960_2017)
#> [1] 959010
```

```
ncol(offenses_known_yearly_1960_2017)
#> [1] 159
```

This is a large file with 159 columns and nearly a million rows. Normally we wouldn't want to print out the names of all 159 columns but let's do this here as we want to know the variables available to subset.

```
names(offenses_known_yearly_1960_2017)
#> [1] "ori"                                "ori9"
#> [3] "agency_name"                         "state"
#> [5] "state_abb"                            "year"
#> [7] "number_of_months_reported"           "fips_state_code"
#> [9] "fips_county_code"                     "fips_state_county_code"
#> [11] "fips_place_code"                    "fips_state_place_code"
```

```

#> [13] "agency_type"
#> [15] "agency_subtype_2"
#> [17] "census_name"
#> [19] "population_group"
#> [21] "juvenile_age"
#> [23] "last_update"
#> [25] "followup_indication"
#> [27] "covered_by_ori"
#> [29] "date_of_last_update"
#> [31] "special_mailing_group"
#> [33] "first_line_of_mailing_address"
#> [35] "third_line_of_mailing_address"
#> [37] "officers_killed_by_felony"
#> [39] "officers_assaulted"
#> [41] "actual_manslaughter"
#> [43] "actual Rape by force"
#> [45] "actual Robbery total"
#> [47] "actual Robbery with a knife"
#> [49] "actual Robbery unarmed"
#> [51] "actual assault With a gun"
#> [53] "actual assault other weapon"
#> [55] "actual assault simple"
#> [57] "actual Burg force entry"
#> [59] "actual Burg attempted"
#> [61] "actual mtr veh theft total"
#> [63] "actual mtr veh theft truck"
#> [65] "actual all crimes"
#> [67] "actual index violent"
#> [69] "actual index total"
#> [71] "tot_clr_manslaughter"
#> [73] "tot_clr_Rape by force"
#> [75] "tot_clr_Robbery total"
#> [77] "tot_clr_Robbery With a knife"
#> [79] "tot_clr_Robbery unarmed"
#> [81] "tot_clr_assault With a gun"
#> [83] "tot_clr_assault other weapon"
#> [85] "tot_clr_assault simple"
#> [87] "tot_clr_burg total"
#> [89] "tot_clr_murder"
#> [91] "tot_clr_theft total"
#> [93] "tot_clr_theft car"
#> [95] "tot_clr_theft other"
#> [97] "tot_clr_assault aggravated"
#> [99] "tot_clr_index property"
#> [101] "tot_clr_murder"
#> [103] "tot_clr_Rape total"
#> [105] "tot_clr_Rape attempted"
#> [107] "tot_clr_Robbery With a gun"
#> [109] "tot_clr_Robbery other weapon"
#> [111] "tot_clr_assault total"
#> [113] "tot_clr_assault With a knife"
#> [115] "tot_clr_assault unarmed"
#> [117] "tot_clr_burg total"
#> [119] "crosswalk_agency_name"
#> [121] "population"
#> [123] "country_division"
#> [125] "core_city_indication"
#> [127] "fbi_field_office"
#> [129] "zip_code"
#> [131] "agency_count"
#> [133] "month_included_in"
#> [135] "special_mailing_address"
#> [137] "second_line_of_mailing_address"
#> [139] "fourth_line_of_mailing_address"
#> [141] "officers_killed_by_accident"
#> [143] "actual_murder"
#> [145] "actual_rape_total"
#> [147] "actual_rape_attempted"
#> [149] "actual_robbery_with_a_gun"
#> [151] "actual_robbery_other_weapon"
#> [153] "actual_assault_total"
#> [155] "actual_assault_with_a_knife"
#> [157] "actual_assault_unarmed"
#> [159] "actual_burg_total"
#> [161] "actual_burg_nonforce_entry"
#> [163] "actual_theft_total"
#> [165] "actual_mtr_veh_theft_car"
#> [167] "actual_mtr_veh_theft_other"
#> [169] "actual_assault_aggravated"
#> [171] "actual_index_property"
#> [173] "tot_clr_murder"
#> [175] "tot_clr_Rape total"
#> [177] "tot_clr_Rape attempted"
#> [179] "tot_clr_Robbery With a gun"
#> [181] "tot_clr_Robbery other weapon"
#> [183] "tot_clr_assault total"
#> [185] "tot_clr_assault With a knife"
#> [187] "tot_clr_assault unarmed"
#> [189] "tot_clr_burg total"

```

```

#> [87] "tot_clr_burg_force_entry"
#> [89] "tot_clr_burg_attempted"
#> [91] "tot_clr_mtr_veh_theft_total"
#> [93] "tot_clr_mtr_veh_theft_truck"
#> [95] "tot_clr_all_crimes"
#> [97] "tot_clr_index_violent"
#> [99] "tot_clr_index_total"
#> [101] "clr_18_manslaughter"
#> [103] "clr_18_rape_by_force"
#> [105] "clr_18_robbery_total"
#> [107] "clr_18_robbery_with_a_knife"
#> [109] "clr_18_robbery_unarmed"
#> [111] "clr_18_assault_with_a_gun"
#> [113] "clr_18_assault_other_weapon"
#> [115] "clr_18_assault_simple"
#> [117] "clr_18_burg_force_entry"
#> [119] "clr_18_burg_attempted"
#> [121] "clr_18_mtr_veh_theft_total"
#> [123] "clr_18_mtr_veh_theft_truck"
#> [125] "clr_18_all_crimes"
#> [127] "clr_18_index_violent"
#> [129] "clr_18_index_total"
#> [131] "unfound_manslaughter"
#> [133] "unfound_rape_by_force"
#> [135] "unfound_robbery_total"
#> [137] "unfound_robbery_with_a_knife"
#> [139] "unfound_robbery_unarmed"
#> [141] "unfound_assault_with_a_gun"
#> [143] "unfound_assault_other_weapon"
#> [145] "unfound_assault_simple"
#> [147] "unfound_burg_force_entry"
#> [149] "unfound_burg_attempted"
#> [151] "unfound_mtr_veh_theft_total"
#> [153] "unfound_mtr_veh_theft_truck"
#> [155] "unfound_all_crimes"
#> [157] "unfound_index_violent"
#> [159] "unfound_index_total"
"tot_clr_burg_nonforce_entry"
"tot_clr_theft_total"
"tot_clr_mtr_veh_theft_car"
"tot_clr_mtr_veh_theft_other"
"tot_clr_assault_aggravated"
"tot_clr_index_property"
"clr_18_murder"
"clr_18_rape_total"
"clr_18_rape_attempted"
"clr_18_robbery_with_a_gun"
"clr_18_robbery_other_weapon"
"clr_18_assault_total"
"clr_18_assault_with_a_knife"
"clr_18_assault_unarmed"
"clr_18_burg_total"
"clr_18_burg_nonforce_entry"
"clr_18_theft_total"
"clr_18_mtr_veh_theft_car"
"clr_18_mtr_veh_theft_other"
"clr_18_assault_aggravated"
"clr_18_index_property"
"unfound_murder"
"unfound_rape_total"
"unfound_rape_attempted"
"unfound_robbery_with_a_gun"
"unfound_robbery_other_weapon"
"unfound_assault_total"
"unfound_assault_with_a_knife"
"unfound_assault_unarmed"
"unfound_burg_total"
"unfound_burg_nonforce_entry"
"unfound_theft_total"
"unfound_mtr_veh_theft_car"
"unfound_mtr_veh_theft_other"
"unfound_assault_aggravated"
"unfound_index_property"

```

Now let's discuss how to subset this data into a smaller data set to answer a specific question. Let's subset the data to answer our above question of "did Colorado's marijuana legalization affect crime in the state?" Like mentioned above, we need data just from Colorado and just for years around the legalization year - we can do 2011-2017 for simplicity.

We also don't need all 159 columns in the current data. Let's say we're only interested in if murder changes. We'd need the column called *actual_murder*, the *state* column (as a check to make sure we subset only Colorado), the *year* column, the *population* column, the *ori* column, and the *agency_name* column (a real analysis would likely grab geographic variables too to see if changes depended on location but here we're just using it as an example). The last two columns - *ori* and *agency_name* - aren't strictly necessary but would be useful if checking if an agency's values are reasonable when checking for outliers, a step we won't do here.

Before explaining how to subset from a data.frame, let's write pseudocode (essentially a description of what we are going to do that is readable to people but isn't real code) for our subset.

We want

- Only rows where the state equals Colorado
- Only rows where the year is 2011-2017
- Only the following columns: *actual_murder*, *state*, *year*, *population*, *ori*, *agency_name*

2.5.1 Select specific columns

The way to select a specific column in R is called the dollar sign notation.

`data$column`

We write the data name followed by a \$ and then the column name. Make sure there are no spaces, quotes, or misspellings (or capitalization issues). Just the `data$column` exactly as it is spelled. Since we are referring to data already read into R, there should not be any quotes for either the data or the column name.

We can do this for the column *agency_name* in our UCR data. If we wrote this in the console it would print out every single row in the column. Because

this data is large (nearly a million rows), I am going to wrap this in `head()` so it only displays the first 6 rows of the column rather than printing the entire column.

```
head(offenses_known_yearly_1960_2017$agency_name)
```

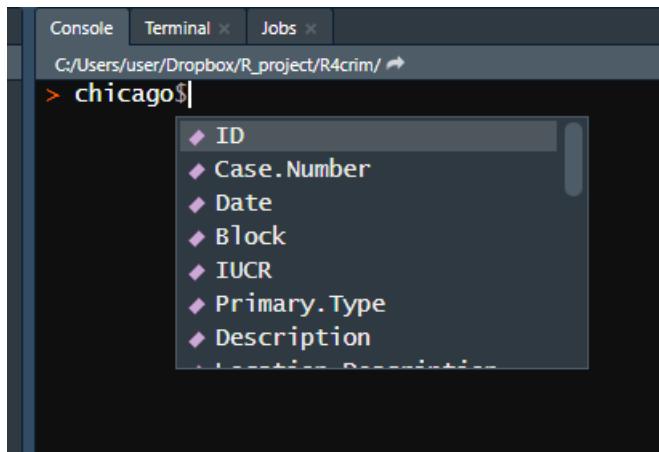
```
#> [1] "anchorage" "anchorage" "anchorage" "anchorage" "anchorage" "anchorage"
```

They're all the same name because Anchorage Police reported many times and are in the data set multiple times. Let's look at the column `actual_murder` which shows the annual number of murders in that agency.

```
head(offenses_known_yearly_1960_2017$actual_murder)
```

```
#> [1] 27 28 26 12 14 15
```

One hint is to write out the data set name in the console and hit the Tab key. Wait a couple of seconds and a popup will appear listing every column in the data set. You can scroll through this and then hit enter to select that column.



2.5.2 Select specific rows

In the earlier examples we used square bracket notation `[]` and just put a number or several numbers in the `[]`. When dealing with `data.frames`, however, you need an extra step to tell R which columns to keep. The syntax in the square bracket is

```
[row, column]
```

As we did earlier, we start in the square bracket by saying which row we want. Now, since we also have to consider the columns, we need to tell it the number or name (in a vector using `c()` if more than one name and putting column names in quotes) of the column or columns we want.

The exception to this is when we use the dollar sign notation to select a single column. In that case we don't need a comma (and indeed it will give us an error!). Let's see a few examples and then explain why this works the way it does.

```
offenses_known_yearly_1960_2017[1, 1]
#> [1] "AK00101"
```

If we input multiple numbers, we can get multiple rows and columns.

```
offenses_known_yearly_1960_2017[1:6, 1:6]
#>      ori      orig agency_name state state_abb year
#> 1 AK00101 AK0010100 anchorage alaska      AK 2017
#> 2 AK00101 AK0010100 anchorage alaska      AK 2016
#> 3 AK00101 AK0010100 anchorage alaska      AK 2015
#> 4 AK00101 AK0010100 anchorage alaska      AK 2014
#> 5 AK00101 AK0010100 anchorage alaska      AK 2013
#> 6 AK00101 AK0010100 anchorage alaska      AK 2012
```

The column section also accepts a vector of the names of the columns. These names must be spelled correctly and in quotes.

```
offenses_known_yearly_1960_2017[1:6, c("ori", "year")]
#>      ori year
#> 1 AK00101 2017
#> 2 AK00101 2016
#> 3 AK00101 2015
#> 4 AK00101 2014
#> 5 AK00101 2013
#> 6 AK00101 2012
```

In cases where we want every row or every column, we just don't put a number. By default, R will return every row/column if you don't specify which ones you want. However, you will still need to include the comma.

Here is every column in the first row.

```

offenses_known_yearly_1960_2017[1, ]
#>      ori      ori9 agency_name state state_abb year number_of_months_re
#> 1 AK00101 AK0010100 anchorage alaska          AK 2017
#>   fips_state_code fips_county_code fips_state_county_code fips_place_code
#> 1           02           020           02020          03000
#>   fips_state_place_code      agency_type agency_subtype_1
#> 1           0203000 local police department    not applicable
#>   agency_subtype_2      crosswalk_agency_name      census_name
#> 1    not applicable anchorage police department anchorage municipality
#>   population      population_group country_division juvenile_age
#> 1     296188 city 250,000 thru 499,999      pacific        18
#>   core_city_indication last_update fbi_field_office followup_indication
#> 1    core city of msa       42094           3030 send a follow-up
#>   zip_code covered_by_ori agency_count date_of_last_update month_included
#> 1     99507           <NA>            1           120717
#>                                         special_mail
#> 1 the agency is a contributor but not on the mailing list, they are not se
#>   special_mailing_address first_line_of_mailing_address
#> 1 not a special mailing address      chief of police
#>   second_line_of_mailing_address third_line_of_mailing_address
#> 1     anchorage police department      4501 elmore rd
#>   fourth_line_of_mailing_address officers_killed_by_felony
#> 1           anchorage, ak            0
#>   officers_killed_by_accident officers_assaulted actual_murder
#> 1           0           426           27
#>   actual_manslaughter actual Rape_total actual_rape_by_force
#> 1           3           391           350
#>   actual_rape_attempted actual_robbery_total actual_robbery_with_a_gun
#> 1           41           778           249
#>   actual_robbery_with_a_knife actual_robbery_other_weapon
#> 1           69           116
#>   actual_robbery_unarmed actual_assault_total actual_assault_with_a_gun
#> 1           344           6448          621
#>   actual_assault_with_a_knife actual_assault_other_weapon
#> 1           392           704
#>   actual_assault_unarmed actual_assault_simple actual_burg_total
#> 1           651           4080          2216

```

```

#>   actual_burg_force_entry actual_burg_nonforce_entry actual_burg_attempted
#> 1           1537                  521                 158
#>   actual_theft_total actual_mtr_veh_theft_total actual_mtr_veh_theft_car
#> 1           10721                 3104                1934
#>   actual_mtr_veh_theft_truck actual_mtr_veh_theft_other actual_all_crimes
#> 1           971                  199                23688
#>   actual_assault_aggravated actual_index_violent actual_index_property
#> 1           2368                 3564                16041
#>   actual_index_total tot_clr_murder tot_clr_manslaughter tot_clr_rape_total
#> 1           19605                 28                  0                  58
#>   tot_clr_rape_by_force tot_clr_rape_attempted tot_clr_robbery_total
#> 1           48                   10                 216
#>   tot_clr_robbery_with_a_gun tot_clr_robbery_with_a_knife
#> 1           47                   22
#>   tot_clr_robbery_other_weapon tot_clr_robbery_unarmed tot_clr_assault_total
#> 1           37                   110                3576
#>   tot_clr_assault_with_a_gun tot_clr_assault_with_a_knife
#> 1           249                  250
#>   tot_clr_assault_other_weapon tot_clr_assault_unarmed tot_clr_assault_simple
#> 1           413                  436                2228
#>   tot_clr_burg_total tot_clr_burg_force_entry tot_clr_burg_nonforce_entry
#> 1           250                  129                114
#>   tot_clr_burg_attempted tot_clr_theft_total tot_clr_mtr_veh_theft_total
#> 1           7                   1358                497
#>   tot_clr_mtr_veh_theft_car tot_clr_mtr_veh_theft_truck
#> 1           335                  145
#>   tot_clr_mtr_veh_theft_other tot_clr_all_crimes tot_clr_assault_aggravated
#> 1           17                  5983                1348
#>   tot_clr_index_violent tot_clr_index_property tot_clr_index_total
#> 1           1650                 2105                3755
#>   clr_18_murder clr_18_manslaughter clr_18_rape_total clr_18_rape_by_force
#> 1           1                   0                   5                  4
#>   clr_18_rape_attempted clr_18_robbery_total clr_18_robbery_with_a_gun
#> 1           1                   9                   0                  1
#>   clr_18_robbery_with_a_knife clr_18_robbery_other_weapon
#> 1           1                   0
#>   clr_18_robbery_unarmed clr_18_assault_total clr_18_assault_with_a_gun

```

```
#> 1 7 277 37
#> clr_18_assault_with_a_knife clr_18_assault_other_weapon
#> 1 17 19
#> clr_18_assault_unarmed clr_18_assault_simple clr_18_burg_total
#> 1 17 187 8
#> clr_18_burg_force_entry clr_18_burg_nonforce_entry clr_18_burg_attempte
#> 1 4 2
#> clr_18_theft_total clr_18_mtr_veh_theft_total clr_18_mtr_veh_theft_car
#> 1 107 22 17
#> clr_18_mtr_veh_theft_truck clr_18_mtr_veh_theft_other clr_18_all_crimes
#> 1 2 3 429
#> clr_18_assault_aggravated clr_18_index_violent clr_18_index_property
#> 1 90 105 137
#> clr_18_index_total unfound_murder unfound_manslaughter unfound_rape_tot
#> 1 242 5 0
#> unfound_rape_by_force unfound_rape_attempted unfound_robbery_total
#> 1 16 0 1
#> unfound_robbery_with_a_gun unfound_robbery_with_a_knife
#> 1 1 0
#> unfound_robbery_other_weapon unfound_robbery_unarmed unfound_assault_to
#> 1 0 0
#> unfound_assault_with_a_gun unfound_assault_with_a_knife
#> 1 0 1
#> unfound_assault_other_weapon unfound_assault_unarmed unfound_assault_si
#> 1 1 0
#> unfound_burg_total unfound_burg_force_entry unfound_burg_nonforce_entry
#> 1 0 0 0
#> unfound_burg_attempted unfound_theft_total unfound_mtr_veh_theft_total
#> 1 0 40 70
#> unfound_mtr_veh_theft_car unfound_mtr_veh_theft_truck
#> 1 53 16
#> unfound_mtr_veh_theft_other unfound_all_crimes unfound_assault_aggravat
#> 1 1 138
#> unfound_index_violent unfound_index_property unfound_index_total
#> 1 24 110 134
```

Since there are 159 columns in our data, normally we'd want to avoid printing

out all of them. And in most cases, we would save the output of subsets to a new object to be used later rather than just printing the output in the console.

What happens if we forget the comma? If we put in numbers for both rows and columns but don't include a comma between them it will have an error.

```
offenses_known_yearly_1960_2017[1 1]
#> Error: <text>:1:35: unexpected numeric constant
#> 1: offenses_known_yearly_1960_2017[1 1
#> ^
```

If we only put in a single number and no comma, it will return the column that matches that number. Here we have number 1 and it will return the first column. We'll wrap it in `head()` so it doesn't print out a million rows.

```
head(offenses_known_yearly_1960_2017[1])
#>      ori
#> 1 AK00101
#> 2 AK00101
#> 3 AK00101
#> 4 AK00101
#> 5 AK00101
#> 6 AK00101
```

Since R thinks you are requesting a column, and we only have 159 columns in the data, asking for any number above 159 will return an error.

```
head(offenses_known_yearly_1960_2017[1000])
#> Error in `.[.data.frame`(`offenses_known_yearly_1960_2017`, 1000): undefined column
```

If you already specify a column using dollar sign notation `$`, you do not need to indicate any column in the square brackets`[]`. All you need to do is say which row or rows you want.

```
offenses_known_yearly_1960_2017$agency_name[15]
#> [1] "anchorage"
```

So make sure when you want a row from a data.frame you always include the comma!

2.5.3 Subset Colorado data

Finally we have the tools to subset our UCR data to just be Colorado from 2011-2017. There are three conditional statements we need to make, two for rows and one for columns.

- Only rows where the state equals Colorado
- Only rows where the year is 2011-2017
- Only the following columns: actual_murder, state, year, population, ori, agency_name

We could use the `&` operator to say rows must meet condition 1 and condition 2. Since this is an intro lesson, we will do them as two separate conditional statements. For the first step we want to get all rows in the data where the state equals “colorado” (in this data all state names are lowercase). And at this point we want to keep all columns in the data. So let’s make a new object called `colorado` to save the result of this subset.

Remember that we want to put the object to the left of the `[]` (and touching the `[]`) to make sure it returns the data. Just having the conditional statement will only return TRUE or FALSE values. Since we want all columns, we don’t need to put anything after the comma (but we must include the comma!).

```
colorado <- offenses_known_yearly_1960_2017[offenses_known_yearly_1960_2017$st
```

Now we want to get all the rows where the year is 2011-2017. Since we want to check if the year is one of the years 2011-2017, we will use `%in%` and put the years in a vector `2011:2017`. This time our primary data set is `colorado`, not `offenses_known_yearly_1960_2017` since `colorado` has already subsetted to just the state we want. This is how subsetting generally works. You take a large data set, subset it to a smaller one and continue to subset the smaller one to only the data you want.

```
colorado <- colorado[colorado$year %in% 2011:2017, ]
```

Finally we want the columns stated above and to keep every row in the current data. Since the format is `[row, column]` in this case we keep the “row” part blank to indicate that we want every row.

```
colorado <- colorado[ , c("actual_murder", "state", "year", "population", "ori", "age")]
```

We can do a quick check using the `unique()` function. The `unique()` prints all the unique values in a category, such as a column. We will use it on the *state* and *year* columns to make sure only the values that we want are present.

```
unique(colorado$state)
#> [1] "colorado"
```

```
unique(colorado$year)
#> [1] 2017 2016 2015 2014 2013 2012 2011
```

The only state is Colorado and the only years are 2011-2017 so our subset worked! This data shows the number of murders in each agency. We want to look at state trends so in Section 3.3 we will sum up all the murders per year and see if marijuana legalization affected it.

Chapter 3

Exploratory data analysis

When you first start working on new data it is important to spend some time getting familiar with the data. This includes understanding how many rows and columns it has, what each row means (is each row an offender? a victim? crime in a city over a day/month/year?, etc.), and what columns it has. **Basically you want to know if the data is capable of answering the question you are asking.**

While not a comprehensive list, the following is a good start for exploratory data analysis of new data sets.

- What are the units (what does each row represent)?
- What variables are available?
- What time period does it cover?
- Are there outliers? How many?
- Are there missing values? How many?

For this lesson we will use a data set of FBI Uniform Crime Reporting (UCR) data for 2017. This data includes every agency that reported their data for all 12 months of the year. Throughout this lesson we will look at some summary statistics for the variables we are interested in and make some basic graphs to visualize the data. We'll return to UCR data in Chapter 23 when focusing on what the UCR is and how to use it.

First, we need to load the data. Make sure your working directory is set to the folder where the data is.

```
load("data/ucr2017.rda")
```

As a first step, let's see how many rows and columns are in the data, and glance at the first several rows from each column. `nrow()` and `ncol()` tell us the number of rows and columns it has, respectively. Like most functions, what you need to do is put the data set name inside the () (exactly as it is spelled without any quotes).

```
nrow(ucr2017)
#> [1] 15764
```

```
ncol(ucr2017)
#> [1] 9
```

The function `head()` will print out the first 6 rows of every column in the data. Since we only have 9 columns, we will use this function. Be careful when you have many columns (100+) as printing all of them out makes it read to read.

```
head(ucr2017)
#>      ori year agency_name state population actual_murder actual_rape_to
#> 1 AK00101 2017 anchorage alaska     296188          27
#> 2 AK00102 2017 fairbanks alaska     32937           10
#> 3 AK00103 2017 juneau alaska     32344            1
#> 4 AK00104 2017 ketchikan alaska     8230            1
#> 5 AK00105 2017 kodiak alaska      6198            0
#> 6 AK00106 2017 nome alaska       3829            0
#>   actual_robbery_total actual_assault_aggravated
#> 1                  778                 2368
#> 2                  40                  131
#> 3                  46                  206
#> 4                  0                   14
#> 5                  4                   41
#> 6                  0                   52
```

From these results it appears that each row is a single agency's annual data for 2017 and the columns show the number of crimes for four crime categories included (the full UCR data contains many more crimes which we'll see in a later lesson).

Finally, we can run `names()` to print out every column name. We can already see every name from `head()` but this is useful when we have many columns and don't want to use `head()`.

```
names(ucr2017)
#> [1] "ori"                      "year"
#> [3] "agency_name"              "state"
#> [5] "population"               "actual_murder"
#> [7] "actual Rape total"       "actual_robbery_total"
#> [9] "actual assault aggravated"
```

3.1 Summary and Table

An important function in understanding the data you have is `summary()` which, as discussed in Section 1.3, provides summary statistics on the numeric columns you have. Let's take a look at the results before seeing how to do something similar for categorical columns.

```
summary(ucr2017)
#>      ori                  year      agency_name      state
#>  Length:15764      Min.   :2017  Length:15764      Length:15764
#>  Class :character  1st Qu.:2017  Class :character  Class :character
#>  Mode  :character  Median :2017  Mode  :character  Mode  :character
#>                    Mean   :2017
#>                    3rd Qu.:2017
#>                    Max.   :2017
#>
#>      population      actual_murder      actual Rape total  actual_robbery_total
#>  Min.   :     0  Min.   : 0.000  Min.   :-2.000  Min.   : -1.00
#>  1st Qu.:  914  1st Qu.: 0.000  1st Qu.: 0.000  1st Qu.:  0.00
#>  Median : 4460  Median : 0.000  Median : 1.000  Median :  0.00
#>  Mean   : 19872  Mean   : 1.069  Mean   : 8.262  Mean   : 19.85
#>  3rd Qu.: 15390  3rd Qu.: 0.000  3rd Qu.: 5.000  3rd Qu.:  4.00
#>  Max.   :8616333  Max.   :653.000  Max.   :2455.000  Max.   :13995.00
#>
#>      actual assault aggravated
#>  Min.   : -1.00
#>  1st Qu.:  1.00
#>  Median :  5.00
#>  Mean   : 49.98
```

```
#> 3rd Qu.: 21.00
#> Max. :29771.00
```

The `table()` function returns every unique value in a category **and** how often that value appears. Unlike `summary()` we can't just put the entire data set into the `()`, we need to specify a single column. To specify a column you use the dollar sign notation which is `data$column`. For most functions we use to examine the data as a whole, you can do the same for a specific column.

```
head(ucr2017$agency_name)
#> [1] "anchorage" "fairbanks" "juneau"      "ketchikan" "kodiak"      "nome"
```

There are only two columns in our data with categorical values that we can use - *year* and *state* so let's use `table()` on both of them. The columns *ori* and *agency_name* are also categorical but as each row of data has a unique ORI and name, running `table()` on those columns would not be helpful.

```
table(ucr2017$year)
#>
#> 2017
#> 15764
```

We can see that every year in our data is 2017, as expected based on the data name. *year* is a numerical column so why can we use `table()` on it? R doesn't differentiate between numbers and characters when seeing how often each value appears. If we ran `table()` on the column "actual_murder" it would tell us how many times each unique value in the column appeared in the data. That wouldn't be very useful as we don't really care how many times an agency has 7 murders, for example (though looking for how often a numeric column has the value 0 can be helpful in finding likely erroneous data). As numeric variables often have many more unique values than character variables, it also leads to many values being printed, making it harder to understand. For columns where the number of categories is important to us, such as years, states, neighborhoods, we should use `table()`.

```
table(ucr2017$state)
#>
#>           alabama          alaska          arizona
#>             305              32            107
```

#>	<i>arkansas</i>	<i>california</i>	<i>colorado</i>
#>	273	732	213
#>	<i>connecticut</i>	<i>delaware</i>	<i>district of columbia</i>
#>	107	63	3
#>	<i>florida</i>	<i>georgia</i>	<i>guam</i>
#>	603	522	1
#>	<i>hawaii</i>	<i>idaho</i>	<i>illinois</i>
#>	4	95	696
#>	<i>indiana</i>	<i>iowa</i>	<i>kansas</i>
#>	247	216	309
#>	<i>kentucky</i>	<i>louisiana</i>	<i>maine</i>
#>	352	192	135
#>	<i>maryland</i>	<i>massachusetts</i>	<i>michigan</i>
#>	152	346	625
#>	<i>minnesota</i>	<i>mississippi</i>	<i>missouri</i>
#>	397	71	580
#>	<i>montana</i>	<i>nebraska</i>	<i>nevada</i>
#>	108	225	59
#>	<i>new hampshire</i>	<i>new jersey</i>	<i>new mexico</i>
#>	176	576	116
#>	<i>new york</i>	<i>north carolina</i>	<i>north dakota</i>
#>	532	310	108
#>	<i>ohio</i>	<i>oklahoma</i>	<i>oregon</i>
#>	532	409	172
#>	<i>pennsylvania</i>	<i>rhode island</i>	<i>south carolina</i>
#>	1473	49	427
#>	<i>south dakota</i>	<i>tennessee</i>	<i>texas</i>
#>	92	466	999
#>	<i>utah</i>	<i>vermont</i>	<i>virginia</i>
#>	125	85	407
#>	<i>washington</i>	<i>west virginia</i>	<i>wisconsin</i>
#>	250	200	433
#>	<i>wyoming</i>		
#>	57		

This shows us how many times each state is present in the data. States with a larger population tend to appear more often, this makes sense as

those states have more agencies to report. Right now the results are in alphabetical order, but when knowing how frequently something appears, we usually want it ordered by frequency. We can use the `sort()` function to order the results from `table()`. Just put the entire `table()` function inside of the () in `sort()`.

```
sort(table(ucr2017$state))
#>
#>      guam district of columbia          hawaii
#>      1                      3                  4
#>      alaska      rhode island      wyoming
#>      32                      49                  57
#>      nevada      delaware      mississippi
#>      59                      63                  71
#>      vermont      south dakota      idaho
#>      85                      92                  95
#>      arizona      connecticut      montana
#>      107                     107                 108
#>      north dakota      new mexico      utah
#>      108                     116                 125
#>      maine      maryland      oregon
#>      135                     152                 172
#>      new hampshire      louisiana      west virginia
#>      176                     192                 200
#>      colorado      iowa      nebraska
#>      213                     216                 225
#>      indiana      washington      arkansas
#>      247                     250                 273
#>      alabama      kansas      north carolina
#>      305                     309                 310
#>      massachusetts      kentucky      minnesota
#>      346                     352                 397
#>      virginia      oklahoma      south carolina
#>      407                     409                 427
#>      wisconsin      tennessee      georgia
#>      433                     466                 522
#>      new york      ohio      new jersey
#>      532                     532                 576
```

#>	<i>missouri</i>	<i>florida</i>	<i>michigan</i>
#>	580	603	625
#>	<i>illinois</i>	<i>california</i>	<i>texas</i>
#>	696	732	999
#>	<i>pennsylvania</i>		
#>	1473		

And if we want to sort it in decreasing order of frequency, we can use the parameter `decreasing` in `sort()` and set it to TRUE. A parameter is just an option used in an R function to change the way the function is used or what output it gives. Almost all functions have these parameters and they are useful if you don't want to use the default setting in the function. This parameter, `decreasing` changes the `sort()` output to print from largest to smallest. By default this parameter is set to FALSE and here we say it is equal to TRUE.

sort(table(ucr2017\$state), decreasing = TRUE)			
#>			
#>	<i>pennsylvania</i>	<i>texas</i>	<i>california</i>
#>	1473	999	732
#>	<i>illinois</i>	<i>michigan</i>	<i>florida</i>
#>	696	625	603
#>	<i>missouri</i>	<i>new jersey</i>	<i>new york</i>
#>	580	576	532
#>	<i>ohio</i>	<i>georgia</i>	<i>tennessee</i>
#>	532	522	466
#>	<i>wisconsin</i>	<i>south carolina</i>	<i>oklahoma</i>
#>	433	427	409
#>	<i>virginia</i>	<i>minnesota</i>	<i>kentucky</i>
#>	407	397	352
#>	<i>massachusetts</i>	<i>north carolina</i>	<i>kansas</i>
#>	346	310	309
#>	<i>alabama</i>	<i>arkansas</i>	<i>washington</i>
#>	305	273	250
#>	<i>indiana</i>	<i>nebraska</i>	<i>iowa</i>
#>	247	225	216
#>	<i>colorado</i>	<i>west virginia</i>	<i>louisiana</i>
#>	213	200	192

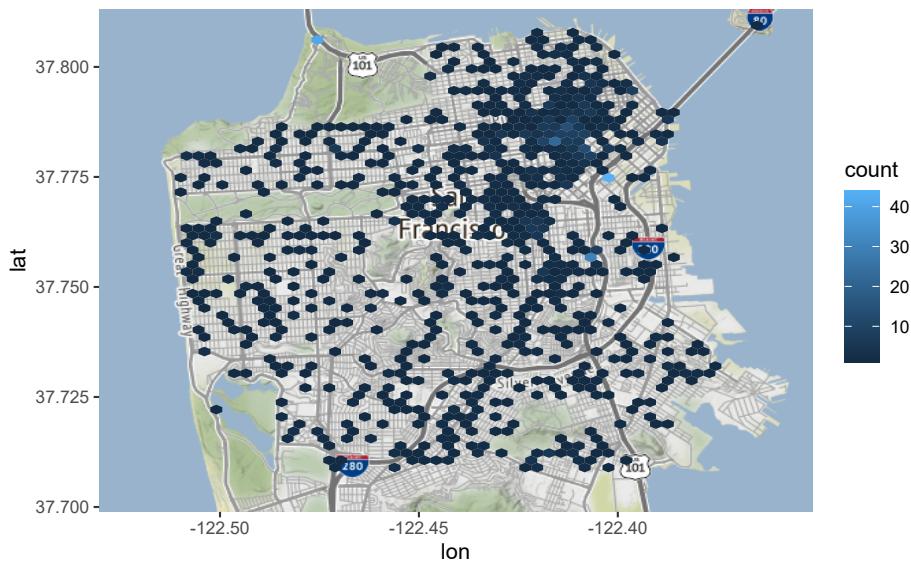
#>	<i>new hampshire</i>	<i>oregon</i>	<i>maryland</i>
#>	176	172	152
#>	<i>maine</i>	<i>utah</i>	<i>new mexico</i>
#>	135	125	116
#>	<i>montana</i>	<i>north dakota</i>	<i>arizona</i>
#>	108	108	107
#>	<i>connecticut</i>	<i>idaho</i>	<i>south dakota</i>
#>	107	95	92
#>	<i>vermont</i>	<i>mississippi</i>	<i>delaware</i>
#>	85	71	63
#>	<i>nevada</i>	<i>wyoming</i>	<i>rhode island</i>
#>	59	57	49
#>	<i>alaska</i>	<i>hawaii</i>	<i>district of columbia</i>
#>	32	4	3
#>	<i>guam</i>		
#>	1		

3.2 Graphing

We often want to make quick plots of our data to get a visual understanding of the data. We will learn a more different way to make graphs in Chapter 6 but for now let's use the function `plot()`.

Let's make a few scatterplots showing the relationship between two variables. With `plot()` the syntax (how you write the code) is `plot(x_axis_variable, y_axis_variable)`. So all we need to do is give it the variable for the x- and y-axis. Each dot will represent a single agency (a single row in our data).

```
plot(ucr2017$actual_murder, ucr2017$actual_robbery_total)
```



Above we are telling R to plot the number of murders on the x-axis and the number of robberies on the y-axis. This shows the relationship between a city's number of murders and number of robberies. We can see that there is a relationship where more murders is correlated with more robberies. However, there are a huge number of agencies in the bottom-left corner which have very few murders or robberies. This makes sense as - as we see in the `summary()` above - most agencies are small, with the median population under 5,000 people.

To try to avoid that clump of small agencies at the bottom, let's make a new data set of only agencies with a population over 1 million. We will use the square bracket [] notation to subset. Remember it is

```
[rows, columns]
```

where we either say exactly which rows or columns we want or give a conditional statement and it'll return only those that meet the condition. We will use the condition that we only want rows where the population is over 1 million.

```
[ucr2017$population > 1000000, ]
```

Now our row conditional is done. We want all the columns in the data so leave the section after the comma empty (don't forget to include that

comma!). Now our square bracket notation is done but we need to put it directly to the right of our data so that we take the rows from the right data set.

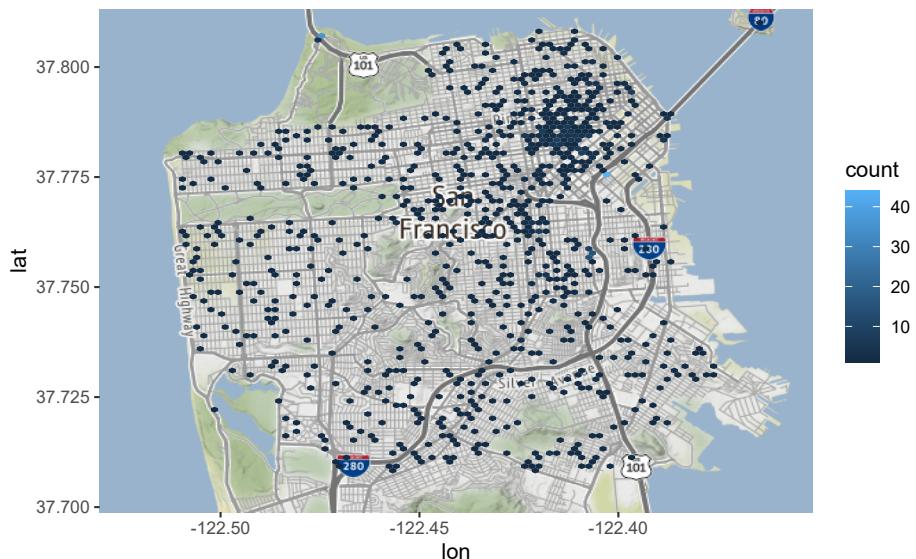
```
ucr2017[ucr2017$population > 1000000, ]
```

And let's save the results in a new object called "ucr2017_big_cities".

```
ucr2017_big_cities <- ucr2017[ucr2017$population > 1000000, ]
```

Now we can do the same graph as above but using this new data set.

```
plot(ucr2017_big_cities$actual_murder, ucr2017_big_cities$actual_robbery_total
```



The problem is somewhat solved. There is still a small clumping of agencies with few robberies or aggravated assaults but the issue is much better. And interestingly the trend is similar with this small subset of data as with all agencies included.

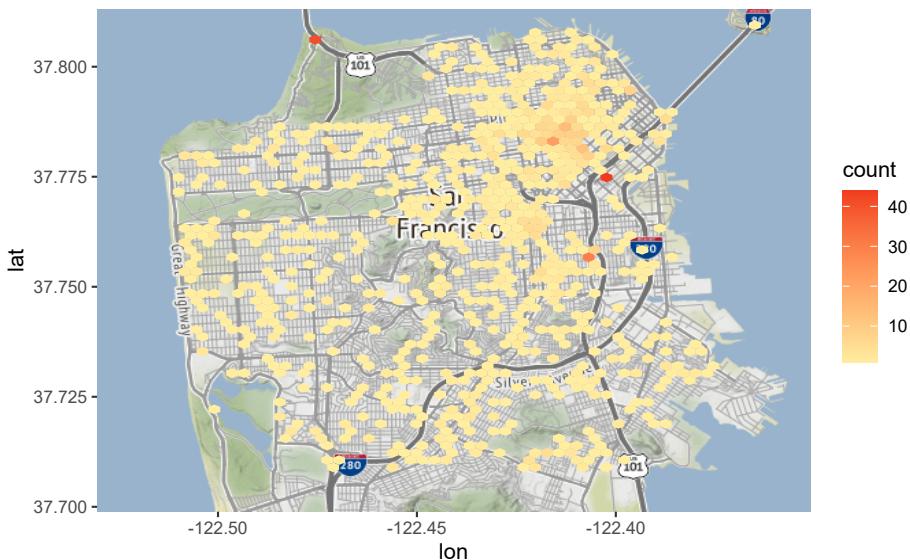
To make our graph look better, we can add labels for the axes and a title (there are many options for changing the appears of this graph, we will just use these three).

- `xlab` - X-axis label
- `ylab` - Y-axis label

- main - Graph title

Like all parameters, we add them in the () of `plot()` and separate each parameter by a comma. Since we are adding text to write in the plot, all of these parameter inputs must be in quotes.

```
plot(ucr2017_big_cities$actual_murder, ucr2017_big_cities$actual_robbery_total,
      xlab = "Murder",
      ylab = "Robberies",
      main = "Relationship between murder and robbery")
```



3.3 Aggregating (summaries of groups)

Right now we have the number of crimes in each agency. For many policy analyses we'd be looking at the effect on the state as a whole, rather than at the agency-level. If we wanted to do this in our data, we would need to **aggregate** up to the state level. What the `aggregate()` function does is group values at some higher level than they currently are (e.g. from agency to state, from day to month, from city street to city neighborhood) and then do some mathematical operation of our choosing (in our case usually sum) to that group.

In Section 2.5.3 we started to see if marijuana legalization affected murder in

Colorado. We subsetted the data to only include agencies in Colorado from 2011-2017. Now we can continue to answer the question by aggregating to the state-level to see the total number of murders per year.

Let's think about how our data are and how we would (theoretically, before we write any code) find that out.

Our data is a single row for each agency and we have a column indicating the year the agency reported. So how would we find out how many murders happened in Colorado for each year? Well, first we take all the agencies in 2011 (the first year available) and add up the murders for all agencies that reported that year. Then take all the rows in 2012 and add up their murders. And so on for all the years. That is essentially what `aggregate()` does. It takes each row and groups them according to the category we specify and then adds up (or does the mathematical operator we specify) each value in each group.

The syntax (how we write the code) is as follows

```
aggregate(numerical_column ~ category_column, FUN = math, data = data_set)
```

The numerical column is the column that we are doing the mathematical operation (sum, mean, median) on. The category column is the one we are using to group (e.g. state, year). Note the `~` between the numerical and category columns. Unlike most functions where we specify a column name, in `aggregate()` we do **not** use quotes for the columns.

`FUN` is the parameter where we tell `aggregate()` which mathematical operator to use. Note that `FUN` is all in capital letters. That is just how this function calls the parameter so we need to make sure we write it in capital letters. `data_set` is the name of the data set we are aggregating.

In Chapter 2 we wanted to see if marijuana legalization in Colorado affected murder. To do this we need to have data showing the number of murders for a few years before and after legalization. We have subsetted UCR data to get all agencies in Colorado for the 3 years before and after 2014, the year of legalization. Let's reload that data and rerun the subsetting code.

```
load("data/offenses_known_yearly_1960_2017.rda")
colorado <- offenses_known_yearly_1960_2017[offenses_known_yearly_1960_2017$st
```

```
colorado <- colorado[colorado$year %in% 2011:2017, ]
colorado <- colorado[ , c("actual_murder", "state", "year", "population", "ori", "age")]
```

Now we can run `aggregate()` to get the number of murders per year.

```
aggregate(actual_murder ~ year, FUN = sum, data = colorado)
  year actual_murder
1 2011      154
2 2012      163
3 2013      172
4 2014      148
5 2015      173
6 2016      203
7 2017      218
```

If we had more grouping categories we could add them by literally using `+` and then writing the next grouping variable name. In our case since all agencies are in the same state it doesn't actually change the results.

```
aggregate(actual_murder ~ year + state, FUN = sum, data = colorado)
  year   state actual_murder
1 2011  colorado      154
2 2012  colorado      163
3 2013  colorado      172
4 2014  colorado      148
5 2015  colorado      173
6 2016  colorado      203
7 2017  colorado      218
```

If we want to aggregate multiple numeric columns, we would use the `cbind()` function which binds together columns. Many times we care more about the crime rate (per 100,000 population usually) than the total number of crimes as a larger population tends to also mean more crime. We can aggregate both the population column and the `actual_murder` column to get totals for each year which we can use to make a murder rate column. Since we need the output of this aggregate saved somewhere to make that column, let's call it `colorado_agg`.

```
colorado_agg <- aggregate(cbind(population, actual_murder) ~ year, FUN = sum,
```

To make the murder rate we simply make a new column, which we can call *murder_rate* which is the number of murders divided by population multiplied by 100,000.

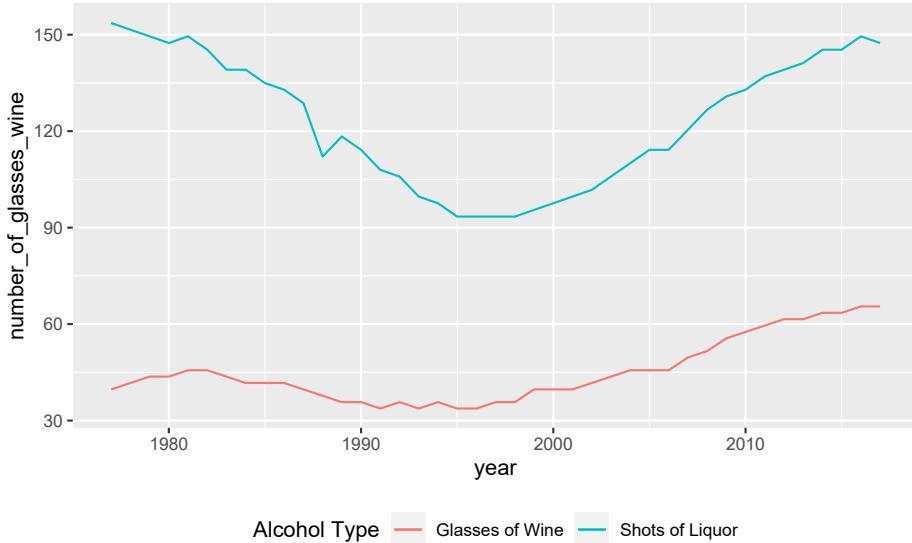
```
colorado_agg$murder_rate <- colorado_agg$actual_murder / colorado_agg$population * 100000
```

	year	population	actual_murder	murder_rate
#>	1 2011	5155993	154	2.986816
#>	2 2012	5227884	163	3.117896
#>	3 2013	5308236	172	3.240248
#>	4 2014	5402555	148	2.739445
#>	5 2015	5505856	173	3.142109
#>	6 2016	5590124	203	3.631404
#>	7 2017	5661529	218	3.850550

Now we can see that the total number of murders increased as did the murder rate. So can we conclude that marijuana legalization increases murder? No, all this analysis shows is that the years following marijuana legalization, murders increased in Colorado. But that can be due to many reasons other than marijuana. For a proper analysis you'd need a comparison area that is similar to Colorado prior to legalization (and didn't legalize marijuana) and see if the murder rates changes following Colorado's legalization.

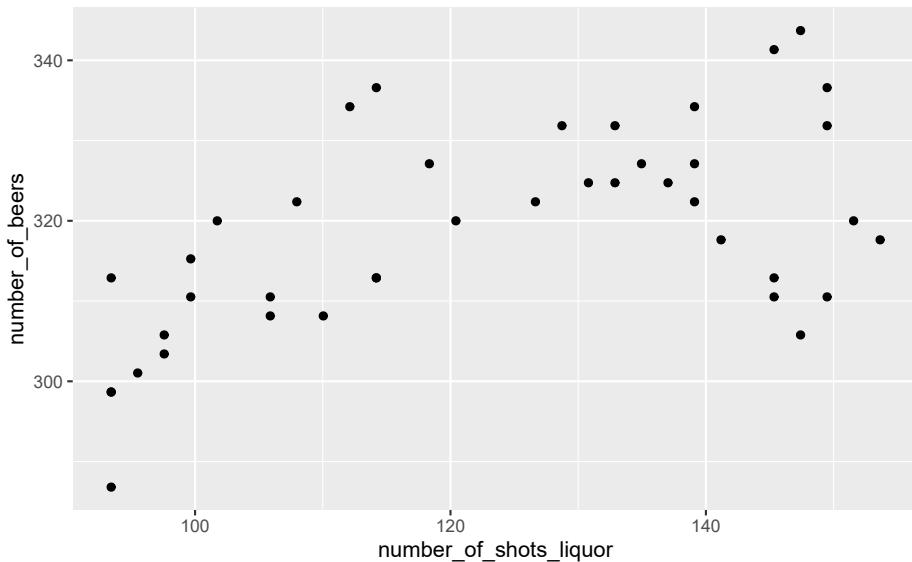
We can also make a plot of this data showing the murder rate over time. With time-series graphs we want the time variable to be on the x-axis and the numeric variable we are measuring to the on the y-axis.

```
plot(x = colorado_agg$year, y = colorado_agg$murder_rate)
```



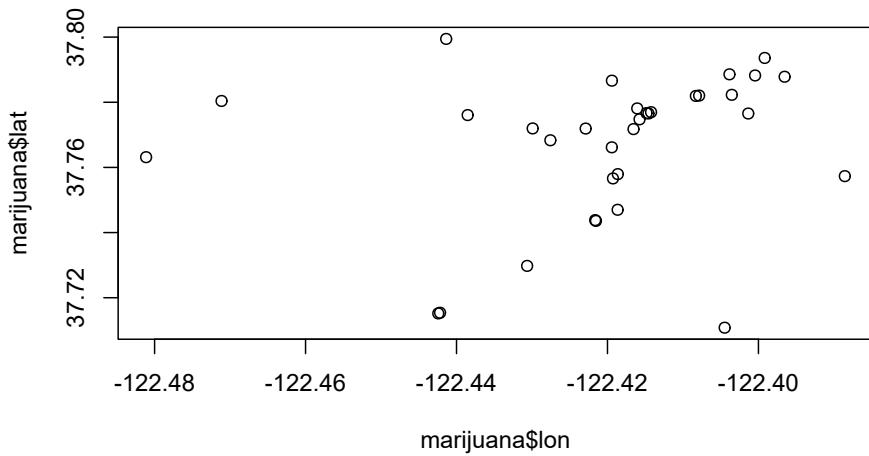
By default `plot()` makes a scatterplot. If we set the parameter `type` to "l" it will be a line plot.

```
plot(x = colorado_agg$year, y = colorado_agg$murder_rate, type = "l")
```



We can add some labels and a title to make this graph easier to read.

```
plot(x = colorado_agg$year, y = colorado_agg$murder_rate, type = "l",
      xlab = "Year",
      ylab = "Murders per 100k Population",
      main = "Murder Rate in Colorado, 2011-2017")
```



Chapter 4

Regular Expressions

Many word processing programs like Microsoft Word or Google Docs let you search for a pattern - usually a word or phrase - and it will show you where on the page that pattern appears. It also lets you replace that word or phrase with something new. R does the same using the function `grep()` to search for a pattern and tell you where in the data it appears, and `gsub()` which lets you search for a pattern and then replace it with a new pattern.

- `grep()` - Find
- `gsub()` - Find and Replace

The `grep()` function lets you find a pattern in the text and it will return a number saying which element has the pattern (in a `data.frame` this tells you which row has a match). `gsub()` lets you input a pattern to find and a pattern to replace it with, just like Find and Replace features elsewhere. You can remember the difference because `gsub()` has the word “sub” in it and what it does is substitute text with new text.

A useful cheat sheet on regular expressions is available [here](#).

For this lesson we will use a vector of 50 crime categories. These are all of the crimes in San Francisco Police data. As we'll see, there are some issues with the crime names that we need to fix.

```
crimes <- c(  
  "Arson",  
  "Assault",
```

```
"Burglary",
"Case Closure",
"Civil Sidewalks",
"Courtesy Report",
"Disorderly Conduct",
"Drug Offense",
"Drug Violation",
"Embezzlement",
"Family Offense",
"Fire Report",
"Forgery And Counterfeiting",
"Fraud",
"Gambling",
"Homicide",
"Human Trafficking (A), Commercial Sex Acts",
"Human Trafficking, Commercial Sex Acts",
"Juvenile Offenses",
"Larceny Theft",
"Liquor Laws",
"Lost Property",
"Malicious Mischief",
"Miscellaneous Investigation",
"Missing Person",
"Motor Vehicle Theft",
"Motor Vehicle Theft?",
"Non-Criminal",
"Offences Against The Family And Children",
"Other",
"Other Miscellaneous",
"Other Offenses",
"Prostitution",
"Rape",
"Recovered Vehicle",
"Robbery",
"Sex Offense",
"Stolen Property",
"Suicide",
```

```
"Suspicious",
"Suspicious Occ",
"Traffic Collision",
"Traffic Violation Arrest",
"Vandalism",
"Vehicle Impounded",
"Vehicle Misplaced",
"Warrant",
"Weapons Carrying Etc",
"Weapons Offence",
"Weapons Offense"
)
```

When looking closely at these crimes it is clear that some may overlap in certain categories such as theft, and there are several duplicates with slight differences in spelling. For example the last two crimes are “Weapons Offence” and “Weapons Offense”. These should be the same crime but the first one spelled “offense” wrong. And take a look at “motor vehicle theft”. There are two crimes here because one of them adds a question mark at the end for some reason.

4.1 Finding patterns in text with grep()

We'll start with `grep()` which allows us to search a vector of data (in R, columns in a `data.frame` operate the same as a vector) and find where there is a match for the pattern we want to look for.

The syntax for `grep()` is

```
grep("pattern", data)
```

Where `pattern` is the pattern you are searching for, such as “a” if you want to find all values with the letter a. The pattern must always be in quotes. `data` is a vector of strings (such as `crimes` we made above or a column in a `data.frame`) that you are searching in to find the pattern.

The output of this function is a number which says which element(s) in the vector the pattern was found in. If it returns, for example, the numbers 1 and 3 you know that the first and third element in your vector has the pattern

- and no other elements do. It is essentially returning the index where the conditional statement “is this pattern present” is true.

So since our data is *crimes* our `grep()` function will be `grep("", crimes)`. What we put in the "" is the pattern we want to search for.

Let's start with the letter “a”.

```
grep("a", crimes)
#> [1] 2 3 4 5 9 11 14 15 17 18 20 21 23 24 28 29 31 34 42 43 44 46 47
#> [26] 50
```

It gives us a bunch of numbers where the letter “a” is present in that element of *crimes*. What this is useful for is subsetting. We can use `grep()` to find all values that match a pattern we want and subset to keep just those values.

```
crimes[grep("a", crimes)]
#> [1] "Assault"
#> [2] "Burglary"
#> [3] "Case Closure"
#> [4] "Civil Sidewalks"
#> [5] "Drug Violation"
#> [6] "Family Offense"
#> [7] "Fraud"
#> [8] "Gambling"
#> [9] "Human Trafficking (A), Commercial Sex Acts"
#> [10] "Human Trafficking, Commercial Sex Acts"
#> [11] "Larceny Theft"
#> [12] "Liquor Laws"
#> [13] "Malicious Mischief"
#> [14] "Miscellaneous Investigation"
#> [15] "Non-Criminal"
#> [16] "Offences Against The Family And Children"
#> [17] "Other Miscellaneous"
#> [18] "Rape"
#> [19] "Traffic Collision"
#> [20] "Traffic Violation Arrest"
#> [21] "Vandalism"
#> [22] "Vehicle Misplaced"
#> [23] "Warrant"
```

```
#> [24] "Weapons Carrying Etc"  
#> [25] "Weapons Offence"  
#> [26] "Weapons Offense"
```

Searching for the letter “a” isn’t that useful. Let’s say we want to subset the data to only include theft related crimes. From reading the list of crimes we can see there are multiple theft crimes - “Larceny Theft”, “Motor Vehicle Theft”, and “Motor Vehicle Theft?”. We may also want to include “Stolen Property” in this search but we’ll wait until later in this lesson for how to search for multiple patterns. Since those three crimes all have the word “Theft” in the name we can search for the pattern and it will return only those crimes

```
grep("Theft", crimes)  
#> [1] 20 26 27  
  
crimes[grep("Theft", crimes)]  
#> [1] "Larceny Theft"           "Motor Vehicle Theft"   "Motor Vehicle Theft?"
```

A very useful parameter is `value`. When we set `value` to TRUE, it will print out the actual strings that are a match rather than the element number. While this prevents us from using it to subset (since R no longer knows which rows are a match), it is an excellent tool to check if the `grep()` was successful as we can visually confirm it returns what we want. When we start to learn about special characters which make the patterns more complicated, this will be important.

```
grep("Theft", crimes, value = TRUE)  
#> [1] "Larceny Theft"           "Motor Vehicle Theft"   "Motor Vehicle Theft?"
```

Note that `grep()` (and `gsub()`) is case sensitive so you must capitalize properly.

```
grep("theft", value = TRUE, crimes)  
#> character(0)
```

Setting the parameter `ignore.case` to be TRUE makes `grep()` ignore capitalization.

```
grep("theft", crimes, value = TRUE, ignore.case = TRUE)
#> [1] "Larceny Theft"           "Motor Vehicle Theft"  "Motor Vehicle Theft?"
```

If we want to find values which do *not* match with “theft”, we can set the parameter `invert` to TRUE.

```
grep("theft", crimes, value = TRUE, ignore.case = TRUE, invert = TRUE)
#> [1] "Arson"
#> [2] "Assault"
#> [3] "Burglary"
#> [4] "Case Closure"
#> [5] "Civil Sidewalks"
#> [6] "Courtesy Report"
#> [7] "Disorderly Conduct"
#> [8] "Drug Offense"
#> [9] "Drug Violation"
#> [10] "Embezzlement"
#> [11] "Family Offense"
#> [12] "Fire Report"
#> [13] "Forgery And Counterfeiting"
#> [14] "Fraud"
#> [15] "Gambling"
#> [16] "Homicide"
#> [17] "Human Trafficking (A), Commercial Sex Acts"
#> [18] "Human Trafficking, Commercial Sex Acts"
#> [19] "Juvenile Offenses"
#> [20] "Liquor Laws"
#> [21] "Lost Property"
#> [22] "Malicious Mischief"
#> [23] "Miscellaneous Investigation"
#> [24] "Missing Person"
#> [25] "Non-Criminal"
#> [26] "Offences Against The Family And Children"
#> [27] "Other"
#> [28] "Other Miscellaneous"
#> [29] "Other Offenses"
#> [30] "Prostitution"
#> [31] "Rape"
```

4.2. FINDING AND REPLACING PATTERNS IN TEXT WITH `gsub()` 79

```
#> [32] "Recovered Vehicle"
#> [33] "Robbery"
#> [34] "Sex Offense"
#> [35] "Stolen Property"
#> [36] "Suicide"
#> [37] "Suspicious"
#> [38] "Suspicious Occ"
#> [39] "Traffic Collision"
#> [40] "Traffic Violation Arrest"
#> [41] "Vandalism"
#> [42] "Vehicle Impounded"
#> [43] "Vehicle Misplaced"
#> [44] "Warrant"
#> [45] "Weapons Carrying Etc"
#> [46] "Weapons Offence"
#> [47] "Weapons Offense"
```

4.2 Finding and replacing patterns in text with `gsub()`

`gsub()` takes patterns and replaces them with other patterns. An important use in criminology for `gsub()` is to fix spelling mistakes in the text such as the way “offense” was spelled wrong in our data. This will be a standard part of your data cleaning process and is important as a misspelled word can cause significant issues. For example if our previous example of marijuana legalization in Colorado had half of agencies misspelling the name “Colorado”, aggregating the data by the state (or simply subsetting to just Colorado agencies) would give completely different results as you’d lose half your data.

`gsub()` is also useful when you want to take subcategories and change the value to larger categories. For example we could take any crime with the word “Theft” in it and change the whole crime name to “Theft”. In our data that would take 3 subcategories of thefts and turn it into a larger category we could aggregate to. This will be useful in city-level data where you may only care about a certain type of crime but it has many subcategories that you need to aggregate.

The syntax of `gsub()` is similar to `grep()` with the addition of a pattern to replace the pattern we found.

```
gsub("find_pattern", "replace_pattern", data)
```

Let's start with a simple example of finding the letter "a" and replacing it with "z". Our data will be the word "cat".

```
gsub("a", "z", "cat")
#> [1] "czt"
```

Like `grep()`, `gsub()` is case sensitive and has the parameter `ignore.case` to ignore capitalization.

```
gsub("A", "z", "cat")
#> [1] "cat"
```

```
gsub("A", "z", "cat", ignore.case = TRUE)
#> [1] "czt"
```

`gsub()` returns the same data you input but with the pattern already replaced. Above you can see that when using capital A, it returns "cat" unchanged as it never found the pattern. When `ignore.case` was set to TRUE it returned "czt" as it then matched to letter "A".

We can use `gsub()` to replace some issues in the crimes data such as "Offense" being spelled "Offence".

```
gsub("Offence", "Offense", crimes)
#> [1] "Arson"
#> [2] "Assault"
#> [3] "Burglary"
#> [4] "Case Closure"
#> [5] "Civil Sidewalks"
#> [6] "Courtesy Report"
#> [7] "Disorderly Conduct"
#> [8] "Drug Offense"
#> [9] "Drug Violation"
#> [10] "Embezzlement"
#> [11] "Family Offense"
#> [12] "Fire Report"
```

4.2. FINDING AND REPLACING PATTERNS IN TEXT WITH GSUB()81

```
#> [13] "Forgery And Counterfeiting"
#> [14] "Fraud"
#> [15] "Gambling"
#> [16] "Homicide"
#> [17] "Human Trafficking (A), Commercial Sex Acts"
#> [18] "Human Trafficking, Commercial Sex Acts"
#> [19] "Juvenile Offenses"
#> [20] "Larceny Theft"
#> [21] "Liquor Laws"
#> [22] "Lost Property"
#> [23] "Malicious Mischief"
#> [24] "Miscellaneous Investigation"
#> [25] "Missing Person"
#> [26] "Motor Vehicle Theft"
#> [27] "Motor Vehicle Theft?"
#> [28] "Non-Criminal"
#> [29] "Offenses Against The Family And Children"
#> [30] "Other"
#> [31] "Other Miscellaneous"
#> [32] "Other Offenses"
#> [33] "Prostitution"
#> [34] "Rape"
#> [35] "Recovered Vehicle"
#> [36] "Robbery"
#> [37] "Sex Offense"
#> [38] "Stolen Property"
#> [39] "Suicide"
#> [40] "Suspicious"
#> [41] "Suspicious Occ"
#> [42] "Traffic Collision"
#> [43] "Traffic Violation Arrest"
#> [44] "Vandalism"
#> [45] "Vehicle Impounded"
#> [46] "Vehicle Misplaced"
#> [47] "Warrant"
#> [48] "Weapons Carrying Etc"
#> [49] "Weapons Offense"
```

```
#> [50] "Weapons Offense"
```

A useful pattern is an empty string "" which says replace whatever the find_pattern is with nothing, deleting it. Let's delete the letter "a" (lowercase only) from the data.

```
gsub("a", "", crimes)
#> [1] "Arson"
#> [2] "Assult"
#> [3] "Burglry"
#> [4] "Cse Closure"
#> [5] "Civil Sidewlks"
#> [6] "Courtesy Report"
#> [7] "Disorderly Conduct"
#> [8] "Drug Offense"
#> [9] "Drug Violtion"
#> [10] "Embezzlement"
#> [11] "Fmily Offense"
#> [12] "Fire Report"
#> [13] "Forgery And Counterfeiting"
#> [14] "Frud"
#> [15] "Gmbling"
#> [16] "Homicide"
#> [17] "Humn Trfficking (A), Commercil Sex Acts"
#> [18] "Humn Trfficking, Commercil Sex Acts"
#> [19] "Juvenile Offenses"
#> [20] "Lrceny Theft"
#> [21] "Liquor Lws"
#> [22] "Lost Property"
#> [23] "Mlicious Mischief"
#> [24] "Miscellneous Investigtion"
#> [25] "Missing Person"
#> [26] "Motor Vehicle Theft"
#> [27] "Motor Vehicle Theft?"
#> [28] "Non-Criminl"
#> [29] "Offences Against The Fmily And Children"
#> [30] "Other"
#> [31] "Other Miscellneous"
```

```
#> [32] "Other Offenses"
#> [33] "Prostitution"
#> [34] "Rpe"
#> [35] "Recovered Vehicle"
#> [36] "Robbery"
#> [37] "Sex Offense"
#> [38] "Stolen Property"
#> [39] "Suicide"
#> [40] "Suspicious"
#> [41] "Suspicious Occ"
#> [42] "Trffic Collision"
#> [43] "Trffic Violtion Arrest"
#> [44] "Vndlism"
#> [45] "Vehicle Impounded"
#> [46] "Vehicle Misplced"
#> [47] "Wrrnt"
#> [48] "Wepons Crrying Etc"
#> [49] "Wepons Offence"
#> [50] "Wepons Offense"
```

4.3 Useful special characters

So far, we have just searched for a single character or word and expected a return only if an exact match was found. Now we'll discuss a number of characters called “special characters” that allow us to make more complex `grep()` and `gsub()` pattern searches.

4.3.1 Multiple characters []

To search for multiple matches we can put the pattern we want to search for inside square brackets [] (note that we use the same square brackets for subsetting but they operate very differently in this context). For example, we can find all the crimes that contain the letters “x”, “y”, or “z”.

The `grep()` searches if any of the letters inside of the [] are present in our *crimes* vector.

```
grep("[xyz]", crimes, value = TRUE)
#> [1] "Burglary"
#> [2] "Courtesy Report"
#> [3] "Disorderly Conduct"
#> [4] "Embezzlement"
#> [5] "Family Offense"
#> [6] "Forgery And Counterfeiting"
#> [7] "Human Trafficking (A), Commercial Sex Acts"
#> [8] "Human Trafficking, Commercial Sex Acts"
#> [9] "Larceny Theft"
#> [10] "Lost Property"
#> [11] "Offences Against The Family And Children"
#> [12] "Robbery"
#> [13] "Sex Offense"
#> [14] "Stolen Property"
#> [15] "Weapons Carrying Etc"
```

As it searches for any letter inside of the square brackets, the order does not matter.

```
grep("[zyx]", crimes, value = TRUE)
#> [1] "Burglary"
#> [2] "Courtesy Report"
#> [3] "Disorderly Conduct"
#> [4] "Embezzlement"
#> [5] "Family Offense"
#> [6] "Forgery And Counterfeiting"
#> [7] "Human Trafficking (A), Commercial Sex Acts"
#> [8] "Human Trafficking, Commercial Sex Acts"
#> [9] "Larceny Theft"
#> [10] "Lost Property"
#> [11] "Offences Against The Family And Children"
#> [12] "Robbery"
#> [13] "Sex Offense"
#> [14] "Stolen Property"
#> [15] "Weapons Carrying Etc"
```

This also works for numbers though we do not have any numbers in the data.

```
grep("[01234567890]", crimes, value = TRUE)
#> character(0)
```

If we wanted to search for a pattern, such as vowels, that is repeated we could put multiple [] patterns together. We will see another way to search for a repeated pattern soon.

```
grep("[aeiou][aeiou][aeiou]", crimes, value = TRUE)
#> [1] "Malicious Mischief"                  "Miscellaneous Investigation"
#> [3] "Other Miscellaneous"                "Suspicious"
#> [5] "Suspicious Occ"
```

Inside the [] we can also use the - to make intervals between certain values. For numbers, n-m means any number between n and m (inclusive). For letters, a-z means all lowercase letters and A-Z means all uppercase letters in that range (inclusive).

```
grep("[x-z]", crimes, value = TRUE)
#> [1] "Burglary"
#> [2] "Courtesy Report"
#> [3] "Disorderly Conduct"
#> [4] "Embezzlement"
#> [5] "Family Offense"
#> [6] "Forgery And Counterfeiting"
#> [7] "Human Trafficking (A), Commercial Sex Acts"
#> [8] "Human Trafficking, Commercial Sex Acts"
#> [9] "Larceny Theft"
#> [10] "Lost Property"
#> [11] "Offences Against The Family And Children"
#> [12] "Robbery"
#> [13] "Sex Offense"
#> [14] "Stolen Property"
#> [15] "Weapons Carrying Etc"
```

4.3.2 n-many of previous character {n}

{n} means the preceding item will be matched exactly n times.

We can use it to rewrite the above grep() to saw the values in the [] should

be repeated three times.

```
grep("[aeiou]{3}", crimes, value = TRUE)
#> [1] "Malicious Mischief"           "Miscellaneous Investigation"
#> [3] "Other Miscellaneous"         "Suspicious"
#> [5] "Suspicious Occ"
```

4.3.3 n-many to m-many of previous character {n,m}

While {n} says “the previous character (or characters inside a []) must be present exactly n times”, we can allow a range by using {n,m}. Here the previous character must be present between n and m times.

We can check for values where there are 2-3 vowels in a row. Note that there cannot be a space before or after the comma.

```
grep("[aeiou]{2,3}", crimes, value = TRUE)
#> [1] "Assault"
#> [2] "Courtesy Report"
#> [3] "Drug Violation"
#> [4] "Forgery And Counterfeiting"
#> [5] "Fraud"
#> [6] "Human Trafficking (A), Commercial Sex Acts"
#> [7] "Human Trafficking, Commercial Sex Acts"
#> [8] "Liquor Laws"
#> [9] "Malicious Mischief"
#> [10] "Miscellaneous Investigation"
#> [11] "Offences Against The Family And Children"
#> [12] "Other Miscellaneous"
#> [13] "Prostitution"
#> [14] "Suicide"
#> [15] "Suspicious"
#> [16] "Suspicious Occ"
#> [17] "Traffic Collision"
#> [18] "Traffic Violation Arrest"
#> [19] "Vehicle Impounded"
#> [20] "Weapons Carrying Etc"
#> [21] "Weapons Offence"
#> [22] "Weapons Offense"
```

If we wanted only crimes with exactly three vowels in a row we'd use {3,3}.

```
grep("[aeiou]{3,3}", crimes, value = TRUE)
#> [1] "Malicious Mischief"           "Miscellaneous Investigation"
#> [3] "Other Miscellaneous"         "Suspicious"
#> [5] "Suspicious Occ"
```

If we leave n blank, such as {,m} it says, “previous character must be present up to m times.”

```
grep("[aeiou]{,3}", crimes, value = TRUE)
#> [1] "Arson"
#> [2] "Assault"
#> [3] "Burglary"
#> [4] "Case Closure"
#> [5] "Civil Sidewalks"
#> [6] "Courtesy Report"
#> [7] "Disorderly Conduct"
#> [8] "Drug Offense"
#> [9] "Drug Violation"
#> [10] "Embezzlement"
#> [11] "Family Offense"
#> [12] "Fire Report"
#> [13] "Forgery And Counterfeiting"
#> [14] "Fraud"
#> [15] "Gambling"
#> [16] "Homicide"
#> [17] "Human Trafficking (A), Commercial Sex Acts"
#> [18] "Human Trafficking, Commercial Sex Acts"
#> [19] "Juvenile Offenses"
#> [20] "Larceny Theft"
#> [21] "Liquor Laws"
#> [22] "Lost Property"
#> [23] "Malicious Mischief"
#> [24] "Miscellaneous Investigation"
#> [25] "Missing Person"
#> [26] "Motor Vehicle Theft"
#> [27] "Motor Vehicle Theft?"
#> [28] "Non-Criminal"
```

```
#> [29] "Offences Against The Family And Children"
#> [30] "Other"
#> [31] "Other Miscellaneous"
#> [32] "Other Offenses"
#> [33] "Prostitution"
#> [34] "Rape"
#> [35] "Recovered Vehicle"
#> [36] "Robbery"
#> [37] "Sex Offense"
#> [38] "Stolen Property"
#> [39] "Suicide"
#> [40] "Suspicious"
#> [41] "Suspicious Occ"
#> [42] "Traffic Collision"
#> [43] "Traffic Violation Arrest"
#> [44] "Vandalism"
#> [45] "Vehicle Impounded"
#> [46] "Vehicle Misplaced"
#> [47] "Warrant"
#> [48] "Weapons Carrying Etc"
#> [49] "Weapons Offence"
#> [50] "Weapons Offense"
```

This returns every crime as “up to m times” includes zero times.

And the same works for leaving m blank but it will be “present at least n times”.

```
grep("[aeiou]{3,}", crimes, value = TRUE)
#> [1] "Malicious Mischief"           "Miscellaneous Investigation"
#> [3] "Other Miscellaneous"          "Suspicious"
#> [5] "Suspicious Occ"
```

4.3.4 Start of string

The `^` symbol (called a caret) signifies that what follows it is the start of the string. We put the `^` at the beginning of the quotes and then anything that follows it must be the very start of the string. As an example let’s

search for “Family”. Our data has both the “Family Offense” crime and the “Offences Against The Family And Children” crime (which likely are the same crime written differently). If we use `^` then we should only have the first one returned.

```
grep("^Family", crimes, value = TRUE)
#> [1] "Family Offense"
```

4.3.5 End of string \$

The dollar sign `$` acts similar to the caret `^` except that it signifies that the value before it is the **end** of the string. We put the `$` at the very end of our search pattern and whatever character is before it is the end of the string. For example, let’s search for all crimes that end with the word “Theft”.

```
grep("Theft$", crimes, value = TRUE)
#> [1] "Larceny Theft"      "Motor Vehicle Theft"
```

Note that the crime “Motor Vehicle Theft?” doesn’t get selected as it ends with a question mark.

4.3.6 Anything .

The `.` symbol is a stand-in for any value. This is useful when you aren’t sure about every part of the pattern you are searching. It can also be used when there are slight differences in words such as our incorrect “Offence” and “Offense”. We can replace the “c” and “s” with the `..`

```
grep("Weapons Offen.e", crimes, value = TRUE)
#> [1] "Weapons Offence" "Weapons Offense"
```

4.3.7 One or more of previous +

The `+` means that the character immediately before it is present at least one time. This is the same as writing `{1,}`. If we wanted to find all values with only two words, we would start with some number of letters followed by a space followed by some more letters and the string would end.

```
grep("^[A-Za-z]+ [A-Za-z]+$", crimes, value = TRUE)
#> [1] "Case Closure"                      "Civil Sidewalks"
#> [3] "Courtesy Report"                   "Disorderly Conduct"
#> [5] "Drug Offense"                     "Drug Violation"
#> [7] "Family Offense"                   "Fire Report"
#> [9] "Juvenile Offenses"                "Larceny Theft"
#> [11] "Liquor Laws"                    "Lost Property"
#> [13] "Malicious Mischief"              "Miscellaneous Investigation"
#> [15] "Missing Person"                  "Other Miscellaneous"
#> [17] "Other Offenses"                  "Recovered Vehicle"
#> [19] "Sex Offense"                    "Stolen Property"
#> [21] "Suspicious Occ"                 "Traffic Collision"
#> [23] "Vehicle Impounded"               "Vehicle Misplaced"
#> [25] "Weapons Offence"                "Weapons Offense"
```

4.3.8 Zero or more of previous *

The * special character says match zero or more of the previous character and is the same as {0,}. Combining . with * is powerful when used in gsub() to delete text before or after a pattern. Let's write a pattern that searches the text for the word "Weapons" and then deletes any text after that.

Our pattern would be "Weapons.*" which is the word "Weapons" followed by anything zero or more times.

```
gsub("Weapons.*", "Weapons", crimes)
#> [1] "Arson"
#> [2] "Assault"
#> [3] "Burglary"
#> [4] "Case Closure"
#> [5] "Civil Sidewalks"
#> [6] "Courtesy Report"
#> [7] "Disorderly Conduct"
#> [8] "Drug Offense"
#> [9] "Drug Violation"
#> [10] "Embezzlement"
#> [11] "Family Offense"
```

```
#> [12] "Fire Report"
#> [13] "Forgery And Counterfeiting"
#> [14] "Fraud"
#> [15] "Gambling"
#> [16] "Homicide"
#> [17] "Human Trafficking (A), Commercial Sex Acts"
#> [18] "Human Trafficking, Commercial Sex Acts"
#> [19] "Juvenile Offenses"
#> [20] "Larceny Theft"
#> [21] "Liquor Laws"
#> [22] "Lost Property"
#> [23] "Malicious Mischief"
#> [24] "Miscellaneous Investigation"
#> [25] "Missing Person"
#> [26] "Motor Vehicle Theft"
#> [27] "Motor Vehicle Theft?"
#> [28] "Non-Criminal"
#> [29] "Offences Against The Family And Children"
#> [30] "Other"
#> [31] "Other Miscellaneous"
#> [32] "Other Offenses"
#> [33] "Prostitution"
#> [34] "Rape"
#> [35] "Recovered Vehicle"
#> [36] "Robbery"
#> [37] "Sex Offense"
#> [38] "Stolen Property"
#> [39] "Suicide"
#> [40] "Suspicious"
#> [41] "Suspicious Occ"
#> [42] "Traffic Collision"
#> [43] "Traffic Violation Arrest"
#> [44] "Vandalism"
#> [45] "Vehicle Impounded"
#> [46] "Vehicle Misplaced"
#> [47] "Warrant"
#> [48] "Weapons"
```

```
#> [49] "Weapons"
#> [50] "Weapons"
```

And now our last three crimes are all identical.

4.3.9 Multiple patterns |

The vertical bar | special character allows us to check for multiple patterns. It essentially functions as “pattern A or Pattern B” with the | symbol replacing the word “or” (and making sure to not have any space between patterns.). To check our crimes for the word “Drug” or the word “Weapons” we could write “Drug|Weapon” which searches for “Drug” or “Weapons” in the text.

```
grep("Drug|Weapons", crimes, value = TRUE)
#> [1] "Drug Offense"           "Drug Violation"          "Weapons Carrying Etc"
#> [4] "Weapons Offence"        "Weapons Offense"
```

4.3.10 Parentheses ()

Parentheses act similar to the square brackets [] where we want everything inside but with parentheses the values must be in the proper order.

```
grep("(Offense)", crimes, value = TRUE)
#> [1] "Drug Offense"           "Family Offense"         "Juvenile Offenses"
#> [4] "Other Offenses"         "Sex Offense"           "Weapons Offense"
```

Running the above code returns the same results as if we didn’t include the parentheses. The usefulness of parentheses comes when combining it with the | symbol to be able to check “(X|Y) Z”), which says, “look for either X or Y which must be followed by Z”.

Running just “(Offense)” returns values for multiple types of offenses. Let’s say we just care about Drug and Weapon Offenses. We can search for “Offense” normally and combine () and | to say, “search for either the word “Drug” or the word “Family” and they should be followed by the word “Offense”.

```
grep("(Drug|Weapons) Offense", crimes, value = TRUE)
#> [1] "Drug Offense"           "Weapons Offense"
```

4.3.11 Optional text ?

The question mark indicates that the character immediately before the ? is optional.

Let's search for the term "offens" and add a ? at the end. This says search for the pattern "offen" and we expect an exact match for that pattern. And if the letter "s" follows "offen" return that too, but it isn't required to be there.

```
grep("Offens?", crimes, value = TRUE)
#> [1] "Drug Offense"
#> [2] "Family Offense"
#> [3] "Juvenile Offenses"
#> [4] "Offences Against The Family And Children"
#> [5] "Other Offenses"
#> [6] "Sex Offense"
#> [7] "Weapons Offence"
#> [8] "Weapons Offense"
```

We can further combine it with () and | to get both spellings of Weapon Offense.

```
grep("(Drug|Weapons) Offens?", crimes, value = TRUE)
#> [1] "Drug Offense"    "Weapons Offence" "Weapons Offense"
```

4.4 Changing capitalization

If you're dealing with data where the only difference is capitalization (as is common in crime data) instead of using `gsub()` to change individual values, you can use the functions `toupper()` and `tolower()` to change every letter's capitalization. These functions take as an input a vector of strings (or a column from a `data.frame`) and return those strings either upper or lowercase.

```
toupper(crimes)
#> [1] "ARSON"
#> [2] "ASSAULT"
#> [3] "BURGLARY"
#> [4] "CASE CLOSURE"
```

```
#> [5] "CIVIL SIDEWALKS"
#> [6] "COURTESY REPORT"
#> [7] "DISORDERLY CONDUCT"
#> [8] "DRUG OFFENSE"
#> [9] "DRUG VIOLATION"
#> [10] "EMBEZZLEMENT"
#> [11] "FAMILY OFFENSE"
#> [12] "FIRE REPORT"
#> [13] "FORGERY AND COUNTERFEITING"
#> [14] "FRAUD"
#> [15] "GAMBLING"
#> [16] "HOMICIDE"
#> [17] "HUMAN TRAFFICKING (A), COMMERCIAL SEX ACTS"
#> [18] "HUMAN TRAFFICKING, COMMERCIAL SEX ACTS"
#> [19] "JUVENILE OFFENSES"
#> [20] "LARCENY THEFT"
#> [21] "LIQUOR LAWS"
#> [22] "LOST PROPERTY"
#> [23] "MALICIOUS MISCHIEF"
#> [24] "MISCELLANEOUS INVESTIGATION"
#> [25] "MISSING PERSON"
#> [26] "MOTOR VEHICLE THEFT"
#> [27] "MOTOR VEHICLE THEFT?"
#> [28] "NON-CRIMINAL"
#> [29] "OFFENCES AGAINST THE FAMILY AND CHILDREN"
#> [30] "OTHER"
#> [31] "OTHER MISCELLANEOUS"
#> [32] "OTHER OFFENSES"
#> [33] "PROSTITUTION"
#> [34] "RAPE"
#> [35] "RECOVERED VEHICLE"
#> [36] "ROBBERY"
#> [37] "SEX OFFENSE"
#> [38] "STOLEN PROPERTY"
#> [39] "SUICIDE"
#> [40] "SUSPICIOUS"
#> [41] "SUSPICIOUS OCC"
```

```
#> [42] "TRAFFIC COLLISION"
#> [43] "TRAFFIC VIOLATION ARREST"
#> [44] "VANDALISM"
#> [45] "VEHICLE IMPOUNDED"
#> [46] "VEHICLE MISPLACED"
#> [47] "WARRANT"
#> [48] "WEAPONS CARRYING ETC"
#> [49] "WEAPONS OFFENCE"
#> [50] "WEAPONS OFFENSE"

tolower(crimes)
#> [1] "arson"
#> [2] "assault"
#> [3] "burglary"
#> [4] "case closure"
#> [5] "civil sidewalks"
#> [6] "courtesy report"
#> [7] "disorderly conduct"
#> [8] "drug offense"
#> [9] "drug violation"
#> [10] "embezzlement"
#> [11] "family offense"
#> [12] "fire report"
#> [13] "forgery and counterfeiting"
#> [14] "fraud"
#> [15] "gambling"
#> [16] "homicide"
#> [17] "human trafficking (a), commercial sex acts"
#> [18] "human trafficking, commercial sex acts"
#> [19] "juvenile offenses"
#> [20] "larceny theft"
#> [21] "liquor laws"
#> [22] "lost property"
#> [23] "malicious mischief"
#> [24] "miscellaneous investigation"
#> [25] "missing person"
#> [26] "motor vehicle theft"
```

```
#> [27] "motor vehicle theft?"  
#> [28] "non-criminal"  
#> [29] "offences against the family and children"  
#> [30] "other"  
#> [31] "other miscellaneous"  
#> [32] "other offenses"  
#> [33] "prostitution"  
#> [34] "rape"  
#> [35] "recovered vehicle"  
#> [36] "robbery"  
#> [37] "sex offense"  
#> [38] "stolen property"  
#> [39] "suicide"  
#> [40] "suspicious"  
#> [41] "suspicious occ"  
#> [42] "traffic collision"  
#> [43] "traffic violation arrest"  
#> [44] "vandalism"  
#> [45] "vehicle impounded"  
#> [46] "vehicle misplaced"  
#> [47] "warrant"  
#> [48] "weapons carrying etc"  
#> [49] "weapons offence"  
#> [50] "weapons offense"
```

Chapter 5

Reading and Writing Data

So far in these lessons we've used data from a number of sources but which all came as .rda files which is the standard R data format. Many data sets, particularly older government data, will not come as .rda file but rather as Excel, Stata, SAS, SPSS, or fixed-width ASCII files. In this brief lesson we'll cover how to read these formats into R as well as how to save data into these formats. Since many criminologists do not use R, it is important to be able to save the data in the language they use to be able to collaborate with them.

Fixed-width ASCII files are not very common and require a bit more effort than the other formats so we'll leave those until later to discuss.

In this lesson we'll use data about officer-involved shootings.

5.1 Reading Data into R

5.1.1 R

As we've seen earlier, to read in data with a .rda or .rdata extension you use the function `load()` with the file name (including the extension) in quotation marks inside of the parentheses. This loads the data into R and calls the object the name it was when it was saved. Therefore we do not need to give it a name ourselves.

For each of the other types of data we'll need to assign a name to the data

we're reading in so it has a name. Whereas we've done `x <- 2` to say *x* gets the value of 2, now we'd do `x <- DATA` where DATA is the way to load in the data and *x* will get the entire data.frame that is read in.

5.1.2 Excel

To read in Excel files, those ending in .csv, we can use the function `read_csv()` from the package `readr` (the function `read.csv()` is included in R by default so it doesn't require any packages but is far slower than `read_csv()` so we will not use it).

```
install.packages("readr")
```

```
library(readr)
```

The input in the () is the file name ending in “.csv”. As it is telling R to read a file that is stored on your computer, the whole name must be in quotes. Unlike loading an .rda file using `load()`, there is no name for the object that gets read in so we must assign the data a name. We can use the name *shootings* as it's relatively descriptive and easy for us to write.

```
shootings <- read_csv("data/fatal-police-shootings-data.csv")
#> Parsed with column specification:
#> cols(
#>   id = col_double(),
#>   name = col_character(),
#>   date = col_date(format = ""),
#>   manner_of_death = col_character(),
#>   armed = col_character(),
#>   age = col_double(),
#>   gender = col_character(),
#>   race = col_character(),
#>   city = col_character(),
#>   state = col_character(),
#>   signs_of_mental_illness = col_logical(),
#>   threat_level = col_character(),
#>   flee = col_character(),
#>   body_camera = col_logical()
#> )
```

`read_csv()` also reads in data to an object called a `tibble` which is very similar to a `data.frame` but has some differences in displaying the data. If we run `head()` on the data it doesn't show all columns. This is useful to avoid accidentally printing out a massive amounts of columns.

```
head(shootings)
#> # A tibble: 6 x 14
#>   id     name    date manner_of_death armed   age gender race city state
#>   <dbl> <chr>   <date> <chr>        <chr> <dbl> <chr> <chr> <chr> <chr>
#> 1     3 Tim ~ 2015-01-02 shot       gun      53 M    A   Shel~ WA
#> 2     4 Lewi~ 2015-01-02 shot       gun      47 M    W   Aloha OR
#> 3     5 John~ 2015-01-03 shot and Taser~ unar~  23 M    H   Wich~ KS
#> 4     8 Matt~ 2015-01-04 shot       toy ~  32 M    W   San ~ CA
#> 5     9 Mich~ 2015-01-04 shot       nail~  39 M    H   Evans CO
#> 6    11 Kenn~ 2015-01-04 shot       gun      18 M    W   Guth~ OK
#> # ... with 4 more variables: signs_of_mental_illness <lgl>, threat_level <chr>,
#> #   flee <chr>, body_camera <lgl>
```

We can convert it to a `data.frame` using the function `as.data.frame()` though that isn't strictly necessary since `tibbles` and `data.frames` operate so similarly.

```
shootings <- as.data.frame(shootings)
```

5.1.3 Stata

For the remaining three data types we'll use the package `haven`.

```
install.packages("haven")
```

```
library(haven)
```

`haven` follows the same syntax for each data type and is the same as with `read_csv()` - for each data type we simply include the file name (in quotes, with the extension) and designate an name to be assigned the data.

Like with `read_csv()` the functions to read data through `haven` all start with `read_` and end with the extension you're reading in.

- `read_dta()` - Stata file, extension “dta”
- `read_sas()` - SAS file, extension “sas”

- `read_sav()` - SPSS file, extension “sav”

To read the data as a .dta format we can copy the code to read it as a .csv but change .csv to .dta.

```
shootings <- read_dta("data/fatal-police-shootings-data.dta")
```

Since we called this new data *shootings*, R overwrote that object (without warning us!). This is useful because we often want to subset or aggregate data and call it by the same name to avoid making too many objects to keep track of, but watch out for accidentally overwriting an object without noticing!

5.1.4 SAS

```
shootings <- read_sas("data/fatal-police-shootings-data.sas")
```

5.1.5 SPSS

```
shootings <- read_sav("data/fatal-police-shootings-data.sav")
```

5.2 Writing Data

When we’re done with a project (or an important part of a project) or when we need to send data to someone, we need to save the data we’ve worked on in a suitable format. For each format, we are saving the data in we will follow the same syntax of

```
function_name(data, "file_name")
```

As usual we start with the function name. Then inside the parentheses we have the name of the object we are saving (as it refers to an object in R, we do not use quotations) and then the file name, in quotes, ending with the extension you want.

For saving an .rda file we use the `save()` function, otherwise we follow the syntax of `write_` ending with the file extension.

- `write_csv()` - Excel file, extension “csv”

- `write_dta()` - Stata file, extension “dta”
- `write_sas()` - SAS file, extension “sas”
- `write_sav()` - SPSS file, extension “sav”

As with reading the data, `write_csv()` comes from the `readr` package while the other formats are from the `haven` package.

5.2.1 R

For saving an .rda file we must set the parameter `file` to be the name we’re saving. For the other types of data they use the parameter `path` rather than `file` but it is not necessary to call them explicitly.

```
save(shootings, file = "data/shootings.rda")
```

5.2.2 Excel

```
write_csv(shootings, "data/shootings.csv")
```

5.2.3 Stata

```
write_dta(shootings, "data/shootings.dta")
```

5.2.4 SAS

```
write_sas(shootings, "data/shootings.sas")
```

5.2.5 SPSS

```
write_sav(shootings, "data/shootings.sav")
```


Part II

Visualize

Chapter 6

Graphing with ggplot2

We've made some simple graphs earlier; in this lesson we will use the package `ggplot2` to make simple and elegant looking graphs.

The ‘gg’ part of `ggplot2` stands for ‘grammar of graphics’ which is the idea that most graphs can be made using the same few ‘pieces.’ We’ll get into those pieces during this lesson. For a useful cheat sheet for this package see [here](#)

```
install.packages("ggplot2")  
library(ggplot2)
```

When working with new data, It’s often useful to quickly graph the data to try to understand what you’re working with. It is also useful when understanding how much to trust the data.

The data we will work on is data about alcohol consumption in U.S. states from 1977-2017 from the National Institute of Health. It contains the per capita alcohol consumption for each state for every year. Their method to determine per capita consumption is amount of alcohol sold / number of people aged 14+ living in the state. More details on the data are available [here](#).

Now we need to load the data.

```
load("data/apparent_per_capita_alcohol_consumption.rda")
```

The name of the data is quite long so for convenience let's copy it to a new object with a better name, *alcohol*.

```
alcohol <- apparent_per_capita_alcohol_consumption
```

The original data has every state, region, and the US as a whole. For this lesson we're using data subsetted to just include states. For now let's just look at Pennsylvania.

```
penn_alcohol <- alcohol[alcohol$state == "pennsylvania", ]
```

6.1 What does the data look like?

Before graphing, it's helpful to see what the data includes. An important thing to check is what variables are available and the units of these variables.

```
names(penn_alcohol)
#> [1] "state"
#> [2] "year"
#> [3] "ethanol_beer_gallons_per_capita"
#> [4] "ethanol_wine_gallons_per_capita"
#> [5] "ethanol_spirit_gallons_per_capita"
#> [6] "ethanol_all_drinks_gallons_per_capita"
#> [7] "number_of_beers"
#> [8] "number_of_glasses_wine"
#> [9] "number_of_shots_liquor"
#> [10] "number_of_drinks_total"

summary(penn_alcohol)
#> state year ethanol_beer_gallons_per_capita
#> Length:41 Length:41 Min. :1.210
#> Class :character Class :character 1st Qu.:1.310
#> Mode :character Mode :character Median :1.350
#> Mean :1.344
#> 3rd Qu.:1.380
#> Max. :1.450
```

```

#>   ethanol_wine_gallons_per_capita ethanol_spirit_gallons_per_capita
#> Min.    :0.1700                  Min.    :0.4500
#> 1st Qu.:0.1900                  1st Qu.:0.5100
#> Median  :0.2100                  Median  :0.6100
#> Mean    :0.2276                  Mean    :0.5939
#> 3rd Qu.:0.2500                  3rd Qu.:0.6800
#> Max.    :0.3300                  Max.    :0.7400
#> 
#>   ethanol_all_drinks_gallons_per_capita number_of_beers number_of_glasses_wine
#> Min.    :1.850                   Min.    :286.8   Min.    :33.74
#> 1st Qu.:2.040                   1st Qu.:310.5   1st Qu.:37.71
#> Median  :2.220                   Median  :320.0   Median  :41.67
#> Mean    :2.167                   Mean    :318.7   Mean    :45.16
#> 3rd Qu.:2.330                   3rd Qu.:327.1   3rd Qu.:49.61
#> Max.    :2.390                   Max.    :343.7   Max.    :65.49
#> 
#>   number_of_shots_liquor number_of_drinks_total
#> Min.    : 93.43                 Min.    :394.7
#> 1st Qu.:105.89                 1st Qu.:435.2
#> Median  :126.65                 Median  :473.6
#> Mean    :123.31                 Mean    :462.3
#> 3rd Qu.:141.18                 3rd Qu.:497.1
#> Max.    :153.64                 Max.    :509.9

head(penn_alcohol)
#>           state year ethanol_beer_gallons_per_capita
#> 1559 pennsylvania 2017          1.29
#> 1560 pennsylvania 2016          1.31
#> 1561 pennsylvania 2015          1.31
#> 1562 pennsylvania 2014          1.32
#> 1563 pennsylvania 2013          1.34
#> 1564 pennsylvania 2012          1.36
#> 
#>   ethanol_wine_gallons_per_capita ethanol_spirit_gallons_per_capita
#> 1559                      0.33                      0.71
#> 1560                      0.33                      0.72
#> 1561                      0.32                      0.70
#> 1562                      0.32                      0.70
#> 1563                      0.31                      0.68
#> 1564                      0.31                      0.67

```

	<i>ethanol_all_drinks_gallons_per_capita</i>	<i>number_of_beers</i>	
#> 1559		2.34	305.7778
#> 1560		2.36	310.5185
#> 1561		2.33	310.5185
#> 1562		2.34	312.8889
#> 1563		2.33	317.6296
#> 1564		2.34	322.3704
	<i>number_of_glasses_wine</i>	<i>number_of_shots_liquor</i>	<i>number_of_drinks_total</i>
#> 1559	65.48837	147.4128	499.2000
#> 1560	65.48837	149.4891	503.4667
#> 1561	63.50388	145.3366	497.0667
#> 1562	63.50388	145.3366	499.2000
#> 1563	61.51938	141.1841	497.0667
#> 1564	61.51938	139.1079	499.2000

So each row of the data is a single year of data for Pennsylvania. It includes alcohol consumption for wine, liquor, beer, and total drinks - both as gallons of ethanol (a hard unit to interpret) and more traditional measures such as glasses of wine or number of beers. The original data only included the gallons of ethanol data which I converted to the more understandable units. If you encounter data with odd units, it is a good idea to convert it to something easier to understand - especially if you intend to show someone else the data or results!

6.2 Graphing data

To make a plot using `ggplot()`, all you need to do is specify the data set and the variables you want to plot. From there you add on pieces of the graph using the `+` symbol and then specify what you want added.

For `ggplot()` we need to specify 4 things

1. The data set
2. The x-axis variable
3. The y-axis variable
4. The type of graph - e.g. line, point, etc.

Some useful types of graphs are

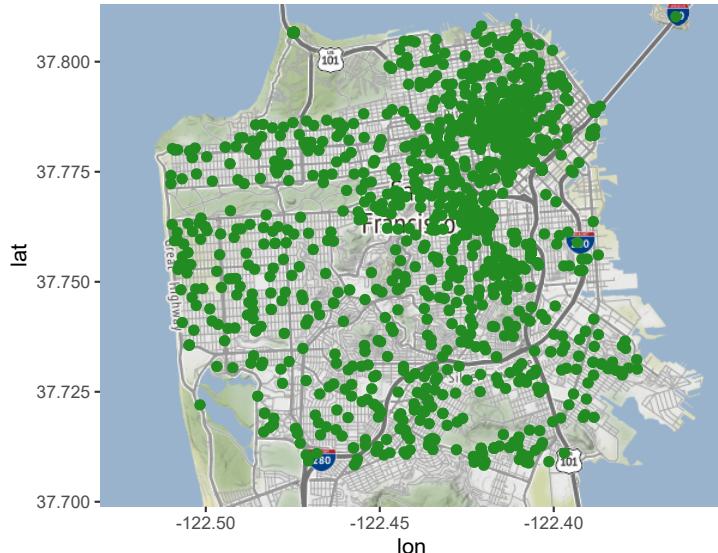
- `geom_point()` - A point graph, can be used for scatter plots
- `geom_line()` - A line graph
- `geom_smooth()` - Adds a regression line to the graph
- `geom_bar()` - A barplot

6.3 Time-Series Plots

Let's start with a time-series of beer consumption in Pennsylvania. In time-series plots the x-axis is always the time variable while the y-axis is the variable whose trend over time is what we're interested in. When you see a graph showing crime rates over time, this is the type of graph you're looking at.

The code below starts by writing our data set name. Then says what our x- and y-axis variables are called. The x- and y-axis variables are within parentheses of the function called `aes()`. `aes()` stands for aesthetic and what's included inside here describes how the graph will look. It's not intuitive to remember, but you need to include it.

```
ggplot(penn_alcohol, aes(x = year,  
                           y = number_of_beers))
```

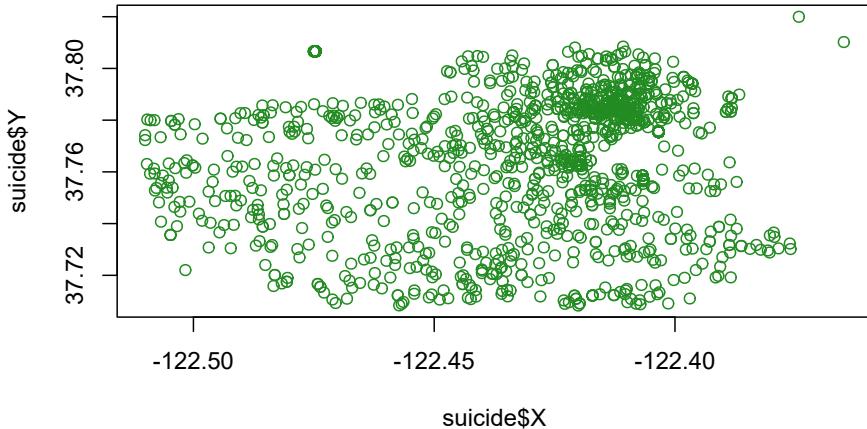


Note that on the x-axis it prints out every single year and makes it completely

unreadable. That is because the “year” column is a character type, so R thinks each year is its own category. It prints every single year because it thinks we want every category shown. To fix this we can make the column numeric and `ggplot()` will be smarter about printing fewer years.

```
penn_alcohol$year <- as.numeric(penn_alcohol$year)
```

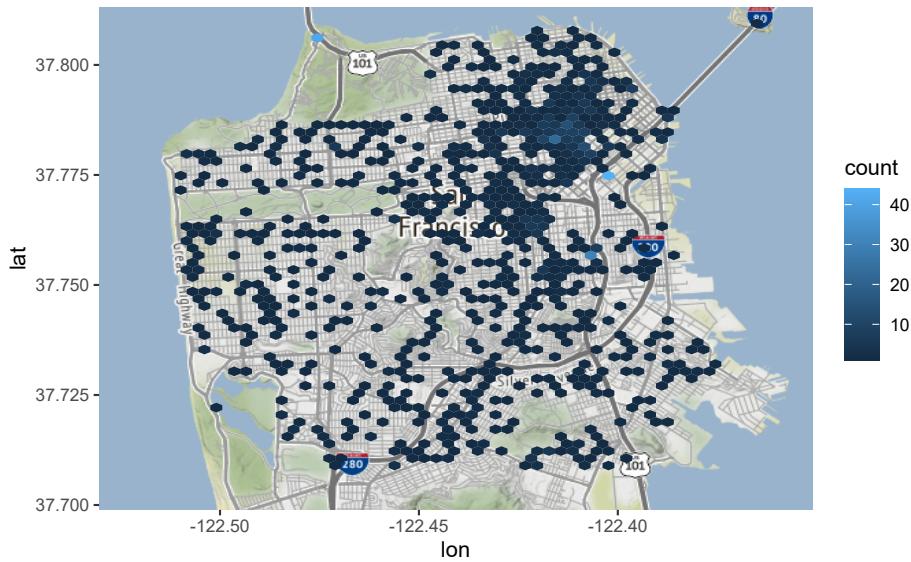
```
ggplot(penn_alcohol, aes(x = year,
                           y = number_of_beers))
```



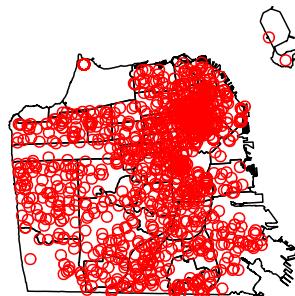
When we run it, we get our graph. It includes the variable names for each axis and shows the range of data through the tick marks. What is missing is the actual data. For that we need to specify what type of graph it is. We literally add it with the `+` followed by the type of graph we want. Make sure that the `+` is at the end of a line, not the start of one. Starting a line with the `+` will not work.

Let’s start with point and line graphs.

```
ggplot(penn_alcohol, aes(x = year, y = number_of_beers)) +
  geom_point()
```



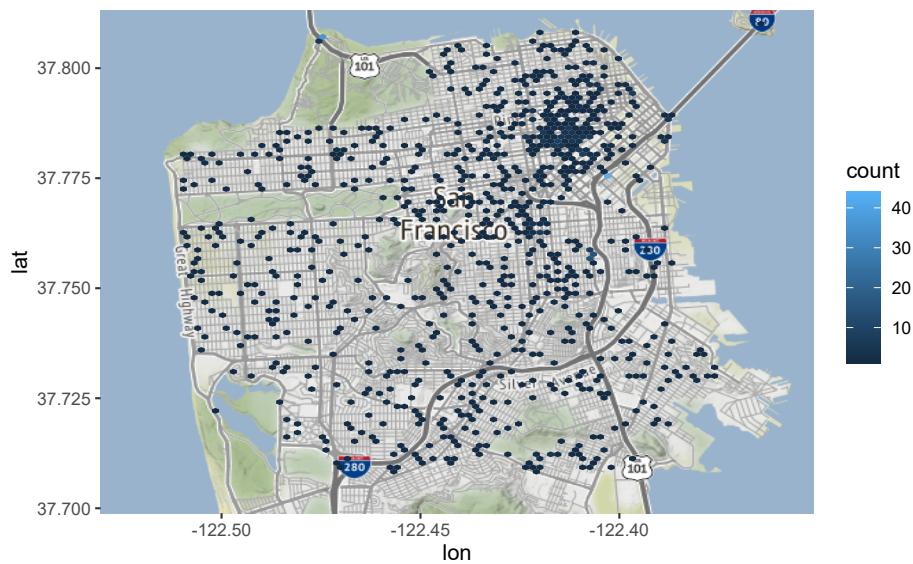
```
ggplot(penn_alcohol, aes(x = year, y = number_of_beers)) +  
  geom_line()
```



We can also combine different types of graphs.

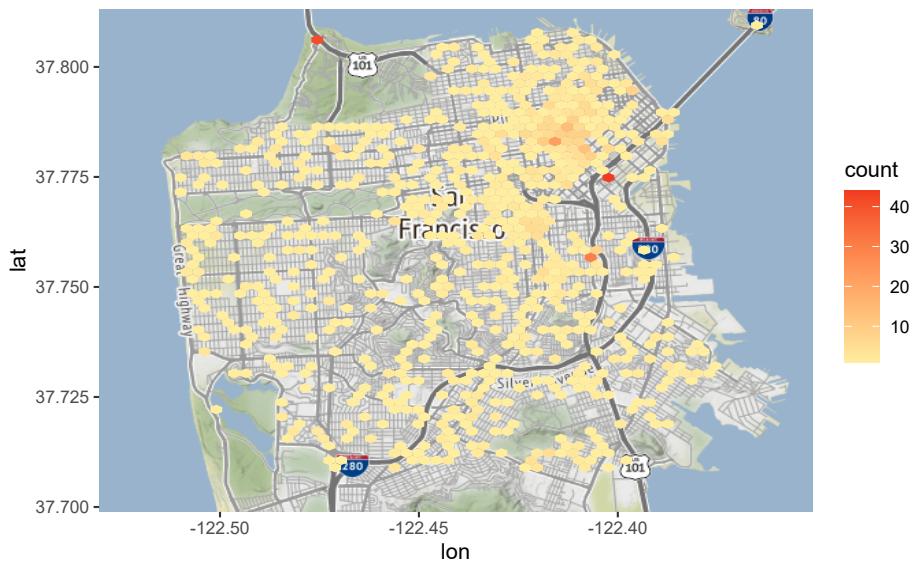
```
ggplot(penn_alcohol, aes(x = year, y = number_of_beers)) +  
  geom_point() +
```

```
geom_line()
```



It looks like there's a huge change in beer consumption over time. But look at where they y-axis starts. It starts around 280 so really that change is only ~60 beers. That's because when graphs don't start at 0, it can make small changes appear big. We can fix this by forcing the y-axis to begin at 0. We can add `expand_limits(y = 0)` to the graph to say that the value 0 must always appear on the y-axis, even if no data is close to that value.

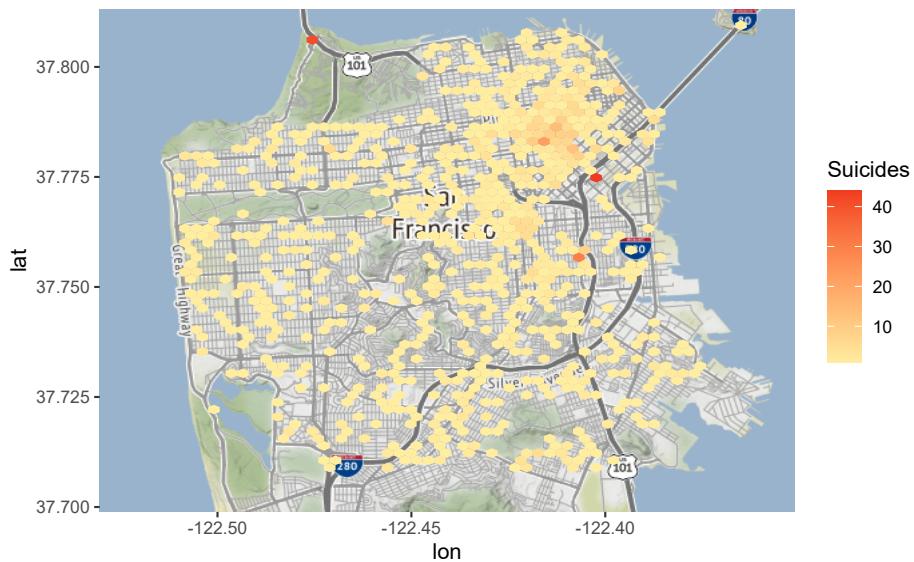
```
ggplot(penn_alcohol, aes(x = year, y = number_of_beers)) +
  geom_point() +
  geom_line() +
  expand_limits(y = 0)
```



Now that graph shows what looks like nearly no change even though that is also not true. Which graph is best? It's hard to say.

Inside the types of graphs we can change how it is displayed. As with using `plot()`, we can specify the color and size of our lines or points.

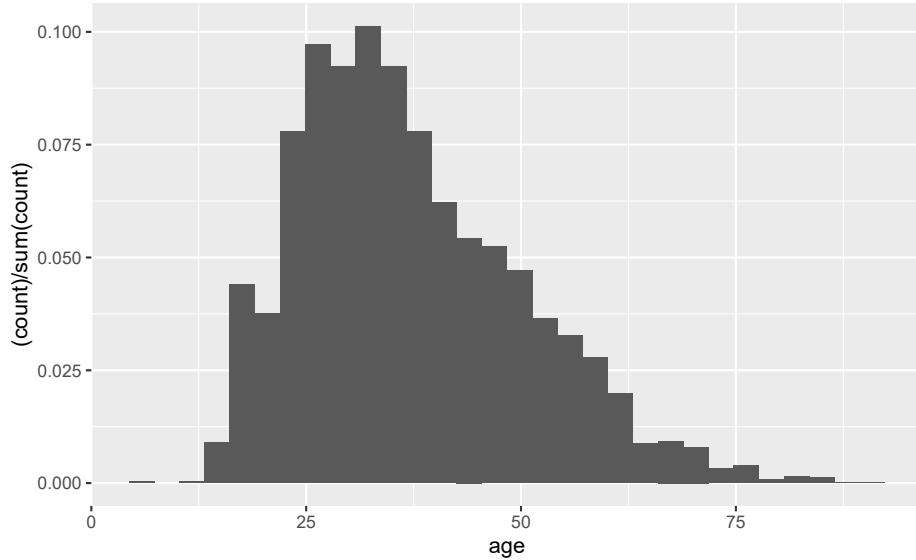
```
ggplot(penn_alcohol, aes(x = year, y = number_of_beers)) +  
  geom_line(color = "forestgreen", size = 1.3)
```



Some other useful features are changing the axis labels and the graph title. Unlike in `plot()` we do not need to include it in the () of `ggplot()` but use their own functions to add them to the graph.

- `xlab()` - x-axis label
- `ylab()` - y-axis label
- `ggttitle()` - graph title

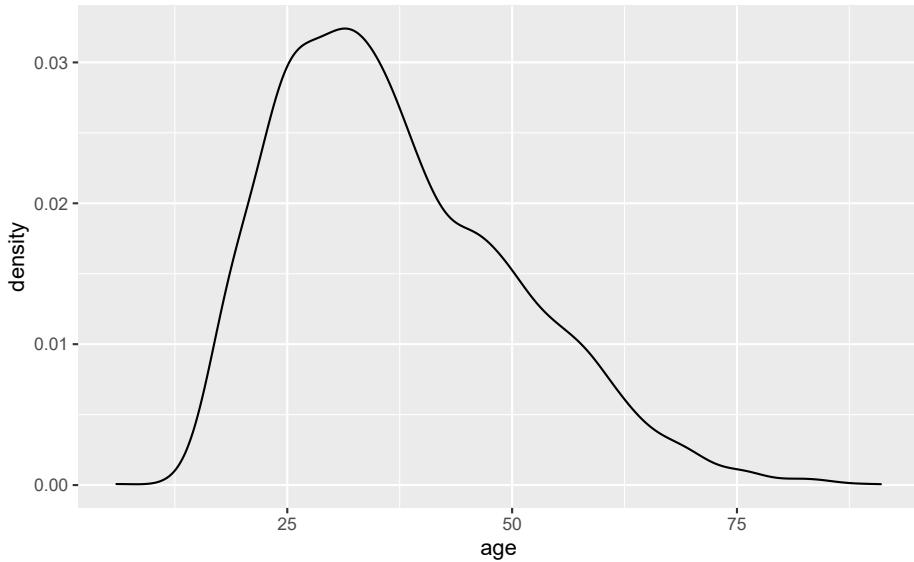
```
ggplot(penn_alcohol, aes(x = year, y = number_of_beers)) +
  geom_line(color = "forestgreen", size = 1.3) +
  xlab("Year") +
  ylab("Number of Beers") +
  ggttitle("PA Annual Beer Consumption Per Capita (1977-2017)")
```



Many time-series plots show multiple variables over the same time period (e.g. murder and robbery over time). There are ways to change the data itself to make creating graphs like this easier, but let's stick with the data we currently have and just change `ggplot()`.

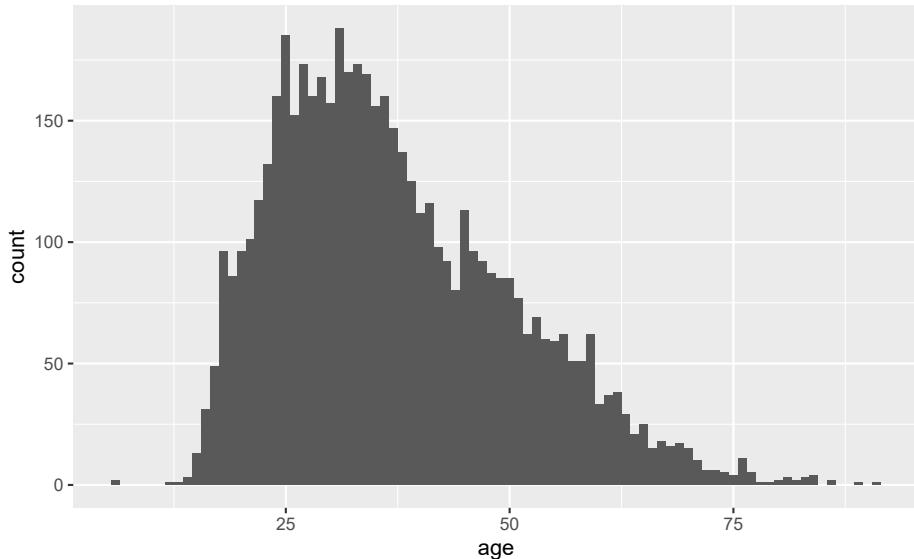
Start with a normal line graph, this time looking at wine.

```
ggplot(penn_alcohol, aes(x = year, y = number_of_glasses_wine)) +  
  geom_line()
```



Then include a second geom_line() with its own aes() for the second variable.

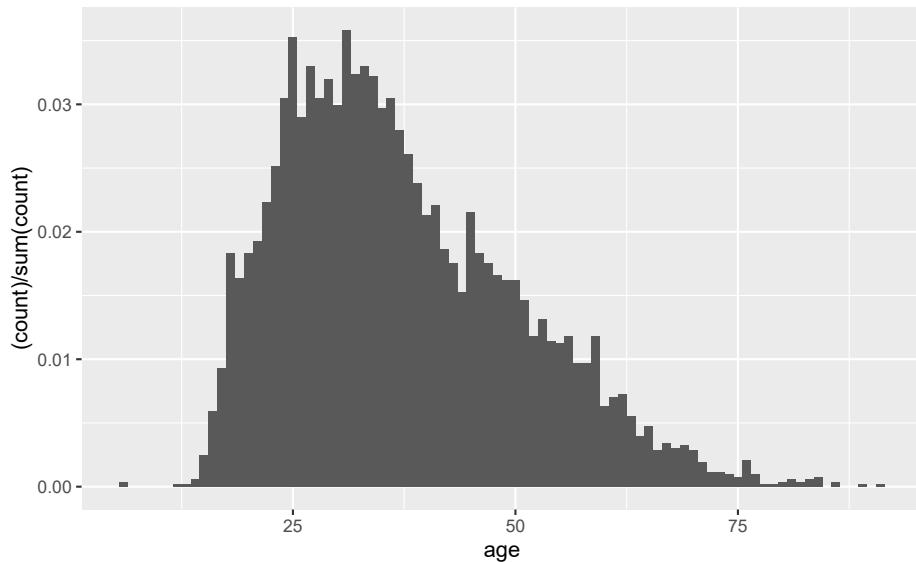
```
ggplot(penn_alcohol, aes(x = year, y = number_of_glasses_wine)) +
  geom_line() +
  geom_line(aes(x = year, y = number_of_shots_liquor))
```



A problem with this is that both lines are the same color. We need to set

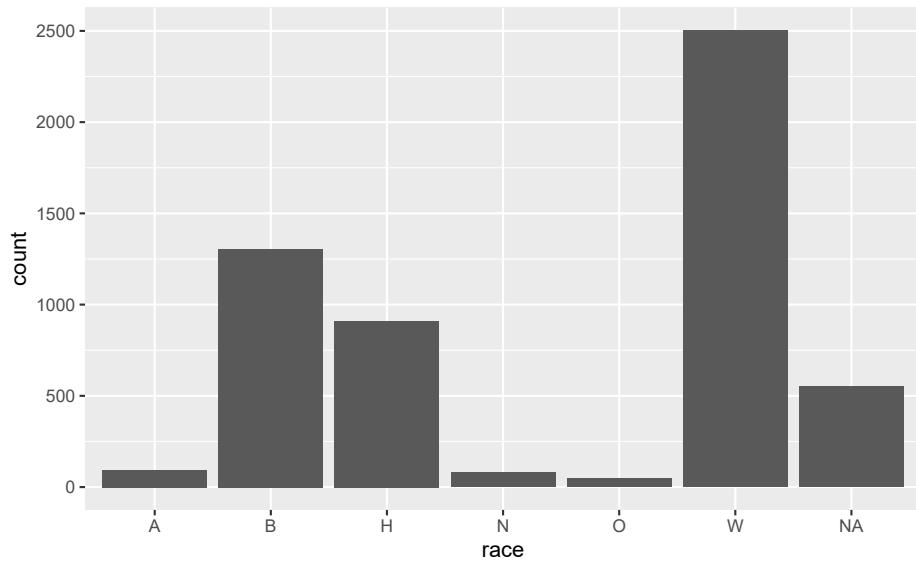
a color for each line and do so within `aes()`. Instead of providing a color name, we need to provide the name the color will have in the legend. Do so for both lines.

```
ggplot(penn_alcohol, aes(x = year, y = number_of_glasses_wine,
                          color = "Glasses of Wine")) +
  geom_line() +
  geom_line(aes(x = year, y = number_of_shots_liquor,
                color = "Shots of Liquor"))
```



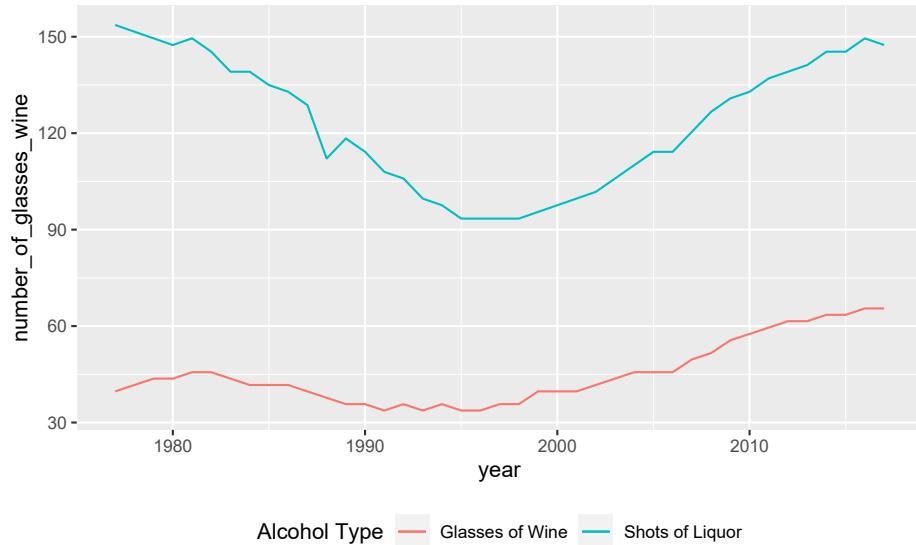
We can change the legend title by using the function `labs()` and changing the value `color` to what we want the legend title to be.

```
ggplot(penn_alcohol, aes(x = year, y = number_of_glasses_wine,
                          color = "Glasses of Wine")) +
  geom_line() +
  geom_line(aes(x = year, y = number_of_shots_liquor,
                color = "Shots of Liquor")) +
  labs(color = "Alcohol Type")
```



Finally, a useful option to move the legend from the side to the bottom is setting the `theme()` function to move the `legend.position` to “bottom”. This will allow the graph to be wider.

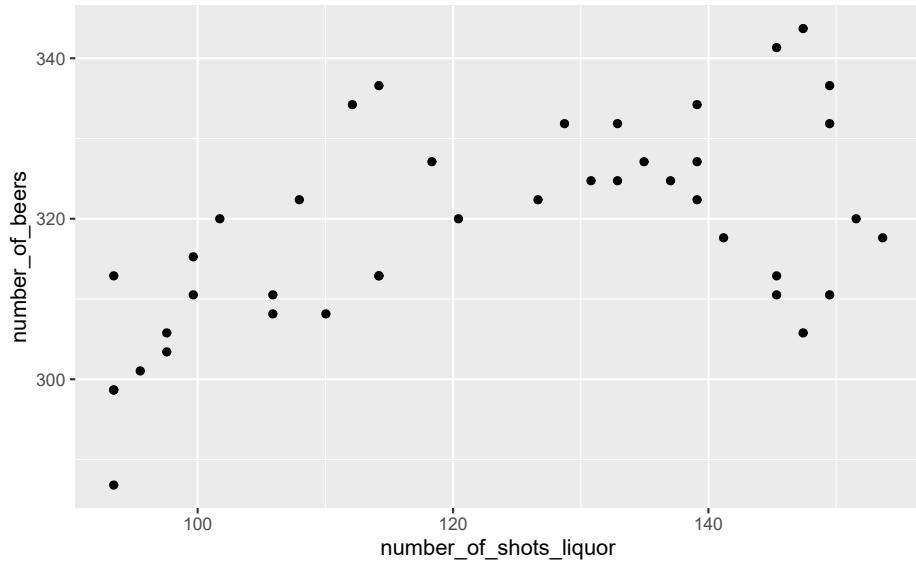
```
ggplot(penn_alcohol, aes(x = year, y = number_of_glasses_wine,
                           color = "Glasses of Wine")) +
  geom_line() +
  geom_line(aes(x = year, y = number_of_shots_liquor,
                color = "Shots of Liquor")) +
  labs(color = "Alcohol Type") +
  theme(legend.position = "bottom")
```



6.4 Scatter Plots

Making a scatter plot simply requires changing the x-axis from year to another numerical variable and using `geom_point()`.

```
ggplot(penn_alcohol, aes(x = number_of_shots_liquor,  
                           y = number_of_beers)) +  
  geom_point()
```

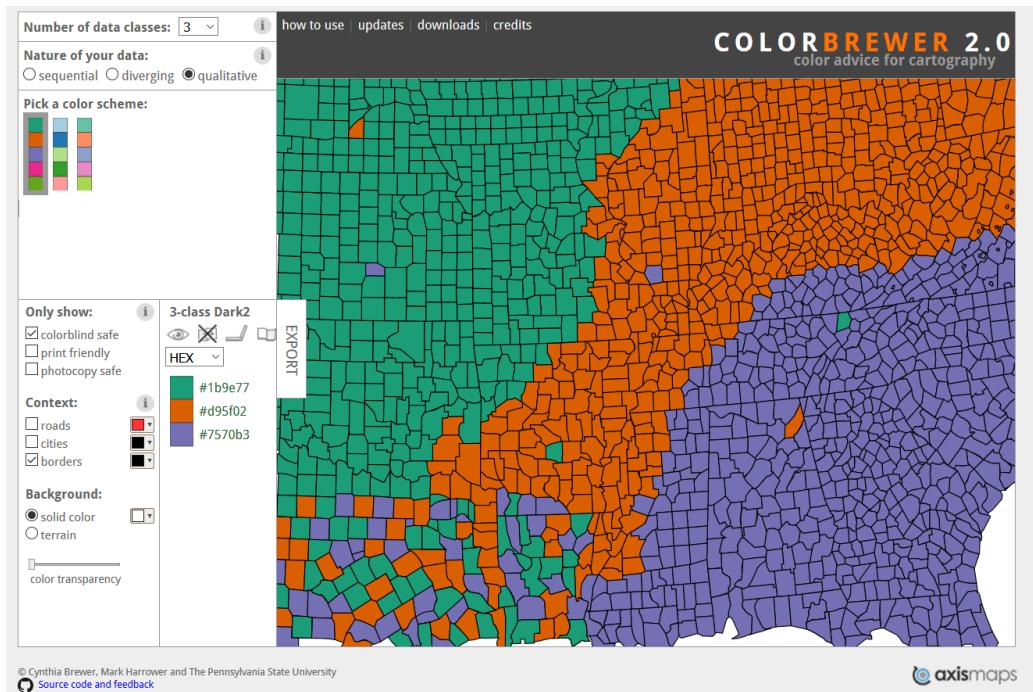


This graph shows us that when liquor consumption increases, beer consumption also tends to increase.

While scatterplots can help show the relationship between variables, we lose the information of how consumption changes over time.

6.5 Color blindness

Please keep in mind that some people are color blind so graphs (or maps which we will learn about soon) will be hard to read for these people if we choose the incorrect colors. A helpful site for choosing colors for graphs is colorbrewer2.org

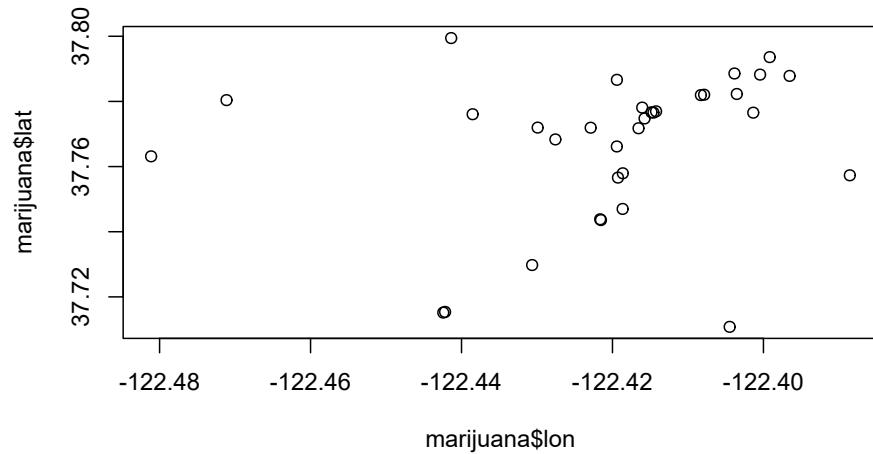


This site lets you select which type of colors you want (sequential and diverging such as shades in a hotspot map, and qualitative such as for data like what we used in this lesson). In the “Only show:” section you can set it to “colorblind safe” to restrict it to colors that allow people with color blindness to read your graph. To the right of this section it shows the HEX codes for each color (a HEX code is just a code that a computer can read and know exactly which color it is).

Let’s use an example of a color blind friendly color from the “qualitative” section of ColorBrewer. We have three options on this page (we can change how many colors we want but it defaults to showing 3): green (HEX = #1b9e77), orange (HEX = #d95f02), and purple (HEX = #7570b3). We’ll use the orange and purple colors. To manually set colors in `ggplot()` we use `scale_color_manual(values = c())` and include a vector of color names or HEX codes inside the `c()`. Doing that using the orange and purple HEX codes will change our graph colors to these two colors.

```
ggplot(penn_alcohol, aes(x = year, y = number_of_glasses_wine,
                         color = "Glasses of Wine")) +
  geom_line() +
```

```
geom_line(aes(x = year, y = number_of_shots_liquor,
               color = "number_of_shots_liquor")) +
  labs(color = "Alcohol Type") +
  theme(legend.position = "bottom") +
  scale_color_manual(values = c("#7570b3", "#d95f02"))
```



Chapter 7

More graphing with ggplot2

In this lesson we will continue to explore graphing using `ggplot()`. The data we will use is a database of officer-involved shootings that result in a death in the United States since January 1st, 2015. This data has been compiled and released by the Washington Post so it will be a useful exercise in exploring data from non-government sources. This data is useful for our purposes as it has a number of variables related to the person who was shot, allowing us to practice making many types of graphs.

To explore the data on their website, see [here](#). To examine their methodology, see [here](#).

The data initially comes as a .csv file so we'll use the `read_csv()` function from the `readr` package. Since it's available on GitHub, we can download it by directing `read_csv()` to read the file at its URL on GitHub.

```
library(readr)
shootings <- read_csv("https://raw.githubusercontent.com/washingtonpost/data-police-shootings/master/shootings.csv")
#> Parsed with column specification:
#> cols(
#>   id = col_double(),
#>   name = col_character(),
#>   date = col_date(format = ""),
#>   manner_of_death = col_character(),
#>   armed = col_character(),
#>   age = col_double(),
```

```
#>   gender = col_character(),
#>   race = col_character(),
#>   city = col_character(),
#>   state = col_character(),
#>   signs_of_mental_illness = col_logical(),
#>   threat_level = col_character(),
#>   flee = col_character(),
#>   body_camera = col_logical()
#> )
```

Since `read_csv()` reads files into a tibble object, we'll turn it into a `data.frame` so `head()` shows every single column.

```
shootings <- as.data.frame(shootings)
```

7.1 Exploring Data

Now that we have the data read in, let's look at it.

```
nrow(shootings)
#> [1] 5489
ncol(shootings)
#> [1] 14
```

The data has 14 variables and covers 5489 shootings. Let's check out some of the variables, first using `head()` then using `summary()` and `table()`.

```
head(shootings)
#>   id           name      date manner_of_death      armed age gender
#> 1  3     Tim Elliot 2015-01-02       shot      gun  53    M
#> 2  4 Lewis Lee Lembke 2015-01-02       shot      gun  47    M
#> 3  5 John Paul Quintero 2015-01-03 shot and Tasered unarmed  23    M
#> 4  8 Matthew Hoffman 2015-01-04       shot toy weapon  32    M
#> 5  9 Michael Rodriguez 2015-01-04       shot nail gun   39    M
#> 6 11 Kenneth Joe Brown 2015-01-04       shot      gun  18    M
#>   city state signs_of_mental_illness threat_level      flee
#> 1  Shelton    WA          TRUE      attack Not fleeing
#> 2  Aloha     OR         FALSE      attack Not fleeing
```

```
#> 3      Wichita    KS          FALSE   other Not fleeing
#> 4 San Francisco CA          TRUE    attack Not fleeing
#> 5      Evans     CO          FALSE   attack Not fleeing
#> 6      Guthrie   OK          FALSE   attack Not fleeing
#> body_camera
#> 1      FALSE
#> 2      FALSE
#> 3      FALSE
#> 4      FALSE
#> 5      FALSE
#> 6      FALSE
```

Each row is a single shooting and it includes variables such as the victim's name, the date of the shooting, demographic information about that person, the city and state where the shooting occurred, and some information about the incident. It is clear from these first 6 rows that most variables are categorical so we can't use `summary()` on them. Let's use `summary()` on the date and age columns and then use `table()` for the rest.

```
summary(shootings$date)
#>      Min.    1st Qu.    Median    Mean    3rd Qu.    Max.
#> "2015-01-02" "2016-05-21" "2017-10-19" "2017-10-12" "2019-03-05" "2020-07-28"
summary(shootings$age)
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#>     6.00  27.00  35.00 37.12  46.00  91.00 241
```

From this we can see that the data is from early January through about a week ago. From the age column we can see that the average age is about 37 with most people around that range. Now we can use `table()` to see how often each value appears in each variable. We don't want to do this for city or name as there would be too many values, but it will work for the other columns. Let's start with the "manner_of_death" column.

```
table(shootings$manner_of_death)
#>
#>      shot shot and Tasered
#>      5215            274
```

To turn these counts into percentages we can divide the results by the number

of rows in our data and multiply by 100.

```
table(shootings$manner_of_death) / nrow(shootings) * 100
#>
#>           shot shot and Tasered
#> 95.008198      4.991802
```

Now it is clear to see that in about 95% of shootings, officers used a gun and in 5% of shootings they also used a Taser. As this is data on officer shooting deaths, this is unsurprising. Let's take a look at whether the victim was armed.

```
table(shootings$armed) / nrow(shootings) * 100
#>
#>           air conditioner          air pistol
#>           0.01821825          0.01821825
#>           Airsoft pistol          ax
#>           0.01821825          0.43723811
#>           barstool            baseball bat
#>           0.01821825          0.32792858
#>           baseball bat and bottle baseball bat and fireplace poker
#>           0.01821825          0.01821825
#>           baseball bat and knife          baton
#>           0.01821825          0.09109127
#>           bayonet              BB gun
#>           0.01821825          0.09109127
#>           BB gun and vehicle        bean-bag gun
#>           0.01821825          0.01821825
#>           beer bottle          blunt object
#>           0.05465476          0.09109127
#>           bow and arrow          box cutter
#>           0.01821825          0.21861906
#>           brick                car, knife and mace
#>           0.03643651          0.01821825
#>           carjack              chain
#>           0.01821825          0.05465476
#>           chain saw            chainsaw
#>           0.03643651          0.01821825
#>           chair                claimed to be armed
```

#>	0.07287302	0.01821825
#>	<i>contractor's level</i>	<i>cordless drill</i>
#>	0.01821825	0.01821825
#>	<i>crossbow</i>	<i>crowbar</i>
#>	0.16396429	0.07287302
#>	<i>fireworks</i>	<i>flagpole</i>
#>	0.01821825	0.01821825
#>	<i>flashlight</i>	<i>garden tool</i>
#>	0.03643651	0.03643651
#>	<i>glass shard</i>	<i>grenade</i>
#>	0.05465476	0.01821825
#>	<i>gun</i>	<i>gun and car</i>
#>	56.64055383	0.20040080
#>	<i>gun and knife</i>	<i>gun and sword</i>
#>	0.30971033	0.01821825
#>	<i>gun and vehicle</i>	<i>guns and explosives</i>
#>	0.18218255	0.05465476
#>	<i>hammer</i>	<i>hand torch</i>
#>	0.29149208	0.01821825
#>	<i>hatchet</i>	<i>hatchet and gun</i>
#>	0.20040080	0.03643651
#>	<i>ice pick</i>	<i>incendiary device</i>
#>	0.01821825	0.03643651
#>	<i>knife</i>	<i>lawn mower blade</i>
#>	14.64747677	0.03643651
#>	<i>machete</i>	<i>machete and gun</i>
#>	0.81982146	0.01821825
#>	<i>meat cleaver</i>	<i>metal hand tool</i>
#>	0.09109127	0.01821825
#>	<i>metal object</i>	<i>metal pipe</i>
#>	0.07287302	0.23683731
#>	<i>metal pole</i>	<i>metal rake</i>
#>	0.05465476	0.01821825
#>	<i>metal stick</i>	<i>motorcycle</i>
#>	0.05465476	0.01821825
#>	<i>nail gun</i>	<i>oar</i>
#>	0.01821825	0.01821825

#>	<i>pellet gun</i>	<i>pen</i>
#>	0.05465476	0.01821825
#>	<i>pepper spray</i>	<i>pick-axe</i>
#>	0.01821825	0.07287302
#>	<i>piece of wood</i>	<i>pipe</i>
#>	0.09109127	0.10930953
#>	<i>pitchfork</i>	<i>pole</i>
#>	0.03643651	0.03643651
#>	<i>pole and knife</i>	<i>rock</i>
#>	0.03643651	0.10930953
#>	<i>samurai sword</i>	<i>scissors</i>
#>	0.05465476	0.12752778
#>	<i>screwdriver</i>	<i>sharp object</i>
#>	0.23683731	0.23683731
#>	<i>shovel</i>	<i>spear</i>
#>	0.10930953	0.01821825
#>	<i>stapler</i>	<i>straight edge razor</i>
#>	0.01821825	0.07287302
#>	<i>sword</i>	<i>Taser</i>
#>	0.40080160	0.47367462
#>	<i>tire iron</i>	<i>toy weapon</i>
#>	0.01821825	3.49790490
#>	<i>unarmed</i>	<i>undetermined</i>
#>	6.44926216	3.04244853
#>	<i>unknown weapon</i>	<i>vehicle</i>
#>	1.43924212	2.84204773
#>	<i>vehicle and gun</i>	<i>vehicle and machete</i>
#>	0.07287302	0.01821825
#>	<i>walking stick</i>	<i>wasp spray</i>
#>	0.01821825	0.01821825
#>	<i>wrench</i>	
#>	0.01821825	

This is fairly hard to interpret as it is sorted alphabetically when we'd prefer it to be sorted by most common weapon. It also doesn't round the numbers so there are many numbers past the decimal point shown. Let's solve these two issues using `sort()` and `round()`. We could just wrap our initial code inside

each of these functions but to avoid making too complicated code, we save the results in a `temp` object and incrementally use `sort()` and `round()` on that. We'll set the parameter `decreasing` to `TRUE` in the `sort()` function so that it is in descending order of how common each value is. And we'll round to two decimal places by setting the parameter `digits` to 2.

```
temp <- table(shootings$armed) / nrow(shootings) * 100
temp <- sort(temp, decreasing = TRUE)
temp <- round(temp, digits = 2)
temp
#>
#>          gun                 knife
#>      56.64                14.65
#>      unarmed              toy weapon
#>      6.45                  3.50
#>      undetermined         vehicle
#>      3.04                  2.84
#>      unknown weapon       machete
#>      1.44                  0.82
#>      Taser                  ax
#>      0.47                  0.44
#>      sword                 baseball bat
#>      0.40                  0.33
#>      gun and knife        hammer
#>      0.31                  0.29
#>      metal pipe            screwdriver
#>      0.24                  0.24
#>      sharp object           box cutter
#>      0.24                  0.22
#>      gun and car           hatchet
#>      0.20                  0.20
#>      gun and vehicle       crossbow
#>      0.18                  0.16
#>      scissors               pipe
#>      0.13                  0.11
#>      rock                  shovel
#>      0.11                  0.11
#>      baton                 BB gun
```

#>	0.09	0.09
#>	<i>blunt object</i>	<i>meat cleaver</i>
#>	0.09	0.09
#>	<i>piece of wood</i>	<i>chair</i>
#>	0.09	0.07
#>	<i>crowbar</i>	<i>metal object</i>
#>	0.07	0.07
#>	<i>pick-axe</i>	<i>straight edge razor</i>
#>	0.07	0.07
#>	<i>vehicle and gun</i>	<i>beer bottle</i>
#>	0.07	0.05
#>	<i>chain</i>	<i>glass shard</i>
#>	0.05	0.05
#>	<i>guns and explosives</i>	<i>metal pole</i>
#>	0.05	0.05
#>	<i>metal stick</i>	<i>pellet gun</i>
#>	0.05	0.05
#>	<i>samurai sword</i>	<i>brick</i>
#>	0.05	0.04
#>	<i>chain saw</i>	<i>flashlight</i>
#>	0.04	0.04
#>	<i>garden tool</i>	<i>hatchet and gun</i>
#>	0.04	0.04
#>	<i>incendiary device</i>	<i>lawn mower blade</i>
#>	0.04	0.04
#>	<i>pitchfork</i>	<i>pole</i>
#>	0.04	0.04
#>	<i>pole and knife</i>	<i>air conditioner</i>
#>	0.04	0.02
#>	<i>air pistol</i>	<i>Airsoft pistol</i>
#>	0.02	0.02
#>	<i>barstool</i>	<i>baseball bat and bottle</i>
#>	0.02	0.02
#> baseball bat and fireplace poker		<i>baseball bat and knife</i>
#>	0.02	0.02
#>	<i>bayonet</i>	<i>BB gun and vehicle</i>
#>	0.02	0.02

#>	<i>bean-bag gun</i>	<i>bow and arrow</i>
#>	0.02	0.02
#>	<i>car, knife and mace</i>	<i>carjack</i>
#>	0.02	0.02
#>	<i>chainsaw</i>	<i>claimed to be armed</i>
#>	0.02	0.02
#>	<i>contractor's level</i>	<i>cordless drill</i>
#>	0.02	0.02
#>	<i>fireworks</i>	<i>flagpole</i>
#>	0.02	0.02
#>	<i>grenade</i>	<i>gun and sword</i>
#>	0.02	0.02
#>	<i>hand torch</i>	<i>ice pick</i>
#>	0.02	0.02
#>	<i>machete and gun</i>	<i>metal hand tool</i>
#>	0.02	0.02
#>	<i>metal rake</i>	<i>motorcycle</i>
#>	0.02	0.02
#>	<i>nail gun</i>	<i>oar</i>
#>	0.02	0.02
#>	<i>pen</i>	<i>pepper spray</i>
#>	0.02	0.02
#>	<i>spear</i>	<i>stapler</i>
#>	0.02	0.02
#>	<i>tire iron</i>	<i>vehicle and machete</i>
#>	0.02	0.02
#>	<i>walking stick</i>	<i>wasp spray</i>
#>	0.02	0.02
#>	<i>wrench</i>	
#>	0.02	

Now it is a little easier to interpret. In over half of the cases the victim was carrying a gun. 15% of the time they had a knife. And 6% of the time they were unarmed. In 4% of cases there is no data on any weapon. That leaves about 20% of cases where one of the many rare weapons were used, including some that overlap with one of the more common categories.

Think about how you'd graph this data. There are 94 unique values in this

column though fewer than ten of them are common enough to appear more than 1% of the time. Should we graph all of them? No, that would overwhelm any graph. For a useful graph we would need to combine many of these into a single category - possibly called “other weapons.” And how do we deal with values where they could meet multiple larger categories? There is not always a clear answer for these types of questions. It depends on what data you’re interested in, the goal of the graph, the target audience, and personal preference.

Let’s keep exploring the data by looking at gender and race.

```
table(shootings$gender) / nrow(shootings) * 100
#>
#>      F          M
#> 4.445254 95.536528
```

Nearly all of the shootings are of a man. Given that we saw most shootings involved a person with a weapon and that most violent crimes are committed by men, this shouldn’t be too surprising.

```
temp <- table(shootings$race) / nrow(shootings) * 100
temp <- sort(temp)
temp <- round(temp, digits = 2)
temp
#>
#>      O          N          A          H          B          W
#>  0.87   1.42   1.71  16.58  23.77  45.58
```

White people are the largest race group that is killed by police, followed by Black people and Hispanic people. In fact, there are about twice as many White people killed than Black people killed, and about 2.5 times as many White people killed than Hispanic people killed. Does this mean that the oft-repeated claim that Black people are killed at disproportionate rates is wrong? No. This data simply shows the number of people killed; it doesn’t give any indication on rates of death per group. You’d need to merge it with Census data to get population to determine a rate per race group. And even that would be insufficient since people are, for example, stopped by police at different rates. This data provides a lot of information on people killed by the police, but even so it is insufficient to answer many of the questions

on that topic. It's important to understand the data not only to be able to answer questions about it, but to know what questions you can't answer - and you'll find when using criminology data that there are a *lot* of questions that you can't answer.¹

One annoying thing with the gender and race variables is that they don't spell out the name. Instead of "Female", for example, it has "F". For our graphs we want to spell out the words so it is clear to viewers. We'll fix this issue, and the issue of having many weapon categories, as we graph each variable.

7.2 Graphing a Single Numeric Variable

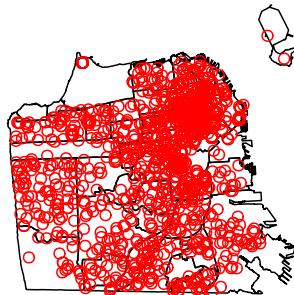
We've spent some time looking at the data so now we're ready to make the graphs. We need to load the `ggplot2` package if we haven't done so already this session (i.e. since you last closed RStudio).

```
library(ggplot2)
```

As a reminder, the benefit of using `ggplot()` is we can start with a simple plot and build our way up to more complicated graphs. We'll start here by building some graphs to depict a numeric variable - in this case the "age" column. We start every `ggplot()` the same, by inserting the dataset first and then put our x and y variables inside of the `aes()` parameter. In this case we're only going to be plotting an x variable so we don't need to write anything for y.

```
ggplot(shootings, aes(x = age))
```

¹It is especially important to not overreach when trying to answer a question when the data can't do it well. Often, no answer is better than a wrong one - especially in a field with serious consequences like criminology. For example, using the current data we'd determine that there's no (or not as much as people claim) racial bias in police killings. If we come to that conclusion based on the best possible evidence, that's okay - even if we're wrong. But coming to that conclusion based on inadequate data could lead to policies that actually cause harm. This isn't to say that you should never try to answer questions since no data is perfect and you may be wrong. You should try to develop a deep understanding of the data and be confident that you can actually answer those questions with confidence.

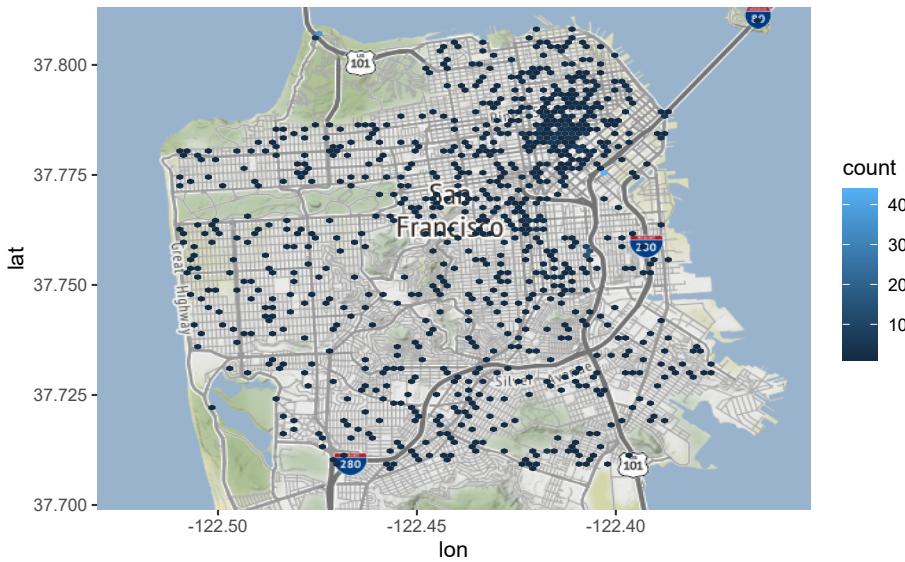


Running the above code returns a blank graph since we haven't told `ggplot()` what type of graph we want yet. Below are a few different types of ways to display a single numeric variable. They're essentially all variations of each other and show the data at different levels of precision. It's hard to say which is best - you'll need to use your best judgment and consider your audience.

7.2.1 Histogram

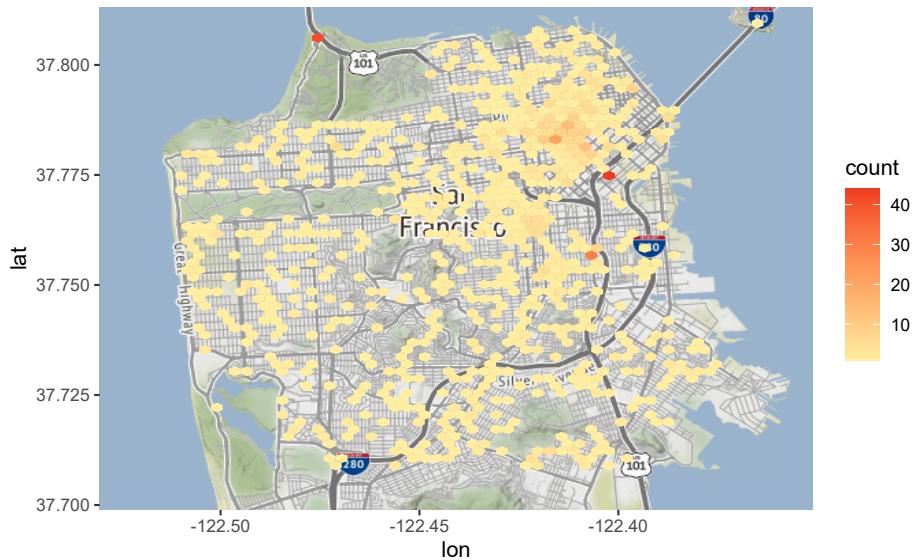
The histogram is a very common type of graph for a single numeric variable. Histograms group a numeric variable into categories and then plot them, with the heights of each bar indicating how common the group is. We can make a histogram by adding `geom_histogram()` to the `ggplot()`.

```
ggplot(shootings, aes(x = age)) +  
  geom_histogram()  
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.  
#> Warning: Removed 241 rows containing non-finite values (stat_bin).
```

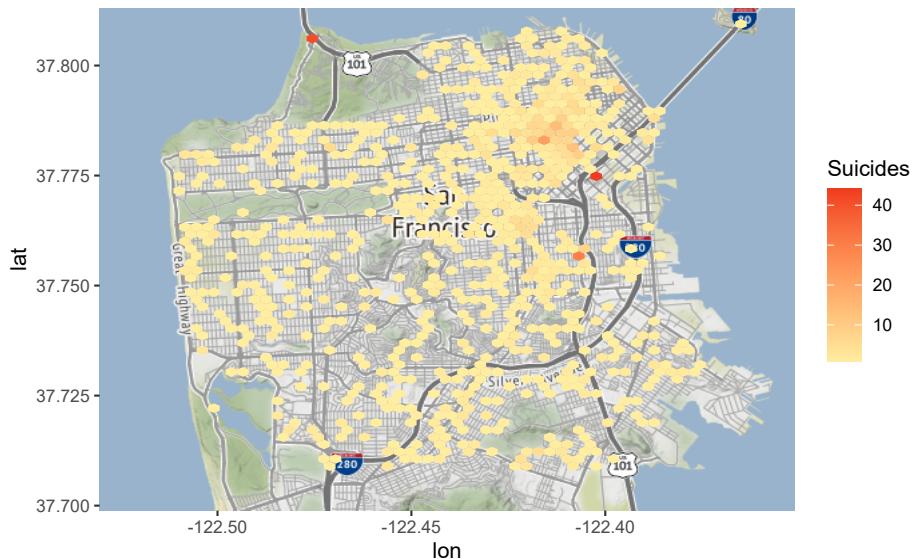


The x-axis is ages with each bar being a group of certain ages, and the y-axis is how many people are in each group. The grouping is done automatically and we can alter it by changing the `bin` parameter in `geom_histogram()`. By default this parameter is set to 30 but we can make each group smaller (have fewer ages per group) by **increasing** it from 30 or make each group larger by **decreasing** it.

```
ggplot(shootings, aes(x = age)) +  
  geom_histogram(bins = 15)  
#> Warning: Removed 241 rows containing non-finite values (stat_bin).
```



```
ggplot(shootings, aes(x = age)) +
  geom_histogram(bins = 45)
#> Warning: Removed 241 rows containing non-finite values (stat_bin).
```

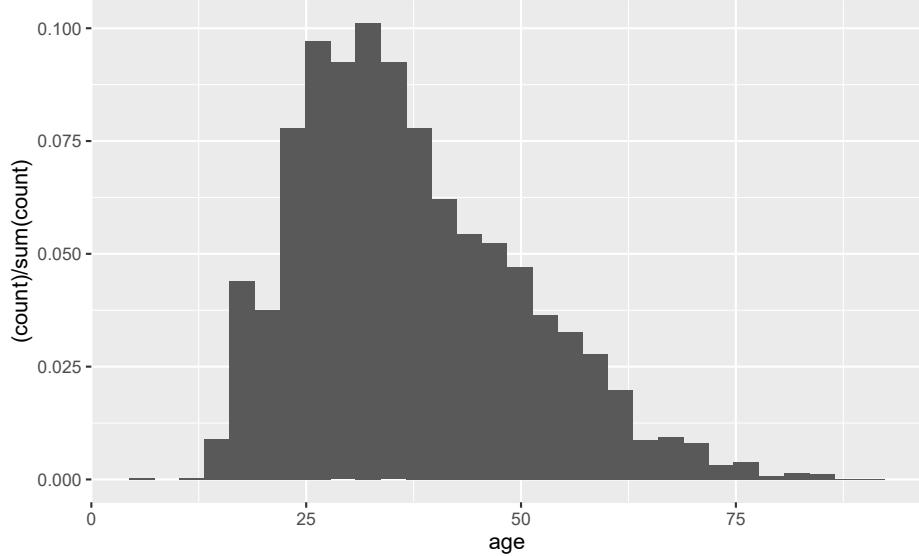


Note that while the overall trend (of most deaths being around age 25) doesn't change when we alter `bin`, the data gets more or less precise. Having fewer bins means fewer, but larger, bars which can obscure trends that more,

smaller, bars would show. But having too many bars may make you focus on minor variations that could occur randomly and take away attention from the overall trend. I prefer to err on the side of more precise graphs (more, smaller bars) but be careful over-interpreting data from small groups.

These graphs show the y-axis as the number of people in each bar. If we want to show percent instead, we can add in a parameter for `y` in the `aes()` of the `geom_histogram()`. We add in `y =(..count..)/sum(..count..))` which automatically converts the counts to percentages. The “`(..count..)/sum(..count..))`” stuff is just taking each group and dividing it from the sum of all groups. You could, of course, do this yourself before making the graph, but it’s an easy helper. If you do this, make sure to relabel the y-axis so you don’t accidentally call the percent a count!

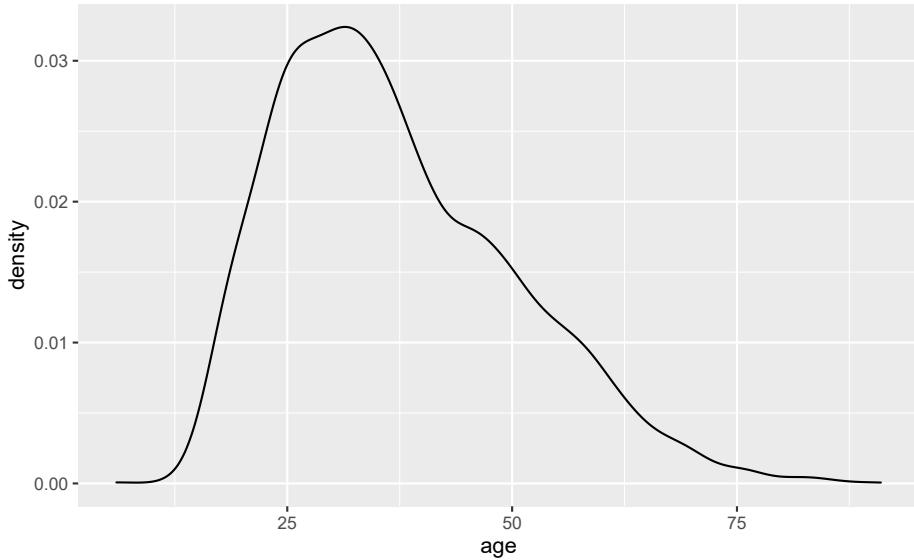
```
ggplot(shootings, aes(x = age)) +
  geom_histogram(aes(y = (..count..)/sum(..count..)))
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
#> Warning: Removed 241 rows containing non-finite values (stat_bin).
```



7.2.2 Density plot

Density plots are essentially smoothed versions of histograms. They're especially useful for numeric variables which are not integers (integers are whole numbers). They're also useful when you want to be more precise than a histogram as they are - to simplify - histograms where each bar is very narrow. Note that the y-axis of a density plot is automatically labeled "density" and has very small numbers. Interpreting the y-axis is fairly hard to explain to someone not familiar with statistics so I'd caution against using this graph unless your audience is already familiar with it. To interpret these kinds of graphs, I recommend looking for trends rather than trying to identify specific points. For example, in the below graph we can see that shootings rise rapidly starting around age 10, peak at around age 30 (if we were presenting this graph to other people we'd probably want more ages shown on the x-axis), and then steadily decline until about age 80 where it's nearly flat.

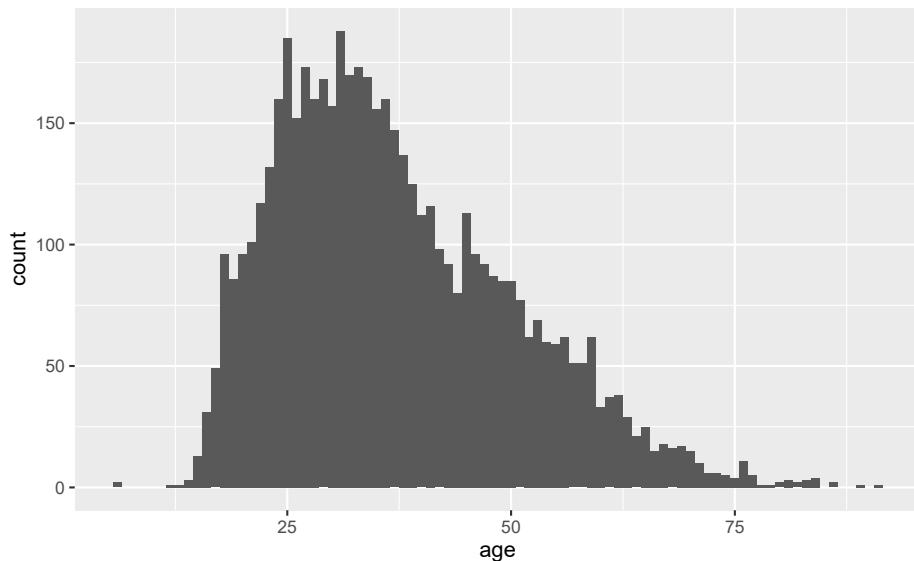
```
ggplot(shootings, aes(x = age)) +
  geom_density()
#> Warning: Removed 241 rows containing non-finite values (stat_density).
```



7.2.3 Count Graph

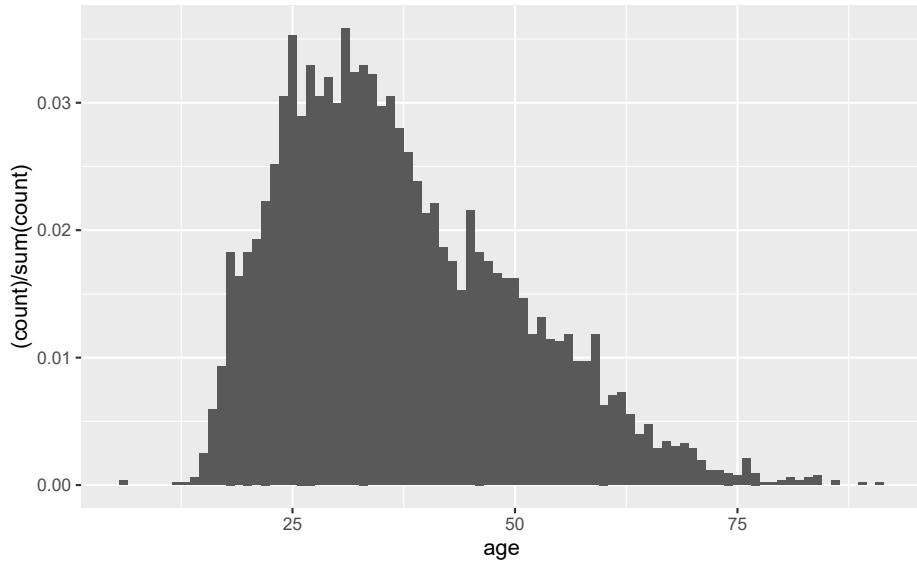
A count graph is essentially a histogram with a bar for every value in the numeric variable - like a less-smooth density plot. Note that this won't work well if you have too many unique values so I'd strongly recommend rounding the data to the nearest whole number first. Our age variable is already rounded so we don't need to do that. To make a count graph, we add `stat_count()` to the `ggplot()`.

```
ggplot(shootings, aes(x = age)) +
  stat_count()
#> Warning: Removed 241 rows containing non-finite values (stat_count).
```



Now we have a single bar for every age in the data. Like the histogram, the y-axis shows the number of people that are that age. And like the histogram, we can change this from number of people to percent of people using the exact same code.

```
ggplot(shootings, aes(x = age)) +
  stat_count(aes(y = (..count..)/sum(..count..)))
#> Warning: Removed 241 rows containing non-finite values (stat_count).
```

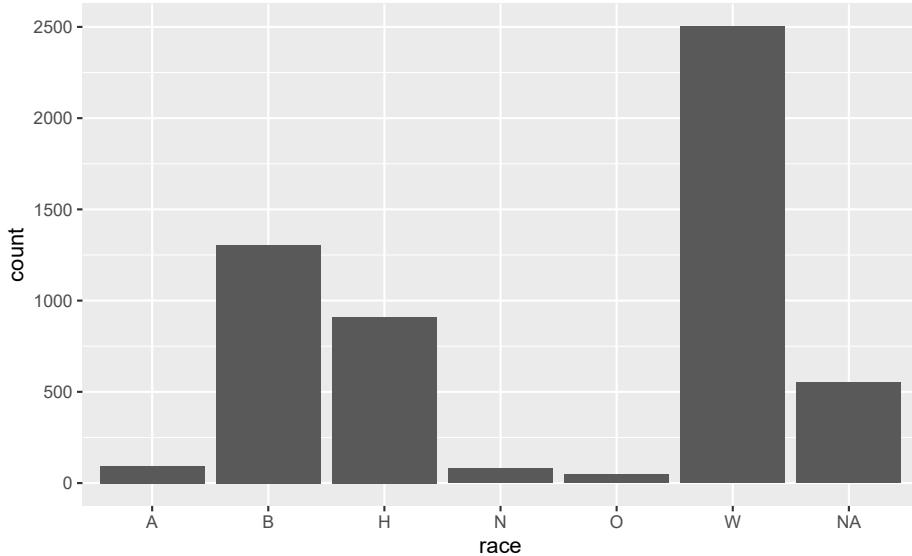


7.2.4 Graphing a Categorical Variable

7.3 Bar graph

To make this barplot we'll set the x-axis variable to our "race" column and add `geom_bar()` to the end.

```
ggplot(shootings, aes(x = race)) +  
  geom_bar()
```



This gives us a barplot in alphabetical order. In most cases we want the data sorted by frequency, so we can easily see what value is the most common, second most common, etc. There are a few ways to do this but we'll do this by turning the "race" variable into a factor and ordering it by frequency. We can do that using the `factor()` function. The first input will be the "race" variable and then we will need to set the `levels` parameter to a vector of values sorted by frequency. An easy way to know how often values are in a column is to use the `table()` function on that column, such as below.

```
table(shootings$race)
#>
#>     A      B      H      N      O      W
#>   94  1305  910   78    48 2502
```

It's still alphabetical so let's wrap that in a `sort()` function.

```
sort(table(shootings$race))
#>
#>     O      N      A      H      B      W
#>   48    78    94  910  1305 2502
```

It's sorted from smallest to largest. We usually want to graph from largest to smallest so let's set the parameter `decreasing` in `sort()` to TRUE.

```
sort(table(shootings$race), decreasing = TRUE)
#>
#>   W     B     H     A     N     O
#> 2502 1305  910   94   78   48
```

Now, we only need the names of each value, not how often they occur. So we can again wrap this whole thing in `names()` to get just the names.

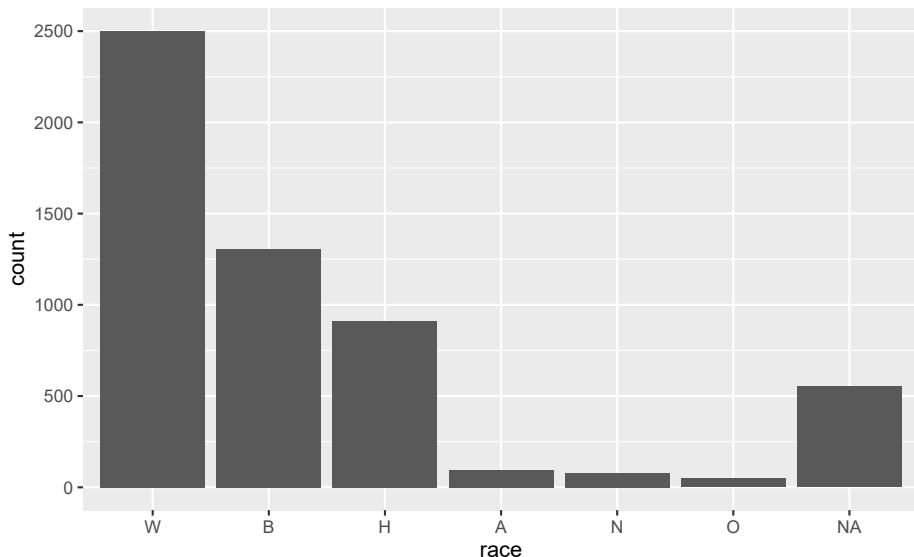
```
names(sort(table(shootings$race), decreasing = TRUE))
#> [1] "W" "B" "H" "A" "N" "O"
```

If we tie it all together, we can make the “race” column into a factor variable.

```
shootings$race <- factor(shootings$race,
                         levels = names(sort(table(shootings$race), decreasing = TRUE)))
```

Now let’s try that barplot again.

```
ggplot(shootings, aes(x = race)) +
  geom_bar()
```

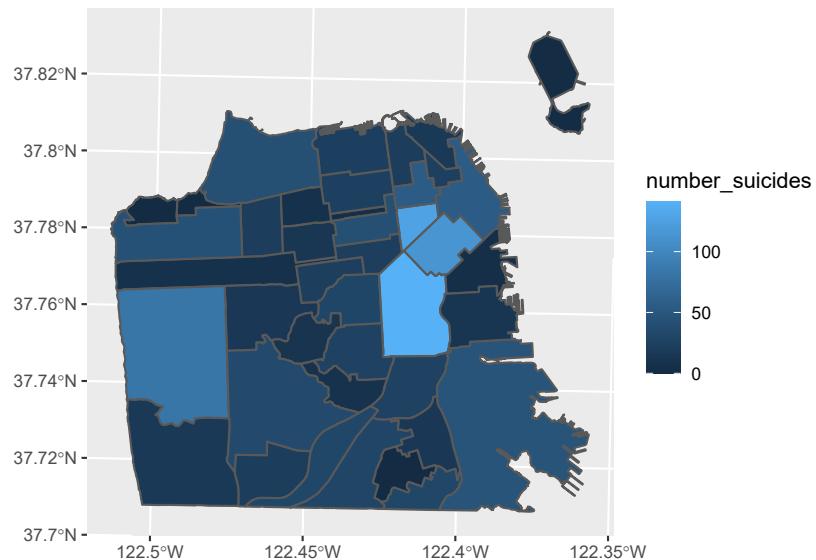


It works! Note that all the values that are missing in our data are still reported in the barplot under a column called “NA”. This is not sorted properly since there are more NA values than three of the other values but is still at

the far right of the graph. We can change this if we want to make all the NA values an actual character type and call it something like “Unknown”. But this way it does draw attention to how many values are missing from this column. Like most things in graphing, this is a personal choice as to what to do.

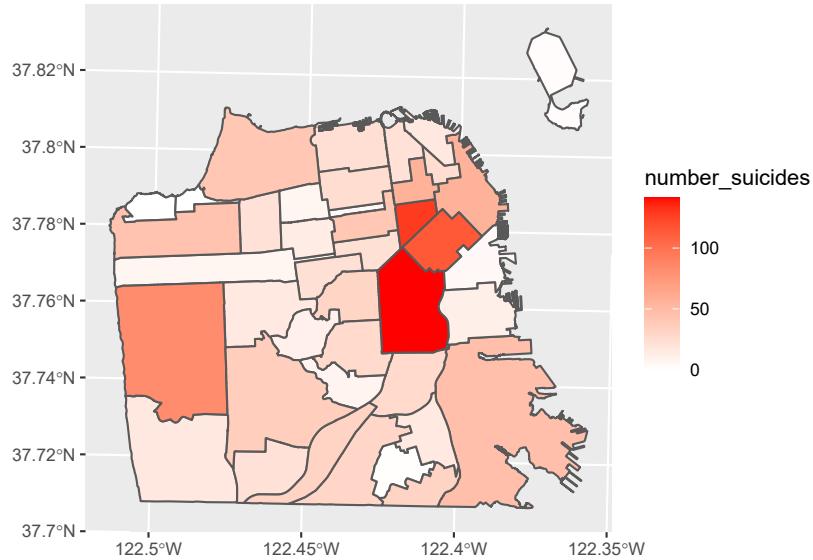
For bar graphs it is often useful to flip the graph so each value is a row in the graph rather than a column. This also makes it much easier to read the value name. If the value names are long, it’ll shrink the graph to accommodate the name. This is usually a sign that you should try to shorten the name to avoid reducing the size of the graph.

```
ggplot(shootings, aes(x = race)) +
  geom_bar() +
  coord_flip()
```



Since it’s flipped, now it’s sorted from smallest to largest. So we’ll need to change the `factor()` code to fix that.

```
shootings$race <- factor(shootings$race,
                           levels = names(sort(table(shootings$race), decreasing = FALSE)))
ggplot(shootings, aes(x = race)) +
  geom_bar() +
  coord_flip()
```



The NA value is now at the top, which looks fairly bad. Let's change all NA values to the string "Unknown". And while we're at it, let's change all the abbreviated race values to actual names. We can get all the NA values by using `is.na(shootings$race)` and using a conditional statement to get all rows that meet that condition, then assign them the value "Unknown". Instead of trying to subset a factor variable to change the values, we should convert it back to a character type first using `as.character()`, and then convert it to a factor again once we're done.

```
shootings$race <- as.character(shootings$race)
shootings$race[is.na(shootings$race)] <- "Unknown"
```

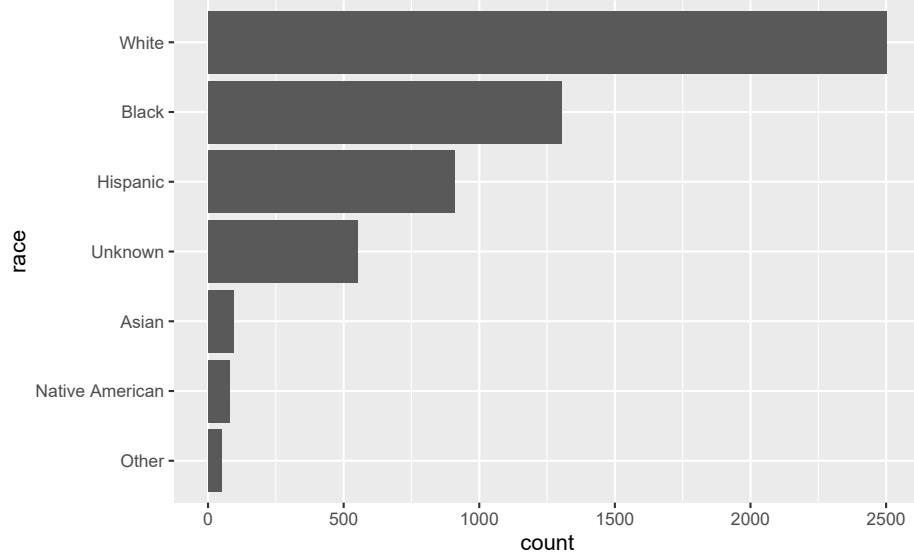
Now we can use conditional statements to change all the race letters to names. It's not clear what race "O" and "N" are so I checked the [Washington Post's GitHub page](#) which explains. Instead of `is.na()` we'll use `shootings$race == ""` where we put the letter inside of the quotes.

```
shootings$race[shootings$race == "O"] <- "Other"
shootings$race[shootings$race == "N"] <- "Native American"
shootings$race[shootings$race == "A"] <- "Asian"
shootings$race[shootings$race == "H"] <- "Hispanic"
shootings$race[shootings$race == "B"] <- "Black"
```

```
shootings$race[shootings$race == "W"] <- "White"
```

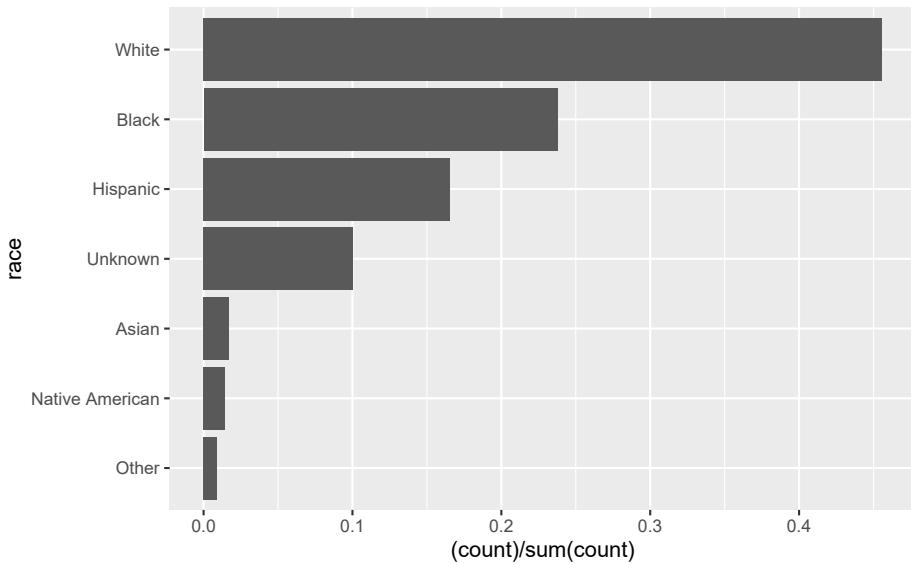
Now let's see how our graph looks. We'll need to rerun the `factor()` code since now all of the values are changed.

```
shootings$race <- factor(shootings$race,
                           levels = names(sort(table(shootings$race), decreasing = FALSE))
ggplot(shootings, aes(x = race)) +
  geom_bar() +
  coord_flip()
```



As earlier, we can show percentage instead of count by adding `y = ..count../sum(..count..)` to the `aes()` in `geom_bar()`.

```
ggplot(shootings, aes(x = race)) +
  geom_bar(aes(y = (..count../sum(..count..)))) +
  coord_flip()
```

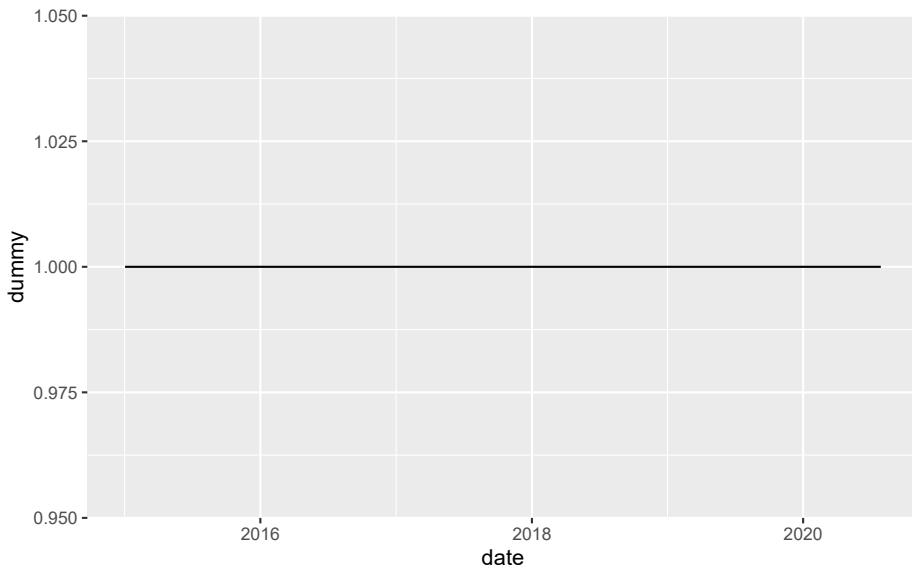


7.4 Graphing Data Over Time

We went over time-series graphs in Chapter 6 but it's such an important topic we'll cover it again. A lot of criminology research is seeing if a policy had an effect, which means we generally want to compare an outcome over time (and compare the treated group to a similar untreated group). To graph that we look at an outcome, in this case numbers of killings, over time. In our case we aren't evaluating any policy, just seeing if the number of police killings change over time.

We'll need to make a variable to indicate that the row is for one shooting. We can call this "dummy" and assign it a value of 1. Then we can make the `ggplot()` and set this "dummy" column to the y-axis value and set our date variable "date" to the x-axis (the time variable is **always** on the x-axis). Then we'll set the type of plot to `geom_line()` so we have a line graph showing killings over time.

```
shootings$dummy <- 1
ggplot(shootings, aes(x = date, y = dummy)) +
  geom_line()
```



This graph is clearly wrong. Why? Well, our y-axis variable is always 1 so there's no variation to plot. Every single value, even if there are more than one shooting per day, is on the 1 line on the y-axis. And the fact that we have multiple killings per day is an issue because we only want a single line in our graph. We'll need to aggregate our data to some time period (e.g. day, month, year) so that we have one row per time-period and know how many people were killed in that period. We'll start with yearly data and then move to monthly data. Since we're going to be dealing with dates, lets load the `lubridate()` package that is well-suited for this task.

```
library(lubridate)
#>
#> Attaching package: 'lubridate'
#> The following objects are masked from 'package:base':
#>
#>     date, intersect, setdiff, union
```

We'll use two functions to create variables that tell us the month and the year of each date in our data. We'll use these new variables to aggregate our data to that time unit. First, the `floor_date()` function is a very useful tool that essentially rounds a date. Here we have the exact date the killing happened on, and we want to determine what month that date is from. So we'll use the parameter `unit` in `floor_date()` and tell the

function we want to know the “month” (for a full set of options please see the documentation for `floor_date()` by entering `?floor_date` in the console). So we can do `floor_date(shootings$date, unit = "month")` to get the month - specifically, it returns the date that is the first of the month for that month - the killing happened on. Even simpler, to get the year, we simply use `year()` and put our “date” variable in the parentheses. We’ll call the new variables “month_year” and “year”, respectively.

```
shootings$month_year <- floor_date(shootings$date, unit = "month")
shootings$year <- year(shootings$date)

head(shootings$month_year)
#> [1] "2015-01-01" "2015-01-01" "2015-01-01" "2015-01-01" "2015-01-01"
#> [6] "2015-01-01"
head(shootings$year)
#> [1] 2015 2015 2015 2015 2015 2015
```

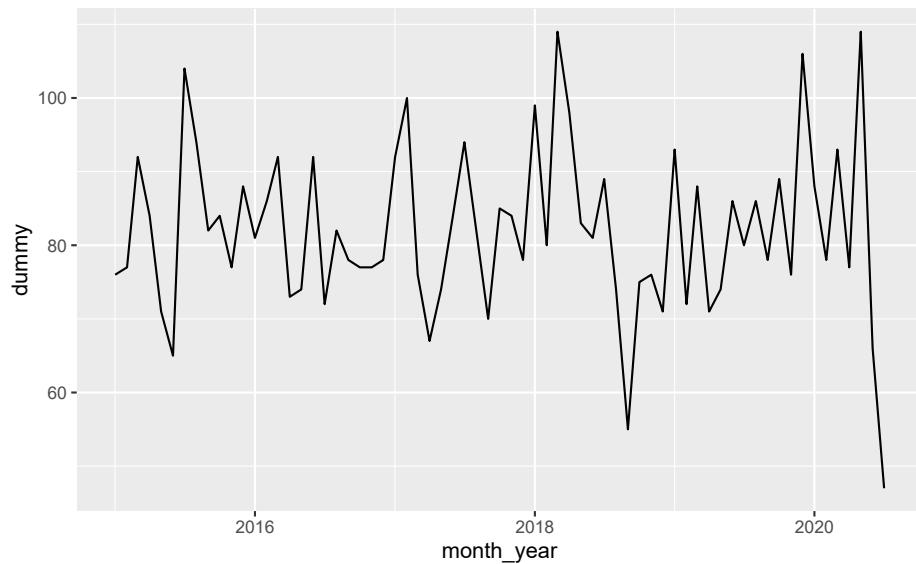
Since the data is already sorted by date, all the values printed from `head()` are the same. But you can look at the data using `View()` to confirm that the code worked properly.

We can now aggregate the data by the “month_year” variable and save the result into a new dataset we’ll call *monthly_shootings*. For a refresher on aggregating, please see Section 3.3

```
monthly_shootings <- aggregate(dummy ~ month_year, data = shootings, FUN = sum)
head(monthly_shootings)
#> month_year dummy
#> 1 2015-01-01    76
#> 2 2015-02-01    77
#> 3 2015-03-01    92
#> 4 2015-04-01    84
#> 5 2015-05-01    71
#> 6 2015-06-01    65
```

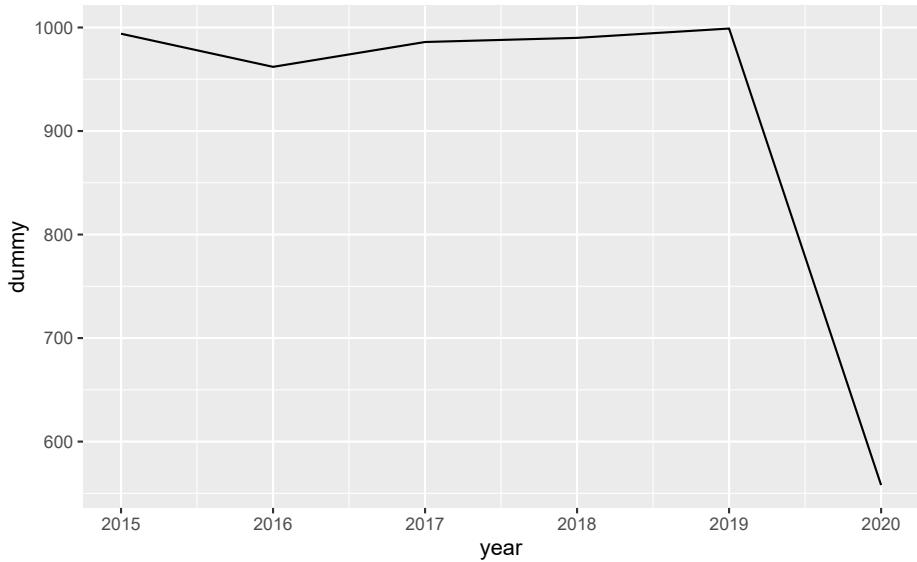
Since we now have a variable that shows for each month the number of people killed, we can graph this new dataset. We’ll use the same process as earlier but our dataset is now `monthly_shootings` instead of `shootings` and the x-axis variable is “month_year” instead of “date”.

```
ggplot(monthly_shootings, aes(x = month_year, y = dummy)) +  
  geom_line()
```



The process is the same for yearly data.

```
yearly_shootings <- aggregate(dummy ~ year, data = shootings, FUN = sum)  
ggplot(yearly_shootings, aes(x = year, y = dummy)) +  
  geom_line()
```



Note the steep drop-off at the end of each graph. Is that due to fewer shooting occurring more recently? No, it's simply an artifact of the graph comparing whole months (years) to parts of a month (year) since we haven't finished this month (year) yet (and the data has a small lag in reporting).

7.5 Pretty Graphs

What's next for these graphs? You'll likely want to add labels for the axes and the title. We went over how to do this in Section 6.3 so please refer to that for more info. Also, check out [ggplot2's website](#) to see more on this very versatile package. As I've said all chapter, a lot of this is going to be personal taste so please spend some time exploring the package and changing the appearance of the graph to learn what looks right to you.

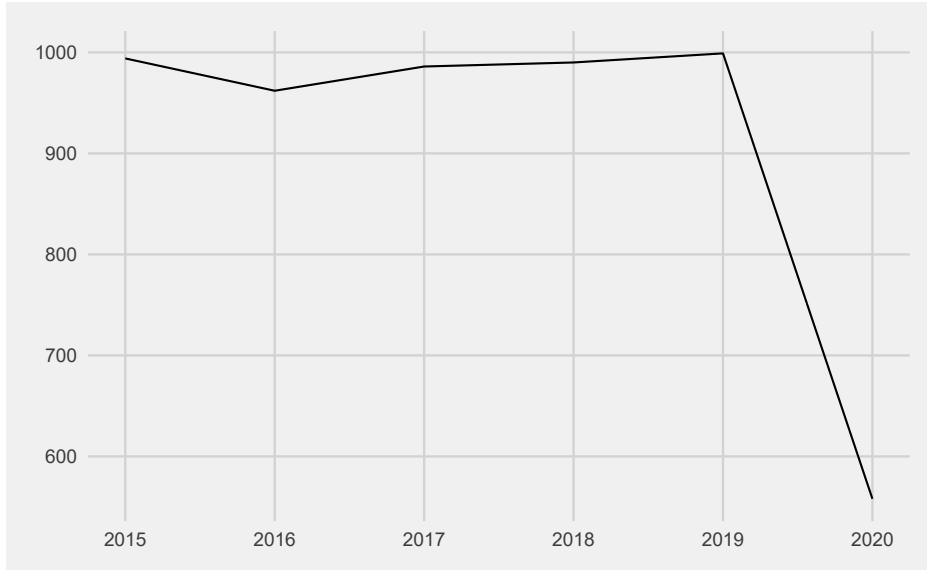
7.5.1 Themes

In addition to making changes to the graph's appearance yourself, you can use a theme that someone else made. A theme is just a collection of changes to the graph's appearance that someone put in a function for others to use. Each theme is different and is fairly opinionated, so you should only use one that you think looks best for your graph. To use a theme, simply add the

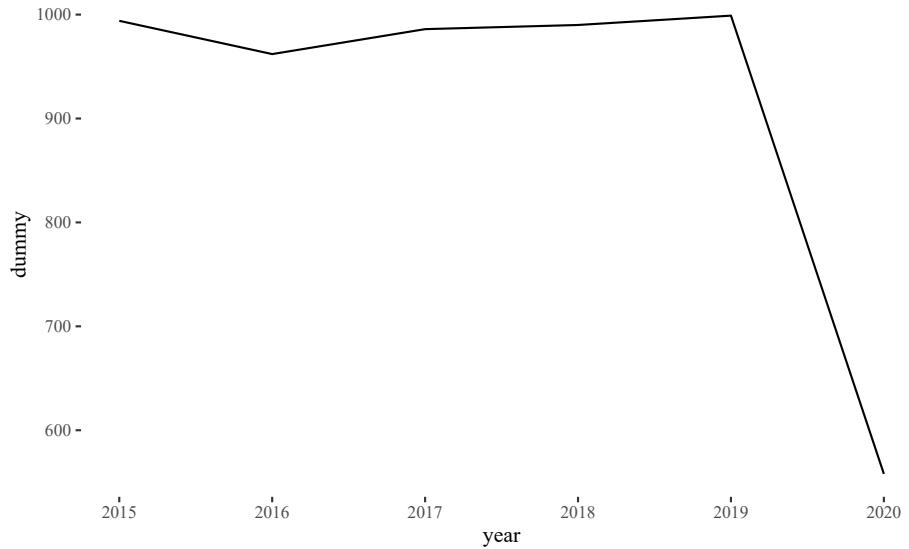
theme (exactly as spelled on the site) to your ggplot using the `+` as normal (and make sure to include the `()` since each theme is actually a function). `ggplot2` comes with a series of themes that you can look at [here](#). Here, we'll be looking at themes from the `ggthemes` package which is a great source of different themes to modify the appearance of your graph. Check out this [website](#) to see a depiction of all of the possible themes. If you don't have the `ggthemes` package installed, do so using `'install.packages("ggthemes")'`.

Let's do a few examples using the graph made above. First, we'll need to load the `ggthemes` library.

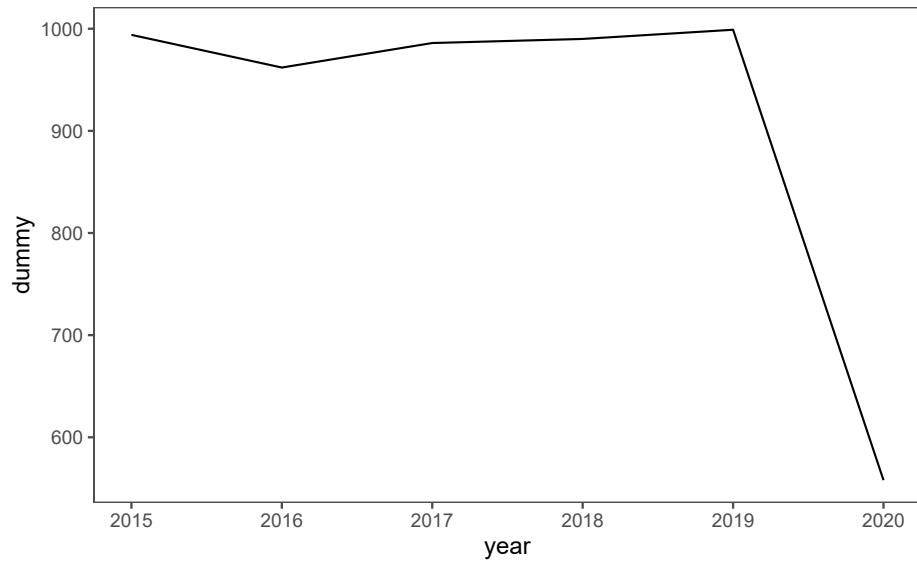
```
library(ggthemes)
ggplot(yearly_shootings, aes(x = year, y = dummy)) +
  geom_line() +
  theme_fivethirtyeight()
```



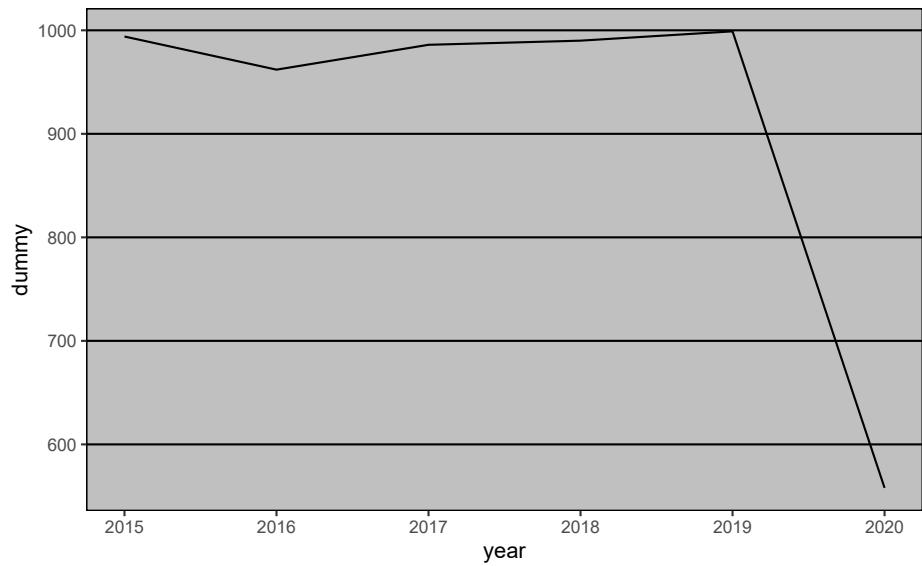
```
ggplot(yearly_shootings, aes(x = year, y = dummy)) +
  geom_line() +
  theme_tufte()
```



```
ggplot(yearly_shootings, aes(x = year, y = dummy)) +  
  geom_line() +  
  theme_few()
```



```
ggplot(yearly_shootings, aes(x = year, y = dummy)) +  
  geom_line() +  
  theme_excel()
```



Chapter 8

Hotspot maps

Hotspot maps are used to find where events (marijuana dispensaries, crimes, liquors stores) are especially prevalent. These maps are frequently used by police departments, particularly in determining where to do hotspot policing (which is focusing patrols on high-crime areas).

However, there are significant flaws with these kinds of maps. As we'll see during this lesson, minor changes to how we make the maps can cause significant differences in interpretation. For example, determining the size of the clusters that make up the hotspots can make it seem like there are much larger or smaller areas with hotspots than there actually are. These clusters are also drawn fairly arbitrarily, without considering context such as neighborhoods (In Chapter 9 we'll make maps that try to account for these types of areas). This makes it more difficult to interpret because even though maps give us the context of location, it can combine different areas in an arbitrary way. We'll explore these issues in more detail throughout the lesson but keep in mind these risks as you make your own hotspot maps.

Here, we will make hotspot maps using data on suicides in San Francisco between 2003 and 2017. First, we need to read the data, which is called "san_francisco_suicide_2003_2017.csv". We can name the object we make *suicide*.

```
library(readr)
suicide <- read_csv("data/san_francisco_suicide_2003_2017.csv")
#> Parsed with column specification:
```

```
#> cols(
#>   IncidntNum = col_double(),
#>   Category = col_character(),
#>   Descript = col_character(),
#>   DayOfWeek = col_character(),
#>   Date = col_character(),
#>   Time = col_time(format = ""),
#>   PdDistrict = col_character(),
#>   Resolution = col_character(),
#>   Address = col_character(),
#>   X = col_double(),
#>   Y = col_double(),
#>   Location = col_character(),
#>   PdId = col_double(),
#>   year = col_double()
#> )
suicide <- as.data.frame(suicide)
```

This data contains information on each crime reported in San Francisco including the type of crime (in our case always suicide), a more detailed crime category, and a number of date and location variables. The columns X and Y are our longitude and latitude columns which we will use to graph the data.

```
head(suicide)
#>   IncidntNum Category                               Descript DayOfWeek
#> 1 180318931 SUICIDE ATTEMPTED SUICIDE BY STRANGULATION Monday 04/30/
#> 2 180315501 SUICIDE      ATTEMPTED SUICIDE BY JUMPING Saturday 04/28/
#> 3 180295674 SUICIDE      SUICIDE BY LACERATION Saturday 04/21/
#> 4 180263659 SUICIDE          SUICIDE Tuesday 04/10/
#> 5 180235523 SUICIDE      ATTEMPTED SUICIDE BY INGESTION Friday 03/30/
#> 6 180236515 SUICIDE      SUICIDE BY ASPHYXIATION Thursday 03/29/
#>   Time PdDistrict Resolution           Address      X
#> 1 06:30:00    TARAVAL      NONE 0 Block of BRUCE AV -122.4517 37.722
#> 2 17:54:00    NORTHERN     NONE 700 Block of HAYES ST -122.4288 37.776
#> 3 12:20:00    RICHMOND     NONE 3700 Block of CLAY ST -122.4546 37.788
#> 4 05:13:00    CENTRAL      NONE 0 Block of DRUMM ST -122.3964 37.794
#> 5 09:15:00    TARAVAL      NONE 0 Block of FAIRFIELD WY -122.4632 37.726
#> 6 17:30:00    RICHMOND     NONE 300 Block of 29TH AV -122.4893 37.782
```

```
#> 
#> 1 POINT (-122.45168059935614 37.72218061554315) 1.803189e+13 2018
#> 2 POINT (-122.42876060987851 37.77620120112792) 1.803155e+13 2018
#> 3 POINT (-122.45462091999406 37.7881754224736) 1.802957e+13 2018
#> 4 POINT (-122.39642194376758 37.79414474237039) 1.802637e+13 2018
#> 5 POINT (-122.46324153155875 37.72679184368551) 1.802355e+13 2018
#> 6 POINT (-122.48929119750689 37.782735835121265) 1.802365e+13 2018
```

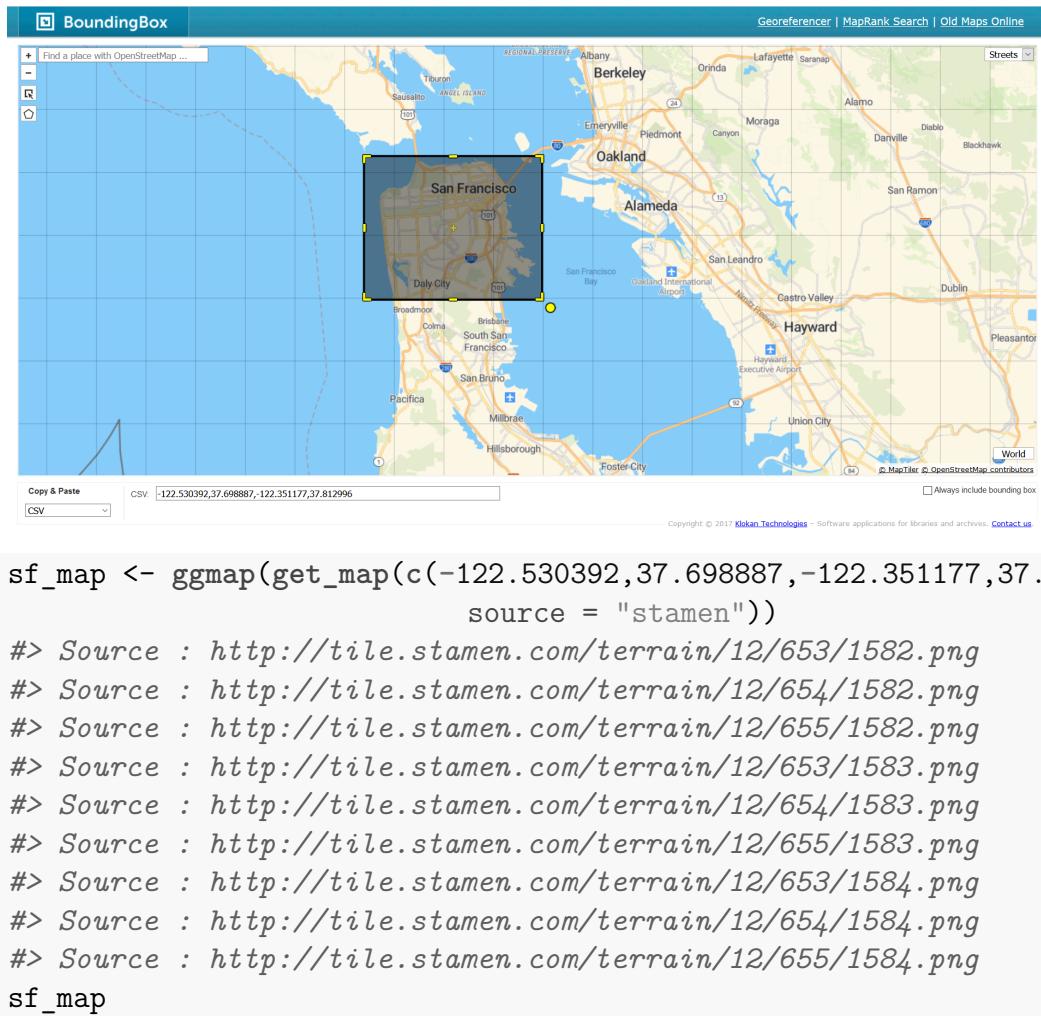
8.1 A simple map

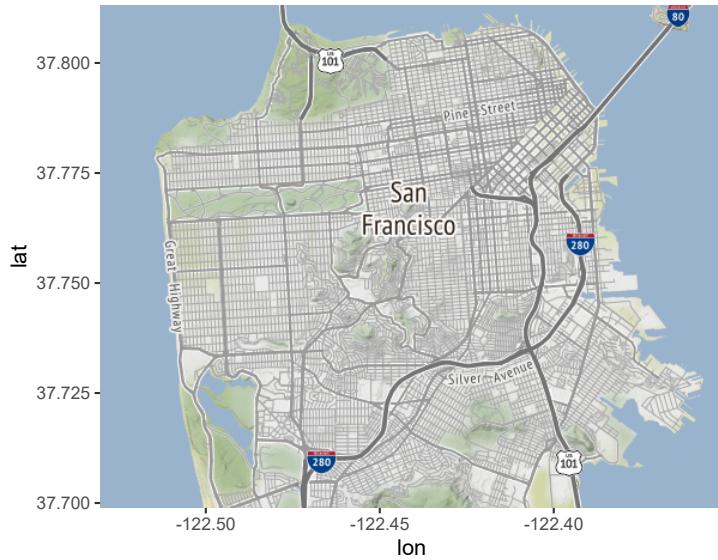
To make these maps we will use the package `ggmap`.

```
install.packages("ggmap")
library(ggmap)
#> Loading required package: ggplot2
#> Google's Terms of Service: https://cloud.google.com/maps-platform/terms/.
#> Please cite ggmap if you use it! See citation("ggmap") for details.
```

We'll start by making the background to our map, showing San Francisco. We do so by using the `get_map()` function from `ggmap` which gets a map background from a number of sources. We'll set the source to "stamen" since Google no longer allows us to get a map without creating an account. The first parameter in `get_map()` is simply coordinates for San Francisco's bounding box to ensure we get a map of the right spot. A bounding box is four coordinates that connect to make a rectangle, used for determining where in the world to show.

An easy way to find the four coordinates for a bounding box is to go to the site [Bounding Box](#). This site has a map of the world and a box on the screen. Move the box to the area you want the map of. You may need to resize the box to cover the area you want. Then in the section that says "Copy & Paste", change the dropdown box to "CSV". In the section to the right of this are the four numbers that make up the bounding box. You can copy those numbers into `get_map()`





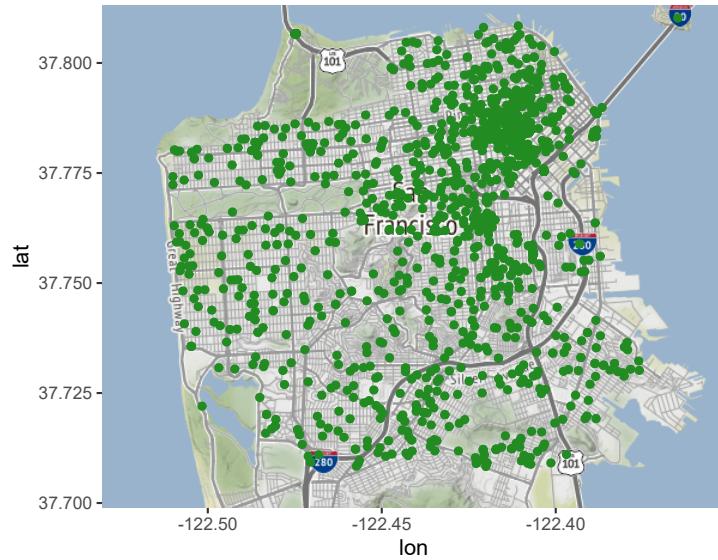
Since we saved the map output into `sf_map` we can reuse this map background for all the maps we're making in this lesson. This saves us time as we don't have to wait to download the map every time. Let's plot the shootings from our data set. Just as with a scatterplot we use the `geom_point()` function from the `ggplot2` package and set our longitude and latitude variables on the x- and y-axis, respectively.

```
sf_map +
  geom_point(aes(x = X, y = Y),
             data = suicide)
#> Warning: Removed 1 rows containing missing values (geom_point).
```



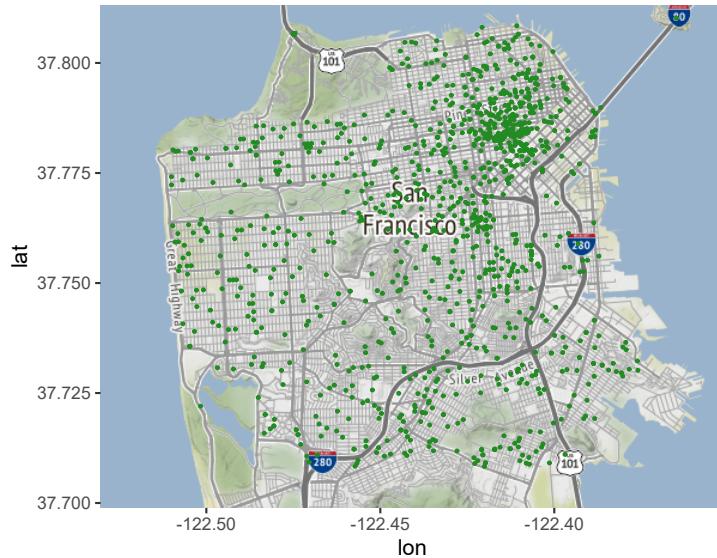
If we wanted to color the dots, we can use `color =` and then select a color. Let's try it with "forestgreen".

```
sf_map +
  geom_point(aes(x = X, y = Y),
             data = suicide,
             color = "forestgreen")
#> Warning: Removed 1 rows containing missing values (geom_point).
```

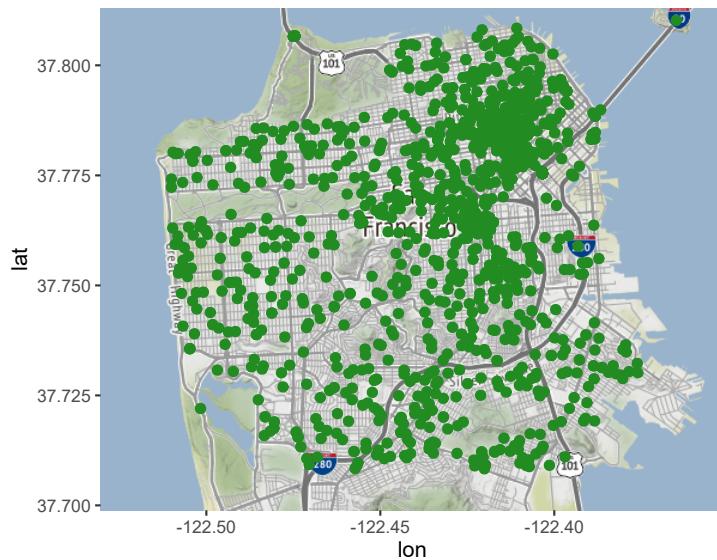


As with other graphs we can change the size of the dot using `size =`.

```
sf_map +  
  geom_point(aes(x = X, y = Y),  
             data = suicide,  
             color = "forestgreen",  
             size = 0.5)  
#> Warning: Removed 1 rows containing missing values (geom_point).
```

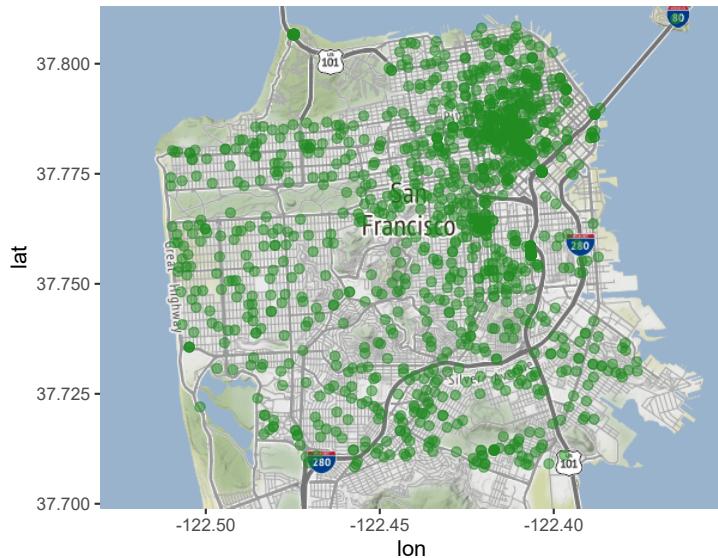


```
sf_map +
  geom_point(aes(x = X, y = Y),
             data = suicide,
             color = "forestgreen",
             size = 2)
#> Warning: Removed 1 rows containing missing values (geom_point).
```



For maps like this - with one point per event - it is hard to tell if any events happen on the same, or nearly the same, location as each point is solid green. We want to make the dots semi-transparent so if multiple suicides happen at the same place that dot will be shaded darker than if only one suicide happened there. To do so we use the parameter `alpha =` which takes an input between 0 and 1 (inclusive). The lower the value the more transparent it is.

```
sf_map +
  geom_point(aes(x = X, y = Y),
             data = suicide,
             color = "forestgreen",
             size = 2,
             alpha = 0.5)
#> Warning: Removed 1 rows containing missing values (geom_point).
```

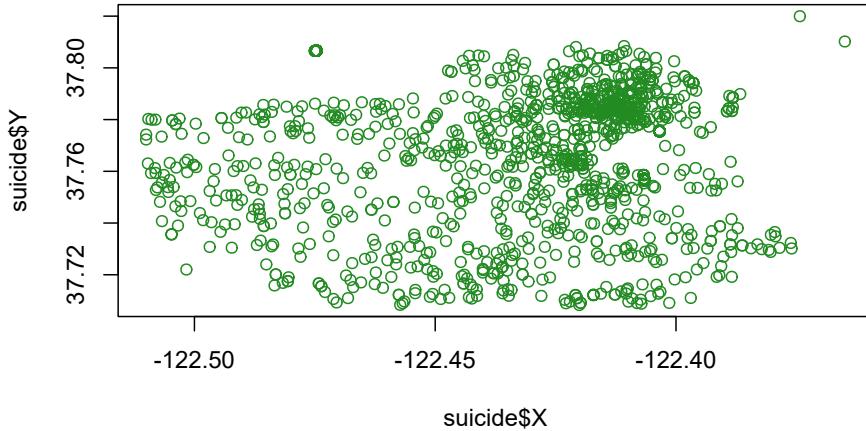


This map is useful because it allows us to easily see where each suicide in San Francisco happened between 2003 and 2017. There are some limitations though. This shows all suicides in a single map, meaning that any time trends are lost.

8.2 What really are maps?

Let's pause for a moment to think about what a map really is. Below, I made a simple scatterplot of our data with one dot per shooting (minus the one without coordinates). Compare this to the map above and you'll see that they are the same except the map has a useful background while the plot has a blank background. That is all static maps are (in Chapter 10 we'll learn about interactive maps), scatterplots of coordinates overlayed on a map background. Basically, they are scatterplots with context. And this context is useful, we can interpret the map to see that there are lots of suicides in the northeast part of San Francisco but not so many elsewhere, for example. The exact same pattern is present in the scatterplot but without the ability to tell "where" a dot is.

```
plot(suicide$X, suicide$Y, col = "forestgreen")
```



8.3 Making a hotspot map

Now we can start making hotspot maps which help to show areas with clusters of events. We'll do this using hexagonal bins which are an efficient way of showing clusters of events on a map. Our syntax will be similar to the map above but now we want to use the function `stat_binhex()` rather than

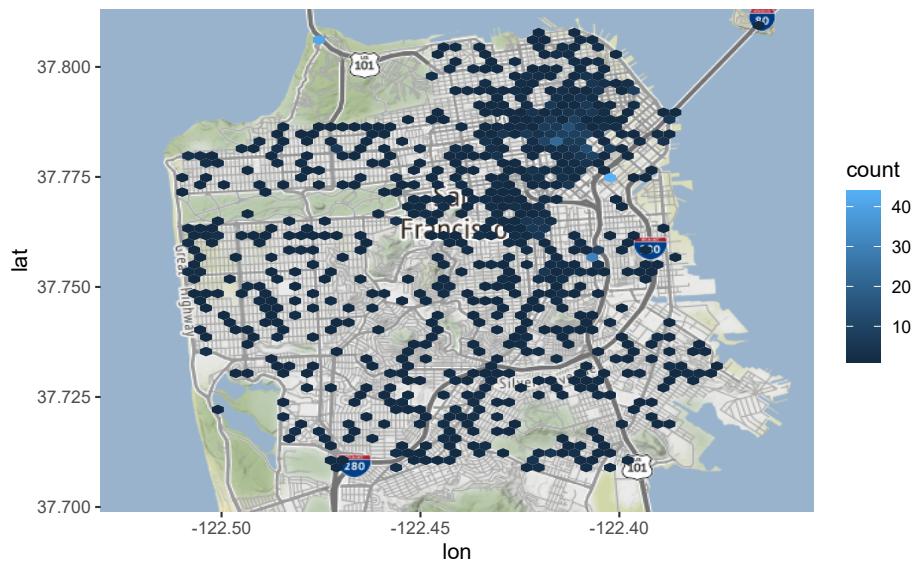
`geom_point()`. It starts the same as before with `aes(x = X, y = Y)` (or whatever the longitude and latitude columns are called in your data), as well as `data = suicide` outside of the `aes()` parameter.

There are two new things we need to make the hotspot map. First, we add the parameter `bins = number_of_bins` where “number_of_bins” is a number we select. `bins` essentially says how large or small we want each cluster of events to be. A smaller value for `bins` says we want more events clustered together, making larger bins. A larger value for `bins` has each bin be smaller on the map and capture fewer events. This will become clearer with examples.

The second thing is to add the function `coord_cartesian()` which just tells `ggplot()` we are going to do some spatial analysis in the making of the bins. We don’t need to add any parameters in this.

Let’s start with 60 bins and then try some other number of bins to see how it changes the map.

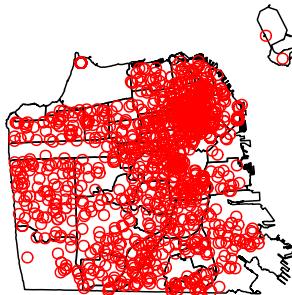
```
sf_map +
  stat_binhex(aes(x = X, y = Y),
              bins = 60,
              data = suicide) +
  coord_cartesian()
#> Coordinate system already present. Adding new coordinate system, which will replace the existing one.
#> Warning: Removed 1 rows containing non-finite values (stat_binhex).
```



From this map we can see that most areas in the city had no suicides and that the areas with the most suicides are in downtown San Francisco.

What happens when we drop the number of bins to 30?

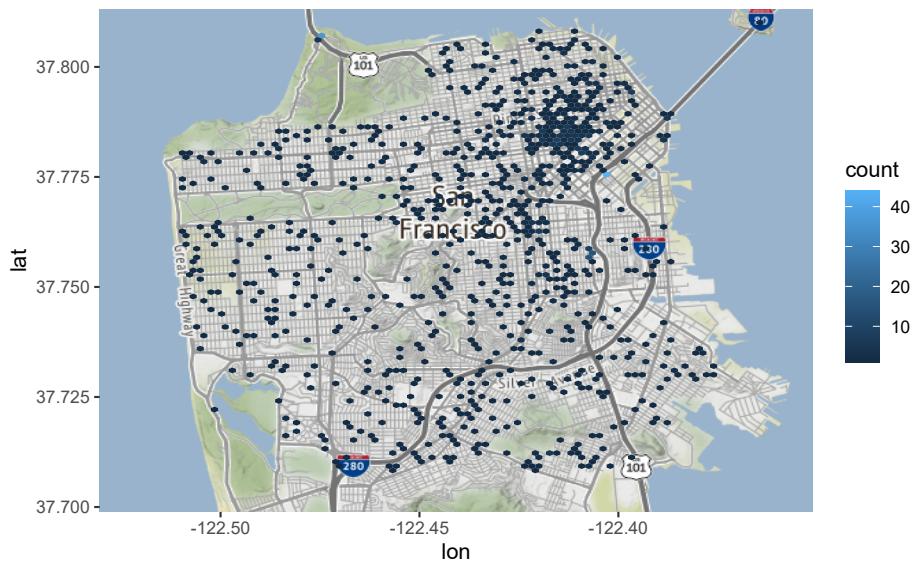
```
sf_map +
  stat_binhex(aes(x = X, y = Y),
              bins = 30,
              data = suicide) +
  coord_cartesian()
#> Coordinate system already present. Adding new coordinate system, which will
#> Warning: Removed 1 rows containing non-finite values (stat_binhex).
```



Each bin is much larger and covers nearly all of San Francisco. Be careful with maps like these! This map is so broad that it appears that suicides are ubiquitous across the city. We know from the map showing each suicide as a dot, and that there are <1,300 suicides, that this is not true. Making maps like this make it easy to mislead the reader, including yourself!

What about looking at 100 bins?

```
sf_map +
  stat_binhex(aes(x = X, y = Y),
               bins = 100,
               data = suicide) +
  coord_cartesian()
#> Coordinate system already present. Adding new coordinate system, which will replace the existing one.
#> Warning: Removed 1 rows containing non-finite values (stat_binhex).
```



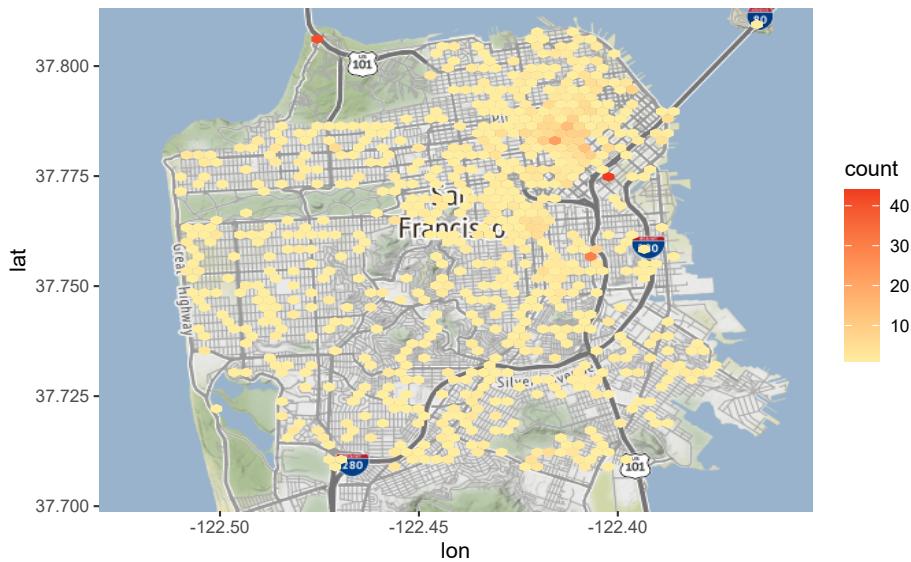
Now each bin is very small and a much smaller area in San Francisco has had a suicide. So what is the right number of bins to use? There is no correct universal answer - you must decide what the goal is with the data you are using. This opens up serious issues for manipulation - intentional or not - of the data as the map is so easily changeable without ever changing the data itself.

8.3.1 Colors

To change the bin colors we can use the parameter `scale_fill_gradient()`. This accepts a color for “low” which is when the events are rare and “high” for the bins with frequent events. We’ll use colors from [ColorBrewer](#), selecting the yellow-reddish theme (“3-class YlOrRd”) from the Multi-hue section of the “sequential” data on the page.

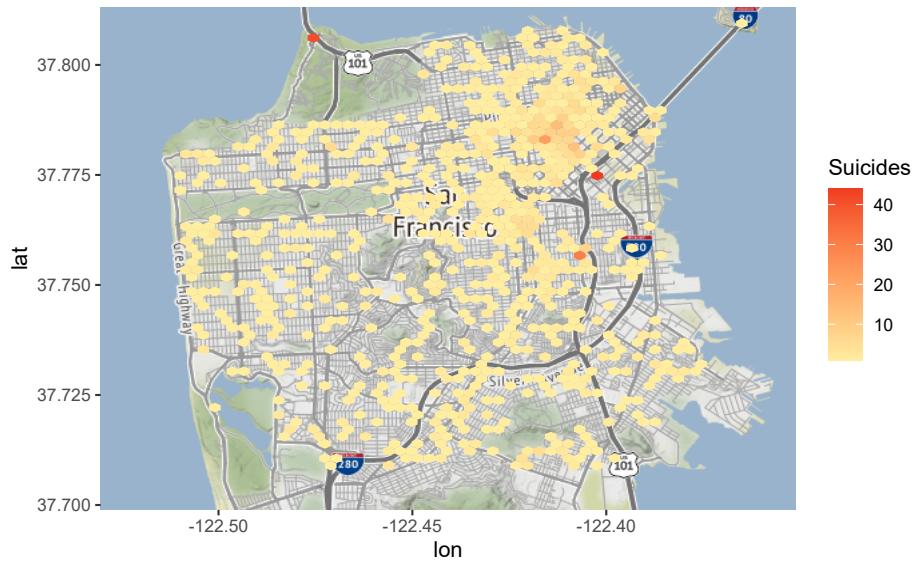
```
sf_map +
  stat_binhex(aes(x = X, y = Y),
              bins = 60,
              data = suicide) +
  coord_cartesian() +
  scale_fill_gradient(low = "#ffeda0",
                      high = "#f03b20")
```

```
#> Coordinate system already present. Adding new coordinate system, which will replace it.
#> Warning: Removed 1 rows containing non-finite values (stat_binhex).
```



By default it labels the legend as “count”. Since we know these are counts of suicides let’s relabel that as such.

```
sf_map +
  stat_binhex(aes(x = X, y = Y),
              bins = 60,
              data = suicide) +
  coord_cartesian() +
  scale_fill_gradient('Suicides',
                      low = "#ffeda0",
                      high = "#f03b20")
#> Coordinate system already present. Adding new coordinate system, which will replace it.
#> Warning: Removed 1 rows containing non-finite values (stat_binhex).
```



Chapter 9

Choropleth maps

In Chapter 8 we made hotspot maps to show which areas in San Francisco had the most suicides. We made the maps in a number of ways and consistently found that suicides were most prevalent in northeast San Francisco. In this lesson we will make choropleth maps, which are shaded maps where each “unit” is some known area such as a state or neighborhood. Think of election maps where states are colored blue when a Democratic candidate wins that state and red when a Republican candidate wins. These are choropleth maps - each state is colored to indicate something. In this lesson we will continue to work on the suicide data and make choropleth maps shaded by the number of suicides in each neighborhood (we will define this later in the lesson) in the city.

Since we will be working more on the suicide data from San Francisco, let's read it in now.

```
library(readr)
suicide <- read_csv("data/san_francisco_suicide_2003_2017.csv")
#> Parsed with column specification:
#> cols(
#>   IncidntNum = col_double(),
#>   Category = col_character(),
#>   Descript = col_character(),
#>   DayOfWeek = col_character(),
#>   Date = col_character(),
```

```
#>   Time = col_time(format = ""),
#>   PdDistrict = col_character(),
#>   Resolution = col_character(),
#>   Address = col_character(),
#>   X = col_double(),
#>   Y = col_double(),
#>   Location = col_character(),
#>   PdId = col_double(),
#>   year = col_double()
#> )
suicide <- as.data.frame(suicide)
```

The package that we will use to handle geographic data and do most of the work in this lesson is `sf`. `sf` is a sophisticated package and does far more than what we will cover in this lesson. For more information about the package's features please see the website for it [here](#).

```
install.packages("sf")

library(sf)
#> Linking to GEOS 3.8.0, GDAL 3.0.4, PROJ 6.3.1
```

For this lesson we will need to read in a shapefile that depicts the boundaries of each neighborhood in San Francisco. A shapefile is similar to a `data.frame` but has information on how to draw a geographic boundary such as a state. The way `sf` reads in the shapefiles is through the `st_read()` function. Our input inside the `()` is a string with the name of the “.shp” file we want to read in (since we are telling R to read a file on the computer rather than an object that exists, it needs to be in quotes). This shapefile contains neighborhoods in San Francisco so we’ll call the object `sf_neighborhoods`.

I downloaded this data from San Francisco’s Open Data site [here](#), selecting the Shapefile format in the Export tab. If you do so yourself it’ll give you a zip file with multiple files in there. This is normal with shapefiles, you will have multiple files and only read in the file with the “.shp” extension to R. We still **do** need all of the files and `st_read()` is using them even if not explicitly called. So make sure every file downloaded is in the same working directory as the .shp file. The files from this site had hard to understand file names so I relabeled them all as “san_francisco_neighborhoods” though

that doesn't matter once it's read into R.

```
sf_neighborhoods <- st_read("data/san_francisco_neighborhoods.shp")
#> Reading layer `san_francisco_neighborhoods' from data source `C:\Users\user\Drop...
#> Simple feature collection with 41 features and 1 field
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -122.5149 ymin: 37.70813 xmax: -122.357 ymax: 37.8333
#> geographic CRS: WGS84 (DD)
```

As usual when dealing with a new data set, let's look at the first 6 rows.

```
head(sf_neighborhoods)
#> Simple feature collection with 6 features and 1 field
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -122.4543 ymin: 37.70822 xmax: -122.357 ymax: 37.80602
#> geographic CRS: WGS84 (DD)
#> 
#>          nhood                geometry
#> 1 Bayview Hunters Point MULTIPOLYGON (((-122.3816 3...
#> 2 Bernal Heights MULTIPOLYGON (((-122.4036 3...
#> 3 Castro/Upper Market MULTIPOLYGON (((-122.4266 3...
#> 4 Chinatown MULTIPOLYGON (((-122.4062 3...
#> 5 Excelsior MULTIPOLYGON (((-122.424 37...
#> 6 Financial District/South Beach MULTIPOLYGON (((-122.3875 3...
```

The last column is important. In shapefiles, the “geometry” column is the one with the instructions to make the map. This data has a single row for each neighborhood in the city. So the “geometry” column in each row has a list of coordinates which, if connected in order, make up that neighborhood. Since the “geometry” column contains the instructions to map, we can `plot()` it to show a map of the data.

```
plot(sf_neighborhoods$geometry)
```



Here we have a map of San Francisco broken up into neighborhoods. Is this a perfect representation of the neighborhoods in San Francisco? No. It is simply the city’s attempt to create definitions of neighborhoods. Indeed, you’re likely to find that areas at the border of neighborhoods are more similar to each other than they are to areas at the opposite side of their designated neighborhood. You can read a bit about how San Francisco determined the neighborhood boundaries [here](#) but know that this, like all geographic areas that someone has designated, has some degree of inaccuracy and arbitrariness in it. Like many things in criminology, this is just another limitation we will have to keep in mind.

In the `head()` results there was a section about something called “epsg” and “proj4string”. Let’s talk about that specifically since they are important for working with spatial data. A way to get just those two results in the `st_crs()` function which is part of `sf`. Let’s look at the “coordinate reference system” (CRS) for `sf_neighborhoods`.

```
st_crs(sf_neighborhoods)
Coordinate Reference System:
  User input: WGS84(DD)
  wkt:
GEOGCRS["WGS84(DD)",
  DATUM["WGS84",
```

```

ELLIPSOID["WGS84",6378137,298.257223563,
  LENGTHUNIT["metre",1,
    ID["EPSG",9001]]],
PRIMEM["Greenwich",0,
  ANGLEUNIT["degree",0.0174532925199433]],
CS[ellipsoidal,2],
  AXIS["geodetic longitude",east,
    ORDER[1],
    ANGLEUNIT["degree",0.0174532925199433]],
  AXIS["geodetic latitude",north,
    ORDER[2],
    ANGLEUNIT["degree",0.0174532925199433]]

```

An issue with working with geographic data is that [the Earth is not flat](#). Since the Earth is spherical, there will always be some distortion when trying to plot the data on a flat surface such as a map. To account for this we need to transform the longitude and latitude values we generally have to work properly on a map. We do so by “projecting” our data onto the areas of the Earth we want. This is a complex field with lots of work done on it (both abstractly and for R specifically) so this lesson will be an extremely brief overview of the topic and oversimplify some aspects of it.

If we look at the output of `st_crs(sf_neighborhoods)` we can see that the EPSG is set to 4326 and the proj4string (which tells us the current map projection) is “`+proj=longlat +datum=WGS84 +no_defs`”. This CRS, WGS84, is a standard CRS and is the one used whenever you use a GPS to find a location. To find the CRS for certain parts of the world see [here](#). If you search that site for “California” you’ll see that California is broken into 6 zones. The site isn’t that helpful on which zones are which but some Googling can often find state or region maps with the zones depicted there. We want California zone 3 which has the EPSG code 2227. We’ll use this code to project this data properly.

If we want to get the proj4string for 2227 we can use

```

st_crs(2227)
#> Coordinate Reference System:
#>   User input: EPSG:2227
#>   wkt:

```

```

#> PROJCRS["NAD83 / California zone 3 (ftUS)",
#>     BASEGEOGCRS["NAD83",
#>         DATUM["North American Datum 1983",
#>             ELLIPSOID["GRS 1980",6378137,298.257222101,
#>                 LENGTHUNIT["metre",1]],
#>             PRIMEM["Greenwich",0,
#>                 ANGLEUNIT["degree",0.0174532925199433]],
#>             ID["EPSG",4269]],
#>         CONVERSION["SPCS83 California zone 3 (US Survey feet)",
#>             METHOD["Lambert Conic Conformal (2SP)",
#>                 ID["EPSG",9802]],
#>             PARAMETER["Latitude of false origin",36.5,
#>                 ANGLEUNIT["degree",0.0174532925199433],
#>                 ID["EPSG",8821]],
#>             PARAMETER["Longitude of false origin",-120.5,
#>                 ANGLEUNIT["degree",0.0174532925199433],
#>                 ID["EPSG",8822]],
#>             PARAMETER["Latitude of 1st standard parallel",38.4333333333333,
#>                 ANGLEUNIT["degree",0.0174532925199433],
#>                 ID["EPSG",8823]],
#>             PARAMETER["Latitude of 2nd standard parallel",37.0666666666667,
#>                 ANGLEUNIT["degree",0.0174532925199433],
#>                 ID["EPSG",8824]],
#>             PARAMETER["Easting at false origin",6561666.667,
#>                 LENGTHUNIT["US survey foot",0.304800609601219],
#>                 ID["EPSG",8826]],
#>             PARAMETER["Northing at false origin",1640416.667,
#>                 LENGTHUNIT["US survey foot",0.304800609601219],
#>                 ID["EPSG",8827]]],
#>             CS[Cartesian,2],
#>                 AXIS["easting (X)",east,
#>                     ORDER[1],
#>                     LENGTHUNIT["US survey foot",0.304800609601219]],
#>                 AXIS["northing (Y)",north,
#>                     ORDER[2],
#>                     LENGTHUNIT["US survey foot",0.304800609601219]],
#>             USAGE[

```

```
#>      SCOPE["unknown"],
#>      AREA["USA - California - SPCS - 3"],
#>      BBOX[36.73,-123.02,38.71,-117.83],
#>      ID["EPSG",2227]
```

Note the text in “prj4string” that says “+units=us-ft”. This means that the units are in feet. Some projections have units in meters so be mindful of this when doing some analysis such as seeing if a point is within X feet of a certain area.

Let’s convert our sf_neighborhoods data to coordinate reference system 2227.

```
sf_neighborhoods <- st_transform(sf_neighborhoods, crs = 2227)
st_crs(sf_neighborhoods)
Coordinate Reference System:
  User input: EPSG:2227
  wkt:
PROJCRS["NAD83 / California zone 3 (ftUS)",
  BASEGEOGCRS["NAD83",
    DATUM["North American Datum 1983",
      ELLIPSOID["GRS 1980",6378137,298.257222101,
        LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
      ANGLEUNIT["degree",0.0174532925199433]],
  ID["EPSG",4269]],
CONVERSION["SPCS83 California zone 3 (US Survey feet)",
  METHOD["Lambert Conic Conformal (2SP)",
    ID["EPSG",9802]],
  PARAMETER["Latitude of false origin",36.5,
    ANGLEUNIT["degree",0.0174532925199433],
    ID["EPSG",8821]],
  PARAMETER["Longitude of false origin",-120.5,
    ANGLEUNIT["degree",0.0174532925199433],
    ID["EPSG",8822]],
  PARAMETER["Latitude of 1st standard parallel",38.4333333333333,
    ANGLEUNIT["degree",0.0174532925199433],
    ID["EPSG",8823]],
  PARAMETER["Latitude of 2nd standard parallel",37.0666666666667,
```

```

ANGLEUNIT["degree",0.0174532925199433],
ID["EPSG",8824]],
PARAMETER["Easting at false origin",6561666.667,
LENGTHUNIT["US survey foot",0.304800609601219],
ID["EPSG",8826]],
PARAMETER["Northing at false origin",1640416.667,
LENGTHUNIT["US survey foot",0.304800609601219],
ID["EPSG",8827]]],
CS[Cartesian,2],
AXIS["easting (X)",east,
ORDER[1],
LENGTHUNIT["US survey foot",0.304800609601219]],
AXIS["northing (Y)",north,
ORDER[2],
LENGTHUNIT["US survey foot",0.304800609601219]],
USAGE[
SCOPE["unknown"],
AREA["USA - California - SPCS - 3"],
BBOX[36.73,-123.02,38.71,-117.83]],
ID["EPSG",2227]]

```

9.1 Spatial joins

What we want to do with these neighborhoods is to find out which neighborhood each suicide occurred in and sum up the number of suicides per neighborhood. Once we do that, we can make a map at the neighborhood level and be able to measure suicides-per-neighborhood. A spatial join is very similar to regular joins where we merge two data sets based on common variables (such as state name or unique ID code of a person). In this case it merges based on some shared geographic feature such as if two lines intersect or (as we will do so here) if a point is within some geographic area.

Right now our *suicide* data is in a `data.frame` with some info on each suicide and the longitude and latitude of the suicide in separate columns. We want to turn this `data.frame` into a spatial object to allow us to find which neighborhood each suicide happened in. We can convert it into a spatial object using the `st_as_sf()` function from `sf`. Our input is first our data,

suicide. Then in the `coords` parameter we put a vector of the column names so the function knows which columns the longitude and latitude columns are so it can convert those columns to a “geometry” column like we saw in `sf_neighborhoods` earlier. We’ll set the CRS to be the WGS84 standard we saw earlier but we will change it to match the CRS that the neighborhood data has.

```
suicide <- st_as_sf(suicide,
                      coords = c("X", "Y"),
                      crs = "+proj=longlat +ellps=WGS84 +no_defs")
```

We want our suicides data in the same projection as the neighborhoods data so we need to use `st_transform()` to change the projection. Since we want the CRS to be the same as in `sf_neighborhoods`, we can set it using `st_crs(sf_neighborhoods)` to use the right CRS.

```
suicide <- st_transform(suicide,
                        crs = st_crs(sf_neighborhoods))
```

Now we can take a look at `head()` to see if it was projected.

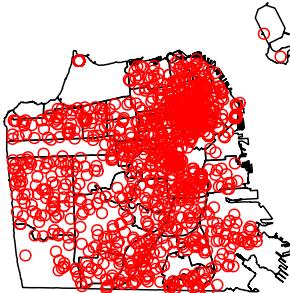
```
head(suicide)
#> Simple feature collection with 6 features and 12 fields
#> geometry type: POINT
#> dimension: XY
#> bbox: xmin: 5986822 ymin: 2091310 xmax: 6013739 ymax: 2117180
#> projected CRS: NAD83 / California zone 3 (ftUS)
#> IncidntNum Category Descript DayOfWeek Date
#> 1 180318931 SUICIDE ATTEMPTED SUICIDE BY STRANGULATION Monday 04/30/2018
#> 2 180315501 SUICIDE ATTEMPTED SUICIDE BY JUMPING Saturday 04/28/2018
#> 3 180295674 SUICIDE SUICIDE BY LACERATION Saturday 04/21/2018
#> 4 180263659 SUICIDE SUICIDE Tuesday 04/10/2018
#> 5 180235523 SUICIDE ATTEMPTED SUICIDE BY INGESTION Friday 03/30/2018
#> 6 180236515 SUICIDE SUICIDE BY ASPHYXIATION Thursday 03/29/2018
#> Time PdDistrict Resolution Address
#> 1 06:30:00 TARAVAL NONE 0 Block of BRUCE AV
#> 2 17:54:00 NORTHERN NONE 700 Block of HAYES ST
#> 3 12:20:00 RICHMOND NONE 3700 Block of CLAY ST
#> 4 05:13:00 CENTRAL NONE 0 Block of DRUMM ST
```

```
#> 5 09:15:00      TARAVAL      NONE 0 Block of FAIRFIELD WY
#> 6 17:30:00      RICHMOND      NONE    300 Block of 29TH AV
#>                                         Location      PdId year
#> 1 POINT (-122.45168059935614 37.72218061554315) 1.803189e+13 2018
#> 2 POINT (-122.42876060987851 37.77620120112792) 1.803155e+13 2018
#> 3 POINT (-122.45462091999406 37.7881754224736) 1.802957e+13 2018
#> 4 POINT (-122.39642194376758 37.79414474237039) 1.802637e+13 2018
#> 5 POINT (-122.46324153155875 37.72679184368551) 1.802355e+13 2018
#> 6 POINT (-122.48929119750689 37.782735835121265) 1.802365e+13 2018
#>                                         geometry
#> 1 POINT (5997229 2091310)
#> 2 POINT (6004262 2110838)
#> 3 POINT (5996881 2115353)
#> 4 POINT (6013739 2117180)
#> 5 POINT (5993921 2093059)
#> 6 POINT (5986822 2113584)
```

We can see it is now a “simple feature collection” with the correct projection. And we can see there is a new column called “geometry” just like in *sf_neighborhoods*. The type of data in “geometry” is POINT since our data is just a single location instead of a polygon like in the neighborhoods data.

Since we have both the neighborhoods and the suicides data let’s make a quick map to see the data.

```
plot(sf_neighborhoods$geometry)
plot(suicide$geometry, add = TRUE, col = "red")
```



Our next step is to combine these two data sets to figure out how many suicides occurred in each neighborhood. This will be a multi-step process so let's plan it out before beginning. Our suicide data is one row for each suicide, our neighborhood data is one row for each neighborhood. Since our goal is to map at the neighborhood-level we need to get the neighborhood where each suicide occurred then aggregate up to the neighborhood-level to get a count of the suicides-per-neighborhood. Then we need to combine that with that the original neighborhood data (since we need the "geometry" column) and we can then map it.

1. Find which neighborhood each suicide happened in
2. Aggregate suicide data until we get one row per neighborhood and a column showing the number of suicides in that neighborhood
3. Combine with the neighborhood data
4. Make a map

We'll start by finding the neighborhood where each suicide occurred using the function `st_join()` which is a function in `sf`. This does a spatial join and finds the polygon (neighborhood in our case) where each point is located in. Since we will be aggregating the data, let's call the output of this function `suicide_agg`. The order in the `()` is important! For our aggregation we want the output to be at the suicide-level so we start with the `suicide` data. In the next step we'll see why this matters.

```
suicide_agg <- st_join(suicide, sf_neighborhoods)
```

Let's look at the first 6 rows.

```
head(suicide_agg)
#> Simple feature collection with 6 features and 13 fields
#> geometry type: POINT
#> dimension: XY
#> bbox: xmin: 5986822 ymin: 2091310 xmax: 6013739 ymax: 2117180
#> projected CRS: NAD83 / California zone 3 (ftUS)
#>   IncidntNum Category Descript DayOfWeek
#> 1 180318931 SUICIDE ATTEMPTED SUICIDE BY STRANGULATION Monday 04/30/
#> 2 180315501 SUICIDE ATTEMPTED SUICIDE BY JUMPING Saturday 04/28/
#> 3 180295674 SUICIDE SUICIDE BY LACERATION Saturday 04/21/
#> 4 180263659 SUICIDE SUICIDE Tuesday 04/10/
#> 5 180235523 SUICIDE ATTEMPTED SUICIDE BY INGESTION Friday 03/30/
#> 6 180236515 SUICIDE SUICIDE BY ASPHYXIATION Thursday 03/29/
#>   Time PdDistrict Resolution Address
#> 1 06:30:00 TARAVAL NONE 0 Block of BRUCE AV
#> 2 17:54:00 NORTHERN NONE 700 Block of HAYES ST
#> 3 12:20:00 RICHMOND NONE 3700 Block of CLAY ST
#> 4 05:13:00 CENTRAL NONE 0 Block of DRUMM ST
#> 5 09:15:00 TARAVAL NONE 0 Block of FAIRFIELD WY
#> 6 17:30:00 RICHMOND NONE 300 Block of 29TH AV
#>   Location PdId year
#> 1 POINT (-122.45168059935614 37.72218061554315) 1.803189e+13 2018
#> 2 POINT (-122.42876060987851 37.77620120112792) 1.803155e+13 2018
#> 3 POINT (-122.45462091999406 37.7881754224736) 1.802957e+13 2018
#> 4 POINT (-122.39642194376758 37.79414474237039) 1.802637e+13 2018
#> 5 POINT (-122.46324153155875 37.72679184368551) 1.802355e+13 2018
#> 6 POINT (-122.48929119750689 37.782735835121265) 1.802365e+13 2018
#>   nhood geometry
#> 1 Oceanview/Merced/Ingleside POINT (5997229 2091310)
#> 2 Hayes Valley POINT (6004262 2110838)
#> 3 Presidio Heights POINT (5996881 2115353)
#> 4 Financial District/South Beach POINT (6013739 2117180)
#> 5 West of Twin Peaks POINT (5993921 2093059)
#> 6 Outer Richmond POINT (5986822 2113584)
```

There is now the *nhood* column from the neighborhoods data which says which neighborhood the suicide happened in. Now we can aggregate up to the neighborhood-level. For now we will use the code to aggregate the number of suicides per neighborhood. Remember, the `aggregate()` command aggregates a numeric value by some categorical value. Here we aggregate the number of suicides per neighborhood. So our code will be

```
aggregate(number_suicides ~ nhood, data = suicide_agg, FUN =
sum)
```

We actually don't have a variable with the number of suicides so we need to make that. We can simply call it *number_suicides* and give it that value of 1 since each row is only one suicide.

```
suicide_agg$number_suicides <- 1
```

Now we can write the `aggregate()` code and save the results back into *suicide_agg*.

```
suicide_agg <- aggregate(number_suicides ~ nhood, data = suicide_agg, FUN = sum)
```

Let's check a summary of the *number_suicides* variable we made.

```
summary(suicide_agg$number_suicides)
#>   Min. 1st Qu. Median   Mean 3rd Qu.    Max.
#>   1.00  15.00  24.00  33.08  38.50  141.00
```

The minimum is one suicide per neighborhood, 33 on average, and 141 in the neighborhood with the most suicides. So what do we make of this data? Well, there are some data issues that cause problems in these results. Let's think about the minimum value. Did every single neighborhood in the city have at least one suicide? No. Take a look at the number of rows in this data, keeping in mind there should be one row per neighborhood.

```
nrow(suicide_agg)
#> [1] 39
```

And let's compare it to the *sf_neighborhoods* data.

```
nrow(sf_neighborhoods)
#> [1] 41
```

The suicides data is missing 2 neighborhoods. That is because if no suicides occurred there, there would never be a matching row in the data so that neighborhood wouldn't appear in the suicide data. That's not going to be a major issue here but is something to keep in mind in future research.

The data is ready to merge with the *sf_neighborhoods* data. We'll introduce a new function that makes merging data simple. This function comes from the *dplyr* package so we need to install and tell R we want to use it using *library()*.

```
install.packages("dplyr")

library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

The function we will use is *left_join()* which takes two parameters, the two data sets to join together.

```
left_join(data1, data2)
```

This function joins these data and keeps all of the rows from the left data and every column from both data sets. It combines the data based on any matching columns (matching meaning same column name) in both data sets. Since in our data sets, the column *nhood* exists in both, it will merge the data based on that column.

There are two other functions that are similar but differ based on which rows they keep.

- *left_join()* - All rows from Left data and all columns from Left and Right data

- `right_join()` - All rows from Right data and all columns from Left and Right data
- `full_join()` - All rows and all columns from Left and Right data

We could alternatively use the `merge()` function which is built into R but that function is slower than the `dplyr` functions and requires us to manually set the matching columns.

We want to keep all rows in `sf_neighborhoods` (keep all neighborhoods) so we can use `left_join(sf_neighborhoods, suicide_agg)`. Let's save the results into a new data.frame called `sf_neighborhoods_suicide`.

```
sf_neighborhoods_suicide <- left_join(sf_neighborhoods, suicide_agg)
#> Joining, by = "nhood"
```

If we look at `summary()` again for `number_suicides` we can see that there are now 2 rows with NAs. These are the neighborhoods where there were no suicides so they weren't present in the `suicide_agg` data.

```
summary(sf_neighborhoods_suicide$number_suicides)
#>      Min. 1st Qu. Median     Mean 3rd Qu.      Max.    NA 's
#>      1.00   15.00  24.00   33.08  38.50  141.00        2
```

We need to convert these values to 0. We will use the `is.na()` function to conditionally find all rows with an NA value in the `number_suicides` column and use square bracket notation to change the value to 0.

```
sf_neighborhoods_suicide$number_suicides[is.na(sf_neighborhoods_suicide$number_suicid
```

Checking it again we see that the minimum is now 0 and the mean number of suicides decreases a bit to about 31.5 per neighborhood.

```
summary(sf_neighborhoods_suicide$number_suicides)
#>      Min. 1st Qu. Median     Mean 3rd Qu.      Max.
#>      0.00   12.00  23.00   31.46  36.00  141.00
```

9.2 Making choropleth maps

Finally we are ready to make some choropleth maps.

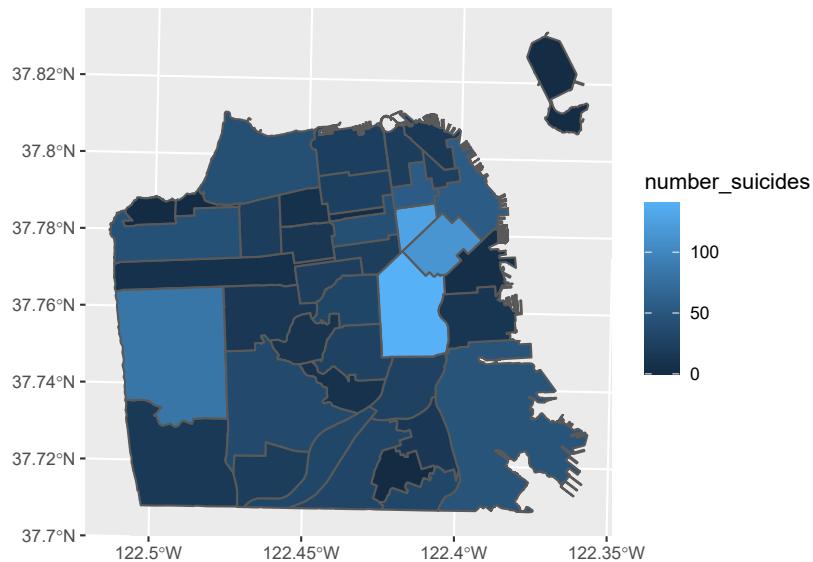
For these maps we are going to use `ggplot2` again so we need to load it.

```
library(ggplot2)
```

`ggplot2`'s benefit is you can slowly build graphs or maps and improve the graph at every step. Earlier, we used functions such as `geom_line()` for line graphs and `geom_point()` for scatter plots. For mapping these polygons we will use `geom_sf()` which knows how to handle spatial data.

As usual we will start with `ggplot()`, inputting our data first. Then inside of `aes` (the aesthetics of the graph/map) we use a new parameter `fill`. In `fill` we will put in the `number_suicides` column and it will color the polygons (neighborhoods) based on values in that column. Then we can add the `geom_sf()`.

```
ggplot(sf_neighborhoods_suicide, aes(fill = number_suicides)) +
  geom_sf()
```

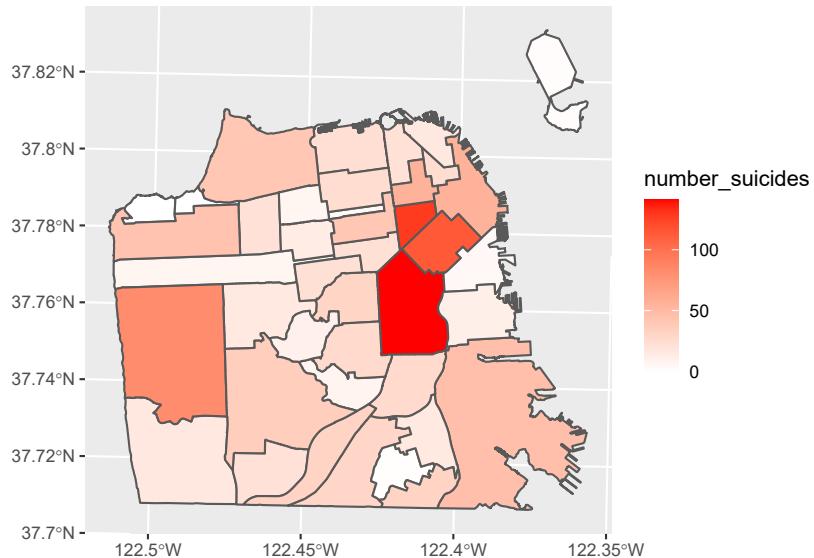


We have now created a choropleth map showing the number of suicides per neighborhood in San Francisco! Based on the legend, neighborhoods that are light blue have the most suicides while neighborhoods that are dark blue have the fewest (or none at all). Normally we'd want the opposite, with darker areas signifying a greater amount of whatever the map is showing.

We can use `scale_fill_gradient()` to set the colors to what we want. We

input a color for low value and a color for high value and it'll make the map shade by those colors.

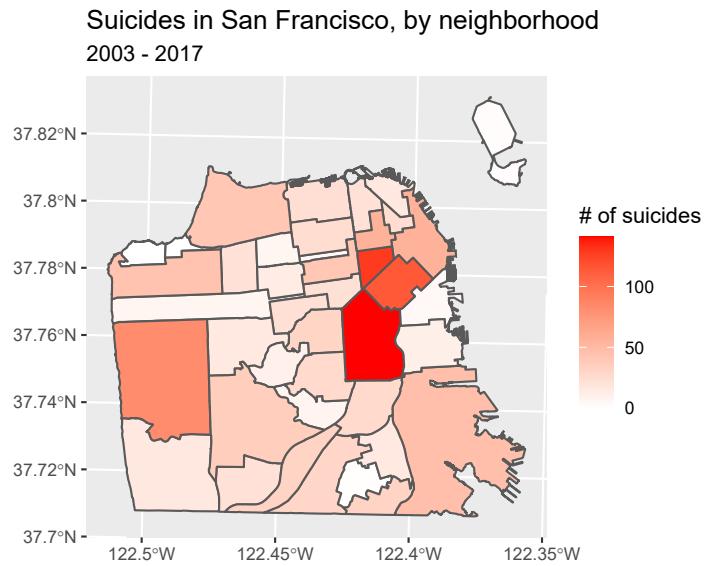
```
ggplot(sf_neighborhoods_suicide, aes(fill = number_suicides)) +
  geom_sf() +
  scale_fill_gradient(low = "white", high = "red")
```



This gives a much better map and clearly shows the areas where suicides are most common and where there were no suicides.

To make this map easier to read and look better, let's add a title to the map and to the legend.

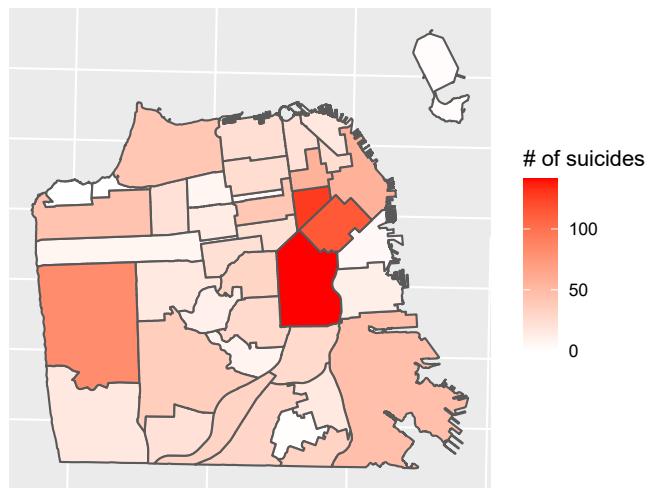
```
ggplot(sf_neighborhoods_suicide, aes(fill = number_suicides)) +
  geom_sf() +
  scale_fill_gradient(low = "white", high = "red") +
  labs(fill = "# of suicides",
       title = "Suicides in San Francisco, by neighborhood",
       subtitle = "2003 - 2017")
```



Since the coordinates don't add anything to the map, let's get rid of them.

```
ggplot(sf_neighborhoods_suicide, aes(fill = number_suicides)) +  
  geom_sf() +  
  scale_fill_gradient(low = "white", high = "red") +  
  labs(fill = "# of suicides",  
       title = "Suicides in San Francisco, by neighborhood",  
       subtitle = "2003 - 2017") +  
  theme(axis.text.x = element_blank(),  
        axis.text.y = element_blank(),  
        axis.ticks = element_blank())
```

Suicides in San Francisco, by neighborhood
2003 - 2017



So what should we take away from this map? There are more suicides in the downtown area than any other place in the city. Does this mean that people are more likely to kill themselves there than elsewhere? Not necessarily. A major mistake people make when making a choropleth map (or really any type of map) is accidentally making a population map. The darker shaded parts of our map are also where a lot of people live. So if there are more people, it is reasonable that there would be more suicides (or crimes, etc.). What we'd really want to do is make a rate per some population (usually per 100k though this assumes equal risk for every person in the city which isn't really correct) to control for population differences.

We'll use this data in Chapter 10 to make interactive choropleth maps so let's save it.

```
save(sf_neighborhoods_suicide, file = "data/sf_neighborhoods_suicide.rda")
```


Chapter 10

Interactive maps

While maps of data are useful, their ability to show incident-level information is quite limited. They tend to show broad trends - where crime happened in a city - rather than provide information about specific crime incidents. While broad trends are important, there are significant drawbacks about being unable to get important information about an incident without having to check the data. An interactive map bridges this gap by showing trends while allowing you to zoom into individual incidents and see information about each incident.

For this lesson we will be using data on every marijuana dispensary in San Francisco that has an active dispensary license as of late September 2019. The file is called “san_francisco_marijuana_geocoded.csv”.

When downloaded from California’s Bureau of Cannabis Control ([here](#) if you’re interested) the data contains the address of each dispensary but does not have coordinates. Without coordinates we are unable to map points, meaning we need to geocode them. Geocoding is the process of taking an address and getting the longitude and latitude of that address for mapping. For this lesson I’ve already geocoded the data and we’ll learn how to do so in Chapter [17](#).

```
library(readr)
marijuana <- read_csv("data/san_francisco_marijuana_geocoded.csv")
#> Parsed with column specification:
#> cols(
```

```
#> License_Number = col_character(),
#> License_Type = col_character(),
#> Business_Owner = col_character(),
#> Business_Contact_Information = col_character(),
#> Business_Structure = col_character(),
#> Premise_Address = col_character(),
#> Status = col_character(),
#> Issue_Date = col_character(),
#> Expiration_Date = col_character(),
#> Activities = col_character(),
#> `Adult-Use/Medicinal` = col_character(),
#> lon = col_double(),
#> lat = col_double()
#> )
marijuana <- as.data.frame(marijuana)
```

10.1 Why do interactive graphs matter?

10.1.1 Understanding your data

The most important thing to learn from this course is that understanding your data is crucial to good research. Making interactive maps is a very useful way to better understand your data as you can immediately see geographic patterns and quickly look at characteristics of those incidents to understand them.

In this lesson we will make a map of each marijuana dispensary in San Francisco that lets you click on the dispensary and see some information about it. If we see a cluster of dispensaries, we can click on each one to see if they are similar - for example if owned by the same person. Though it is possible to find these patterns just looking at the data, it is easier to be able to see a geographic pattern and immediately look at information about each incident.

10.1.2 Police departments use them

Interactive maps are popular in large police departments such as Philadelphia and New York City. They allow easy understanding of geographic patterns

in the data and, importantly, allow such access to people who do not have the technical skills necessary to create the maps. If nothing else, learning interactive maps may help you with a future job.

10.2 Making the interactive map

As usual, let's take a look at the top 6 rows of the data.

```
head(marijuana)
#>   License_Number          License_Type Business_Owner
#> 1 C10-0000614-LIC Cannabis - Retailer License Terry Muller
#> 2 C10-0000586-LIC Cannabis - Retailer License Jeremy Goodin
#> 3 C10-0000587-LIC Cannabis - Retailer License Justin Jarin
#> 4 C10-0000539-LIC Cannabis - Retailer License Ondyn Herschelle
#> 5 C10-0000522-LIC Cannabis - Retailer License Ryan Hudson
#> 6 C10-0000523-LIC Cannabis - Retailer License Ryan Hudson
#>
#> 1                               OUTER SUNSET HOLDINGS, LLC : Barbary Coast Sunset
#> 2                               URBAN FLOWERS : Urban Pharm : Email- hilary@urbanph
#> 3                               CCPC, INC. : The Green Door : Email- alicia@greendoorsf.
#> 4 SEVENTY SECOND STREET : Flower Power SF : Email- flowerpowersf@hotmail.com :
#> 5 HOWARD STREET PARTNERS, LLC : The Apothecarium : Email- Ryan@apothecarium.c
#> 6 DEEP THOUGHT, LLC : The Apothecarium : Email- ryan@pothecarium.c
#>   Business_Structure           Premise_Address Status
#> 1 Limited Liability Company 2165 IRVING ST san francisco, CA 94122 Active
#> 2 Corporation 122 10TH ST SAN FRANCISCO, CA 941032605 Active
#> 3 Corporation 843 Howard ST SAN FRANCISCO, CA 94103 Active
#> 4 Corporation 70 SECOND ST SAN FRANCISCO, CA 94105 Active
#> 5 Limited Liability Company 527 Howard ST San Francisco, CA 94105 Active
#> 6 Limited Liability Company 2414 Lombard ST San Francisco, CA 94123 Active
#>   Issue_Date Expiration_Date Activities Adult-Use/Medicinal
#> 1 9/13/2019      9/12/2020 N/A for this license type BOTH
#> 2 8/26/2019      8/25/2020 N/A for this license type BOTH
#> 3 8/26/2019      8/25/2020 N/A for this license type BOTH
#> 4 8/5/2019       8/4/2020 N/A for this license type BOTH
#> 5 7/29/2019      7/28/2020 N/A for this license type BOTH
#> 6 7/29/2019      7/28/2020 N/A for this license type BOTH
```

```
#>      lon      lat
#> 1 -122.4811 37.76314
#> 2 -122.4158 37.77476
#> 3 -122.4035 37.78228
#> 4 -122.4004 37.78823
#> 5 -122.3965 37.78784
#> 6 -122.4414 37.79944
```

This data has information about the type of license, who the owner is, where the dispensary is (as an address and as coordinates), and contact information. We'll be making a map showing every dispensary in the city and make it so when you click a dot it'll make a popup showing information about that dispensary.

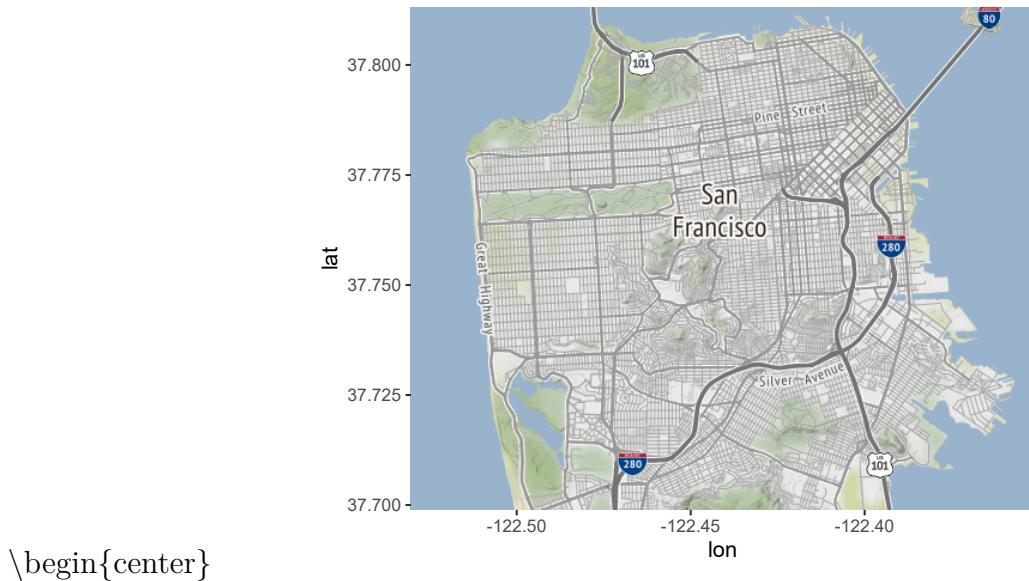
We will use the package `leaflet` for our interactive map. `leaflet` produces maps similar to Google Maps with circles (or any icon we choose) for each value we add to the map. It allows you to zoom in, scroll around, and provides context to each incident that isn't available on a static map.

```
install.packages("leaflet")
library(leaflet)
```

To make a `leaflet` map we need to run the function `leaflet()` and add a tile to the map. A tile is simply the background of the map. This [website](#) provides a large number of potential tiles to use, though many are not relevant to our purposes of crime mapping.

We will use a standard tile from Open Street Maps. This tile gives street names and highlights important features such has parks and large stores which provides useful contexts for looking at the data. The `attribution` parameter isn't strictly necessary but it is good form to say where your tile is from.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors')
```



When you run the above code it shows a world map (copied several times). Zoom into it and it'll start showing relevant features of wherever you're looking.

Note the `%>%` between the `leaflet()` function and the `addTiles()` function. This is called a “pipe” in R and is used like the `+` in `ggplot()` to combine multiple functions together. This is used heavily in what is called the “tidyverse”, a series of packages that are prominent in modern R and useful for data analysis. We won’t be covering them in this book but for more information on them you can check the [tidyverse website](#). For this lesson you need to know that each piece of the `leaflet` function must end with `%>%` for the next line to work.

To add the points to the graph we use the function `addMarkers()` which has two parameters, `lng` and `lat`. For both parameters we put the column in which the longitude and latitude are, respectively.

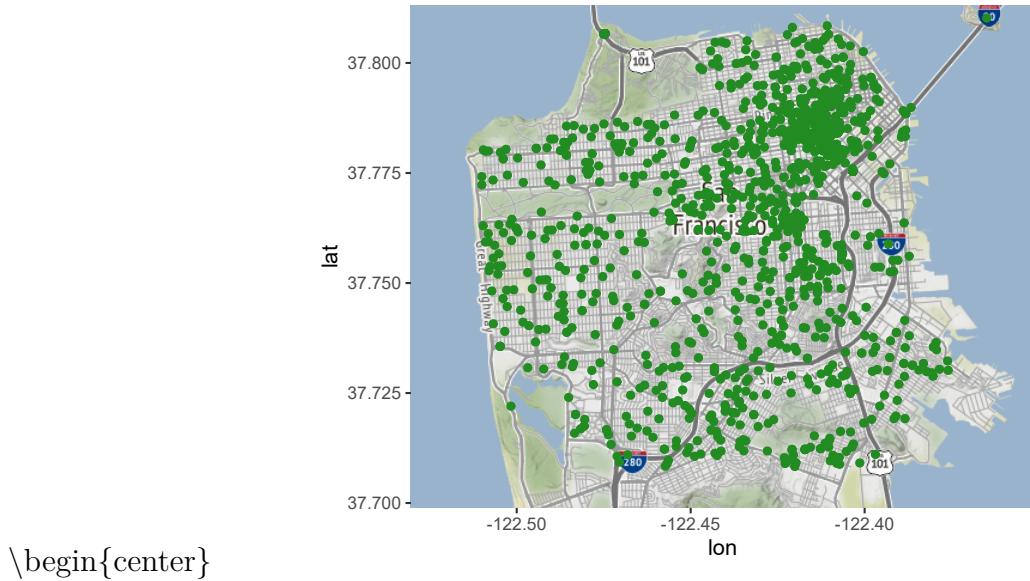
```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
          attribution = '&copy; <a href="http://openstreetmap.org">
          OpenStreetMap</a> contributors') %>%
  addMarkers(lng = marijuana$lon,
            lat = marijuana$lat)
```



```
\begin{center}
```

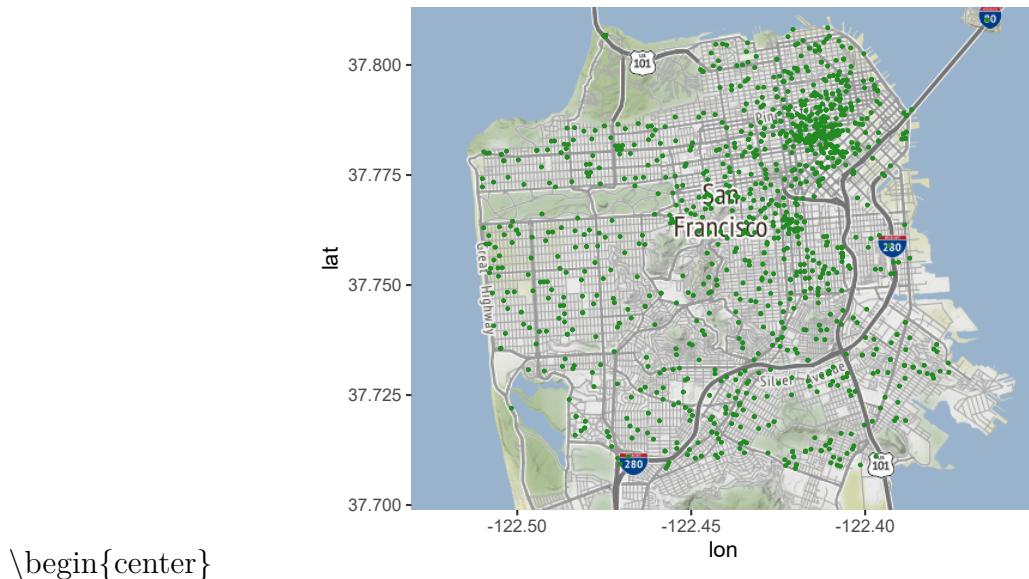
It now adds an icon indicating where every dispensary in our data is. You can zoom in and scroll around to see more about where the dispensaries are. There are only a few dozen locations in the data so the popups overlapping a bit doesn't affect our map too much. If we had more - such as crime data with millions of offenses - it would make it very hard to read. To change the icons to circles we can change the function `addMarkers()` to `addCircleMarkers()`, keeping the rest of the code the same,

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat)
```



This makes the icon into circles which take up less space than icons. To adjust the size of our icons we use the `radius` parameter in `addMarkers()` or `addCircleMarkers()`. The larger the radius, the larger the icons.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
          attribution = '&copy; <a href="http://openstreetmap.org">
          OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat,
                  radius = 5)
```

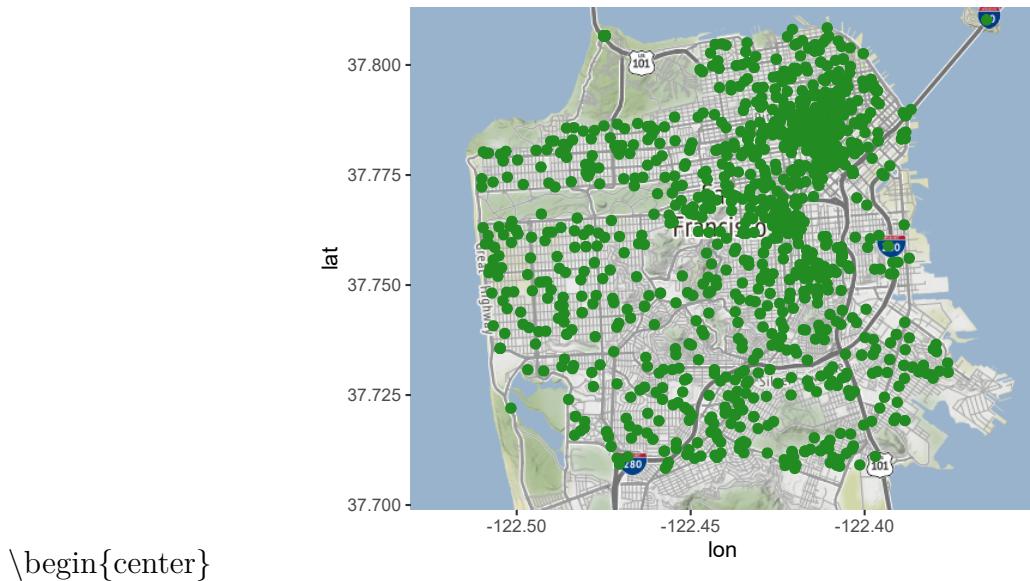


Setting the `radius` option to 5 shrinks the size of the icon a lot. In your own maps you'll have to fiddle with this option to get it to look the way you want. Let's move on to adding information about each icon when clicked upon.

10.3 Adding popup information

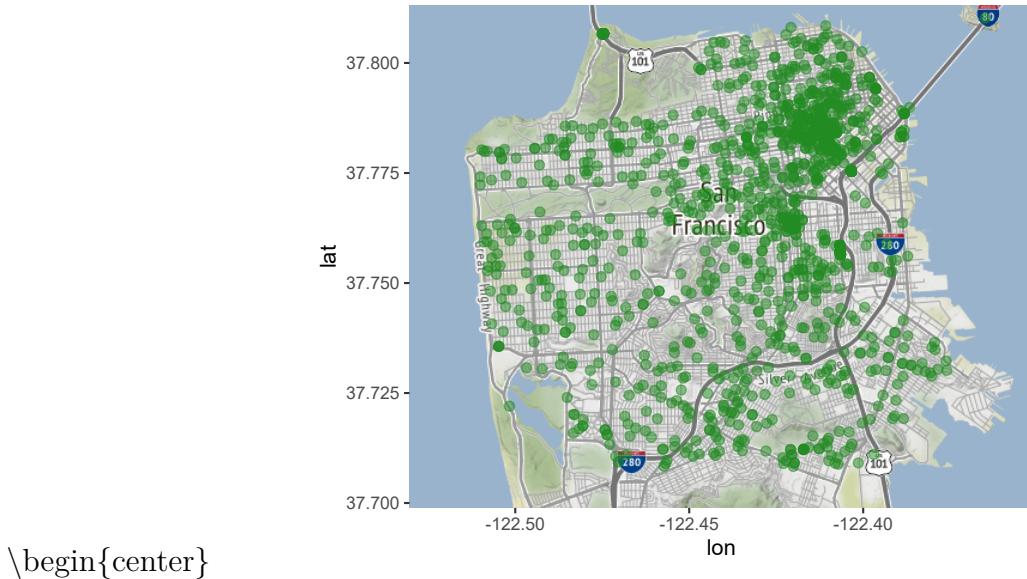
The parameter `popup` in the `addMarkers()` or `addCircleMarkers()` functions lets you input a character value (if not already a character value it will convert it to one) and that will be shown as a popup when you click on the icon. Let's start simple here by inputting the business owner column in our data and then build it up to a more complicated popup.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat,
                  radius = 5,
                  popup = marijuana$Business_Owner)
```



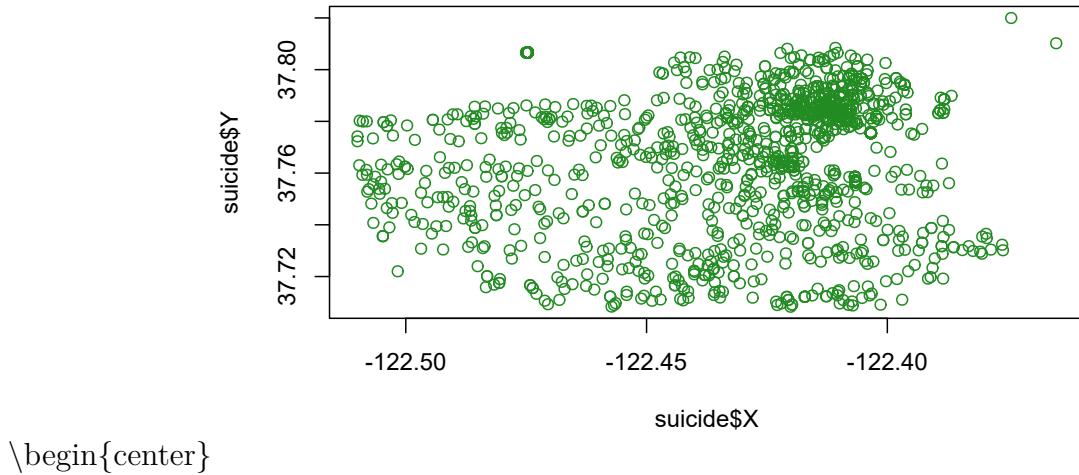
Try clicking around and you'll see that the owner of the dispensary you clicked on appears over the dot. We usually want to have a title indicating what the value in the popup means. We can do this by using the `paste()` function to combine text explaining the value with the value itself. Let's add the words "Business Owner:" before the business owner column.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat,
                  radius = 5,
                  popup = paste("Business Owner:", marijuana$Business_Owner))
```



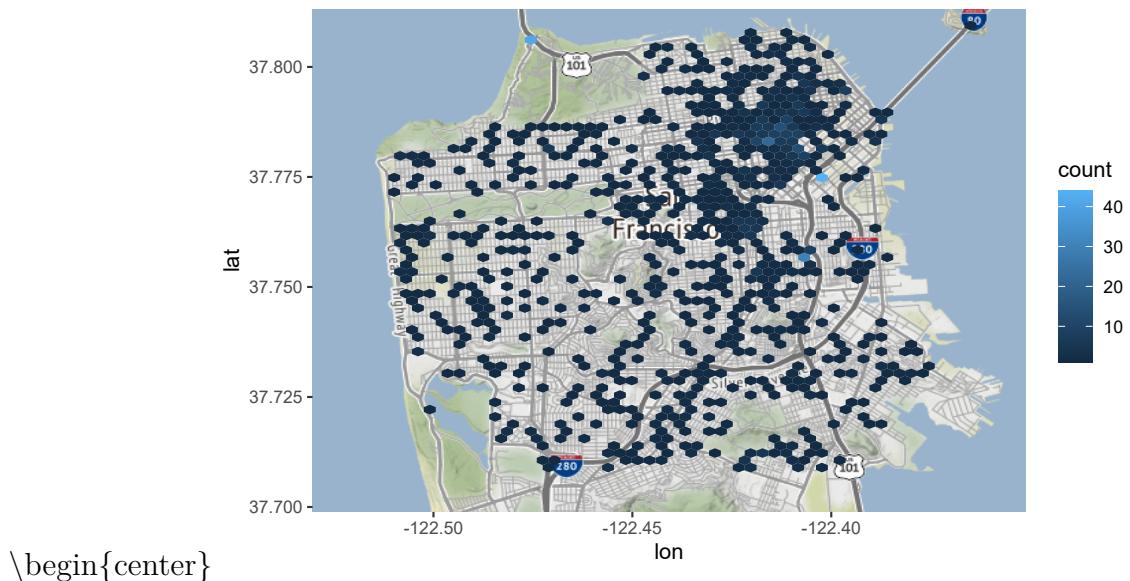
We don't have too much information in the data but we let's add the address and license number to the popup by adding them to the `paste()` function we're using.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat,
                  radius = 5,
                  popup = paste("Business Owner:", marijuana$Business_Owner,
                               "Address:", marijuana$Premise_Address,
                               "License:", marijuana$License_Number))
```



Just adding the location text makes it try to print out everything on one line which is hard to read. If we add the text
 where we want a line break it will make one.
 is the HTML tag for line-break which is why it works making a new line in this case.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat,
                  radius = 5,
                  popup = paste("Business Owner:", marijuana$Business_Owner,
                               "<br>",
                               "Address:", marijuana$Premise_Address,
                               "<br>",
                               "License:", marijuana$License_Number))
```



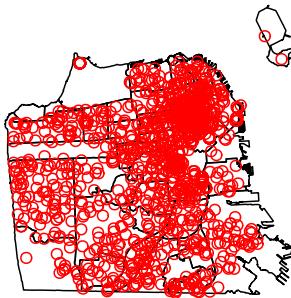
10.4 Dealing with too many markers

In our case with only 33 rows of data, turning the markers to circles solves our visibility issue. In cases with many more rows of data, this doesn't always work. A solution for this is to cluster the data into groups where the dots only show if you zoom down.

If we add the code `clusterOptions = markerClusterOptions()` to our `addCircleMarkers()` it will cluster for us.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addCircleMarkers(lng = marijuana$lon,
                  lat = marijuana$lat,
                  radius = 5,
                  popup = paste("Business Owner:", marijuana$Business_Owner,
                               "<br>",
                               "Address:", marijuana$Premise_Address,
                               "<br>",
                               "License:", marijuana$License_Number),
```

```
clusterOptions = markerClusterOptions()
```



\begin{center}

Locations close to each other are grouped together in fairly arbitrary groupings and we can see how large each grouping is by moving our cursor over the circle. Click on a circle or zoom in and it will show smaller groupings at lower levels of aggregation. Keep clicking or zooming in and it will eventually show each location as its own circle.

This method is very useful for dealing with huge amounts of data as it avoids overflowing the map with too many icons at one time. A downside, however, is that the clusters are created arbitrarily meaning that important context, such as neighborhood, can be lost.

10.5 Interactive choropleth maps

In Chapter 9 we worked on choropleth maps which are maps with shaded regions, such as states colored by which political party won them in an election. Here we will make interactive choropleth maps where you can click on a shaded region and see information about that region. We'll make the same map as before - neighborhoods shaded by the number of suicides.

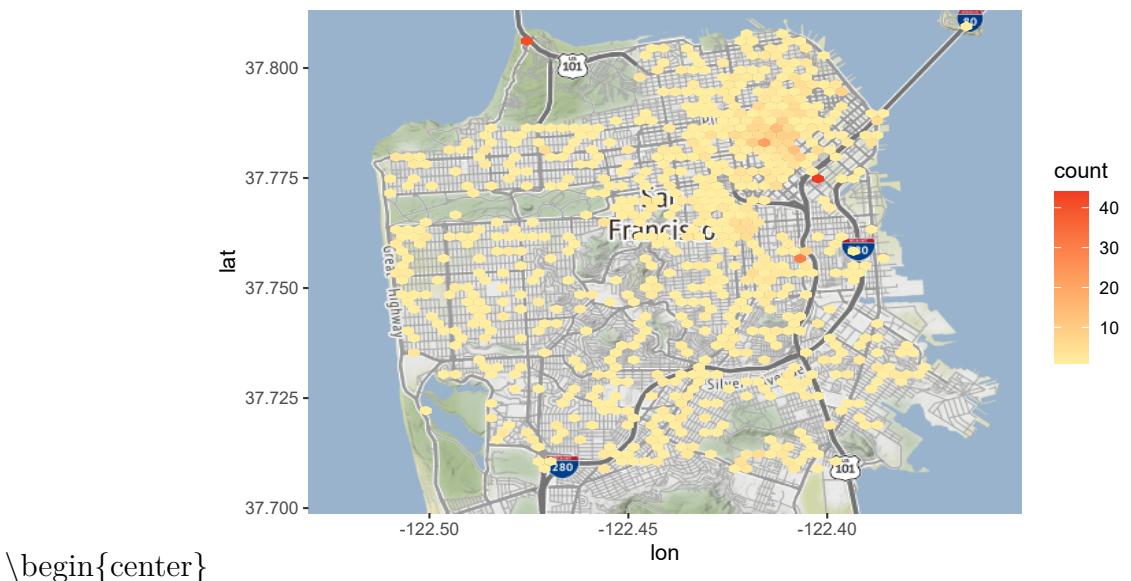
Let's load the San Francisco suicides-by-neighborhood data that we made

earlier.

```
load("data/sf_neighborhoods_suicide.rda")
```

We'll begin the `leaflet` map similar to before but use the function `addPolygons()` and our input here is the geometry column of `sf_neighborhoods_suicide`.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addPolygons(data = sf_neighborhoods_suicide$geometry)
#> Warning: sf layer is not long-lat data
#> Warning: sf layer has inconsistent datum (+proj=lcc +lat_0=36.5 +lon_0=-122.427)
#> Need '+proj=longlat +datum=WGS84'
```



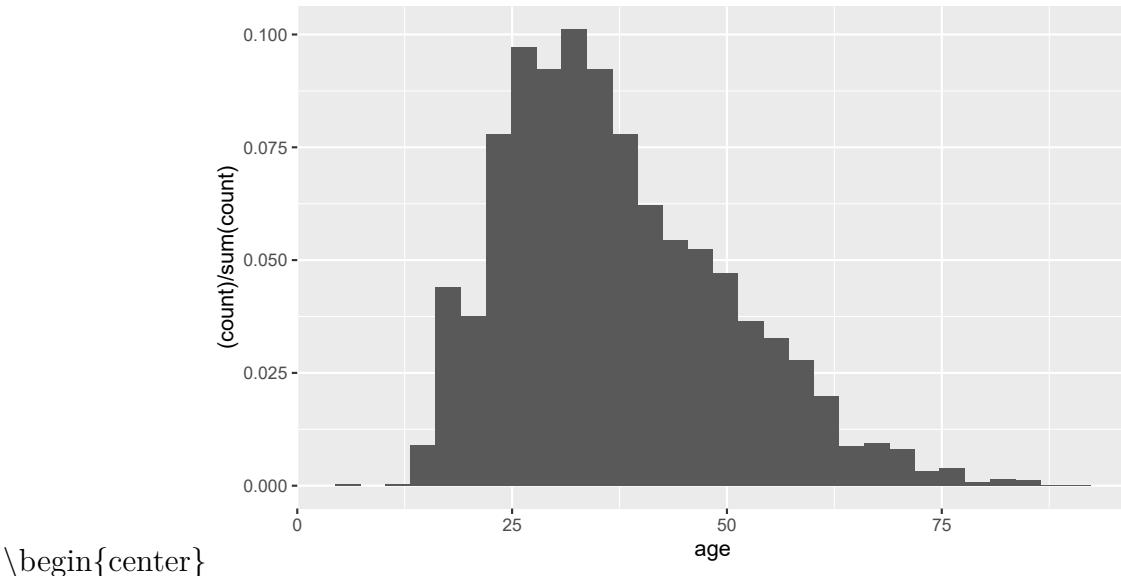
It gives us a blank map because our polygons are projected to San Francisco's projection while the `leaflet` map expects the standard CRS, WGS84 which uses longitude and latitude. So we need to change our projection to that using the `st_transform()` function from the `sf` package.

```
library(sf)
#> Linking to GEOS 3.8.0, GDAL 3.0.4, PROJ 6.3.1
```

```
sf_neighborhoods_suicide <- st_transform(sf_neighborhoods_suicide,
                                         crs = "+proj=longlat +datum=WGS84")
```

Now let's try again.

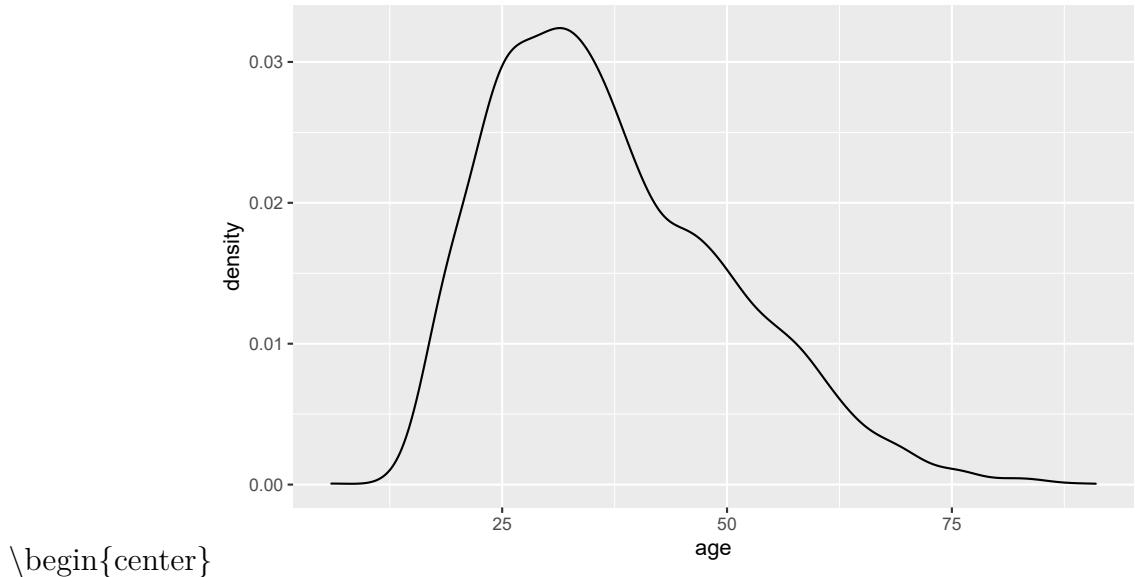
```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addPolygons(data = sf_neighborhoods_suicide$geometry)
```



It made a map with large blue lines indicating each neighborhood. Let's change the appearance of the graph a bit before making a popup or shading the neighborhoods. The parameter **color** in **addPolygons()** changes the color of the lines - let's change it to black. The lines are also very large, blurring into each other and making the neighborhoods hard to see. We can change the **weight** parameter to alter the size of these lines - smaller values are smaller lines. Let's try setting this to 1.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
```

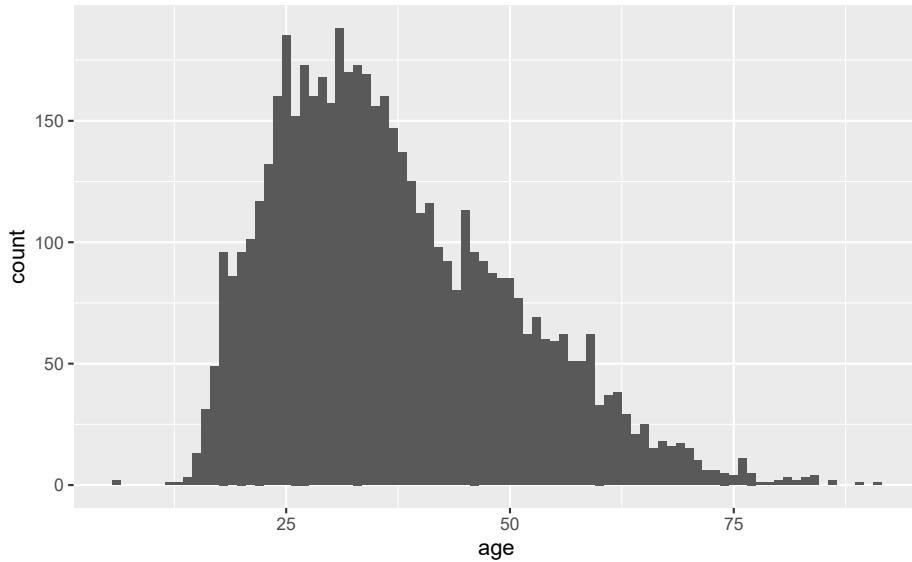
```
addPolygons(data = sf_neighborhoods_suicide$geometry,
            color = "black",
            weight = 1)
```



That looks better and we can clearly distinguish each neighborhood now.

As we did earlier, we can add the popup text directly to the function which makes the geographic shapes, in this case `addPolygons()`. Let's add the *nhood* column value - the name of that neighborhood - and the number of suicides that occurred in that neighborhood. As before, when we click on a neighborhood a popup appears with the output we specified.

```
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addPolygons(data = sf_neighborhoods_suicide$geometry,
              col = "black",
              weight = 1,
              popup = paste0("Neighborhood: ", sf_neighborhoods_suicide$nhood,
                            "<br>",
                            "Number of Suicides: ", sf_neighborhoods_suicide$
```



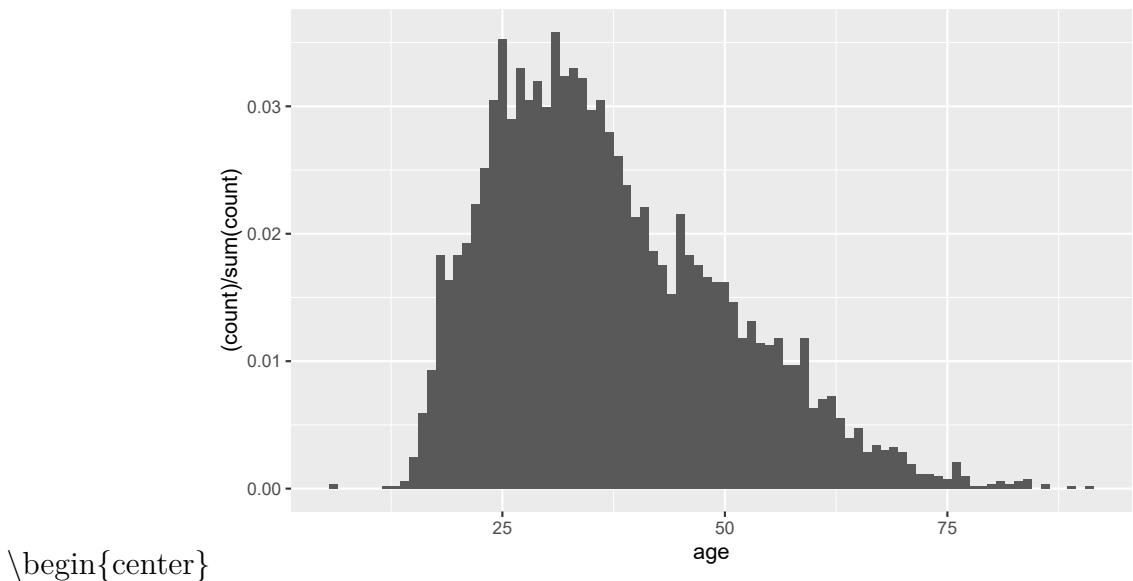
\begin{center}

For these types of maps we generally want to shade each polygon to indicate how frequently the event occurred in the polygon. We'll use the function `colorNumeric()` which takes a lot of the work out of the process of coloring in the map. This function takes two inputs, first a color palette which we can get from the site [colorbrewer2](#). Let's use the fourth bar in the Sequential page, which is light orange to red. If you look in the section with each HEX value it says that the palette is "3-class OrRd". The "3-class" just means we selected 3 colors, the "OrRd" is the part we want. That will tell `colorNumeric()` to make the palette using these colors. The second parameter is the column for our numeric variable, *number_suicides*.

We will save the output of `colorNumeric("OrRd", sf_neighborhoods_suicide$number_suicides)` as a new variable which we'll call *pal* for convenience. Then inside of `addPolygons()` we'll set the parameter `fillColor` to `pal(sf_neighborhoods_suicide$number_suicide)` running this function on the column. What this really does is determine which color every neighborhood should be based on the value in the *number_suicides* column.

```
pal <- colorNumeric("OrRd", sf_neighborhoods_suicide$number_suicides)
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
OpenStreetMap</a> contributors') %>%
```

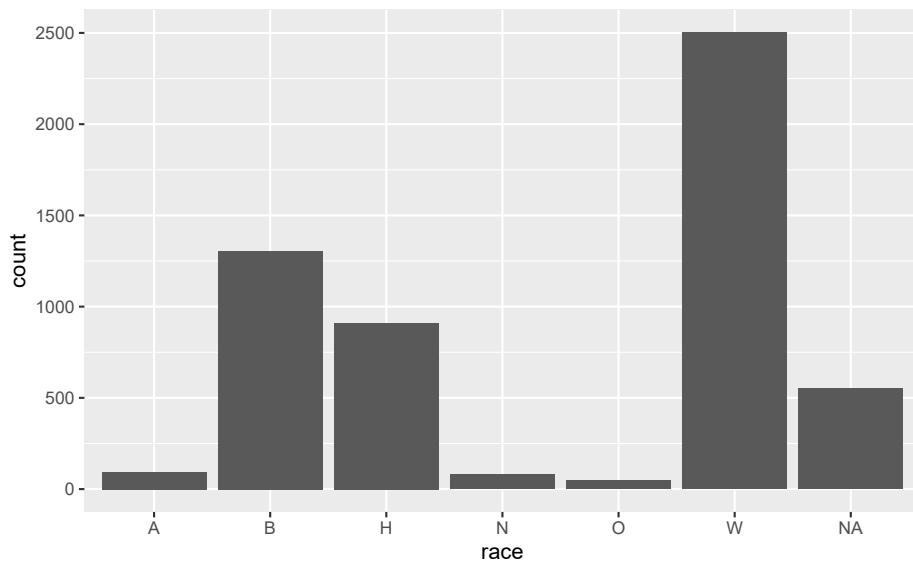
```
addPolygons(data = sf_neighborhoods_suicide$geometry,
            col = "black",
            weight = 1,
            popup = paste0("Neighborhood: ", sf_neighborhoods_suicide$nhood,
                          "<br>",
                          "Number of Suicides: ", sf_neighborhoods_suicide$number_suicides),
            fillColor = pal(sf_neighborhoods_suicide$number_suicides))
```



Since the neighborhoods are transparent, it is hard to distinguish which color is shown. We can make each neighborhood a solid color by setting the parameter `fillOpacity` inside of `addPolygons()` to 1.

```
pal <- colorNumeric("OrRd", sf_neighborhoods_suicide$number_suicides)
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
          attribution = '&copy; <a href="http://openstreetmap.org">
                        OpenStreetMap</a> contributors') %>%
  addPolygons(data = sf_neighborhoods_suicide$geometry,
              col = "black",
              weight = 1,
              popup = paste0("Neighborhood: ", sf_neighborhoods_suicide$nhood,
                            "<br>",
```

```
"Number of Suicides: ", sf_neighborhoods_suicide$number_
fillColor = pal(sf_neighborhoods_suicide$number_suicides),
fillOpacity = 1)
```



\begin{center}

To add a legend to this we use the function `addLegend()` which takes three parameters. `pal` asks which color palette we are using - we want it to be the exact same as we use to color the neighborhoods so we'll use the `pal` object we made. The `values` parameter is used for which column our numeric values are from, in our case the `number_suicides` column so we'll input that. Finally `opacity` determines how transparent the legend will be. As each neighborhood is set to not be transparent at all, we'll also set this to 1 to be consistent.

```
pal <- colorNumeric("OrRd", sf_neighborhoods_suicide$number_suicides)
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
    attribution = '&copy; <a href="http://openstreetmap.org">
      OpenStreetMap</a> contributors') %>%
  addPolygons(data = sf_neighborhoods_suicide$geometry,
    col = "black",
    weight = 1,
    popup = paste0("Neighborhood: ", sf_neighborhoods_suicide$nhood,
```

```

    "<br>",
    "Number of Suicides: ", sf_neighborhoods_suicide$number_suicides),
fillColor = pal(sf_neighborhoods_suicide$number_suicides),
fillOpacity = 1) %>%
addLegend(pal = pal,
values = sf_neighborhoods_suicide$number_suicides,
opacity = 1)

```



\begin{center}

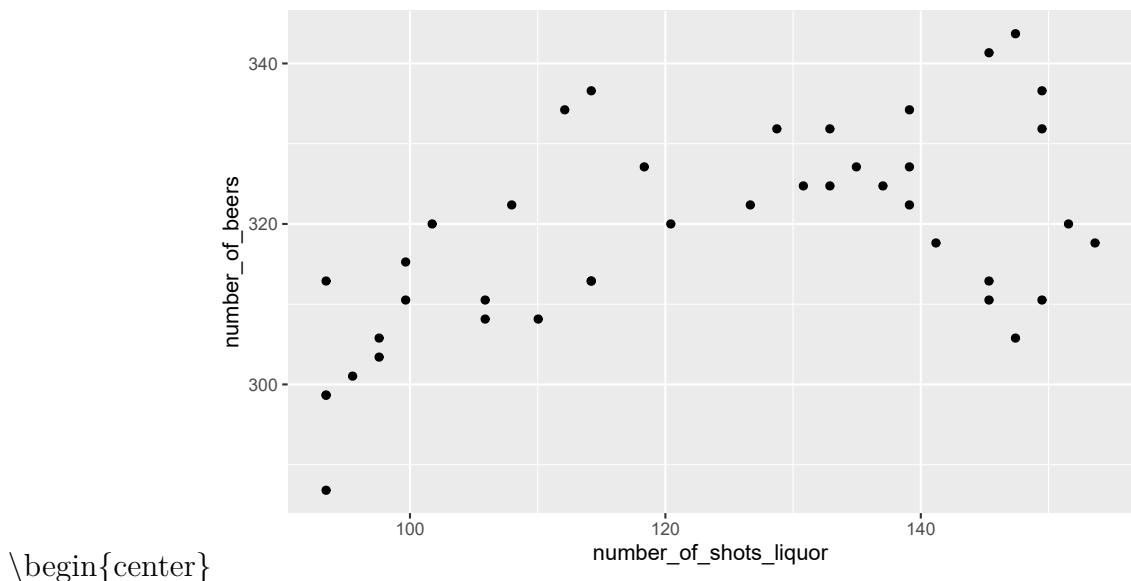
Finally, we can add a title to the legend using the `title` parameter inside of `addLegend()`.

```

pal <- colorNumeric("OrRd", sf_neighborhoods_suicide$number_suicides)
leaflet() %>%
  addTiles('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
           attribution = '&copy; <a href="http://openstreetmap.org">
                         OpenStreetMap</a> contributors') %>%
  addPolygons(data = sf_neighborhoods_suicide$geometry,
              col = "black",
              weight = 1,
              popup = paste0("Neighborhood: ", sf_neighborhoods_suicide$nhood,
                            "<br>",
                            "Number of Suicides: ", sf_neighborhoods_suicide$number_suicides))

```

```
  fillColor = pal(sf_neighborhoods_suicide$number_suicides),  
  fillOpacity = 1) %>%  
addLegend(pal = pal,  
         values = sf_neighborhoods_suicide$number_suicides,  
         opacity = 1,  
         title = "Suicides")
```



\begin{center}

Chapter 11

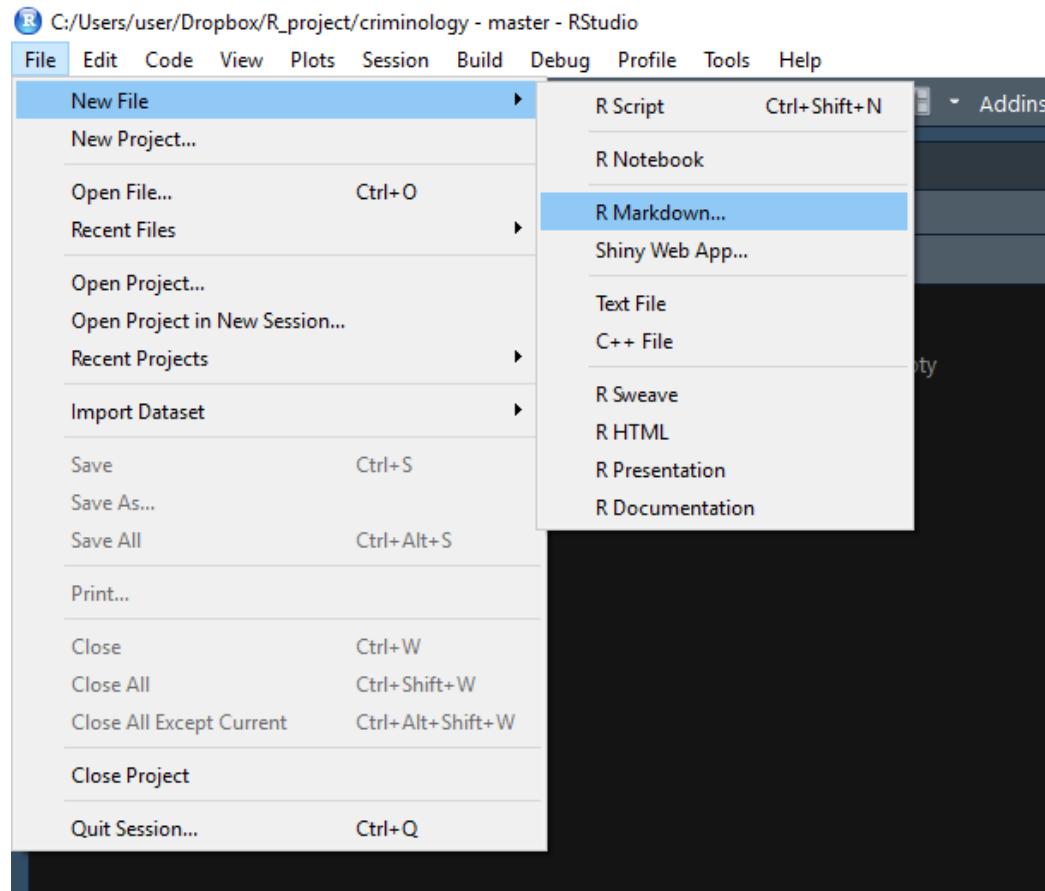
R Markdown

When conducting research your end product is usually a Word Document or a PDF which reports on the research you've done, often including several graphs or tables. In many cases people do the data work in R, producing the graphs or numbers for the table, and then write up the results in Word or LaTeX. While this is a good system, there are significant drawbacks, mainly that if you change the graph or table you need to change it in R **and** change it in the report. If you only do this rarely it isn't much of a problem. However, doing so many times can increase both the amount of work and the likelihood of an error occurring from forgetting to change something or changing it incorrectly. We can avoid this issue by using R Markdown, R's way of writing a document and incorporating R code within.

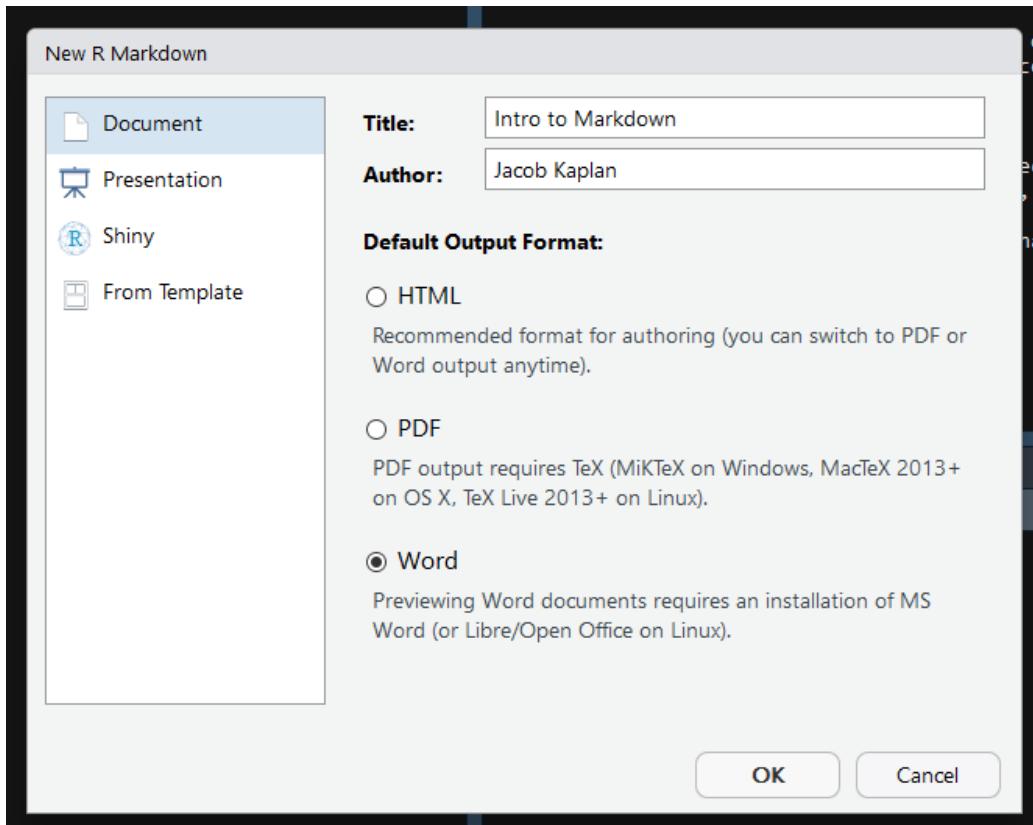
This chapter will only briefly introduce R Markdown, for a comprehensive guide please see [this excellent book](#). For a cheat sheet on R Markdown see [here](#).

What R Markdown does is let you type exactly as you would in Microsoft Word and insert the code to make the table or graph in the places you want it. If you change the code, the document will have the up-to-date result already, reducing your workload. There is some additional formatting you have to do when using R Markdown but it is minimal and is well-worth the return on the effort. This book, for example, was made entirely using R Markdown.

To open up a R Markdown file click File from the top menu, then New File, and then R Markdown...



From here it'll open up a window where you select the title, author, and type of output. You can always change all three of these selections right in the R Markdown file after making your selection here. Selecting PDF may require you to download additional software to get it to output - some OS may already have the software installed. For a nice guide to using PDF with R Markdown, see [here](#).



When you click OK, it will open a new R Markdown file that is already populated with example text and code. You can delete this entirely or modify it as needed.

The screenshot shows the RStudio interface with the 'Untitled1' tab selected. The code editor displays an R Markdown file with the following content:

```
1 ---  
2 title: "Intro to Markdown"  
3 author: "Jacob Kaplan"  
4 date: "March 11, 2018"  
5 output: word_document  
6 ---  
7  
8 ```{r setup, include=FALSE}  
9 knitr::opts_chunk$set(echo = TRUE)  
10  
11  
12 ## R Markdown  
13  
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring  
HTML, PDF, and MS Word documents. For more details on using R Markdown see  
http://rmarkdown.rstudio.com.  
15  
16 When you click the **Knit** button a document will be generated that includes both  
content as well as the output of any embedded R code chunks within the document. You can  
embed an R code chunk like this:  
17  
18 ```{r cars}  
19 summary(cars)  
20  
21  
22 ## Including Plots  
23  
24 You can also embed plots, for example:  
25  
26 ```{r pressure, echo=FALSE}  
27 plot(pressure)
```

When you output that file as a PDF it will look like the image below.

Intro to Markdown

Jacob Kaplan

March 11, 2018

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

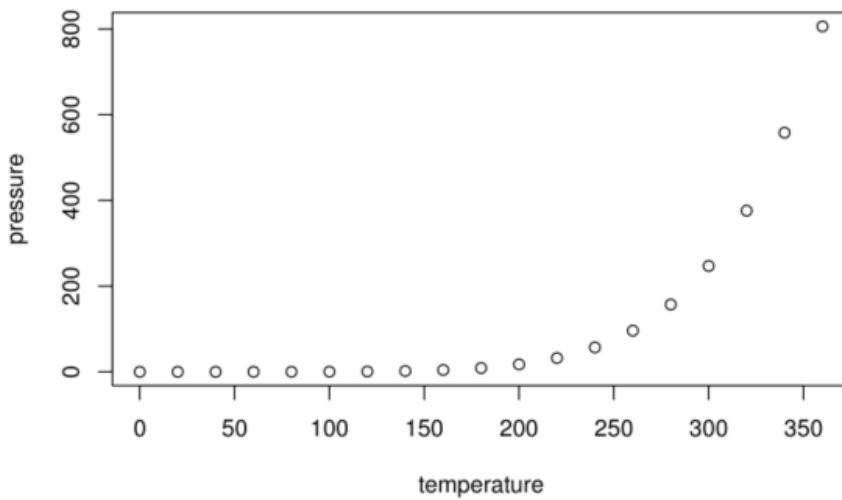
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed          dist
## Min.   : 4.0   Min.   :  2.00
## 1st Qu.:12.0   1st Qu.: 26.00
## Median :15.0   Median : 36.00
## Mean   :15.4   Mean   : 42.98
## 3rd Qu.:19.0   3rd Qu.: 56.00
## Max.   :25.0   Max.   :120.00
```

Including Plots

You can also embed plots, for example:



R converted the file into a PDF, executing the code and using the formatting specified. In an R Script a `#` means that the line is a comment. In an R Markdown file, the `#` signifies that the line is a section header. There are 6 possible headers, made by combining the `#` together - a `#` is the largest header while `#####` is the smallest header. As with comments, they must be at the beginning of a line.

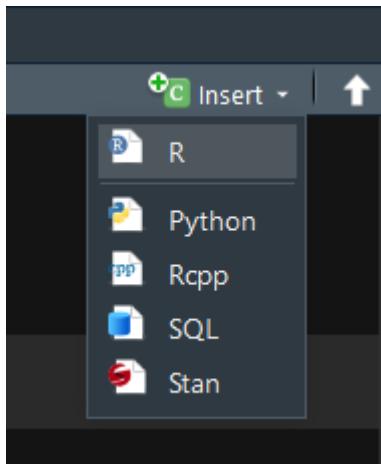
The word “Knit” was surrounded by two asterix `*` in the R Markdown file and became bold in the PDF because that is how R Markdown sets bolding - to make something italics using a single asterix like `this`. If you’re interested in more advanced formatting please see the book or cheat sheet linked earlier.

Other than the section headers, most of what you do in R Markdown is exactly the same as in Word. You can write text as you would normally and it will look exactly as you write it.

11.1 Code

The reason R Markdown is so useful is because you can include code output in the file. In R Markdown we write code in what is called a “code chunk”. These are simply areas in the document which R knows it should evaluate as R code. You can see three of them in the example - at lines 8-9 setting a default for the code, lines 18-20 to run the `summary()` function on the `cars` data (a data set built into R), and lines 26-28 (and cut off in the screenshot) to make a plot of the data set `pressure` (another data set built into R).

To make a chunk click Insert near the top right, then R.



It will then make an empty code chunk where your cursor is.



Notice the three ‘` at the top and bottom of the chunk. Don’t touch these! They tell R that anything in it is a code chunk (i.e. that R should run the code). Inside the squiggly brackets {} are instructions about how the code is outputted. Here you can specify, among other things if the code will be outputted or just the output itself, captions for tables or graphs, and formatting for the output. Include all of these options after the `r` in the squiggly brackets. Multiple options must be separated by a comma (just like options in normal R functions).

If you do not have the R Markdown file in the same folder as your data, you’ll need to set the working directory in a chunk before reading the data (you do so exactly like you would in an R Script). However, once a working directory is set, or the data is read in, it applies for all following chunks. You will also need to run any packages (using `library()`) to use them in a chunk. It is good form to set your working directory, load any data, and load any packages you need in the first chunk to make it easier to keep track of what you’re using.

11.1.1 Hiding code in the output

When you’re making a report for a general audience you generally only want to keep the output (e.g. a graph or table), not the code you used. At early stages in writing the report or when you’re collaborating with someone who wants to see your code, it is useful to include the code in the R Markdown output.

If you look at the second code chunk in the screenshot (lines 18-20) it includes the function `summary(cars)` as the code and the options `{r cars}` (the “cars” simply names the code chunk “cars” for if you want to reference the chunk - or its output if a table or graph - later, but does not change the code chunk’s behavior). In the output it shows both the code it used and the output of the code. This is because by default a code chunk shows both. To set it to only show the output, we need to set the parameter `echo` to `FALSE` inside of the `{}`.

In the third code chunk (lines 26-28), that parameter is set to false as it is `{r pressure, echo=FALSE}`. In the output it only shows the graph, not the code that was used.

11.2 Inline Code

You can also include R code directly in the text of your document and it will return the output of that code. To use it, you simply need to setup an inline code chunk using the tick mark followed by the lowercase letter R, the code you want to use, and then end it using another tick mark. This is called using inline code. When you have a table or visualization to output, this isn’t the proper method, it is best for small pieces of text to add to your document. This is most useful for when you want to include some descriptive info, such as the number of respondents to a survey or the mean of some variable, in the text of your document. Inline code will only present the output of the code and doesn’t show the code itself. Below is an example of inline code - see the image below that for what it looks like with the code.

The dataset `mtcars` has 32 rows and 11 columns. The mean of the `mpg` column is 20.090625. When rounded, the mean is 20.

Table 11.1: This is an example table caption

	mpg	cyl	disp	hp	drat
Mazda RX4	21.0	6	160	110	3.90
Mazda RX4 Wag	21.0	6	160	110	3.90
Datsun 710	22.8	4	108	93	3.85
Hornet 4 Drive	21.4	6	258	110	3.08
Hornet Sportabout	18.7	8	360	175	3.15

The dataset `mtcars` has `r nrow(mtcars)` rows and `r ncol (mtcars)` columns. The mean of the `mpg` column is `r mean(mtcars\$mpg)`.

11.3 Tables

There are a number of packages that make nice tables in R Markdown. We will use the `knitr` package for this example.

The easiest way to make a table in Markdown is to make a `data.frame` with all the data (and column names) you want and then show that `data.frame` (there are also packages that can make tables from regression output though that won't be covered in this lesson). For this example we will subset the `mtcars` data (which is included in R) to just the first 5 rows and columns. The `kable` function from the `knitr` package will then make a nice looking table. With `kable` you can add the caption directly in the `kable()` function. The option `echo` in our code chunk is not set to `FALSE` here so you can see the code.

```
library(knitr)
mtcars_small <- mtcars[1:5, 1:5]
kable(mtcars_small, caption = "This is an example table caption")
```

For another package to make very nice looking tables, see [this guide](#) to the `kableExtra` package.

11.4 Footnotes

In your writing, you'll often have sentences that you want to include but are auxiliary to your main point (or, frequently, to include links to specific resources such as a website where you got data from). In these cases you'll want to include that info as a footnote, which is a section at the bottom of the page for this kind of information. To create a footnote in RMarkdown, you use the carrot ^ followed immediately by square brackets []. Put the text inside of the [] and it'll print that at the bottom of the page. A footnote will look like this: ^[This sentence will be printed as a footnote]. In cases where you have a very long footnote it may extend to the next page and will be again at the bottom of the page. Look down at the bottom of this page to see the footnote (in a PDF or Word Doc, the footnote will be on the page you create it on, however since websites are just one long page without breaks, this footnote is at the very bottom of this entire page.¹ When you use a footnote, you'll usually put it immediately after the punctuation of the sentence it should be after. Note that footnotes are numbered so you can identify them. There's a blue superscript 1 where we made the footnote so people reading know the context - i.e. which part of the text they relate to. If we make another footnote, it'll be numbered sequentially, such that the next one is 2, the next is 3, etc.

If you're familiar with LaTeX you can use LaTeX code such as \footnote{} where the text goes inside the {}. But note that citations (which we'll learn in Section 11.5) won't work properly in the footnote if made this way. You can use LaTeX code - and use LaTeX packages - in RMarkdown if you'd like and it'll operate (in most cases) like normal LaTeX.

11.5 Citation

In academic research you will need to cite the papers that you are referencing. RMarkdown has a built-in way to cite papers, though it's a bit of a process to get everything setup. You'll need the citation data in BibTeX format and we'll walk through the steps from finding an article that you want to cite to citing it in your RMarkdown file. First, a brief overview of what kinds of citations you can use. There are two types of citations you can use, in-

¹This is an example of a footnote.

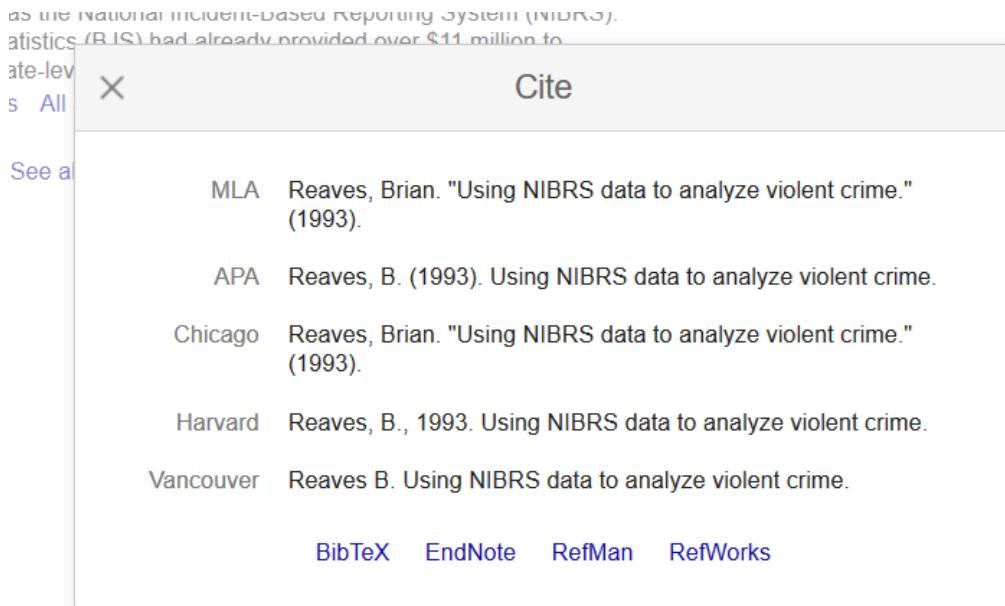
text and parenthetical. You'll use in-text citations when you want to have the author names be in the text, and parenthetical citations when you want everything to be in parentheses.

Note, there may be other ways to get the citations in the right format; I'm just showing you one way to do so. For this example, we'll use the article "Using NIBRS data to analyze violent crime" by Brian Reaves that was published in 1993. We'll walk through the process from finding the article on Google Scholar to citing it in your paper. First, from Google Scholar we'll search for the article title.

The screenshot shows a search bar at the top with the query "Using NIBRS data to analyze violent crime". Below the search bar is a list of search results. The first result is highlighted with a blue circle around the link text. The result is titled "Using NIBRS data to analyze violent crime" by B Reaves - 1993 - books.google.com. The abstract describes the shift from UCR to NIBRS. Below the abstract, there are links for "Cited by 31", "Related articles", and "All 5 versions". A blue circle highlights the "Cited by 31" link. At the bottom of the screenshot, a message says "Showing the best result for this search. See all results".

This returns all articles that meet your search criteria. Since we're searching for a specific article title, we only get one result. The result shows some basic info about the article - title, date, name, abstract. Below the abstract are some important things. First, and circled in blue in the above photo, is a link that looks like quotation marks. This is what we'll click on to get to the BibTeX citation. While not necessary for citation, the next two links may come in handy during your research. 'Cited by 31' means that 31 published (in some format that Google can locate, not necessarily peer-reviewed articles) articles have cited this article. If you click the link it'll open up a Google Scholar page with all of these articles. This is a good way to find relevant literature. Clicking 'Related articles' does the same thing but with articles that Google Scholar deems similar, not necessarily articles linking to the one you're looking up.

But back to the quotes link circled in blue. Click this and it'll make a popup, shown below, of ways to cite this article in various formats. We'll have RMarkdown automatically generate the citation in the format we want so we don't need to worry about this. Instead, click the BibTeX link at the bottom left.



When you click it, it'll open up a new page with that article's citation in BibTeX form, as shown below. This basically is just a way to tell a computer how to cite it properly. Each part of the citation - author, year, title, etc. - is its own piece. Take a close look at the section immediately after the first squiggly bracket, “reaves1993using”. This is how you'll identify the article in RMarkdown so R knows which article to cite. It's essentially the citation's name. It's created automatically by combining the author name (first author if there are more than one author, publication year, and part of the title). You can change it to whatever you want it to be called.

```
@misc{reaves1993using,
  title={Using NIBRS data to analyze violent crime},
  author={Reaves, Brian},
  year={1993},
  publisher={US Department of Justice, Office of Justice Programs, Bureau of Justice~...}
}
```

Note at the end of the publisher section are the characters “~...”. This looks like a mistake made by Google Scholar so we’ll need to delete that so it isn’t included in a paper we use this citation in. When using Google Scholar, you’ll occasionally find issues like this which you’ll need to fix manually - a bigger issue is apostrophes or other punctuation may copy over from Google Scholar weird (meaning that it copies as a character that your computer, and thus RMarkdown, doesn’t understand) and need to be rewritten so RMarkdown will run. You can rewrite it by just deleting the punctuation and typing it using your keyboard. This isn’t always an issue so don’t worry about it unless you get an error with the citations when outputting your document.

Below is the citation included in my .bib file, and the start of another citation also included in the file. A .bib file is basically a text file that programs can read to get citation info. You’ll have all of your citations (in the BibTeX format) in this one file. To make a .bib file you can open up a text document, such as through the Notepad app in Windows, and paste the BibTeX that you’ve copied from Google Scholar. Save this file as a .bib extension (by renaming it filename.bib) and you’ll have a usable .bib file.

Note that I have the word NIBRS surrounded by squiggly brackets {}. That is because by default RMarkdown (and other citation generators such as Overleaf) will only capitalize the first letter of the title or the first letter following a colon. Since NIBRS is an abbreviation and should be capitalized, I put it in the {} to force it to remain capitalized. This is often a problem with abbreviations or country names (such as United States) in the paper title I’ve also deleted the weird characters at the end of the publisher section. Since all citations you use for a project (I have a single .bib file that I use for projects since much of my work is on the same topic and the citations overlap across papers) are in one .bib file, you can see the start of another article cited below the Reaves citation.

```

@misc{reaves1993using,
  title={Using {NIBRS} data to analyze violent crime},
  author={Reaves, Brian},
  year={1993},
  publisher={US Department of Justice, Office of Justice
Programs, Bureau of Justice}
}

@article{jain2000recruitment,
  title={Recruitment, selection and promotion of visible-minority
and {A}boriginal police officers in selected {C}anadian police
services},
  author={Jain, Harish C and Singh, Parbudyal and Agocs, Carol},
  journal={Canadian Public Administration},
  volume={43}
}

```

To use citations from your .bib file, add `bibliography: references_file_name.bib` to the head of your RMarkdown file. If your .bib file isn't in the RMarkdown file's working directory, as my example below is not, you'll need to include the path in the file name.

```

-->
title: "Ambient Lighting and Arrests: Evidence from a Natural Experiment"
author: Jacob Kaplan^([Department of Criminology, 483 McNeil Building, University of
Pennsylvania, jacobkap@sas.upenn.edu. For helpful comments and suggestions, thank you to
Sara-Laure Faraji and Kristina Block. All remaining errors are my own.])
bibliography: C:/Users/user/Dropbox/Penn/PhD/Research/global_references.bib

```

Now that we have the citation in BibTeX format, put it in our .bib file, and told RMarkdown where to look for that file, we are ready to finally cite that article. To use a citation we simply put the @ sign in front of the citation name (in our case “reaves1993using”) so we would write `@reaves1993using`. This will give us an in-text citation, with the author name in the text and the year in parentheses. Adding a - right in front of the @ will cause the citation to show just the year, not the author’s name. You’ll usually want to use this if you’re already named the author earlier in the sentence. Generally we will want parenthetical citations, with both the authors and the year in parentheses. To do this, we put the citation inside of square brackets like this `[@reaves1993using]`. If we’re citing multiple articles, we separate each citation using a semicolon `[@reaves1993using; @jain2000recruitment]`.

Here’s what the results look like when citing that Reaves article, see the photo below for what this looks like just as code.

(Reaves, 1993)

Reaves (1993)

(1993)

(1993)

(Reaves, 1993; Jain et al., 2000)

If you use a citation that isn't in your .bib file, RMarkdown will present three question marks in place of the citation.

(?)

```
[@reaves1993using]
@reaves1993using
-@reaves1993using
[-@reaves1993using]
[@reaves1993using; @jain2000recruitment]
If you use a citation that isn't in your .bib file, RMarkdown will present three question
marks in place of the citation.

[@wrongCitation]
```

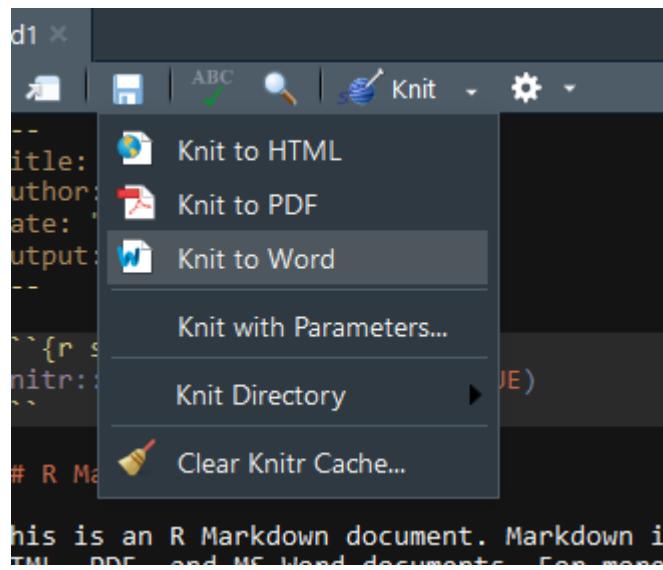
When you use citations, R will automatically put the reference at the very end of the document. Two LaTeX commands may be useful here. `\clearpage` makes a new page so your reference section isn't on the same page as the conclusion. `\singlenspace` makes the reference section single spaced if your document is set to be double spaced. Put these commands at the very end of your document so they only apply to the reference page. You don't need to do anything other than write them (for easier reading, make them on separate lines) at the end of the RMarkdown file. If you want to make the references go in another part of the paper (e.g. after tables and figures), just put this code at the place in the paper where you want to reference section to go: `<div id="refs"></div>`.

11.6 Spell check

RMarkdown does have a built-in spell checker (the ABC above a check mark symbol to the left of the Knit button) but it isn't that great. I recommend that you export to Word (or open up the PDF in Word if you prefer using PDFs) and using Word's superior spell checker.

11.7 Making the output file

To create the Word or PDF output click Knit and it will create the output in the format set in the very top. To change this format click the white down-arrow directly to the right of Knit and it will drop-down a menu with output options. Click the option you want and it will output it in that format and change that to the new default. Sometimes it takes a while for it to output, so be patient.



Part III

Collect

Chapter 12

Webscraping with `rvest`

If I ever stop working in the field of criminology, I would certainly be a baker. So for the next few chapters we are going to work with “data” on baking. What we’ll learn to do is find a recipe from the website [All Recipes](#) and webscrape the ingredients and directions of that recipe.

For our purposes we will be using the package `rvest`. This package makes it relatively easy to scrape data from websites, especially when that data is already in a table on the page as our data will be.

If you haven’t done so before, make sure to install `rvest`.

```
install.packages("rvest")
```

And every time you start R, if you want to use `rvest` you must tell R so by using `library()`.

```
library(rvest)
#> Loading required package: xml2
```

Here is a screenshot of the recipe for the “MMMMM... Brownies” (an excellent brownies recipe) [page](#).

The screenshot shows a recipe page for "Mmm... Brownies" on the Allrecipes website. The page includes the following details:

- Header:** Follow us on: social media icons; Get the Allrecipes magazine.
- Breadcrumbs:** Home > Recipes > Desserts > Cookies > Brownies > Chocolate Brownies
- Image:** A large image of a chocolate brownie with a bite taken out of it.
- Rating:** ★★★★☆ (4 stars) from 4k made it | 1k reviews | 380 photos.
- Author:** Recipe by: cicada77 | "Best brownies I've ever had!"
- Engagement:** 24 saves.
- Buttons:** Watch, Save, I Made It, Rate it, Print, Pin, Share.
- Time and Servings:** 1 h 16 servings.
- Ingredients:**
 - 1/2 cup white sugar
 - 1/2 teaspoon vanilla extract
 - 2 tablespoons butter
 - 2/3 cup all-purpose flour
 - 1/4 teaspoon baking soda
 - 1/2 teaspoon salt
 - 2 tablespoons water
- On Sale:** Whole Foods Market 2001 Pennsylvania Ave PHILADELPHIA, PA 19130 Sponsored.
- Related Recipes:** San Francisco Sourdough Bread (4.5 stars, 410 reviews) and Sourdough Starter (4.5 stars, 197 reviews).

(+) 2 tablespoons water Add all ingredients to list

(+) 1 1/2 cups semisweet chocolate chips

(+) 2 eggs

365 Everyday Value® Large
White Eggs
Everyday Savings
[LEARN MORE](#)
ADVERTISEMENT

Directions Add a note Print Watch

Prep 25 m	Cook 25 m	Ready In 1 h
--------------	--------------	-----------------

- 1 Preheat the oven to 325 degrees F (165 degrees C). Grease an 8x8 inch square pan.
- 2 In a medium saucepan, combine the sugar, butter and water. Cook over medium heat until boiling. Remove from heat and stir in chocolate chips until melted and smooth. Mix in the eggs and vanilla. Combine the flour, baking soda and salt; stir into the chocolate mixture. Spread evenly into the prepared pan.
- 3 Bake for 25 to 30 minutes in the preheated oven, until brownies set up. Do not overbake! Cool in pan and cut into squares.

Nutrition Facts

Per Serving: 141 calories; 6.8 g fat; 20.2 g carbohydrates; 2 g protein; 27 mg cholesterol; 113 mg sodium. [Full nutrition](#)

You might also like

MMMM... Brownies
These chocolate brownies from scratch are easy, moist, and delicious.

Get the magazine

Get a full year for \$5!
Cook 5-star weekday dinners every time.

Share

12.1 Scraping one page

In later lessons we'll learn how to scrape the ingredients of any recipe on the site. For now, we'll focus on just getting data for our brownies recipe.

The first step to scraping a page is to read in that page's information to R using the function `read_html()` from the `rvest` package. The input for the `()` is the URL of the page we want to scrape. In a later lesson, we will manipulate this URL to be able to scrape data from many pages.

```
read_html("https://www.allrecipes.com/recipe/25080/mmmmm-brownies/")
#> {html_document}
#> <html lang="EN">
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=U
#> [2] <body class="template-recipe node- mdex-test karma-site-container no-
```

When running the above code, it returns an XML Document. The `rvest` package is well suited for interpreting this and turning it into something we already know how to work with. To be able to work on this data, we need to save the output of `read_html()` into an object which we'll call *brownies* since that is the recipe we are currently scraping.

```
brownies <- read_html("https://www.allrecipes.com/recipe/25080/mmmmm-brownies")
```

We now need to select only a small part of page which has the relevant information - in this case the ingredients and directions.

We need to find just which parts of the page to scrape. To do so we'll use the helper tool [SelectorGadget](#), a Google Chrome extension that lets you click on parts of the page to get the CSS selector code that we'll use. Install that extension in Chrome and go to the [brownie recipe page](#).

When you open SelectorGadget it allows you click on parts of the page and it will highlight every similar piece and show the CSS selector code in the box near the bottom. Here we clicked on the first ingredient - “1/2 cup white sugar”. Every ingredient is highlighted in yellow as (to oversimplify this explanation) these ingredients are the same “type” in the page. It also highlighted the text “Add all ingredients to list” which we don’t want. As it is always the last line of text in ingredients, we’ll leave it in for now and practice subsetting data through R to remove it.

The screenshot shows a recipe card for brownies. At the top right, there are nutritional facts: 1 h (time), 16 servings, 141 cal, and a bar chart icon. Below the card, there's a sidebar for a Whole Foods Market listing. At the bottom, there's a 'Directions' section and a SelectorGadget bar with the text '.added'.

Note that in the bottom right of the screen, the SelectorGadget bar now has the text “.added”. This is the CSS selector code we can use to get all of the ingredients.

A close-up view of the SelectorGadget bar, which displays the CSS selector ".added".

We will use the function `html_nodes()` to grab the part of the page (based on the CSS selectors) that we want. The input for this function is first the object made from `read_html()` (which we called *brownies*) and then we can paste the CSS selector text - in this case, “.added”. We’ll save the resulting object as *ingredients* since we want to use *brownies* to also get the directions.

```
ingredients <- html_nodes(brownies, ".added")
```

Since we are getting data that is a text format, we need to tell `rvest` that the format of the scraped data is text. We do with using `html_text()` and our input in the () is the object made in the function `html_text()`.

```
ingredients <- html_text(ingredients)
```

Now let’s check what we got.

```
ingredients
#> character(0)
```

We have successfully scraped the ingredients for this brownies recipes - plus the “Add all ingredients to list” (copied twice for some reason).

Now let’s do the same process to get the directions for baking.

In SelectorGadget click clear to unselect the ingredients. Now click one of in lines of directions. It’ll highlight all three directions as they’re all of the same “type” (to be slightly more specific, when the site is made it has to put all of the pieces of the site together, such as links, photos, the section on ingredients, the section on directions, the section on reviews. So in this case we selected a “text” type in the section on directions and SelectorGadget then selected all “text” types inside of that section.).

The screenshot shows a recipe card for brownies. At the top, there are preparation and cooking times: Prep 25m, Cook 25m, Ready In 1h. Below this is the 'Directions' section, which contains three numbered steps:

- 1 Preheat the oven to 325 degrees F (165 degrees C). Grease an 8x8 inch square pan.
- 2 In a medium saucepan, combine the sugar, butter and water. Cook over medium heat until boiling. Remove from heat and stir in chocolate chips until melted and smooth. Mix in the eggs and vanilla. Combine the flour, baking soda and salt; stir into the chocolate mixture. Spread evenly into the prepared pan.
- 3 Bake for 25 to 30 minutes in the preheated oven, until brownies set up. Do not overbake! Cool in pan and cut into squares.

Below the directions is a 'Nutrition Facts' section with serving information. To the right of the directions, there's a sidebar with 'You might also like' and 'Get the magazine' sections. At the bottom left is an orange button labeled 'I made it!' and at the bottom right is the SelectorGadget interface showing the CSS selector used.

The CSS selector code this time is ”.recipe-directions__list-item” so we can put that inside of `html_nodes()`. Let’s save the output as `directions`.

```
directions <- html_nodes(brownies, ".recipe-directions__list-item")
directions <- html_text(directions)
```

Did it work?

```
directions
#> character(0)
```

Yes! The final value in our vector is blank so we will have to remove that.

12.2 Cleaning the webscraped data

We only have three things to do to clean the data. First, we need to remove the “Add all ingredients to list” from the *ingredients* object. Second, we will remove the blank value (“”) from the *directions* object. For both tasks we’ll do conditional subsetting to keep all values that do *not* equal those values. Finally, the directions print out with the text \n at the end. This indicates that it is the end of the line but we’ll want to remove that, which we can do using `gsub()`.

First let’s try out the condition of *ingredients* that do not equal the string “Add all ingredients to list”.

```
ingredients != "Add all ingredients to list"
#> logical(0)
```

It returns TRUE for all values except the last two, the ones which do equal “Add all ingredients to list”. Let’s only keep the elements without this string.

```
ingredients <- ingredients[ingredients != "Add all ingredients to list"]
```

And we can do the same thing for the empty string in *directions*.

```
directions <- directions[directions != ""]
```

To remove the \n we simple find that in `gsub()` and replace it with a blank string.

```
directions <- gsub("\n", "", directions)
```

And let’s print out both objects to make sure it worked.

```
ingredients
#> character(0)
directions
#> character(0)
```

Now *ingredients* is as it should be but *directions* has a bunch of space at the end of the string. Let’s use `gsub()` again to remove multiple spaces.

We’ll search for anything with two or more spaces and replace that with an empty string.

```
directions <- gsub("{2,}", "", directions)
```

And one final check to make sure it worked.

```
directions  
#> character(0)
```

In your own research, you will want to create a data.frame for nearly all data - this is also the way most statistical analysis packages expect data. In our case it doesn't make sense to do so. We'll keep them separate for now and in Chapter 13 we'll learn to make a function to scrape any recipe using just the URL and to print the ingredients and directions to the console.

Chapter 13

Functions

So far, we have been writing code to handle specific situations such as subsetting a single `data.frame`. In cases where you want to reuse the code it is unwise to simply copy and paste the code and make minor changes to handle the new data. Instead we want something that is able to take multiple values and perform the same action (subset, aggregate, make a plot, webscrape, etc) on those values. Code where you can input a value (such as a `data.frame`) and some (often optional) instructions on how to handle that data, and have the code run on the value is called a function. We've used other people's function before, such as `c()`, `mean()`, `grep()`, and `rvest()`.

Think of a function like a stapler - you put the paper in a push down and it staples the paper together. It doesn't matter what papers you are using; it always staples them together. If you needed to buy a new stapler every time you needed to staple something (i.e. copy and pasting code) you'd quickly have way too many staples (and waste a bunch of money).

An important benefit is that you can use this function again and again to help solve other problems. If, for example, you have code that cleans data from Philadelphia's crime data set, if you wanted to use it for Chicago's crime data, making a single function is much easier (to read and to fix if there is an issue) than copying the code. If you wanted to use it for 20 cities, copy and pasting code quickly becomes a terrible solution - functions work much better. If you did copy and paste 20 times and you found a bug, then you'd have to fix the bug 20 times. With a function you would change the code

once.

13.1 A simple function

We'll start with a simple function that takes a number and returns that number plus the value 2.

```
add_2 <- function(number) {  
  number <- number + 2  
  return(number)  
}
```

The syntax (how we write it) of a function is

```
function_name <- function(parameters) {  
  code  
  return(output)  
}
```

There are five essential parts of a function

- **function_name** - This is just the name we give to the function. It can be anything but, like when making other objects, call it something where it is easy to remember what it does.
- **parameters** - Here is where we say what goes into the function. In most cases you will want to put some data in and expect something new out. For example, for the function `mean()` you put in a vector of numbers in the () section and it returns the mean of those numbers. Here is also where you can put any options to affect how the code is run.
- **code** - This is the code you write to do the thing you want the function to do. In the above example our code is `number <- number + 2`. For any number inputted, our code adds 2 to it and saves it back into the object `number`.
- **return** - This is something new in this course, here you use `return()` and inside the () you put the object you want to be outputted. In our example we have "number" inside the `return()` as that's what we want to come out of the function. It is not always necessary to end your function with `return()` but is highly recommended to make sure you're outputting what it is you want to output. If you save the output

of a function (such as by `x <- mean(1:3)`) it will save the output to the variable assigned. Otherwise it will print out the results in the console.

- The final piece is the structure of your function. After the `function_name` (whatever it is you call it) you always need the text `<- function()` where the parameters (if any) are in the `()`. After the closing parentheses put a `{` and at the very end of the function, after the `return()`, close those squiggly brackets with a `"}`. The `<- function()` tells R that you are making a function rather than some other type of object. And the `{` and `}` tell R that all the code in between are part of that function.

Our function here adds 2 to any number we input.

```
add_2(2)
#> [1] 4
```

```
add_2(5)
#> [1] 7
```

13.2 Adding parameters

Let's add a single parameter which multiplies the result by 5 if selected.

```
add_2 <- function(number, times_5 = FALSE) {
  number <- number + 2
  return(number)
}
```

Now we have added a parameter called `time_5` to the `()` part of the function and set it to be FALSE by default. Right now it doesn't do anything so we need to add code to say what happens if it is TRUE (remember in R true and false must always be all capital letters).

```
add_2 <- function(number, times_5 = FALSE) {
  number <- number + 2

  if (times_5 == TRUE) {
    number <- number * 5
```

```

    }

    return(number)
}

```

Now our code says if the parameter `times_5` is TRUE, then do the thing in the squiggly brackets {} below. Note that we use the same squiggly brackets as when making the entire function. That just tells R that the code in those brackets belong together. Let's try out our function.

```

add_2(2)
#> [1] 4

```

It returns 4, as expected. Since the parameter `times_5` is defaulted to FALSE, we don't need to specify that parameter if we want it to stay FALSE. When we don't tell the function that we want it to be TRUE, the code in our "if statement" doesn't run. When we set `times_5` to TRUE, it runs that code.

```

add_2(2, times_5 = TRUE)
#> [1] 20

```

13.3 Making a function to scrape recipes

In Section 12.1 we wrote some code to scrape data from the website All Recipes for a recipe. We are going to turn that code into a function here. The benefit is that our input to the function will be an URL and then it will print out the ingredients and directions for that recipe. If we want multiple recipes (and for webscraping you usually will want to scrape multiple pages), we just change the URL we input without changing the code at all.

We used the `rvest` package so we need to tell R want to use it again.

```

library(rvest)
#> Loading required package: xml2

```

Let's start by writing a shell of the function - everything but the code. We can call it `scrape_recipes` (though any name would work), add in the `<- function()` and put "URL" (without quotes) in the () as our input for the

function is a date. In this case we won't return anything, we will just print things to the console, so we don't need the `return()` value. And don't forget the `{` after the end of the `function()` and `}` at the very end of the function.

```
scrape_recipes <- function(URL) {  
}  
}
```

Now we need to add the code that takes the date, scrapes the website, and saves that data into objects called *ingredients* and *directions*. Since we have the code from an earlier lesson, we can copy and paste that code into the function and make a small change to get a working function.

```
scrape_recipes <- function(URL) {  
  
  brownies <- read_html("https://www.allrecipes.com/recipe/25080/mmmmm-brownies/")  
  
  ingredients <- html_nodes(brownies, ".added")  
  ingredients <- html_text(ingredients)  
  
  directions <- html_nodes(brownies, ".recipe-directions__list--item")  
  directions <- html_text(directions)  
  
  ingredients <- ingredients[ingredients != "Add all ingredients to list"]  
  directions <- directions[directions != ""]  
}
```

The part inside the `()` of `read_html()` is the URL of the page we want to scrape. This is the part of the function that will change based on our input. We want whatever input is in the URL parameter to be the URL we scrape. So let's change the URL of the brownies recipe we scraped previously to simply say "URL" (without quotes).

```
scrape_recipes <- function(URL) {  
  
  brownies <- read_html(URL)  
  
  ingredients <- html_nodes(brownies, ".added")  
  ingredients <- html_text(ingredients)
```

```

directions <- html_nodes(brownies, ".recipe-directions__list--item")
directions <- html_text(directions)

ingredients <- ingredients[ingredients != "Add all ingredients to list"]
directions <- directions[directions != ""]
}

```

To make this function print something to the console we need to specifically tell it to do so in the code. We do this using the `print()` function. Let's print first the ingredients and then the directions. We'll add that add the final lines of the function.

```

scrape_recipes <- function(URL) {

  brownies <- read_html(URL)

  ingredients <- html_nodes(brownies, ".added")
  ingredients <- html_text(ingredients)

  directions <- html_nodes(brownies, ".recipe-directions__list--item")
  directions <- html_text(directions)

  ingredients <- ingredients[ingredients != "Add all ingredients to list"]
  directions <- directions[directions != ""]
  directions <- gsub("\n", "", directions)
  directions <- gsub("{2,}", "", directions)

  print(ingredients)
  print(directions)
}

```

Now we can try it for a new recipe, this one for “The Best Lemon Bars” at URL <https://www.allrecipes.com/recipe/10294/the-best-lemon-bars/>.

```

scrape_recipes("https://www.allrecipes.com/recipe/10294/the-best-lemon-bars/")
#> character(0)
#> character(0)

```

In the next lesson we'll use “for loops” to scrape multiple recipes very quickly.

Chapter 14

For loops

We will often want to perform the same task on a number of different items, such as cleaning every column in a data set. One effective way to do this is through “for loops”. Earlier in this course we learned how to scrape the recipe website [All Recipes](#). We did so for a single recipe, if we wanted to get a feasts worth of recipes, typing out each recipe would be excessively slow, even with the function we made in Section 13.3. In this lesson we will use a for loop to scrape multiple recipes very quickly.

14.1 Basic for loops

We’ll start with a simple example, making R print the numbers 1-10.

```
for (i in 1:10) {  
  print(i)  
}  
#> [1] 1  
#> [1] 2  
#> [1] 3  
#> [1] 4  
#> [1] 5  
#> [1] 6  
#> [1] 7  
#> [1] 8
```

```
#> [1] 9
#> [1] 10
```

The basic concept of a for loop is you have some code that you need to run many times with slight changes to a value or values in the code - somewhat like a function. Like a function, all the code you want to use goes in between the { and } squiggly brackets. And you loop through all the values you specify - meaning the code runs once for each of those values.

Let's look closer at the (i in 1:10). The i is simply a placeholder object which takes the value 1:10 each iteration of the loop. It's not necessary to call it i but that is convention in programming to do so. It takes the value of whatever follows the in which can range from a vector of strings to numbers to lists of data.frames. Especially when you're an early learner of R it could help to call the i something informative to you about what value it has.

Let's go through a few examples with different names for i and different values it is looping through.

```
for (a_number in 1:10) {
  print(a_number)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] 6
#> [1] 7
#> [1] 8
#> [1] 9
#> [1] 10

animals <- c("cat", "dog", "gorilla", "buffalo", "lion", "snake")
for (animal in animals) {
  print(animal)
}
#> [1] "cat"
#> [1] "dog"
```

```
#> [1] "gorilla"
#> [1] "buffalo"
#> [1] "lion"
#> [1] "snake"
```

Now let's make our code a bit more complicated, adding the number 2 every loop.

```
for (a_number in 1:10) {
  print(a_number + 2)
}
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] 6
#> [1] 7
#> [1] 8
#> [1] 9
#> [1] 10
#> [1] 11
#> [1] 12
```

We're keeping the results inside of `print()` since for loops do not print the results by default. Let's try combining this with some subsetting using square bracket notation `[]`. We will look through every value in `numbers`, a vector we will make with the values 1:10 and replace each value with its value plus 2.

The object we're looping through is `numbers`. But we're actually looping through every index it has, hence the `1:length(numbers)`. That is saying, `i` takes the value of each index in `numbers` which is useful when we want to change that element. `length(numbers)` finds how long the vector `numbers` is (if this was a `data.frame` we could use `nrow()`) to find how many elements it has. In the code we take the value at each index `numbers[i]` and add 2 to it.

```
numbers <- 1:10
for (i in 1:length(numbers)) {
  numbers[i] <- numbers[i] + 2
```

```
}
numbers
#> [1] 3 4 5 6 7 8 9 10 11 12
```

We can also include functions we made in for loops. Here's a function we made last lesson which adds 2 to each inputted number.

```
add_2 <- function(number) {
  number <- number + 2
  return(number)
}
```

Let's put that in the loop.

```
for (i in 1:length(numbers)) {
  numbers[i] <- add_2(numbers[i])
}

numbers
#> [1] 5 6 7 8 9 10 11 12 13 14
```

14.2 Scraping multiple recipes

Below is the function copied from Section 13.3 which takes a single URL and scraped the site [All Recipes](#) for that recipe. It printed the ingredients and directions to cook that recipe to the console. If we wanted to get data for multiple recipes, we would need to run the function multiple times. Here we will use a for loop to do this. Since we're using the `read_html()` function from `rvest`, we need to tell R we want to use that package.

```
library(rvest)
#> Loading required package: xml2
scrape_recipes <- function(URL) {

  brownies <- read_html(URL)

  ingredients <- html_nodes(brownies, ".added")
  ingredients <- html_text(ingredients)
```

```

directions <- html_nodes(brownies, ".recipe-directions__list--item")
directions <- html_text(directions)

ingredients <- ingredients[ingredients != "Add all ingredients to list"]
directions <- directions[directions != ""]
directions <- gsub("\n", "", directions)
directions <- gsub("{2,}", "", directions)

print(ingredients)
print(directions)
}

```

With any for loop you need to figure out what is going to be changing, in this case it is the URL. And since we want multiple, we need to make an object with the URLs of all the recipes we want.

Here I am making a vector called *recipe_urls* with the URLs of several recipes that I like on the site. The way I got the URLs was to go to each recipe's page and copy and paste the URL. Is this the right approach? Shouldn't we do everything in R? Not always. In situations like this where we know that there are a small number of links we want - and there is no easy way to get them through R - it is reasonable to do it by hand. Remember that R is a tool to help you. While keeping everything you do in R is good for reproducibility, it is not always reasonable and may take too much time or effort given the constraints - usually limited time - of your project.

```

recipe_urls <- c("https://www.allrecipes.com/recipe/25080/mmmmm-brownies/",
                 "https://www.allrecipes.com/recipe/27188/crepes/",
                 "https://www.allrecipes.com/recipe/84270/slow-cooker-corned-beef-and",
                 "https://www.allrecipes.com/recipe/25130/soft-sugar-cookies-v/",
                 "https://www.allrecipes.com/recipe/53304/cream-corn-like-no-other/",
                 "https://www.allrecipes.com/recipe/10294/the-best-lemon-bars/",
                 "https://www.allrecipes.com/recipe/189058/super-simple-salmon/")

```

Now we can write the for loop to go through every single URL in *recipe_urls* and use the function *scrape_recipes* on that URL.

```

for (recipe_url in recipe_urls) {
  scrape_recipes(recipe_url)
}

```

```
}

#> character(0)
#> character(0)
#> character(0)
#> character(0)
#> [1] "4 large carrots, peeled and cut into matchstick pieces"
#> [2] "10 baby red potatoes, quartered"
#> [3] "1 onion, peeled and cut into bite-sized pieces"
#> [4] "4 cups water"
#> [5] "1 (4 pound) corned beef brisket with spice packet"
#> [6] "6 ounces beer"
#> [7] "1/2 head cabbage, coarsely chopped"
#> [1] "Place the carrots, potatoes, and onion into the bottom of a slow cooker"
#> [2] "Cook the brisket for about 8 hours. An hour before serving, stir in
#> character(0)
#> character(0)
#> [1] "2 (10 ounce) packages frozen corn kernels, thawed"
#> [2] "1 cup heavy cream"
#> [3] "1 teaspoon salt"
#> [4] "2 tablespoons granulated sugar"
#> [5] "1/4 teaspoon freshly ground black pepper"
#> [6] "2 tablespoons butter"
#> [7] "1 cup whole milk"
#> [8] "2 tablespoons all-purpose flour"
#> [9] "1/4 cup freshly grated Parmesan cheese"
#> [1] "In a skillet over medium heat, combine the corn, cream, salt, sugar,
#> character(0)
#> character(0)
#> character(0)
#> character(0)
```

Chapter 15

Scraping tables from PDFs

In the majority of cases when you want data from a PDF it will be in a table. Essentially the data will be an Excel file inside of a PDF. This format is not altogether different than what we've done before. We will be using regular expressions and the function `strsplit()` to get this data into a usable format.

Let's first take a look at the data we will be scraping. The first step in any PDF scraping should be to look at the PDF and try to think about the best way to approach this particular problem - while all PDF scraping follows a general format you cannot necessarily copy and paste your code, each situation is likely slightly different. Our data is from the U.S. Customs and Border Protection (CBP) and contains a wealth of information about apprehensions and contraband seizures in border sectors.

We will be using the Sector Profile 2017 PDF which has information in four tables, three of which we'll scrape and then combine together. The data was downloaded from the U.S. Customs and Border Protection "Stats and Summaries" page [here](#). If you're interested in using more of their data, some of it has been cleaned and made available [here](#).

The file we want to use is called "usbp_stats_fy2017_sector_profile.pdf" and has four tables in the PDF. Let's take a look at them one at a time, understanding what variables are available, and what units each row is in. Then we'll start scraping the tables.

The first table is “Sector Profile - Fiscal Year 2017 (Oct. 1st through Sept. 30th)”. Before we even look down more at the table, the title is important. It is for fiscal year 2017, not calendar year 2017 which is more common in the data we usually use. This is important if we ever want to merge this data with other data sets. If possible, we would have to get data that is monthly so we can just use October 2016 through September 2017 to match up properly.



United States Border Patrol										
Sector Profile - Fiscal Year 2017 (Oct. 1st through Sept. 30th)										
SECTOR	Agent Staffing*	Apprehensions	Other Than Mexican Apprehensions	Marijuana (pounds)	Cocaine (pounds)	Accepted Prosecutions	Assaults	Rescues	Deaths	
Miami	111	2,280	1,646	2,253	231	292	1	N/A	N/A	
New Orleans	63	920	528	21	6	10	0	N/A	N/A	
Ramey	38	388	387	3	2,932	89	0	N/A	N/A	
Coastal Border Sectors Total	212	3,588	2,561	2,277	3,169	391	1	N/A ***	N/A ***	
Blaine	296	288	237	0	0	9	0	N/A	N/A	
Buffalo	277	447	293	228	2	37	2	N/A	N/A	
Detroit	408	1,070	322	124	0	85	1	N/A	N/A	
Grand Forks	189	496	202	0	0	40	2	N/A	N/A	
Havre	183	39	28	98	0	2	0	N/A	N/A	
Houlton	173	30	30	17	0	2	0	N/A	N/A	
Spokane	230	208	67	68	0	24	0	N/A	N/A	
Swanton	292	449	359	531	1	103	6	N/A	N/A	
Northern Border Sectors Total	2,048	3,027	1,538	1,066	3	302	11	N/A ***	N/A ***	
Big Bend (formerly Marfa)	500	6,002	3,346	40,852	45	2,847	11	26	1	
Del Rio	1,391	13,476	6,156	9,482	62	8,022	12	99	18	
El Centro	870	18,633	5,812	5,554	484	1,413	34	4	2	
El Paso	2,182	25,193	15,337	34,189	140	6,996	54	44	8	
Laredo	1,666	25,460	7,891	69,535	757	6,119	31	1,054	83	
Rio Grande Valley (formerly McAllen)	3,130	137,562	107,909	260,020	1,192	7,979	422	1,190	104	
San Diego	2,199	26,086	7,060	10,985	2,903	3,099	84	48	4	
Tucson	3,691	38,657	12,328	397,090	331	20,963	93	750	72	
Yuma	859	12,847	10,139	30,181	261	2,367	33	6	2	
Southwest Border Sectors Total**	16,605	303,916	175,978	857,888	6,174	59,805	774	3,221	294	
Nationwide Total***	19,437	310,531	180,077	861,231	9,346	60,498	786	3,221	294	

* Agent staffing statistics depict FY17 on-board personnel data as of 9/30/2017
** Southwest Border Sectors staffing statistics include: Big Bend, Del Rio, El Centro, El Paso, Laredo, Rio Grande Valley, San Diego, Tucson, Yuma, and the Special Operations Group.
*** Nationwide staffing statistics include: All on-board Border Patrol agents in CBP
**** Rescue and Death statistics are not tracked for Northern and Coastal Border Sectors.

Now if we look more at the table, we can see that each row is a section of the U.S. border. There are three main sections - Coastal, Northern, and Southwest, with subsections of each also included. The bottom row is the sum of all these sections and gives us nationwide data. Many government data will be like this form with sections and subsections in the same table. Watch out when doing mathematical operations! Just summing any of these columns will give you triple the true value due to the presence of nationwide, sectional, and subsectional data.

There are 9 columns in the data other than the border section identifier. It looks like we have total apprehensions, apprehensions for people who are not Mexican citizens, marijuana and cocaine seizures (in pounds), the number of accepted prosecutions (presumably of those apprehended), and the number of CBP agents assaulted. The last two columns have the number of people rescued by CBP and the number of people who died (it is unclear from this data alone if this is solely people in custody or deaths during crossing the border). These two columns are also special as they only have data for the Southwest border.

Table 2 has a similar format with each row being a section or subsection. The columns now have the number of juveniles apprehended, subdivided by if they were accompanied by an adult or not, and the number of adults apprehended. The last column is total apprehensions which is also in Table 1.

United States Border Patrol					
Juvenile (0-17 Years Old) and Adult Apprehensions - Fiscal Year 2017 (Oct. 1st through Sept. 30th)					
SECTOR	Accompanied Juveniles	Unaccompanied Juveniles	Total Juveniles	Total Adults	Total Apprehensions
Miami	19	42	61	2,219	2,280
New Orleans	1	22	23	897	920
Ramey	7	1	8	380	388
Coastal Border Sectors Total	27	65	92	3,496	3,588
Blaine	29	7	36	252	288
Buffalo	3	3	6	441	447
Detroit	5	11	16	1,054	1,070
Grand Forks	5	4	9	487	496
Havre	1	3	4	35	39
Houlton	1	8	9	21	30
Spokane	3	0	3	205	208
Swanton	18	10	28	421	449
Northern Border Sectors Total	65	46	111	2,916	3,027
Big Bend (<i>formerly Marfa</i>)	506	811	1,317	4,685	6,002
Del Rio	1,348	1,349	2,697	10,779	13,476
El Centro	968	1,531	2,499	16,134	18,633
El Paso	4,642	3,926	8,568	16,625	25,193
Laredo	477	2,033	2,510	22,950	25,460
Rio Grande Valley (<i>formerly McAllen</i>)	27,222	23,708	50,930	86,632	137,562
San Diego	1,639	1,551	3,190	22,896	26,086
Tucson	1,088	3,659	4,747	33,910	38,657
Yuma	3,241	2,867	6,108	6,739	12,847
Southwest Border Sectors Total	41,131	41,435	82,566	221,350	303,916
Nationwide Total	41,223	41,546	82,769	227,762	310,531

Table 3 follows the same format and the new columns are number of appre-

hensions by gender.



United States Border Patrol

Apprehensions by Gender - Fiscal Year 2017 (Oct. 1st through Sept. 30th)

SECTOR	Female	Male	Total Apprehensions
Miami	219	2,061	2,280
New Orleans	92	828	920
Ramey	65	323	388
Coastal Border Sectors Total	376	3,212	3,588
Blaine	97	191	288
Buffalo	69	378	447
Detroit	78	992	1,070
Grand Forks	56	440	496
Havre	13	26	39
Houlton	17	13	30
Spokane	17	191	208
Swanton	106	343	449
Northern Border Sectors Total	453	2,574	3,027
Big Bend (formerly Marfa)	985	5,017	6,002
Del Rio	2,622	10,854	13,476
El Centro	2,791	15,842	18,633
El Paso	7,364	17,829	25,193
Laredo	3,651	21,809	25,460
Rio Grande Valley (formerly McAllen)	50,306	87,256	137,562
San Diego	4,117	21,969	26,086
Tucson	4,693	33,964	38,657
Yuma	4,328	8,519	12,847
Southwest Border Sectors Total	80,857	223,059	303,916
Nationwide Total	81,686	228,845	310,531

Finally, Table 4 is a bit different in its format. The rows are now variables and the columns are the locations. In this table it doesn't include subsections, only border sections and nationwide total. The data it has available are partially a repeat of Table 1 but with more drug types and the addition of the number of drug seizures and some firearm seizure information. As this table is formatted differently than the others, we won't scrape it in this lesson - but you can use the skills you'll learn to do so yourself.



United States Border Patrol
Apprehensions / Seizure Statistics - Fiscal Year 2017 (Oct. 1st through Sept. 30th)

Apprehension/Seizure Type	Coastal Border Sectors	Northern Border Sectors	Southwest Border Sectors	Nationwide Total
Apprehensions	3,588	3,027	303,916	310,531
Other Than Mexican Apprehensions	2,561	1,538	175,978	180,077
Marijuana (pounds)	2,277	1,066	857,888	861,231
Cocaine (pounds)	3,169	3	6,174	9,346
Heroin (ounces)	0	62	15,182	15,244
Methamphetamine (pounds)	23	32	10,273	10,328
Ecstasy (pounds)	0	0	1	1
Other Drugs* (pounds)	0	14	554	568
Marijuana Seizures	113	255	9,371	9,739
Cocaine Seizures	33	46	463	542
Heroin Seizures	0	29	219	248
Methamphetamine Seizures	2	68	809	879
Ecstasy Seizures	1	2	48	51
Other Drugs* Seizures	6	99	735	840
Conveyances	86	79	7,388	7,553
Firearms	9	45	369	423
Ammunition (rounds)	217	384	13,938	14,539
Currency (value)	\$325,129	\$374,282	\$5,169,593	\$5,869,004

***Other Drugs include:** All USBP drug seizures excluding marijuana, cocaine, heroin, methamphetamine, and ecstasy (MDMA).

Coastal Border Sectors include: Miami, New Orleans, and Ramey, Puerto Rico.

Northern Border Sectors include: Blaine, Buffalo, Detroit, Grand Forks, Havre, Houlton, Spokane and Swanton.

Southwest Border Sectors include: Big Bend, Del Rio, El Centro, El Paso, Laredo, Rio Grande Valley, San Diego, Tucson, and Yuma.

Drug quantities are rounded to the nearest whole number

15.1 Scraping the first table

We've now seen all three of the tables that we want to scrape so we can begin the process of actually scraping them. Note that each table is very similar meaning we can reuse some code to scrape as well as clean the data. That means that we will want to write some functions to make our work easier and avoid copy and pasting code three times. We will use the `pdf_text()` function from the `pdftools` package to read the PDFs into R.

```
library(pdftools)
#> Using poppler version 0.73.0
```

We can save the output of the `pdf_text()` function as the object `border_patrol` and we'll use it for each table.

```
border_patrol <- pdf_text("data/usbp_stats_fy2017_sector_profile.pdf")
```

We can take a look at the `head()` of the result.

```
head(border_patrol)
#> [1] "
#> [2] "
#> [3] "
#> [4] "
United States Border Patrol\r\n
United States Border Patrol\r\n
United States Border Patrol\r\n
United States Border Patrol\r\n
```

If you look closely in this huge amount of text output, you can see that it is a vector with each table being an element in the vector. We can see this further by checking the `length()` of “border_patrol” and just looking at the first element.

```
length(border_patrol)
#> [1] 4
```

It is four elements long, one for each table.

```
border_patrol[1]
#> [1] "
United States Border Patrol\r\n
United States Border Patrol\r\n
United States Border Patrol\r\n
United States Border Patrol\r\n
```

And this gives us all the values in the first table plus a few sentences at the end detailing some features of the table. At the end of each line (where in the PDF it should end but doesn’t in our data yet) there is a `\r\n` indicating that there should be a new line. We want to use `strsplit()` to split at the `\r\n`.

The `strsplit()` function breaks up a string into pieces based on a value inside of the string. Let’s use the word “criminology” as an example. If we want to split it by the letter “n” we’d have two results, “crimi” and “ology” as these are the pieces of the word after breaking up “criminology” at letter “n”.

```
strsplit("criminology", split = "n")
#> [[1]]
#> [1] "crimi" "ology"
```

Note that it deletes whatever value is used to break up the string.

Let's save a new object with the value in the first element of "border_patrol", calling it *sector_profile* as that's the name of that table, and then using *strsplit()* on it. *strsplit()* returns a list so we will also want to keep just the first element of that list using double square bracket [()] notation.

```
sector_profile <- border_patrol[1]
sector_profile <- strsplit(sector_profile, "\r\n")
sector_profile <- sector_profile[[1]]
```

Now we can look at the first six rows of this data.

<code>head(sector_profile)</code>	<i>United States Border Patrol Sector Profile</i>
#> [1] "	
#> [2] "	
#> [3] "	
#> [4] "SECTOR	<i>Agent Staffing*</i>
#> [5] "	
#> [6] "	<i>Apprehensions**</i>

Notice that there is a lot of empty white space at the beginning of the rows. We want to get rid of that to make our next steps easier. We can use *trimws()* and put the entire "sector_profile" data in the () and it'll remove the white space at the ends of each line for us.

```
sector_profile <- trimws(sector_profile)
```

We have more rows than we want so let's look at the entire data and try to figure out how to keep just the necessary rows.

<code>sector_profile</code>			
#> [1] "United States Border Patrol"			
#> [2] "Sector Profile - Fiscal Year 2017 (Oct. 1st through Sept. 30th)"			
#> [3] "Agent"	<i>Other Than Mexican</i>	<i>Marijuana</i>	
#> [4] "SECTOR"	<i>Staffing*</i>		
#> [5] "Apprehensions"			
#> [6] "Apprehensions"	(pounds)	(pounds)	<i>Prosecutions**</i>
#> [7] "Assaults Rescues"	<i>Deaths</i> "		
#> [8] "Miami"		111	2,280
#> [9] "New Orleans"		63	920
#> [10] "Ramey"		38	388

#> [11]	"Coastal Border Sectors Total	212
#> [12]	"Blaine	296
#> [13]	"Buffalo	277
#> [14]	"Detroit	408
#> [15]	"Grand Forks	189
#> [16]	"Havre	183
#> [17]	"Houlton	173
#> [18]	"Spokane	230
#> [19]	"Swanton	292
#> [20]	"Northern Border Sectors Total	2,048
#> [21]	"Big Bend (formerly Marfa)	500
#> [22]	"Del Rio	1,391
#> [23]	"El Centro	870
#> [24]	"El Paso	2,182
#> [25]	"Laredo	1,666
#> [26]	"Rio Grande Valley (formerly McAllen)	3,130
#> [27]	"San Diego	2,199
#> [28]	"Tucson	3,691
#> [29]	"Yuma	859
#> [30]	"Southwest Border Sectors Total**	16,605
#> [31]	"Nationwide Total***	19,437
#> [32]	"* Agent staffing statistics depict FY17 on-board personnel data as	3
#> [33]	"** Southwest Border Sectors staffing statistics include: Big Bend,	
#> [34]	"*** Nationwide staffing statistics include: All on-board Border Pat	
#> [35]	"**** Rescue and Death statistics are not tracked for Northern and C	

Based on the PDF, we want every row from Miami to Nationwide Total. But here we have several rows with the title of the table and the column names, and at the end we have the sentences with some details that we don't need.

To keep only the rows that we want, we can combine `grep()` and subsetting to find the rows from Miami to Nationwide Total and keep only those rows. We will use `grep()` to find which row has the text "Miami" and which has the text "Nationwide Total" and keep all rows between them (including those matched rows as well). Since each only appears once in the table we don't need to worry about handling duplicate results.

```
grep("Miami", sector_profile)
#> [1] 8

grep("Nationwide Total", sector_profile)
#> [1] 31
```

We'll use square bracket notation to keep all rows between those two values (including each value). Since the data is a vector, not a data.frame, we don't need a comma.

```
sector_profile <- sector_profile[grep("Miami", sector_profile):grep("Nationwide Total", sector_profile)]
```

Note that we're getting rid of the rows which had the column names. It's easier to make the names ourselves than to deal with that mess.

head(sector_profile)			
#> [1] "Miami"	111		2,280
#> [2] "New Orleans"	63		920
#> [3] "Ramey"	38		388
#> [4] "Coastal Border Sectors Total"	212		3,588
#> [5] "Blaine"	296		288
#> [6] "Buffalo"	277		447

The data now has only the rows we want but still doesn't have any columns, it's currently just a vector of strings. We want to make it into a data.frame to be able to work on it like we usually do. When looking at this data it is clear that where the division between columns is a bunch of white space. Take the first row for example, it says "Miami" then after lots of white spaces "111" then again with "2,280" and so on for the rest of the row. We'll use this pattern of columns differentiated by white space to make *sector_profile* into a data.frame.

We will use the function `str_split_fixed()` from the `stringr` package. This function is very similar to `strsplit()` except you can tell it how many columns to expect.

```
install.packages("stringr")
library(stringr)
```

The syntax of `str_split_fixed()` is similar to `strsplit()` except the new

parameter of the number of splits to expect. Looking at the PDF shows us that there are 10 columns so that's the number we'll use. Our split will be ” {2,}“. That is, a space that occurs two or more times. Since there are sectors with spaces in their name, we can't have only one space, we need at least two. If you look carefully at the rows with sectors”Coast Border Sectors Total” and “Northern Border Sectors Total”, the final two columns actually do not have two spaces between them because of the amount of * they have. Normally we'd want to fix this using `gsub()`, but those values will turn to NA anyway so we won't bother in this case.

```
sector_profile <- str_split_fixed(sector_profile, " {2,}", 10)
```

If we check the `head()` we can see that we have the proper columns now but this still isn't a data.frame and has no column names.

```
head(sector_profile)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,"Miami" "111" "2,280" "1,646" "2,253" "231"
#> [2,"New Orleans" "63" "920" "528" "21" "6"
#> [3,"Ramey" "38" "388" "387" "3" "2,932"
#> [4,"Coastal Border Sectors Total" "212" "3,588" "2,561" "2,277" "3,169"
#> [5,"Blaine" "296" "288" "237" "0" "0"
#> [6,"Buffalo" "277" "447" "293" "228" "2"
#>      [,8] [,9] [,10]
#> [1,"1" "N/A" "N/A"
#> [2,"0" "N/A" "N/A"
#> [3,"0" "N/A" "N/A"
#> [4,"1" "N/A **** N/A ****" ""
#> [5,"0" "N/A" "N/A"
#> [6,"2" "N/A" "N/A"
```

We can make it a data.frame just by putting it in `data.frame()`. To avoid making the columns into factors, we'll set the parameter `stringsAsFactors` to FALSE. And we can assign the columns names using a vector of strings we can make. We'll use the same column names as in the PDF but in lowercase and replacing spaces and parentheses with underscores.

```
sector_profile <- data.frame(sector_profile, stringsAsFactors = FALSE)
names(sector_profile) <- c("sector",
```

```
"agent_staffing",
"apprehensions",
"other_than_mexican_apprehensions",
"marijuana_pounds",
"cocaine_pounds",
"accepted_prosecutions",
"assaults",
"rescues",
"deaths")
```

We have now taken a table from a PDF and successfully scraped it to a data.frame in R. Now we can work on it as we would any other data set that we've used previously.

```
head(sector_profile)
#>                                     sector agent_staffing apprehensions
#> 1                               Miami           111        2,280
#> 2                           New Orleans          63         920
#> 3                             Ramey            38        388
#> 4 Coastal Border Sectors Total        212        3,588
#> 5                            Blaine           296        288
#> 6                           Buffalo           277        447
#>   other_than_mexican_apprehensions marijuana_pounds cocaine_pounds
#> 1                         1,646            2,253          231
#> 2                           528              21             6
#> 3                           387              3        2,932
#> 4                         2,561            2,277        3,169
#> 5                           237              0             0
#> 6                           293            228             2
#>   accepted_prosecutions assaults      rescues deaths
#> 1                         292             1            N/A   N/A
#> 2                           10            0            N/A   N/A
#> 3                           89            0            N/A   N/A
#> 4                         391            1  N/A **** N/A ****
#> 5                           9            0            N/A   N/A
#> 6                          37            2            N/A   N/A
```

To really be able to use this data we'll want to clean the columns to turn the

values to numeric type but we can leave that until later. For now let's write a function that replicates much of this work for the next tables.

15.2 Making a function

As we've done before, we want to take the code we wrote for the specific case of the first table in this PDF and turn it into a function for the general case of other tables in the PDF. Let's copy the code we used above then convert it to a function.

```
sector_profile <- border_patrol[1]
sector_profile <- trimws(sector_profile)
sector_profile <- strsplit(sector_profile, "\r\n")
sector_profile <- sector_profile[[1]]
sector_profile <- sector_profile[grep("Miami", sector_profile) : grep("Nationwide", sector_profile)]
sector_profile <- str_split_fixed(sector_profile, " {2,}", 10)
sector_profile <- data.frame(sector_profile, stringsAsFactors = FALSE)
names(sector_profile) <- c("sector",
                           "agent_staffing",
                           "total_apprehensions",
                           "other_than_mexican_apprehensions",
                           "marijuana_pounds",
                           "cocaine_pounds",
                           "accepted_prosecutions",
                           "assaults",
                           "rescues",
                           "deaths")
```

Since each table is so similar our function will only need a few changes in the above code to work for all three tables. The object `border_patrol` has all four of the tables in the data, so we need to say which of these tables we want - we can call the parameter `table_number`. Then each table has a different number of columns so we need to change the `str_split_fixed()` function to take a variable with the number of columns we input, a value we'll call `number_columns`. We rename each column to their proper name so we need to input a vector - which we'll call `column_names` - with the names for each column. Finally, we want to have a parameter where we enter in the data which holds all of the tables, our object `border_patrol`, we can call

this `list_of_tables` as it is fairly descriptive. We do this as it is bad form to have a function that relies on an object that isn't explicitly put in the function. If we change our `border_patrol` object and the function doesn't have that as an input, it will work differently than we expect. Since we called the object we scraped `sector_profile` for the first table, let's change that to `data` as not all tables are called Sector Profile.

```
scrape_pdf <- function(list_of_tables, table_number, number_columns, column_names) {  
  data <- list_of_tables[table_number]  
  data <- trimws(data)  
  data <- strsplit(data, "\r\n")  
  data <- data[[1]]  
  data <- data[grep("Miami", data):grep("Nationwide Total", data)]  
  data <- str_split_fixed(data, " {2,}", number_columns)  
  data <- data.frame(data, stringsAsFactors = FALSE)  
  names(data) <- column_names  
  
  return(data)  
}
```

Now let's run this function for each of the three tables we want to scrape, changing the function's parameters to work for each table. To see what parameter values you need to input, look at the PDF itself or the screenshots in this lesson.

```
table_1 <- scrape_pdf(list_of_tables = border_patrol,  
                      table_number = 1,  
                      number_columns = 10,  
                      column_names = c("sector",  
                                      "agent_staffing",  
                                      "total_apprehensions",  
                                      "other_than_mexican_apprehensions",  
                                      "marijuana_pounds",  
                                      "cocaine_pounds",  
                                      "accepted_prosecutions",  
                                      "assaults",  
                                      "rescues",  
                                      "deaths"))  
table_2 <- scrape_pdf(list_of_tables = border_patrol,
```

```

  table_number = 2,
  number_columns = 6,
  column_names = c("sector",
                  "accompanied_juveniles",
                  "unaccompanied_juveniles",
                  "total_juveniles",
                  "total_adults",
                  "total_apprehensions"))
table_3 <- scrape_pdf(list_of_tables = border_patrol,
                      table_number = 3,
                      number_columns = 4,
                      column_names = c("sector",
                                      "female",
                                      "male",
                                      "total_apprehensions"))

```

We can use the function `left_join()` from the `dplyr` package to combine the three tables into a single object. In the first table there are some asterix after the final two row names in the Sector column. For our match to work properly we need to delete them which we can do using `gsub()`. If you look carefully at the Sector column in `table_1` you'll see that each value starts with a space (this is something that is hard to see just looking at the data and is found primarily when encountering an error that forces you to search as I did here). Since the other tables do not have their values start with a space, it won't match properly in `left_join()`. We'll fix this by running `trimws()` on the column from `table_1`.

```

table_1$sector <- gsub("\\*", "", table_1$sector)
table_1$sector <- trimws(table_1$sector)

```

Now we can run `left_join()`.

```

library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag

```

```
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
final_data <- left_join(table_1, table_2)
#> Joining, by = c("sector", "total_apprehensions")
final_data <- left_join(final_data, table_3)
#> Joining, by = c("sector", "total_apprehensions")
```

Let's take a look at the `head()` of this combined data.

```
head(final_data)
#>                                         sector agent_staffing total_apprehensions
#> 1                               Miami           111             2,280
#> 2                         New Orleans            63              920
#> 3                           Ramey            38              388
#> 4 Coastal Border Sectors Total          212            3,588
#> 5                           Blaine            296              288
#> 6                          Buffalo            277              447
#> other_than_mexican_apprehensions marijuana_pounds cocaine_pounds
#> 1                            1,646            2,253              231
#> 2                             528                21                 6
#> 3                             387                  3            2,932
#> 4                            2,561            2,277            3,169
#> 5                             237                  0                 0
#> 6                             293                228                 2
#> accepted_prosecutions assaults             rescues deaths accompanied_juveniles
#> 1                            292                 1            N/A      N/A               19
#> 2                             10                 0            N/A      N/A               1
#> 3                            89                 0            N/A      N/A               7
#> 4                            391                 1            N/A      ****            27
#> 5                             9                 0            N/A      N/A              29
#> 6                            37                 2            N/A      N/A               3
#> unaccompanied_juveniles total_juveniles total_adults female male
#> 1                            42                 61            2,219        219 2,061
#> 2                            22                 23            897          92   828
#> 3                             1                  8            380          65   323
#> 4                            65                 92            3,496        376 3,212
#> 5                             7                  36            252          97   191
```

#> 6	3	6	441	69	378
------	---	---	-----	----	-----

In one data set we now have information from three separate tables in a PDF. There's still some work to do - primarily convert the numeric columns to be actually numeric using `gsub()` to remove commas then using `as.numeric()` (or the `parse_numeric()` function from `readr`) on each column (probably through a for loop). but we have still made important progress getting useful data from a PDF table.

Chapter 16

More scraping tables from PDFs

In Chapter 15 we used the package `pdftools` to scrape tables on arrests/seizures from the United States Border Patrol that were only available in a PDF. Given the importance of PDF scraping - hopefully by the time you read this chapter more data will be available in reasonable formats and not in PDFs - in this chapter we'll continue working on scraping tables from PDFs. Here we will use the package `tabulizer` which has a number of features making it especially useful for grabbing tables from PDFs. One issue which we saw in Chapter 15 is that the table may not be the only thing on the page - the page could also have a title, page number etc. When using `pdftools` we use regular expressions and subsetting to remove all the extra lines. Using `tabulizer` we can simply say (through a handy function) that we only want a part of the page, so we only scrape the table itself. For more info about the `tabulizer` package please see their site [here](#).

For this chapter we'll scrape data from the Texas Commission on Jail Standards - Abbreviated Population Report. This is a report that shows monthly data on people incarcerated in jails for counties in Texas and is available [here](#). Make sure to download this file and put it in the working directory that you use to follow along.

This PDF is 9 pages long because of how many counties there are in Texas. Let's take a look at what the first page looks like. If you look at the PDF

yourself you'll see that every page follows the format of the 1st page, which greatly simplifies our scrape. The data is in county-month units which means that each row of data has info for a single county in a single month. We know that because the first column is "County" and each row is a single county (this is not true in every case. For example, on page 3 there are the rows 'Fannin 1(P)' and 'Fannin 2(P)', possibly indicating that there are two jails in that county. It is unclear from this PDF what the '(P)' means.). For knowing that the data is monthly, the title of this document says 'for 06/01/2020' indicating that it is for that date, though this doesn't by itself mean the data is monthly - it could be daily based only on this data.

To know that it's monthly data we'd have to go to the original source on the Texas Commission on Jail Standards website [here](#). On this page it says that 'Monthly population reports are available for review below,' which tells us that the data is monthly. It's important to know the unit so you can understand the data properly - primarily so you know what kinds of questions you can answer. If someone asks whether yearly trends on jail incarceration change in Texas, you can answer that with this data. If they ask whether more people are in jail on a Tuesday than on a Friday, you can't.

Just to understand what units our data is in we had to look at both the PDF itself and the site it came from. This kind of multi-step process is tedious but often necessary to truly understand your data. And even now we have questions - what does the (P) that's in some rows mean? For this we'd have to email or call the people who handle the data and ask directly. This is often the easiest way to answer your question, though different organizations have varying speeds in responding - if ever.

Now let's look at what columns are available. It looks like each column is the number of people incarcerated in the jail, broken down into categories of people. For example, the first two columns after County are 'Pretrial Felons' and 'Conv. Felons' so those are probably how many people are incarcerated who are awaiting trial for a felony and those already convicted of a felony. The other columns seem to follow this same format until the last few ones which describe the jail's capacity (i.e. how many people they can hold), what percent of capacity they are at, and specifically how many open beds they have.

Texas Commission on Jail Standards - Abbreviated Population Report for 06/01/2020

County	Conv. Felons Sentenced to County Jail time			Parole Violators with a New Charge			Pretrial Misd.			Conv. Misd.			Bench Warrants			Federal			Pretrial SJF			Conv. SJF Sentenced to Co. Jail Time			Conv. SJF Sentenced to State Jail					
	Pretrial Felons	Conv. Felons		Parole Violators			Pretrial Misd.	Conv. Misd.		Bench Warrants			Federal			Pretrial SJF				Total Others	Total Local	Total Contract	Total Population	Total Capacity	% of Capacity	Available Beds				
Anderson	81	13	3	1	5	12	1	0	0	21	0	1	0	138	0	138	300	45.00	132											
Andrews	23	11	0	2	4	11	0	0	0	5	0	6	0	35	0	35	50	70.00	10											
Angelina	79	35	4	6	0	14	0	3	0	23	0	3	1	168	0	168	279	60.22	83											
Aransas	23	10	0	2	6	7	0	6	73	2	0	0	0	56	73	129	212	60.85	62											
Archer	12	3	0	0	1	3	1	1	2	5	0	0	1	26	9	35	48	72.92	0											
Armstrong	1	1	0	0	0	0	0	0	0	0	0	0	0	2	0	2	8	25.00	0											
Atascosa	54	2	0	8	4	21	0	5	0	29	0	4	0	127	29	156	250	62.40	69											
Austin	17	2	0	1	3	3	0	0	0	5	0	0	0	2	32	0	32	90	35.56	49										
Bailey	10	2	0	0	0	4	0	0	56	0	0	2	1	19	59	78	96	81.25	0											
Bandera	26	8	0	1	3	3	2	0	0	1	0	1	0	45	11	56	96	58.33	30											
Bastrop	111	4	0	13	11	22	1	4	70	17	1	0	0	184	72	256	400	64.00	104											
Baylor	7	1	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0.00	0											
Bee	24	5	1	0	8	4	0	2	21	0	0	0	0	2	46	21	67	128	52.34	48										
Bell	365	74	1	32	53	55	13	1	11	66	0	10	3	672	11	683	1184	57.69	383											
Bexar	1552	225	56	288	302	253	10	52	2	287	5	52	379	3461	11	3472	5108	67.97	1125											
Blanco	7	2	0	1	0	1	0	0	0	1	0	0	0	12	0	12	56	21.43	38											
Borden	1	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0.00	0											
Bosque	4	1	0	0	1	1	1	1	0	1	0	0	2	12	0	12	64	18.75	46											
Bowie (P)	31	29	0	2	5	22	4	2	47	24	2	4	1	126	490	616	921	66.88	213											
Brazoria	219	167	10	17	29	34	18	15	0	96	0	35	14	654	0	654	1170	55.90	399											
Brazos	200	78	3	16	66	37	8	6	0	52	0	20	24	510	0	510	1089	46.83	470											
Brewster	3	0	0	0	0	0	0	0	38	0	0	0	0	3	38	41	56	73.21	0											
Briscoe	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.00	0											
Brooks	15	1	1	1	0	0	0	0	0	0	0	0	0	18	0	18	36	50.00	14											
Brooks (P)	0	0	0	0	0	0	0	0	0	164	0	0	0	0	0	0	408	408	652	62.58	179									
Brown	70	20	0	7	20	9	0	2	0	0	3	0	2	133	7	140	196	71.43	36											
Burleson	19	2	0	2	0	3	0	0	0	0	0	0	1	27	0	27	96	28.13	59											
Burnet	57	23	1	5	9	3	0	1	0	10	1	0	0	110	158	268	595	45.04	268											
CalDWELL	89	4	0	3	2	26	1	2	19	13	0	3	0	143	21	164	301	54.49	107											

6/9/2020

Page 1 of 9

Now that we've familiarized ourselves with the data, let's begin scraping this data using `tabulizer`. If you don't have this package installed, you'll need to install it using `install.packages("tabulizer")`. Then we'll need to run `library(tabulizer)`.

```
library(tabulizer)
```

The main function that we'll be using from the `tabulizer` package is `extract_tables()`. This function basically looks at a PDF page, figures out which part of the page is a table, and then scrapes just that table. As we'll see, it's not always perfect at figuring out what part of the page is a table so we can also tell it exactly where to look. You can look at all of the features of `extract_tables()` by running `help(extract_tables)`.

```
data <- extract_tables(file = "data/AbbreRptCurrent.pdf")
is(data)
#> [1] "list"    "vector"
length(data)
```

```
#> [1] 18
data[[1]]
#> [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
#> [1,]     ""       ""       ""       "Conv. Felons"    ""
#> [2,]     ""       ""       ""       "Sentenced to"   ""
#> [3,]     ""       ""       ""       "County Jail"    ""
#> [4,]     ""       "Pretrial" "Conv."    ""       "Parole"      ""
#> [5,]     ""       ""       ""       "time"        ""       "Charge"
#> [6,]     "County"  "Felons"  "Felons"  ""       "Violators"   ""
#> [,7]      [,8]      [,9]      [,10]     [,11]     [,12]
#> [1,]     ""       ""       ""       ""       ""       "Conv. SJF"
#> [2,]     ""       ""       ""       ""       ""       "Sentenced"
#> [3,]     ""       ""       ""       ""       ""       "to Co. Jail"
#> [4,]     "Pretrial" "Conv."   "Bench"   ""       "Pretrial"   ""
#> [5,]     ""       ""       ""       ""       ""       "Time"
#> [6,]     "Misd."   "Misd."   "Warrants" "Federal"  "SJF"    ""
#> [,13]     [,14]     [,15]     [,16]     [,17]     [,18]
#> [1,]     "Conv."   ""       ""       ""       ""       ""
#> [2,]     "SJF"    ""       ""       ""       ""       ""
#> [3,]     "Sentenced" ""       ""       ""       ""       ""
#> [4,]     ""       "Total"   "Total"   "Total"   "Total"   "Total"
#> [5,]     "to State Jail" ""       ""       ""       ""       ""
#> [6,]     ""       "Others"  "Local"   "Contract" "Population" "Capacity"
#> [,19]     [,20]
#> [1,]     ""
#> [2,]     ""
#> [3,]     ""
#> [4,]     "% of"   "Available"
#> [5,]     ""
#> [6,]     "Capacity" "Beds"
head(data[[2]])
#> [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]      [,9]      [,10]     [,11]     [,12]
#> [1,]     "Anderson" "81"     "13"     "3"      "1"      "5"      "12"     "1"      "0"      "0"      "21"     "0"
#> [2,]     "Andrews"   "23"     "11"     "0"      "2"      "4"      "11"     "0"      "0"      "0"      "5"      "0"
#> [3,]     "Angelina"  "79"     "35"     "4"      "6"      "0"      "14"     "0"      "3"      "0"      "23"     "0"
#> [4,]     "Aransas"   "23"     "10"     "0"      "2"      "6"      "7"      "0"      "6"      "73"    "2"      "0"
#> [5,]     "Archer"    "12"     "3"      "0"      "0"      "1"      "3"      "1"      "1"      "2"      "5"      "0"
```

```
#> [6,] "Armstrong" "1"  "1"  "0"  "0"  "0"  "0"  "0"  "0"  "0"  "0"  "0"  "0"
#>      [,13] [,14] [,15] [,16] [,17] [,18] [,19]   [,20]
#> [1,] "1"    "0"   "138" "0"   "138" "300" "46.00" "132"
#> [2,] "6"    "0"   "35"  "0"   "35"  "50"  "70.00" "10"
#> [3,] "3"    "1"   "168" "0"   "168" "279" "60.22" "83"
#> [4,] "0"    "0"   "56"  "73"  "129" "212" "60.85" "62"
#> [5,] "0"    "1"   "26"  "9"   "35"  "48"  "72.92" "0"
#> [6,] "0"    "0"   "2"   "0"   "2"   "8"   "25.00" "0"
```

Above is scraping code and some output when running `extract_tables()` on our PDF using all of the default options in that function. The only parameter we put in the function is `file = "data/Abbreviated Pop Rpt Dec 2017.pdf"`. This is just telling the function where to look for the PDF. I have the PDF in the data folder of my project; you'll need to change this to have `extract_tables()` look in the right place for the PDF on your computer.

You can see from the output that the scrape was successful - but our work isn't done yet. The results from `is(data)` say that the scrape returned a list, and from `length(data)` we learn that it's a list of length 18. Why is this? We have 9 pages so it is reasonable that we would have 9 lists since we have one table per page, but we shouldn't have 19 tables. Let's look again at just the first table - as it is a list, we'll need double square brackets to pull just the first element in the list.

```
data[[1]]
#>      [,1]     [,2]     [,3]     [,4]      [,5]      [,6]
#> [1,] ""      ""      ""      "Conv. Felons"  ""      "Parole"
#> [2,] ""      ""      ""      "Sentenced to"  ""      "Violators"
#> [3,] ""      ""      ""      "County Jail"   ""      "with a New"
#> [4,] ""      "Pretrial" "Conv."   ""      "Parole"   ""
#> [5,] ""      ""      ""      "time"      ""      "Charge"
#> [6,] "County" "Felons"  "Felons"  ""      "Violators"  ""
#>      [,7]     [,8]     [,9]     [,10]     [,11]     [,12]
#> [1,] ""      ""      ""      ""      ""      "Conv. SJF"
#> [2,] ""      ""      ""      ""      ""      "Sentenced"
#> [3,] ""      ""      ""      ""      ""      "to Co. Jail"
#> [4,] "Pretrial" "Conv."  "Bench"  ""      "Pretrial"  ""
```

```
#> [5,]   ""      ""      ""      ""      ""      "Time"
#> [6,] "Misd." "Misd." "Warrants" "Federal" "SJF"   ""
#> [,13]      [,14]   [,15]   [,16]   [,17]   [,18]
#> [1,] "Conv."   ""      ""      ""      ""      ""
#> [2,] "SJF"     ""      ""      ""      ""      ""
#> [3,] "Sentenced"   ""      ""      ""      ""      ""
#> [4,] ""        "Total"  "Total"  "Total"  "Total"  "Total"
#> [5,] "to State Jail"   ""      ""      ""      ""      ""
#> [6,] ""        "Others" "Local"  "Contract" "Population" "Capacity"
#>      [,19]   [,20]
#> [1,]   ""
#> [2,]   ""
#> [3,]   ""
#> [4,] "% of"   "Available"
#> [5,]   ""
#> [6,] "Capacity" "Beds"
```

The results from `data[[1]]` provide some answers. It has the right number of columns but only 6 rows! This is our first table so should be the entire table we can see on page 1. Instead, it appears to be just the column names, with 6 rows because some column names are on multiple rows. Here's the issue, we can read the table and easily see that the column names may be on multiple rows but belong together, and that they are part of the table. `tabulizer` can't see this obvious fact as we can, it must rely on a series of rules to indicate what is part of a table and what isn't. For example, having white space between columns and thin black lines around rows tells it where each row and column is. Our issue is that the column names appear to just be text until there is a thick black line and (in `tabulizer`'s mind) the table begins, so it keeps the column name part separate from the rest of the table. Now let's look closer at table 2 and see if it is correct for the table on page 1 of our PDF.

```
head(data[[2]])
#>      [,1]      [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
#> [1,] "Anderson" "81" "13" "3"  "1"  "5"  "12" "1"  "0"  "0"   "21"  "0"
#> [2,] "Andrews"   "23" "11" "0"  "2"  "4"  "11" "0"  "0"  "0"   "5"   "0"
#> [3,] "Angelina"  "79" "35" "4"  "6"  "0"  "14" "0"  "3"  "0"   "23"  "0"
#> [4,] "Aransas"   "23" "10" "0"  "2"  "6"  "7"  "0"  "6"  "73"  "2"   "0"
```

```

#> [5,] "Archer"    "12"  "3"   "0"   "0"   "1"   "3"   "1"   "1"   "2"   "5"   "0"
#> [6,] "Armstrong" "1"   "1"   "0"   "0"   "0"   "0"   "0"   "0"   "0"   "0"   "0"
#> [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
#> [1,] "1"   "0"   "138" "0"   "138" "300" "46.00" "132"
#> [2,] "6"   "0"   "35"  "0"   "35"  "50"  "70.00" "10"
#> [3,] "3"   "1"   "168" "0"   "168" "279" "60.22" "83"
#> [4,] "0"   "0"   "56"  "73"  "129" "212" "60.85" "62"
#> [5,] "0"   "1"   "26"  "9"   "35"  "48"  "72.92" "0"
#> [6,] "0"   "0"   "2"   "0"   "2"   "8"   "25.00" "0"
tail(data[[2]])
#>      [,1]      [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
#> [24,] "Brooks" "15"  "1"  "1"  "1"  "0"  "0"  "0"  "0"  "0"  "0"  "0"  "0"
#> [25,] "Brooks (P)" "0"  "0"  "0"  "0"  "0"  "0"  "0"  "0"  "164" "0"  "0"  "0"
#> [26,] "Brown"   "70"  "20" "0"  "7"  "20" "9"  "0"  "2"  "0"  "0"  "0"  "3"
#> [27,] "Burleson" "19" "2"  "0"  "2"  "0"  "3"  "0"  "0"  "0"  "0"  "0"  "0"
#> [28,] "Burnet"  "57"  "23" "1"  "5"  "9"  "3"  "0"  "1"  "0"  "10" "1"  "1"
#> [29,] "Caldwell" "89" "4"  "0"  "3"  "2"  "26" "1"  "2"  "19" "13" "0"  "0"
#>      [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
#> [24,] "0"   "0"   "18"  "0"   "18"  "36"  "50.00" "14"
#> [25,] "0"   "0"   "0"   "408" "408" "652" "62.58" "179"
#> [26,] "0"   "2"   "133" "7"   "140" "196" "71.43" "36"
#> [27,] "0"   "1"   "27"  "0"   "27"  "96"  "28.13" "59"
#> [28,] "0"   "0"   "110" "158" "268" "595" "45.04" "268"
#> [29,] "3"   "0"   "143" "21"  "164" "301" "54.49" "107"

```

We're looking just at the `head()` and `tail()` to get the first and 6 six rows as otherwise we'd print out all 29 rows in that table. When you are exploring your own data, you'll probably want to be more thorough and ensure that rows around the middle are also correct - but this is a good first pass. If you look at the output the table and compare it to the PDF, you'll see that the scrape was successful. Every row is where it should be and the columns are correct - unlike using `pdftools()` we have the results already in proper columns. One thing to note is that this data isn't in a `data.frame` format, it's in a matrix. Matrices are the default output of `extract_tables()` though you can set it to output a `data.frame` by setting the parameter `output = "data.frame"`. In our case we actually wouldn't want that due to the issue of the column names.

```

data <- extract_tables(file = "data/AbbreRptCurrent.pdf", output = "data.frame")
head(data[[2]])
#> Anderson X81 X13 X3 X1 X5 X12 X1.1 X0 X0.1 X21 X0.2 X1.2 X0.3 X138 X0.
#> 1 Andrews 23 11 0 2 4 11 0 0 0 5 0 6 0 35
#> 2 Angelina 79 35 4 6 0 14 0 3 0 23 0 3 1 168
#> 3 Aransas 23 10 0 2 6 7 0 6 73 2 0 0 0 56 7
#> 4 Archer 12 3 0 0 1 3 1 1 2 5 0 0 1 26
#> 5 Armstrong 1 1 0 0 0 0 0 0 0 0 0 0 0 0 2
#> 6 Atascosa 54 2 0 8 4 21 0 5 0 29 0 4 0 127 2
#> X138.1 X300 X46.00 X132
#> 1 35 50 70.00 10
#> 2 168 279 60.22 83
#> 3 129 212 60.85 62
#> 4 35 48 72.92 0
#> 5 2 8 25.00 0
#> 6 156 250 62.40 69

```

Above we reran the `extract_tables()` code and just added a parameter to make the output a `data.frame` instead of a `matrix`. Now it sets the first row - which should be the columns - as the column name, which is not correct. We'll have to fix the column names first before we can convert the result from a `matrix` to a `data.frame`.

We'll use the `apply()` function on our data to create column names from it. The `apply()` function is actual part of a “family” of similar function which essentially operate as quicker for loops - we could have done a for loop to solve this problem. The `apply()` function takes as an input a vector or matrix data set, and then performs some function on either its rows or its columns. Our first input is our data, the first element of the `data` list. Then we put the number 2 to indicate that we want it to perform our function on each column of the matrix - if we put 1, that would perform the function on each row. So now what function to use on this data? We want to combine each row together into a single string per column. To do that we can use the `paste()` function and use the `collapse` parameter to combine multiple strings to a single string. So after the 2 we enter `paste`, and then just add `collapse = ""` (the `""` means that we're not putting anything between the strings when combining them) separated by a comma. Let's see what it returns.

```

apply(data[[1]], 2, paste, collapse = "")
#>                               X                               X.1
#>           "County"                  "PretrialFelons"
#>           X.2                      Conv..Felons
#>           "Conv.Felons"  "Sentenced toCounty Jailtime"
#>           X.3                      Parole
#>           "ParoleViolators"  "Violatorswith a NewCharge"
#>           X.4                               X.5
#>           "PretrialMisd."      "Conv.Misd."
#>           X.6                               X.7
#>           "BenchWarrants"      "Federal"
#>           X.8                      Conv..SJF
#>           "PretrialsSJF"      "Sentencedto Co. JailTime"
#>           Conv.                               X.9
#>           "SJFSentencedto State Jail"      "TotalOthers"
#>           X.10                               X.11
#>           "TotalLocal"      "TotalContract"
#>           X.12                               X.13
#>           "TotalPopulation"      "TotalCapacity"
#>           X.14                               X.15
#>           "% ofCapacity"      "AvailableBeds"

```

Now we have a vector of 20 strings, one per column in our data. We will use this to name the columns in our data set with the actual info from the scraped table. One helper function first. The column names don't follow conventional R style for column names - it has spaces, uppercase letters, punctuation other than the underscore. To easily fix this we can use the `make_clean_names()` function from the `janitor` package. If you don't have this package installed, install it using `install.packages("janitor")`. We'll first save the results of the above `apply()` function as a vector we can call `column_names` and then run the `make_clean_names()` function on it. The input to `make_clean_names()` is the vector of strings (our column names) and it'll return that vector but now with names in R's common style.

```

library(janitor)
#>
#> Attaching package: 'janitor'
#> The following objects are masked from 'package:stats':

```

```
#>
#>      chisq.test, fisher.test
column_names <- apply(data[[1]], 2, paste, collapse = "")
column_names <- make_clean_names(column_names)
column_names
#> [1] "county"                  "pretrial_felons"
#> [3] "conv_felons"             "sentenced_to_county_jailtime"
#> [5] "parole_violators"        "violatorswith_a_new_charge"
#> [7] "pretrial_misd"           "conv_misd"
#> [9] "bench_warrants"          "federal"
#> [11] "pretrial_sjf"           "sentencedto_co_jail_time"
#> [13] "sjf_sentencedto_state_jail" "total_others"
#> [15] "total_local"            "total_contract"
#> [17] "total_population"       "total_capacity"
#> [19] "percent_of_capacity"    "available_beds"
```

Now the column names are in the proper style. And notice the 19th value, it changed from the percent sign to the word “percent”.

We can combine the results from the first table - the column names - with that of the second table - the actual data - to have a complete table from page 1 of our PDF. We do this simply by making table 2 a data.frame and using `names()` to rename the columns to the ones we made above. Since this is the table from page 1 of the PDF, we’ll call the object `page1_table`.

```
page1_table <- data[[2]]
page1_table <- data.frame(page1_table)
names(page1_table) <- column_names
head(page1_table)
#>   county pretrial_felons conv_felons sentenced_to_county_jailtime
#> 1 Andrews      23          11                  0
#> 2 Angelina     79          35                  4
#> 3 Aransas      23          10                  0
#> 4 Archer        12          3                   0
#> 5 Armstrong     1           1                   0
#> 6 Atascosa     54          2                   0
#>   parole_violators violatorswith_a_new_charge pretrial_misd conv_misd
#> 1                      2                         4                 11                  0
```

```

#> 2          6          0        14        0
#> 3          2          6        7        0
#> 4          0          1        3        1
#> 5          0          0        0        0
#> 6          8          4        21        0
#>   bench_warrants federal pretrial_sjf sentencedto_co_jail_time
#> 1          0          0        5        0
#> 2          3          0       23        0
#> 3          6         73        2        0
#> 4          1          2        5        0
#> 5          0          0        0        0
#> 6          5          0       29        0
#>   sjf_sentencedto_state_jail total_others total_local total_contract
#> 1          6          0        35        0
#> 2          3          1       168        0
#> 3          0          0        56       73
#> 4          0          1        26        9
#> 5          0          0        2        0
#> 6          4          0       127       29
#>   total_population total_capacity percent_of_capacity available_beds
#> 1         35         50      70.00       10
#> 2        168        279      60.22       83
#> 3        129        212      60.85       62
#> 4         35         48      72.92        0
#> 5          2          8      25.00        0
#> 6        156        250      62.40       69
tail(page1_table)
#>   county pretrial_felons conv Felons sentenced_to_county_jailtime
#> 23 Brooks           15      1          1
#> 24 Brooks (P)       0       0          0
#> 25 Brown            70     20          0
#> 26 Burleson          19      2          0
#> 27 Burnet            57     23          1
#> 28 Caldwell          89      4          0
#>   parole_violators violatorswith_a_new_charge pretrial_misd conv_misd
#> 23                 1          0          0          0
#> 24                 0          0          0          0

```

#> 25	7	20	9	0
#> 26	2	0	3	0
#> 27	5	9	3	0
#> 28	3	2	26	1
	<i>bench_warrants federal pretrial_sjf sentencedto_co_jail_time</i>			
#> 23	0	0	0	0
#> 24	0	164	0	0
#> 25	2	0	0	3
#> 26	0	0	0	0
#> 27	1	0	10	1
#> 28	2	19	13	0
	<i>sjf_sentencedto_state_jail total_others total_local total_contract</i>			
#> 23	0	0	18	0
#> 24	0	0	0	408
#> 25	0	2	133	7
#> 26	0	1	27	0
#> 27	0	0	110	158
#> 28	3	0	143	21
	<i>total_population total_capacity percent_of_capacity available_beds</i>			
#> 23	18	36	50.00	14
#> 24	408	652	62.58	179
#> 25	140	196	71.43	36
#> 26	27	96	28.13	59
#> 27	268	595	45.04	268
#> 28	164	301	54.49	107

Looking at the results of `head()` and `tail()` (if this was data that you were using in your project you'd want to look closer than just these checks) shows that we've done this correctly. The values are right and the column names are correct. Complete the rest of the PDF on your own. You can follow the same steps as above but now that we've made the `column_names` object you can reuse that for the other tables. This is only true because each page has the same column names. Otherwise you'd have to fix the column names for each page of the PDF.

So why did I choose this example when it highlights a limitation of an otherwise very effective R package? A lot of the work you do in R is going to be like the example we went through - there are tools to solve *most* of the data

problems, but you'll need to spend time fixing the extra issues. And since a lot of problems are fairly unique (at least insofar as there are differences in your exact problem even if problems are generally similar) there's usually not a R function to solve everything. Below is another (brief) example of the `tabulizer` package working perfectly - but with a few issues just due to how the data is arranged on the PDF.

16.1 Pregnant Women Incarcerated

We'll finish this lesson with another example of data from Texas - this time using data on the number of pregnant women booked in Texas county jails. This data has a unique challenge, it has 10 columns but we want to make it have only 2. In the data (shown below), it starts with a column of county names, then a column of the number of pregnant women booked into that county's jail. Next is another column of county names - instead of continuing onto another page, this data just makes new columns when it runs out of room. We'll scrape this PDF using `tabulizer()` and then work to fix this multiple-column issue. The file is called "PregnantFemaleReportingCurrent.pdf" and is available on GitHub [here](#). Make sure to download this file and put it in the proper working directory for the `extract_tables()` function we use below.

Pregnant Females Booked In Texas County Jails for 6/1/2020						Month Total:	305
Anderson	0	Delta	0	Irion	0	Motley	0
Andrews	1	Denton	3	Jack	0	Nacogdoches	2
Angelina	0	DeWitt	0	Jackson	1	Navarro	2
Aransas	0	Dickens	0	Jasper	0	Newton	0
Archer	1	Dickens (P)	0	Jeff Davis	0	Newton (P)	0
Armstrong	0	Dimmit	0	Jefferson	0	Nolan	2
Atascosa	0	Donley	0	Jefferson (P)	0	Nueces	4
Austin	0	Duval	0	Jim Hogg	0	Ochiltree	0
Bailey	0	Eastland	0	Jim Wells	0	Oldham	0
Bandera	0	Ector	3	Johnson	2	Orange	0
Bastrop	0	Edwards	0	Jones	0	Palo Pinto	1
Baylor	0	El Paso	8	Karnes	0	Panola	0
Bee	0	Ellis	0	Karnes (P)	0	Parker	1
Bell	9	Erath	0	Kaufman	3	Parmer	0
Bexar	27	Falls	0	Kendall	1	Pecos	0
Blanco	0	Fannin 1(P)	0	Kenedy	0	Polk	0
Borden	0	Fannin 2(P)	2	Kent	0	Polk (P)	0
Bosque	0	Fayette	0	Kerr	1	Potter	4
Bowie (P)	0	Fisher	0	Kimble	0	Presidio	0
Brazoria	2	Floyd	0	King	0	Rains	0
Brazos	5	Foard	0	Kinney	0	Randall	0
Brewster	0	Fort Bend	4	Kleberg	1	Reagan	0
Briscoe	0	Franklin	0	Knox	0	Real	0
Brooks	0	Freestone	1	La Salle	0	Red River	0
Brooks (P)	1	Frio (P)	0	Lamar	0	Reeves	0
Brown	1	Gaines	0	Lamb	0	Refugio	1
Burleson	1	Galveston	5	Lampasas	0	Roberts	0
Burnet 1(P)	5	Garza	0	Lavaca	0	Robertson	0
Caldwell	0	Gillespie	0	Lee	0	Rockwall	0
Calhoun	1	Glasscock	0	Leon	0	Runnels	0
Callahan	0	Goliad	0	Liberty (P)	3	Rusk	0
Cameron	6	Gonzales	1	Limestone	0	Sabine	0
Camp	0	Gray	1	Lipscomb	0	San Augustine	0
Carson	0	Grayson	4	Live Oak	0	San Jacinto	0
Cass	0	Gregg	1	Llano	1	San Patricio	0
Castro	0	Grimes	0	Loving	0	San Saba	0
Chambers	0	Guadalupe	6	Lubbock	9	Schleicher	0
Cherokee	0	Hale	1	Lynn	0	Scurry	0
Childress	0	Hall	0	Madison	0	Shackelford	0
Clay	0	Hamilton	0	Marion	0	Shelby	0
Cochran	0	Hansford	0	Martin	0	Sherman	0
Coke	0	Hardeman	0	Mason	0	Smith	9
Coleman	0	Hardin	0	Matagorda	0	Somervell	0
Collins	0	Harris	10	Maverick	1	Tarrant	0

Notice that this data doesn't even have column names. Whereas earlier in this chapter we have to combine multiple rows to form the column names, here we will have to create the names entirely ourselves. This is always a bit risky as maybe next month the table will change and if we hard-code any column names, we'll either have code that breaks or - much more dangerous - mislabel the columns without noticing. In cases like this we have no other choice, but if you intend to scrape something that recurring - that is, that you'll scrape a future version of - be careful about situations like this.

We'll start scraping this PDF using the standard `extract_tables()` function without any parameters other than the file name. This is usually a good start since it's quick and often works - and if it doesn't, we haven't lost much time checking. Since we know `extract_tables()` will return a list by default,

we'll save the result of `extract_tables()` as an object called `data` and then just pull the first element (i.e. the only element if this works) from that list.

```
data <- extract_tables(file = "data/PregnantFemaleReportingCurrent.pdf")
data <- data[[1]]
data

#>      [,1]          [,2]        [,3]       [,4]      [,5]      [,6]
#> [1,] "Anderson"    "0"   "Delta"     "0"   "Irion"    "0"
#> [2,] "Andrews"     "1"   "Denton"    "3"   "Jack"     "0"
#> [3,] "Angelina"    "0"   "DeWitt"    "0"   "Jackson"  "1"
#> [4,] "Aransas"     "0"   "Dickens"   "0"   "Jasper"   "0"
#> [5,] "Archer"      "1"   "Dickens (P)" "0"   "Jeff Davis" "0"
#> [6,] "Armstrong"   "0"   "Dimmit"    "0"   "Jefferson" "0"
#> [7,] "Atascosa"    "0"   "Donley"    "0"   "Jefferson (P)" "0"
#> [8,] "Austin"       "0"   "Duval"     "0"   "Jim Hogg"  "0"
#> [9,] "Bailey"      "0"   "Eastland"   "0"   "Jim Wells" "0"
#> [10,] "Bandera"    "0"   "Ector"     "3"   "Johnson"  "2"
#> [11,] "Bastrop"    "0"   "Edwards"   "0"   "Jones"     "0"
#> [12,] "Baylor"     "0"   "El Paso"    "8"   "Karnes"    "0"
#> [13,] "Bee"         "0"   "Ellis"     "0"   "Karnes (P)" "0"
#> [14,] "Bell"        "9"   "Erath"     "0"   "Kaufman"   "3"
#> [15,] "Bexar"       "27"  "Falls"     "0"   "Kendall"   "1"
#> [16,] "Blanco"      "0"   "Fannin 1(P)" "0"   "Kenedy"    "0"
#> [17,] "Borden"      "0"   "Fannin 2(P)" "2"   "Kent"      "0"
#> [18,] "Bosque"      "0"   "Fayette"    "0"   "Kerr"      "1"
#> [19,] "Bowie (P)"   "0"   "Fisher"    "0"   "Kimble"    "0"
#> [20,] "Brazoria"   "2"   "Floyd"     "0"   "King"      "0"
#> [21,] "Brazos"      "5"   "Foard"     "0"   "Kinney"    "0"
#> [22,] "Brewster"    "0"   "Fort Bend"  "4"   "Kleberg"   "1"
#> [23,] "Briscoe"     "0"   "Franklin"  "0"   "Knox"      "0"
#> [24,] "Brooks"      "0"   "Freestone" "1"   "La Salle"   "0"
#> [25,] "Brooks (P)"  "1"   "Frio (P)"  "0"   "Lamar"     "0"
#> [26,] "Brown"        "1"   "Gaines"    "0"   "Lamb"      "0"
#> [27,] "Burleson"    "1"   "Galveston" "5"   "Lampasas"  "0"
#> [28,] "Burnet 1(P)" "5"   "Garza"     "0"   "Lavaca"    "0"
#> [29,] "Caldwell"    "0"   "Gillespie" "0"   "Lee"       "0"
#> [30,] "Calhoun"     "1"   "Glasscock" "0"   "Leon"      "0"
#> [31,] "Callahan"    "0"   "Goliad"    "0"   "Liberty (P)" "3"
```

```

#> [32,] "Cameron"      "6"   "Gonzales"      "1"   "Limestone"    "0"
#> [33,] "Camp"         "0"   "Gray"          "1"   "Lipscomb"     "0"
#> [34,] "Carson"       "0"   "Grayson"       "4"   "Live Oak"     "0"
#> [35,] "Cass"          "0"   "Gregg"          "1"   "Llano"         "1"
#> [36,] "Castro"        "0"   "Grimes"         "0"   "Loving"        "0"
#> [37,] "Chambers"      "0"   "Guadalupe"     "6"   "Lubbock"      "9"
#> [38,] "Cherokee"      "0"   "Hale"           "1"   "Lynn"          "0"
#> [39,] "Childress"     "0"   "Hall"           "0"   "Madison"       "0"
#> [40,] "Clay"          "0"   "Hamilton"      "0"   "Marion"        "0"
#> [41,] "Cochran"       "0"   "Hansford"      "0"   "Martin"        "0"
#> [42,] "Coke"          "0"   "Hardeman"      "0"   "Mason"         "0"
#> [43,] "Coleman"       "0"   "Hardin"         "0"   "Matagorda"    "0"
#> [44,] "Collin"        "9"   "Harris"         "19"  "Maverick"     "1"
#> [45,] "Collingsworth" "0"   "Harrison"      "0"   "Maverick (P)" "0"
#> [46,] "Colorado"       "0"   "Haskell (P)"  "2"   "McCulloch"    "0"
#> [47,] "Comal"          "0"   "Hays"           "0"   "McLennan"      "6"
#> [48,] "Comanche"      "0"   "Hemphill"       "0"   "McLennan 1(P)" "0"
#> [49,] "Concho"         "0"   "Henderson"     "2"   "McLennan 2(P)" "0"
#> [50,] "Cooke"          "2"   "Hidalgo"        "6"   "McMullen"     "0"
#> [51,] "Coryell"        "1"   "Hidalgo (P)"  "0"   "Medina"        "0"
#> [52,] "Cottle"         "0"   "Hill"           "0"   "Menard"        "0"
#> [53,] "Crane"          "0"   "Hockley"        "1"   "Midland"       "2"
#> [54,] "Crockett"       "0"   "Hood"           "3"   "Milam"         "0"
#> [55,] "Crosby"         "0"   "Hopkins"        "1"   "Mills"         "0"
#> [56,] "Culberson"      "0"   "Houston"        "0"   "Mitchell"      "0"
#> [57,] "Dallam"         "0"   "Howard"         "2"   "Montague"     "0"
#> [58,] "Dallas"          "22"  "Hudspeth"      "0"   "Montgomery"   "9"
#> [59,] "Dawson"         "0"   "Hunt"           "2"   "Moore"         "0"
#> [60,] "Deaf Smith"     "0"   "Hutchinson"    "0"   "Morris"        "0"
#> [,7]                  [,8]  [,9]                [,10]
#> [1,] "Motley"          "0"   "Upton"          "0"
#> [2,] "Nacogdoches"     "2"   "Uvalde"         "0"
#> [3,] "Navarro"          "2"   "Val Verde (P)" "1"
#> [4,] "Newton"          "0"   "Van Zandt"      "0"
#> [5,] "Newton (P)"      "0"   "Victoria"      "1"
#> [6,] "Nolan"            "2"   "Walker"         "1"
#> [7,] "Nueces"          "4"   "Waller"         "0"

```

```

#> [8,] "Ochiltree"      "0"  "Ward"        "0"
#> [9,] "Oldham"         "0"  "Washington"   "0"
#> [10,] "Orange"        "0"  "Webb"         "3"
#> [11,] "Palo Pinto"    "1"  "Wharton"     "1"
#> [12,] "Panola"        "0"  "Wheeler"      "0"
#> [13,] "Parker"        "1"  "Wichita"      "3"
#> [14,] "Parmer"        "0"  "Wilbarger"    "0"
#> [15,] "Pecos"         "0"  "Willacy"      "0"
#> [16,] "Polk"          "0"  "Williamson"   "2"
#> [17,] "Polk (P)"      "0"  "Wilson"       "0"
#> [18,] "Potter"        "4"  "Winkler"      "0"
#> [19,] "Presidio"      "0"  "Wise"         "1"
#> [20,] "Rains"         "0"  "Wood"         "0"
#> [21,] "Randall"       "0"  "Yoakum"       "0"
#> [22,] "Reagan"        "0"  "Young"        "0"
#> [23,] "Real"          "0"  "Zapata"       "0"
#> [24,] "Red River"     "0"  "Zavala"       "0"
#> [25,] "Reeves"        "0"  "Zavala (P)"  "0"
#> [26,] "Refugio"       "1"  ""             ""
#> [27,] "Roberts"       "0"  ""             ""
#> [28,] "Robertson"     "0"  ""             ""
#> [29,] "Rockwall"      "0"  ""             ""
#> [30,] "Runnels"       "0"  ""             ""
#> [31,] "Rusk"          "0"  ""             ""
#> [32,] "Sabine"        "0"  ""             ""
#> [33,] "San Augustine" "0"  ""             ""
#> [34,] "San Jacinto"   "0"  ""             ""
#> [35,] "San Patricio"  "0"  ""             ""
#> [36,] "San Saba"      "0"  ""             ""
#> [37,] "Schleicher"    "0"  ""             ""
#> [38,] "Scurry"        "0"  ""             ""
#> [39,] "Shackelford"   "0"  ""             ""
#> [40,] "Shelby"         "0"  ""             ""
#> [41,] "Sherman"       "0"  ""             ""
#> [42,] "Smith"          "9"  ""             ""
#> [43,] "Somervell"     "0"  ""             ""
#> [44,] "Starr"          "0"  ""             "

```

```
#> [45,] "Stephens"      "1"   ""      ""
#> [46,] "Sterling"      "0"   ""      ""
#> [47,] "Stonewall"     "0"   ""      ""
#> [48,] "Sutton"        "0"   ""      ""
#> [49,] "Swisher"       "0"   ""      ""
#> [50,] "Tarrant"       "34"  ""      ""
#> [51,] "Taylor"        "6"   ""      ""
#> [52,] "Terrell"       "0"   ""      ""
#> [53,] "Terry"         "1"   ""      ""
#> [54,] "Throckmorton" "0"   ""      ""
#> [55,] "Titus"          "0"   ""      ""
#> [56,] "Tom Green"     "2"   ""      ""
#> [57,] "Travis"        "10"  ""      ""
#> [58,] "Trinity"       "0"   ""      ""
#> [59,] "Tyler"          "0"   ""      ""
#> [60,] "Upshur"        "0"   ""      ""
```

If we check the output from the above code to the PDF, we can see that it worked. Every column in the PDF is in our output and the values were scraped correctly. This is great! Now we want to make two columns - “county” and “pregnant_females_booked” (or whatever you’d like to call it) - from these 10. As usual with R, there are a few ways we can do this. We’ll just do two ways. First, since there are only 10 columns, we can just do it manually. We can use square bracket [] notation to grab specific columns using the column number (since the data is a matrix and not a data.frame we can’t use dollar sign notation even if we wanted to). Let’s print out the head of all the county columns. We can see from the PDF that these are columns 1, 3, 5, 7, and 9. So can use a vector of numbers to get that c(1, 3, 5, 7, 9).

```
head(data[, c(1, 3, 5, 7, 9)])
#>      [,1]      [,2]      [,3]      [,4]      [,5]
#> [1,] "Anderson" "Delta"  "Irion"  "Motley" "Upton"
#> [2,] "Andrews"   "Denton" "Jack"   "Nacogdoches" "Uvalde"
#> [3,] "Angelina"  "DeWitt" "Jackson" "Navarro" "Val Verde (P)"
#> [4,] "Aransas"   "Dickens" "Jasper" "Newton" "Van Zandt"
#> [5,] "Archer"    "Dickens (P)" "Jeff Davis" "Newton (P)" "Victoria"
#> [6,] "Armstrong" "Dimmit" "Jefferson" "Nolan" "Walker"
```

Now again for the “pregnant_females_booked” column.

```
head(data[, c(2, 4, 6, 8, 10)])
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] "0"  "0"  "0"  "0"  "0"
#> [2,] "1"  "3"  "0"  "2"  "0"
#> [3,] "0"  "0"  "1"  "2"  "1"
#> [4,] "0"  "0"  "0"  "0"  "0"
#> [5,] "1"  "0"  "0"  "0"  "1"
#> [6,] "0"  "0"  "0"  "2"  "1"
```

These results look right so we can make a data.frame using the `data.frame()` and having the input be from the above code - removing the `head()` function since we want every now. Conveniently, `data.frame()` allows us to name the columns we are making so we'll name the two columns “county” and “pregnant_females_booked”. We'll save the result as `data` and check out the `head()` and `tail()` of that data.frame.

```
data <- data.frame(county = c(data[, c(1, 3, 5, 7, 9)]),
                    pregnant_females_booked = c(data[, c(2, 4, 6, 8, 10)]))
head(data)
#>      county pregnant_females_booked
#> 1  Anderson                  0
#> 2  Andrews                   1
#> 3  Angelina                  0
#> 4  Aransas                   0
#> 5  Archer                     1
#> 6 Armstrong                  0
tail(data)
#>      county pregnant_females_booked
#> 295
#> 296
#> 297
#> 298
#> 299
#> 300
```

These results look good! We now have only two columns and the first six rows (from `head()`) look right. Why are the last six rows all empty? Look

back at the PDF. The final two columns are shorter than the others, so `extract_tables()` interprets them as empty strings `""`. We can subset those away using a conditional statement remove any row with an empty string in either column. Since we know that if there's an empty string in one of the columns it will also be there in the other, we only need to run this once.

```
data <- data[data$county != "", ]
head(data)
#>      county pregnant_females_booked
#> 1  Anderson                      0
#> 2  Andrews                        1
#> 3  Angelina                       0
#> 4  Aransas                         0
#> 5  Archer                          1
#> 6  Armstrong                      0
tail(data)
#>      county pregnant_females_booked
#> 260    Wood                           0
#> 261    Yoakum                         0
#> 262    Young                          0
#> 263    Zapata                         0
#> 264    Zavala                         0
#> 265 Zavala (P)                      0
```

Now the results from `tail()` look right. First, I'm rerunning the code to scrape the PDF since now our `data` data set is already cleaned from above.

```
data <- extract_tables(file = "data/PregnantFemaleReportingCurrent.pdf")
data <- data[[1]]
```

We'll use a toy example now with a vector of numbers from 1 to 10 `1:10` which we can call `x`.

```
x <- 1:10
x
#> [1]  1  2  3  4  5  6  7  8  9 10
```

Now say we want every value of `x` and want to use Booleans (true or false value) to get it. Since we need a vector of 10 values since we'd need one for every element in `x`. Specifically, we'd be using square bracket `[]` notation to

subset (in this case not really a true subset since we'd return all the original values) and write ten TRUEs in the square brackets [].

```
x[c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE)]
#> [1] 1 2 3 4 5 6 7 8 9 10
```

If you're reading the code carefully, you might have noticed that I only wrote nine TRUE values. Since R was expecting 10 values, when I only gave it nine, it started again from the beginning and used the first value in place of the expected tenth value. If we only wrote one TRUEs, R would just repeat that all 10 times.

```
x[TRUE]
#> [1] 1 2 3 4 5 6 7 8 9 10
```

What happens when the value isn't always TRUE? It'll recycle it the exact same way. Let's try using now a vector c(TRUE, FALSE).

```
x[c(TRUE, FALSE)]
#> [1] 1 3 5 7 9
```

It returns only the odd numbers. That's because the first value in our vector is TRUE so it returns the first value of x which is 1. The next value is FALSE so it does not return the second value of x which is 2. R then "recycles" our vector and uses the first value in our vector (TRUE) to interpret how to subset the third value of x (3). Since it's TRUE, it returns 3. But now the value for 4 is FALSE so it doesn't return it. The process repeats again until the end of the subset. Since every other value is returned, it returns only the odd numbers. We can use R's method of "recycling" a vector that is shorter than it expects to solve our pregnant females booked issue. Indeed we can use this exact c(TRUE, FALSE) vector to select only the odd columns. Reversing it to c(FALSE, TRUE) gives us only the even columns. So we'll copy over the code that made the data.frame last time and change the c(data[, c(1, 3, 5, 7, 9)] to c(data[, c(TRUE, FALSE)]) and the c(data[, c(2, 4, 6, 8, 10)]) to c(data[, c(FALSE, TRUE)]). Since the issue of empty strings is still there, we'll reuse the data <- data[data\$county != "",] we made above to fix it.

```
data <- data.frame(county = c(data[, c(TRUE, FALSE)]),
                    pregnant_females_booked = c(data[, c(FALSE, TRUE)]))
```

```

data <- data[data$county != "", ]
head(data)
#>      county pregnant_females_booked
#> 1  Anderson                      0
#> 2  Andrews                        1
#> 3  Angelina                       0
#> 4  Aransas                         0
#> 5  Archer                          1
#> 6  Armstrong                      0
tail(data)
#>      county pregnant_females_booked
#> 260    Wood                           0
#> 261    Yoakum                         0
#> 262    Young                          0
#> 263    Zapata                         0
#> 264    Zavala                         0
#> 265 Zavala (P)                      0

```

16.2 Making PDF-scraped data available to others

You've now seen two examples of scraping tables from PDFs using the `tabulizer()` package and a few more from the `pdftools` package in Chapter 15. These lessons should get you started on most PDF scraping, but every PDF is different so don't rely on the functions alone to do all of the work. You'll still likely have to spend some time cleaning up the data afterwards to make it usable. This chapter is being written in 2020 which should be long after we ever need to get data from PDFs - it should be available in much easier to access formats. However, often we still need to scrape PDFs to get data necessary for research.

Given the effort you'll spend in scraping a PDF - and the relative rarity of this skill in criminology - I recommend that you help others by making your data available to the public. There are several current websites that let you do this but I recommend [openICPSR](#). openICPSR is the version of ICPSR (Inter-university Consortium for Political and Social Research) which

is essentially a massive repository of data. openICPSR lets people submit data for free (under a certain limit, 3GB per submission as of mid-2020) and has a number of features to make it easier to store and document the data. This includes a section to describe your data in text form, fill out tags to help people search for the data, and answer (optional) questions on how the data collection and the geographic and temporal scope of the data. If you decide to update the data, it'll keep a link to your older submission so you essentially have versions of the data. When you update the data, I recommend having a section on the submission description describing the changes in each version. This is useful for record-keeping and (though be careful because the link to the submission also changes when you update it so you'll need to change that if you have links on any document such as a CV). Below are a few images showing the submission page for one of my submissions that has many versions (and corresponding version notes).

Jacob Kaplan's Concatenated Files: Uniform Crime Reporting (UCR) Program Data: Supplementary Homicide Reports, 1976-2018 [PUBLISHED] [Edit Project Header](#)

Name	Type	Size	Last Modified	Actions
shr_1976_2018_csv.zip	application/zip	21 MB	10/26/2019 5:42:47 PM	SHOW ACTIONS
shr_1976_2018_dta.zip	application/zip	40 MB	10/26/2019 5:54:57 PM	SHOW ACTIONS
shr_1976_2018_rda.zip	application/zip	16 MB	10/26/2019 5:44:44 PM	SHOW ACTIONS

Project

- Collapse All

Project Description

Principal Investigator(s) (required) [+ add value](#)
Jacob Kaplan, University of Pennsylvania. Department of Criminology [edit](#) [remove](#)

Summary (required) [edit](#)
For any questions about this data please email me at jacob@crimedatatool.com. If you use this data, please cite it.

Version 8 release notes:

- Adds 2018 data.

openicpsr-100699 (Published)

[EDIT PROJECT TO RE-PUBLISH](#)

[Share Project](#)

[Change Owner](#)

[View Log](#)

[Report a Problem](#)

[view storage](#)

Published Versions

[V8 - 10/27/2019 10:18:22 AM]	Edit Permissions
[V7 - 7/15/2019 10:11:47 PM]	Edit Permissions
[V6 - 10/10/2018 6:35:59 PM]	Edit Permissions
[V5 - 6/19/2018 2:24:37 PM]	Edit Permissions
[V4 - 6/6/2018 10:24:51 AM]	

Summary (required) 

For any questions about this data please email me at jacob@crimedatatool.com. If you use this data, please cite it.

Version 8 release notes:

- Adds 2018 data.
- Changes source of data for years 1985-2018 to be directly from the FBI. 2018 data was received via email from the FBI, 2016-2017 is from the FBI who mailed me a DVD, and 1985-2015 data is from the FBI's Crime Data Explorer site (<https://crime-data-explorer.fr.cloud.gov/downloads-and-docs>).
- Adds .csv version of the data.
- Makes minor changes to value labels for consistency and to fix grammar.

Version 7 release notes:

- Changes project name to avoid confusing this data for the ones done by NACJD.

Version 6 release notes:

- Adds 2017 data.

Version 5 release notes:

- Adds 2016 data.
- Standardizes the "group" column which categorizes cities and counties by population.
- Arrange rows in descending order by year and ascending order by ORI.

Version 4 release notes:

- Fix bug where Philadelphia Police Department had incorrect FIPS county code.

Version 3 Release Notes:

- Merges data with LEAIC data to add FIPS codes, census codes, agency type variables, and ORI9 variable.
- Change column names for relationship variables from *offender_n_relation_to_victim_1* to *victim_1_relation_to_offender_n* to better indicate that all relationship are victim 1's relationship to each offender.
- Reorder columns.

This is a single file containing all data from the Supplementary Homicide Reports from 1976 to 2018. The Supplementary Homicide Report provides detailed information about the victim, offender, and circumstances of the murder. Details include victim and offender age, sex, race, ethnicity (Hispanic/not Hispanic), the weapon used, circumstances of the incident, and the number of both offenders and victims.

Years 1976-1984 were downloaded from NACJD, while more recent years are from the FBI. All files came as ASCII+SPSS Setup files and were cleaned using R. The "cleaning" just means that column names were

— Scope of Project

Subject Terms ⓘ [+ add value](#)
Do not copy/paste multiple terms into this field. Terms must be entered individually.
 [SHR](#) [murder](#) [homicide](#) [supplementary homicide report](#)

Geographic Coverage ⓘ [+ add value](#)
United States [edit](#) [remove](#)

Time Period(s) ⓘ [+ add value](#)
1976 – 2018 [edit](#) [remove](#)

Collection Date(s) ⓘ [+ add value](#)

Universe ⓘ
Victims of homicide in the United States between 1976 and 2018. [edit](#) [remove](#)

Data Type(s) ⓘ
[administrative records data](#) [aggregate data](#) [edit](#) [remove](#)

Collection Notes ⓘ
[edit](#)

— Methodology

Response Rate ⓘ
[edit](#)

Sampling ⓘ
[edit](#)

Data Source ⓘ
United States Department of Justice. Federal Bureau of Investigation [edit](#) [remove](#)

Chapter 17

Geocoding

Several recent studies have looked at the effect of marijuana dispensaries on crime around the dispensary. For these analyses they find the coordinates of each crime in the city and see if it occurred in a certain distance from the dispensary. Many crime data sets provide the coordinates of where each occurred, however sometimes the coordinates are missing - and other data such as marijuana dispensary locations give only the address - meaning that we need a way to find the coordinates of these locations.

17.1 Geocoding a single address

In this chapter we will cover using the free geocoder from ArcGIS, a software that people frequently use when dealing primarily with mapping projects. Google Maps used to be easily usable in R but since 2018 requires an account to use its geocoder so we will not be using it.

The URL for geocoding using ArcGIS is the following:

<https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCandidate>

where instead of “ADDRESS” we put in the address whose coordinates we want. As an example, let’s look at Food Network’s Corporate office where they film many of their shows. The address is 75 9th Ave, New York, NY 10011.

<https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCandidate>

Including spaces in the address causes errors so all spaces need to be replaced with %20. Let's see what data we get back from this URL. Enter the URL above in your browser and you'll see these results.

```
JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON

▼ spatialReference:
  wkid: 4326
  latestWkid: 4326
▼ candidates:
  ▼ 0:
    address: "75 9th Ave, New York, 10011"
    ▼ location:
      x: -74.00455581294355
      y: 40.74217626395059
      score: 100
    ▼ attributes:
      Match_addr: "75 9th Ave, New York, 10011"
      Addr_type: "PointAddress"
    ▼ extent:
      xmin: -74.00566
      ymin: 40.74121700000001
      xmax: -74.00366
      ymax: 40.74321700000001
  ▼ 1:
    address: "9th Ave, New York, 10001"
    ▼ location:
      x: -73.99833276262969
      y: 40.750603066007194
      score: 96.16
    ▼ attributes:
      Match_addr: "9th Ave, New York, 10001"
      Addr_type: "StreetName"
    ▼ extent:
      xmin: -73.99933276262969
      ymin: 40.7496030660072
      xmax: -73.99733276262968
      ymax: 40.75160306600719
  ▼ 2:
    address: "9th Ave, New York, 10036"
    ▼ location:
      x: -73.99148456582823
```

It gives us a page with several important values. For our purposes we want the “lat” and “lon” sections which are the latitude and longitude parts of a location’s coordinates.

This data is stored on the page in a JSON format which is a convenient (for computers to read) way data is stored online. We can convert it to a data.frame that we’re more familiar with using the package `jsonlite`.

```
install.packages("jsonlite")
```

We will use the `fromJSON()` function and enter in the URL right in the `()`.

```
library(jsonlite)
fromJSON("https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/
#> $spatialReference
#> $spatialReference$wkid
#> [1] 4326
#>
#> $spatialReference$latestWkid
#> [1] 4326
#>
#>
#> $candidates
#>                               address location.x location.y score
#> 1 75 9th Ave, New York, 10011 -74.00466 40.74222 100
#>           attributes.Match_addr attributes.Addr_type extent.xmin extent.ymi
#> 1 75 9th Ave, New York, 10011             PointAddress -74.00566 40.7412
#>   extent.xmax extent.ymax
#> 1    -74.00366    40.74322
```

It returns a list of objects. This is a named list meaning that we can grab the part of the list we want using dollar sign notation as if it were a column in a data.frame. In this case we want the part of the object called *candidates*. To avoid having a very long line of code, let’s call the list `fromJSON()` returns *address_coordinate* and grab the *candidates* object from that list.

```
address_coordinates <- fromJSON("https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/10011?outFields=address,location.x,location.y,score")
address_coordinates$candidates
#>                               address location.x location.y score
#> 1 75 9th Ave, New York, 10011 -74.00466 40.74222 100
```

```
#>           attributes.Match_addr attributes.Addr_type extent.xmin extent.ymin
#> 1 75 9th Ave, New York, 10011             PointAddress -74.00566 40.74122
#> extent.xmax extent.ymax
#> 1 -74.00366 40.74322
```

candidates is a data.frame which includes 12 (slightly) different coordinates for our address. The first one is the one we want and if you look at the “score” column you can see it has the highest score of those 12. The ArcGIS geocoder provides a number of potential coordinates for an inputted address and ranks them in order of how confident it is that this is the address you want. Since we just want the top address - the “most confident” one - so we will just keep the first row.

Since we are grabbing the first row of a data.frame, our square bracket notation must be [row, column]. For row we put 1 since we want the first row. Since we want every column, we can leave it blank but make sure to keep the comma.

```
address_coordinates <- fromJSON("https://geocode.arcgis.com/arcgis/rest/services/World
address_coordinates <- address_coordinates$candidates
address_coordinates <- address_coordinates[1, ]
address_coordinates
#>           address location.x location.y score
#> 1 75 9th Ave, New York, 10011 -74.00466 40.74222 100
#>           attributes.Match_addr attributes.Addr_type extent.xmin extent.ymin
#> 1 75 9th Ave, New York, 10011             PointAddress -74.00566 40.74122
#> extent.xmax extent.ymax
#> 1 -74.00366 40.74322
```

This data.frame has something we've never seen before. It has columns that are themselves data.frames. For example, the column “location” is a data.frame with the x- and y-coordinates that we want. We can select this exactly as we do with any column but instead of returning a vector of values it returns a data.frame.

```
address_coordinates$location
#>           x      y
#> 1 -74.00466 40.74222
```

Since our end goal is to get the coordinates of an address, the `data.frame` in the “location” column is exactly what we want. It took a few steps but now we have code that returns the coordinates of an address.

17.2 Making a function

We want to geocode every single address from the officer-involved shooting data. As with most things where we do the same thing many times except for one minor change - here, the address being geocoded - we will make a function to help us.

Let’s start by copying the code used to geocode a single address.

```
address_coordinates <- fromJSON("https://geocode.arcgis.com/arcgis/rest/services/Geocoding/Address/GeocodeJSON/JSON")
address_coordinates <- address_coordinates$candidates
address_coordinates <- address_coordinates[1, ]
address_coordinates$location
#>           x         y
#> 1 -74.00466 40.74222
```

Now we can make the skeleton of a function without including any code. What do we want to input to the function and what do we want it to return? We want it so we input an address and it returns the coordinates of that address.

We can call the function `geocode_address`, the input `address` and the returning value `address_coordinates` just to stay consistent with the code we already wrote.

```
geocode_address <- function(address) {
  return(address_coordinates)
}
```

Now we can add the code.

```
geocode_address <- function(address) {
  address_coordinates <- fromJSON("https://geocode.arcgis.com/arcgis/rest/services/Geocoding/Address/GeocodeJSON/JSON")
  address_coordinates <- address_coordinates$candidates
  address_coordinates <- address_coordinates[1, ]
```

```

address_coordinates$location
return(address_coordinates)
}

```

Finally we need to replace the value in `fromJSON()` which is for a specific address with something that works for any address we input.

Since the URL is in the form

`https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCandid`

we can use the `paste()` function to combine the address inputted with the URL format. There is one step necessary before that, however. Since spaces cause issues in the data, we need to replace every space in the address with `%20`. We can do that using `gsub()` which is perfect for replacing characters. Let's try a simple example using `gsub()` before including it in our function. We just want to find every " " and replace it with "%20".

We will use the address for the Food Network's Corporate office as our example.

```

gsub(" ", "%20", "75 9th Ave, New York, NY 10011")
#> [1] "75%209th%20Ave,%20New%20York,%20NY%2010011"

```

It works so we can use the code to fix the address before putting it in the URL. To avoid having very long lines of code, we can break down the code into smaller pieces. We want to use `paste()` to combine the parts of the URL with the address and have that as the input in `fromJSON()`. Let's do that in two steps. First, we do the `paste()`, saving it in an object we can call `url`, and then use `url` as our input in `fromJSON()`. Since we do not want spaces in the URL, we need to set the `sep` parameter in `paste()` to "".

```

geocode_address <- function(address) {
  address <- gsub(" ", "%20", address)
  url <- paste("https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/",
              address,
              "&outFields=Match_addr,Addr_type",
              sep = ""))
  address_coordinates <- fromJSON(url)

  address_coordinates <- address_coordinates$candidates
}

```

```

address_coordinates <- address_coordinates[1, ]
address_coordinates <- address_coordinates$location

return(address_coordinates)
}

```

We can try it using the same address we did earlier, “75 9th Ave, New York, NY 10011”.

```

geocode_address("75 9th Ave, New York, NY 10011")
#>           x          y
#> 1 -74.00466 40.74222

```

It returns the same data.frame as earlier so our function works!

17.3 Geocoding San Francisco marijuana dispensary locations

We now have a function capable of returning the coordinates of every location in our marijuana dispensary data. We can write a for loop to go through every row of data and get the coordinates for that row’s location.

Let’s read in the marijuana dispensary data which is called “san_francisco_active_marijuana” and call the object *marijuana*. Note the “data/” part in front of the name of the .csv file. This is to tell R that the file we want is in the “data” folder of our working directory. Doing this is essentially a shortcut to changing the working directory directly.

```

library(readr)
marijuana <- read_csv("data/san_francisco_active_marijuana_retailers.csv")
#> Parsed with column specification:
#> cols(
#>   `License Number` = col_character(),
#>   `License Type` = col_character(),
#>   `Business Owner` = col_character(),
#>   `Business Contact Information` = col_character(),
#>   `Business Structure` = col_character(),
#>   `Premise Address` = col_character(),

```

17.3. GEOCODING SAN FRANCISCO MARIJUANA DISPENSARY LOCATIONS301

```
#>   Status = col_character(),
#>   `Issue Date` = col_character(),
#>   `Expiration Date` = col_character(),
#>   Activities = col_character(),
#>   `Adult-Use/Medicinal` = col_character()
#> )
marijuana <- as.data.frame(marijuana)
```

Let's look at the top 6 rows.

```
head(marijuana)
#>   License Number           License Type   Business Owner
#> 1 C10-0000614-LIC Cannabis - Retailer License Terry Muller
#> 2 C10-0000586-LIC Cannabis - Retailer License Jeremy Goodin
#> 3 C10-0000587-LIC Cannabis - Retailer License Justin Jarin
#> 4 C10-0000539-LIC Cannabis - Retailer License Ondyn Herschelle
#> 5 C10-0000522-LIC Cannabis - Retailer License Ryan Hudson
#> 6 C10-0000523-LIC Cannabis - Retailer License Ryan Hudson
#>
#> 1                               OUTER SUNSET HOLDINGS, LLC : Barbary Coast Sunset
#> 2                               URBAN FLOWERS : Urban Pharm : Email- hilary@urbanph
#> 3                               CCPC, INC. : The Green Door : Email- alicia@greendoorsf.
#> 4 SEVENTY SECOND STREET : Flower Power SF : Email- flowerpowersf@hotmail.com :
#> 5 HOWARD STREET PARTNERS, LLC : The Apothecarium : Email- Ryan@apothecarium.c
#> 6 DEEP THOUGHT, LLC : The Apothecarium : Email- ryan@pothecarium.c
#>   Business Structure
#> 1 Limited Liability Company
#> 2 Corporation
#> 3 Corporation
#> 4 Corporation
#> 5 Limited Liability Company
#> 6 Limited Liability Company
#>   Premise Address Status
#> 1 2165 IRVING ST san francisco, CA 94122 County: SAN FRANCISCO Active
#> 2 122 10TH ST SAN FRANCISCO, CA 941032605 County: SAN FRANCISCO Active
#> 3 843 Howard ST SAN FRANCISCO, CA 94103 County: SAN FRANCISCO Active
#> 4 70 SECOND ST SAN FRANCISCO, CA 94105 County: SAN FRANCISCO Active
#> 5 527 Howard ST San Francisco, CA 94105 County: SAN FRANCISCO Active
```

```
#> 6 2414 Lombard ST San Francisco, CA 94123 County: SAN FRANCISCO Active
#> Issue Date Expiration Date Activities Adult-Use/Medicinal BOT
#> 1 9/13/2019 9/12/2020 N/A for this license type BOT
#> 2 8/26/2019 8/25/2020 N/A for this license type BOT
#> 3 8/26/2019 8/25/2020 N/A for this license type BOT
#> 4 8/5/2019 8/4/2020 N/A for this license type BOT
#> 5 7/29/2019 7/28/2020 N/A for this license type BOT
#> 6 7/29/2019 7/28/2020 N/A for this license type BOT
```

So the column with the address is called *Premise Address*. Since it's easier to deal with columns that don't have spacing in the name, we will be using `gsub()` to remove spacing from the column names. Each address also ends with "County:" followed by that address's county, which in this case is always San Francisco. That isn't normal in an address so it may affect our geocode. We need to `gsub()` that column to remove that part of the address.

```
names(marijuana) <- gsub(" ", "_", names(marijuana))
```

Since the address issue is always " County: SAN FRANCISCO" we can just `gsub()` out that entire string.

```
marijuana$Premise_Address <- gsub(" County: SAN FRANCISCO", "", marijuana$Premise_Address)
```

Now let's make sure we did it right.

```
names(marijuana)
#> [1] "License_Number"           "License_Type"
#> [3] "Business_Owner"          "Business_Contact_Information"
#> [5] "Business_Structure"       "Premise_Address"
#> [7] "Status"                  "Issue_Date"
#> [9] "Expiration_Date"         "Activities"
#> [11] "Adult-Use/Medicinal"
head(marijuana$Premise_Address)
#> [1] "2165 IRVING ST san francisco, CA 94122"
#> [2] "122 10TH ST SAN FRANCISCO, CA 941032605"
#> [3] "843 Howard ST SAN FRANCISCO, CA 94103"
#> [4] "70 SECOND ST SAN FRANCISCO, CA 94105"
#> [5] "527 Howard ST San Francisco, CA 94105"
#> [6] "2414 Lombard ST San Francisco, CA 94123"
```

17.3. GEOCODING SAN FRANCISCO MARIJUANA DISPENSARY LOCATIONS303

We can now write a for loop to go through every row in our data and geocode that address. The function `geocode_address()` we made returns a `data.frame` with one column for the longitude and one for the latitude. To make it so we only work with the `data.frame` *marijuana* we can save the output of `geocode_address()` to a temporary file and add each of the columns it produces to a column in *marijuana*.

We need to make columns for the coordinates in *marijuana* now to be filled in during the for loop. We can call them *lon* and *lat* for the longitude and latitude values we get from the coordinates. When making a new column which you will fill through a for loop, it is a good idea to start by assigning the column NA. That way any row that you don't fill in during the loop (such as if there is no match for the address), will still be NA. NAs are easy to detect in your data for future subsetting or to ignore in a mathematical operation.

```
marijuana$lon <- NA  
marijuana$lat <- NA
```

Let's start with an example using the first row. Inputting the address from the first row gives a `data.frame` with the coordinates. Let's now save that output to an object we'll call *temp*.

```
temp <- geocode_address(marijuana$Premise_Address[1])  
temp  
#>           x          y  
#> 1 -122.4811 37.76314
```

We can use square bracket [] notation to assign the value from the *x* column of *temp* to our *lon* column in *marijuana* and do the same for the *y* and *lat* columns. Since we got the address from the first row, we need to put the coordinates in the first row so they are with the right address.

```
marijuana$lon[1] <- temp$x  
marijuana$lat[1] <- temp$y
```

And we can check the first 6 rows to make sure the first row is the only one with values in these new columns.

```
head(marijuana)  
#> License_Number          License_Type    Business_Owner
```

```

#> 1 C10-0000614-LIC Cannabis - Retailer License Terry Muller
#> 2 C10-0000586-LIC Cannabis - Retailer License Jeremy Goodin
#> 3 C10-0000587-LIC Cannabis - Retailer License Justin Jarin
#> 4 C10-0000539-LIC Cannabis - Retailer License Ondyn Herschelle
#> 5 C10-0000522-LIC Cannabis - Retailer License Ryan Hudson
#> 6 C10-0000523-LIC Cannabis - Retailer License Ryan Hudson
#>
#> 1 OUTER SUNSET HOLDINGS, LLC : Barbary Coast
#> 2 URBAN FLOWERS : Urban Pharm : Email- hilary@urbanflowers.com
#> 3 CCPC, INC. : The Green Door : Email- alicia@greenpc.com
#> 4 SEVENTY SECOND STREET : Flower Power SF : Email- flowerpowersf@hotmail.com
#> 5 HOWARD STREET PARTNERS, LLC : The Apothecarium : Email- Ryan@apothecarium.com
#> 6 DEEP THOUGHT, LLC : The Apothecarium : Email- ryan@apothecarium.com
#> Business_Structure Premise_Address_Status
#> 1 Limited Liability Company 2165 IRVING ST san francisco, CA 94122 Active
#> 2 Corporation 122 10TH ST SAN FRANCISCO, CA 941032605 Active
#> 3 Corporation 843 Howard ST SAN FRANCISCO, CA 94103 Active
#> 4 Corporation 70 SECOND ST SAN FRANCISCO, CA 94105 Active
#> 5 Limited Liability Company 527 Howard ST San Francisco, CA 94105 Active
#> 6 Limited Liability Company 2414 Lombard ST San Francisco, CA 94123 Active
#> Issue_Date Expiration_Date Activities Adult-Use/Medicinal BOT
#> 1 9/13/2019 9/12/2020 N/A for this license type BOT
#> 2 8/26/2019 8/25/2020 N/A for this license type BOT
#> 3 8/26/2019 8/25/2020 N/A for this license type BOT
#> 4 8/5/2019 8/4/2020 N/A for this license type BOT
#> 5 7/29/2019 7/28/2020 N/A for this license type BOT
#> 6 7/29/2019 7/28/2020 N/A for this license type BOT
#> lon lat
#> 1 -122.4811 37.76314
#> 2 NA NA
#> 3 NA NA
#> 4 NA NA
#> 5 NA NA
#> 6 NA NA

```

Since we are geocoding a few dozen of addresses, this may take some time.

17.3. GEOCODING SAN FRANCISCO MARIJUANA DISPENSARY LOCATIONS 305

```
for (i in 1:nrow(marijuana)) {
  temp <- geocode_address(marijuana$Premise_Address[i])
  marijuana$lon[i] <- temp$x
  marijuana$lat[i] <- temp$y
}
```

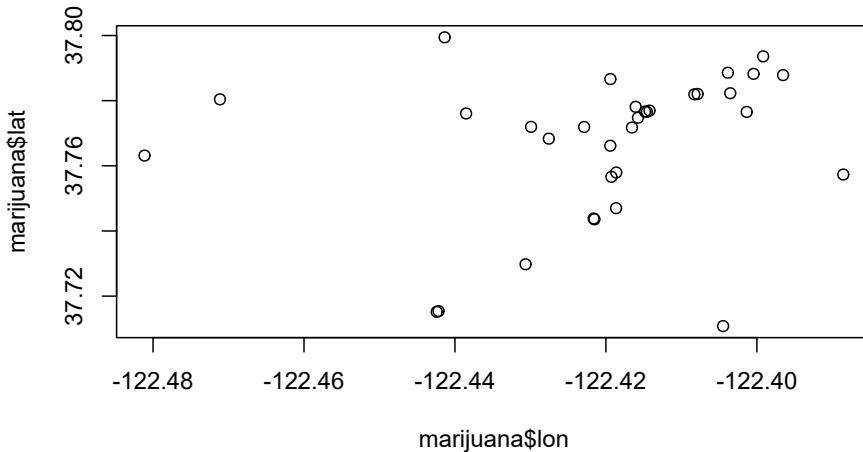
Now it appears that we have longitude and latitude for every dispensary. We should check that they all look sensible.

```
summary(marijuana$lat)
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> 37.71 37.76 37.77 37.77 37.78 37.80

summary(marijuana$lon)
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> -122.5 -122.4 -122.4 -122.4 -122.4 -122.4
```

Another check is to make a simple scatterplot of the data. Since all the data is from San Francisco, they should be relatively close to each other. If there are dots far from the rest, that is probably a geocoding issue.

```
plot(marijuana$lon, marijuana$lat)
```



worked properly.

To finish this lesson we want to save the *marijuana* data.frame. We'll use the `write_csv()` function from the `readr` package to save it as a .csv file. Since this data is now geocoded and it is specifically for San Francisco, we'll save it as "san_francisco_marijuana_geocoded.csv".

```
setwd(here::here("data"))
write_csv(marijuana, path = "san_francisco_marijuana_geocoded.csv")
```

Part IV

Project Management

Chapter 18

Mise en place

If you’re familiar with cooking you might have heard the phrase *mise en place* which is French for “everything in its place”. In cooking this concept means that you get everything - ingredients, pots, pans, bowls, utensils, etc. - needed to cook that item ready before you begin cooking. This saves time as you have everything you need in front of you and can just cook from start to finish without stopping to find something. This is also a useful idea in programming, especially when you’re programming to conduct research.

In this section of the book, we’ll cover how to get *mise en place* for your programming projects. First, we’ll go over what is, in my opinion, the best way to organize your folders, data, and code. This method is particularly suited for research projects, so please feel free to modify my methods to suit your own needs and preferences. We’ll also cover some ways to start working on the project by hand, before writing any code. In the next few chapters we’ll cover collaborating with other people (including yourself in the future who, in all likelihood, will forget a lot of the code you wrote), testing your code, and using the version control software Git.

18.1 Starting with a pencil and paper

This may seem counter-intuitive, but the best way to start any programming project - and in particular, research project - is to use a pencil and paper. On this paper you should outline every step (broadly speaking, not literally every

line of code) that you'll take for the project. This is a useful process at the start of a project to step back from the code and think about the overarching goal of the project - and what you need to do to get there. For example, let's think about doing research using the US Border Patrol data that we scraped in Chapter 15. We want to see if a policy change affected apprehensions at the border. On the data side, that'd require scraping and cleaning the PDFs. On the analysis side, we'd probably want to do a time-series graph showing apprehensions over time, and run a regression to see if the policy had a significant effect. So here we have four broad categories of work (scraping, cleaning, graphing, running a regression) for a fairly simple policy evaluation. Within each category you can make a number of subcategories. For example, in scraping you might want to add download the PDFs, see how each table relates to each other, figure out which parts of the tables are actually relevant, etc. We can probably break down these subcategories even further if we want.

You essentially want to build a roadmap to follow - you can, of course, deviate from this roadmap if necessary - as you work on the project. This is useful for two reasons. First, writing out what you need to do will often clarify exactly what you need to do. Knowing that you'll want a time-series graph, for example, will mean that you need to have your data aggregated into a certain time unit. Knowing this before-hand will save you time as you'll have a tangible goal to work towards and don't have to keep stopping during your work to figure out what to do next.

And second, from my experience helping people at Penn with R, people - especially new programmers (and myself when I was first starting learning R, my first programming language) - can get overwhelmed with programming. One major problem they had is they couldn't articulate what they needed to do since they weren't familiar enough with R to know the right words.¹ They knew the end goal, and what they had at the start, but couldn't articulate the path from start to finish. Writing out each step in plain language allowed them to know the path - it is simpler to know what steps you need to do to complete a project in plain language than to actually write the code (though

¹Knowing the "right words" is surprisingly important when it comes to programming because it is crucial to finding help online. If you don't know how to subset, you can easily find on Google as to how to subset in R. But if you don't know the word subset or how to describe what subsetting does well, it can be very tricky to find help (you won't even know what to Google!). These "right words" are, annoyingly, an important part of programming that I believe isn't given enough focus.

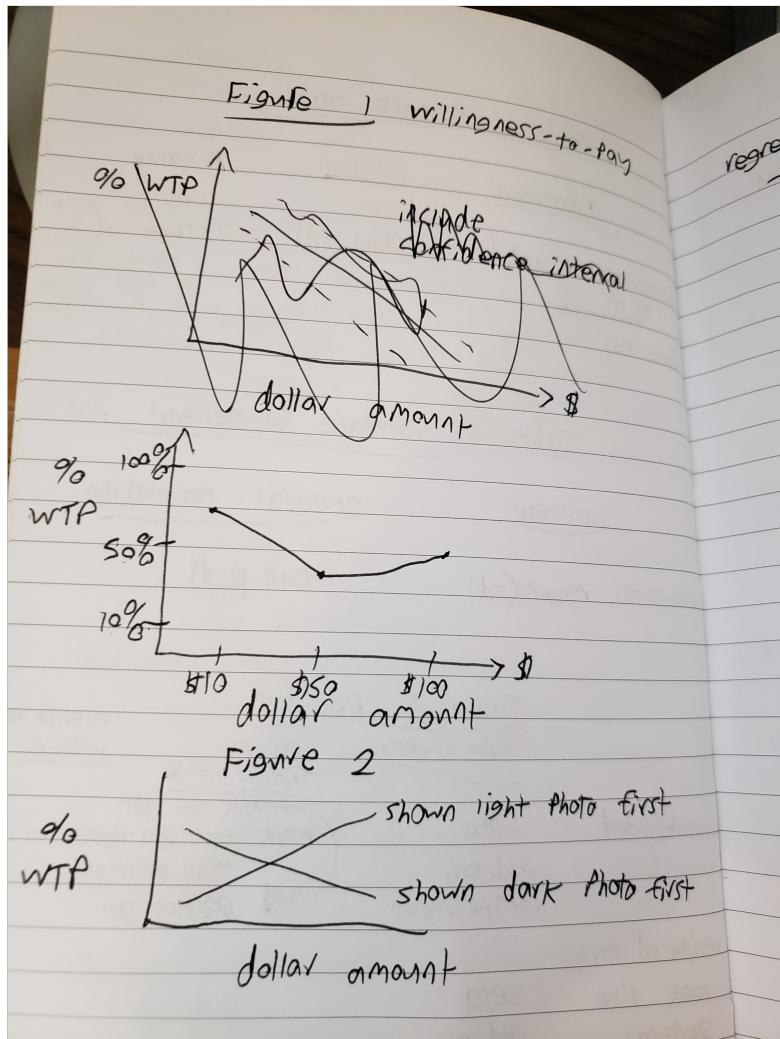
this still requires experience to tell you a lot of the “minor” intermediate steps). Having a game plan helps people avoid being overwhelmed since they could do one step at a time (and feel accomplished at each step).

18.1.1 Tables and graphs

One of the biggest challenges I had early in my PhD was figuring out what data was supposed to look like. I mean that literally. My first research project was analyzing if monthly crime in school buildings changed after a new policy was instituted that increased building security. The data I had available was incident-level so one row for every crime at the school, and I needed to convert it to the building-month level. The code for this is just to run `aggregate()` on the incident-level data and aggregate the data to building-month units. For some reason I just couldn’t think of the proper way for my data to appear in the final data set which prevented me from figuring out what I needed to do. One solution to this - and useful even if you don’t have this problem - is to draw out the graphs and tables you want before starting the code. Like writing out the steps for the code, drawing the graph will help you understand exactly how your data needs to look - and thus what code you need to write - for these graphs.

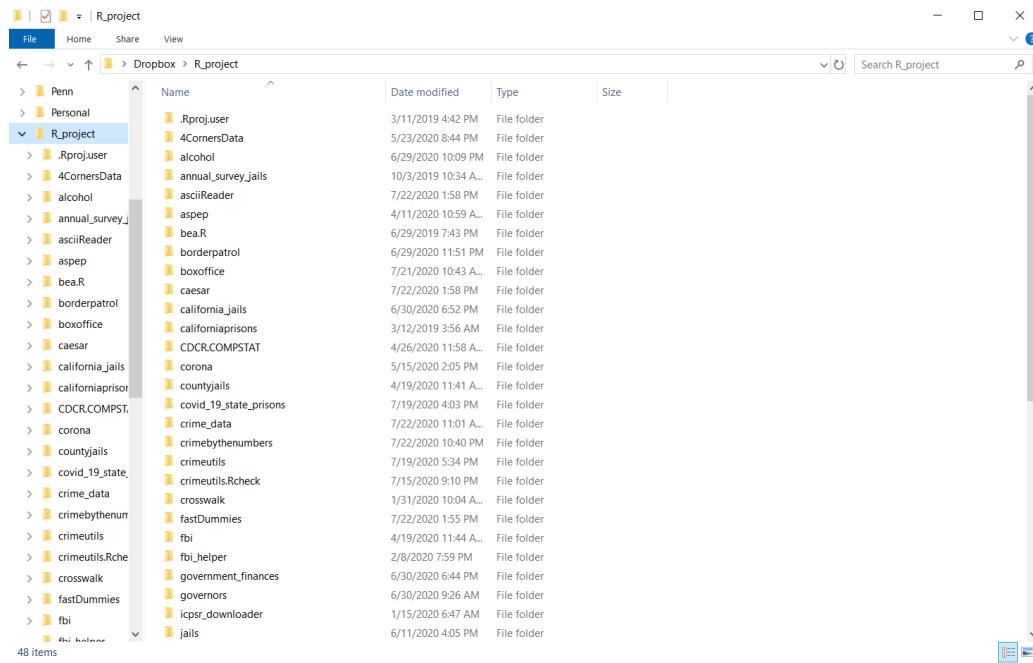
Below are two images from a recent project of mine with the tables and graphs that I wanted sketched out. Note that in the image showing my graphs I have crossed out the first graph. These sketches are just preliminary tools to help your work, you aren’t chained to them. Like any tool, if it is no longer relevant or useful, find something new. For regression result tables especially, sketching these out help you think about what variables you will need to have to run the regression. For example, you may want to have control variables for demographics in your geographic unit (say, for the US Census). If we continue our example of using the US Border Patrol data, this means that you’ll also need to grab, clean, and merge Census data to your other datasets. Sketching out the resulting tables and graphs is a good tool to figure out steps that you’ll need to do for the project but may have not thought of.

		Treatment	Control	P-value
2007	Outcomes - % of each answer	mean (sd)	mean (sd)	T-test
days	covariates			
etc.				
<u>Table 2: Compare to general pop</u>				
<u>Sample</u>		<u>general population</u>		
90	covariates	mean (sd)	mean (sd)	
<u>Table 3: Results</u>				
<u>Feel safety</u>		<u>nights per week</u>		
unweighted	Beta	go to friends house		
	std. err.	walk or not		
	control mean	(walk (exclude those who don't go) and go/not go)		
weighted to look like general pop				
	beta			
	std. err.			
	control mean			



18.2 R Projects

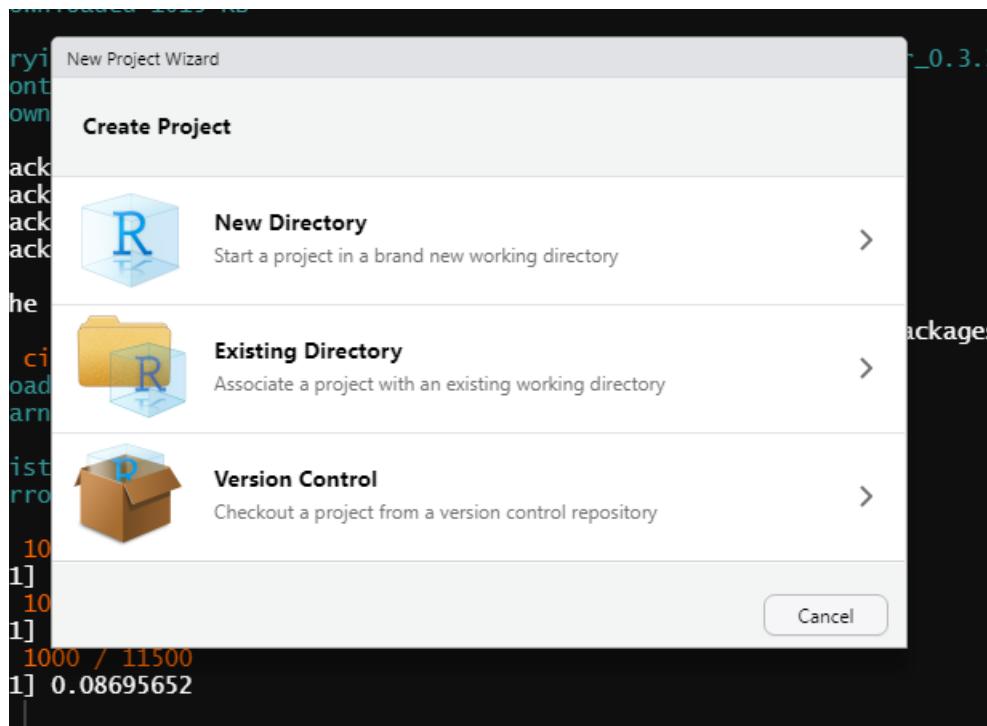
We've talked about projects in an abstract sense - that they are research papers or specific data exploration jobs. RStudio provides, a bit confusingly, something called an R Project which is merely a helpful way to organize folders for a specific project (paper, data exploration, etc.) that you do. When you do a project, I recommend keeping *everything* for that project in a single folder on your computer. Below is an image showing all of the folders I use for my various R work. As you can see from the file names, each folder is for a separate project, and there is not overlap between them - each project is independent. Within each folder is a structured way to organize the folders and files that I believe is one of the best ways to organize data for research projects.



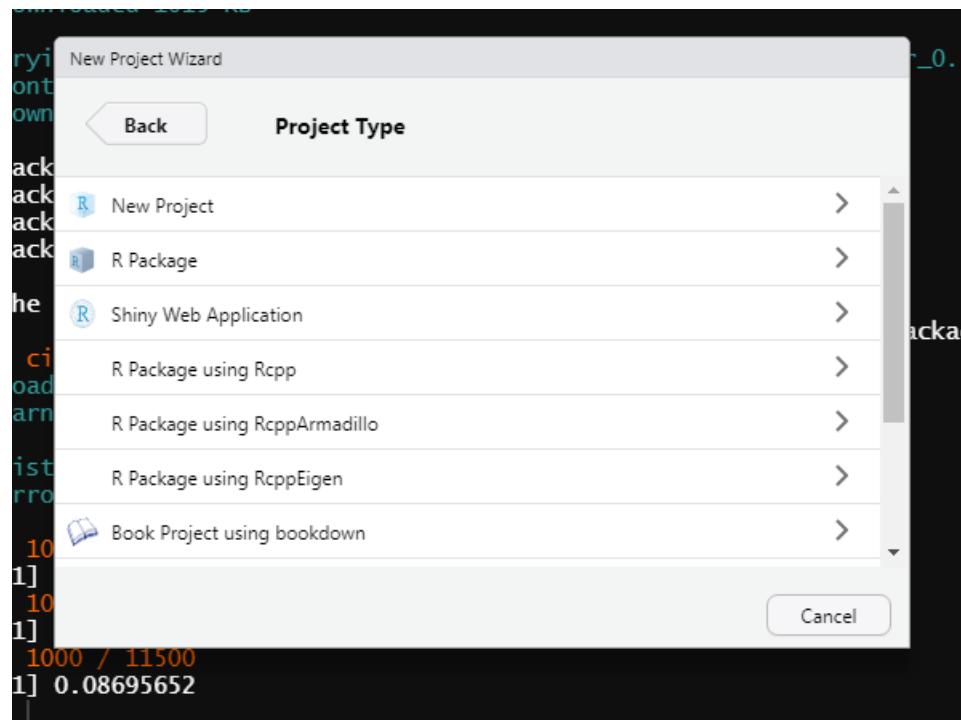
First, we'll explain how to set up an R Project through R Studio, and why you would want to do it. There are two main reasons to want to use an R Project. First, throughout this book I had you set your working directory so that R knew where to look for a particular file. In R Projects, by default the working directory is in that project's folder. So if you had a file `example.csv` in your project folder, you wouldn't need to set a working directory since R

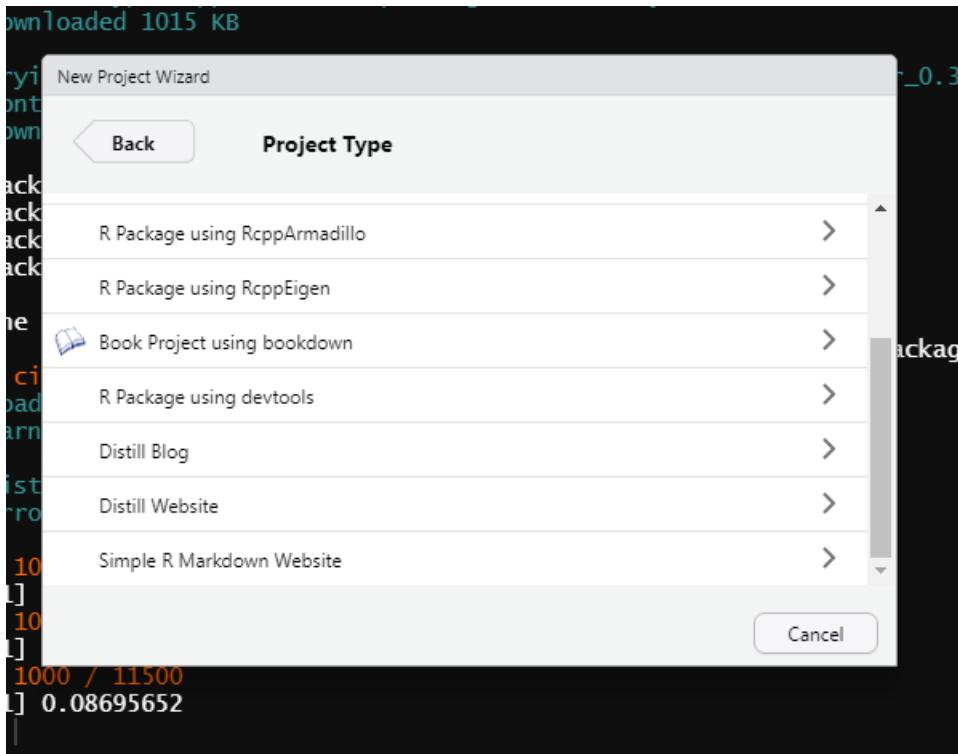
would already be looking that. This may be a minor time-saving method if you're working alone since you'd only need to set the working directory once when not using an R Project. But consider if you're collaborating with three people and you've shared your code. When using an R Project, it just runs. There is no figuring out what the issue is - and determining what the issue is will take them longer than setting the working directory itself. Second, it provides easy access to using the version control software Git, which we'll talk about in detail in Chapter 21.

To make an R Project, start by clicking the *File* button on the top left corner of RStudio and then click *New Project*. This will open up a window that has three options: New Directory, Existing Directory, and Version Control. New Directory says that the project we are making is going to be in a brand new folder that we're (R will do this automatically) going to create. This is the one you'll click on in the majority of cases. Existing Directory is for making a folder in an existing folder, which doesn't have too many useful cases. The Version Control is taking a project that someone else has created and downloading it to your computer. We'll cover this more in Chapter 21.

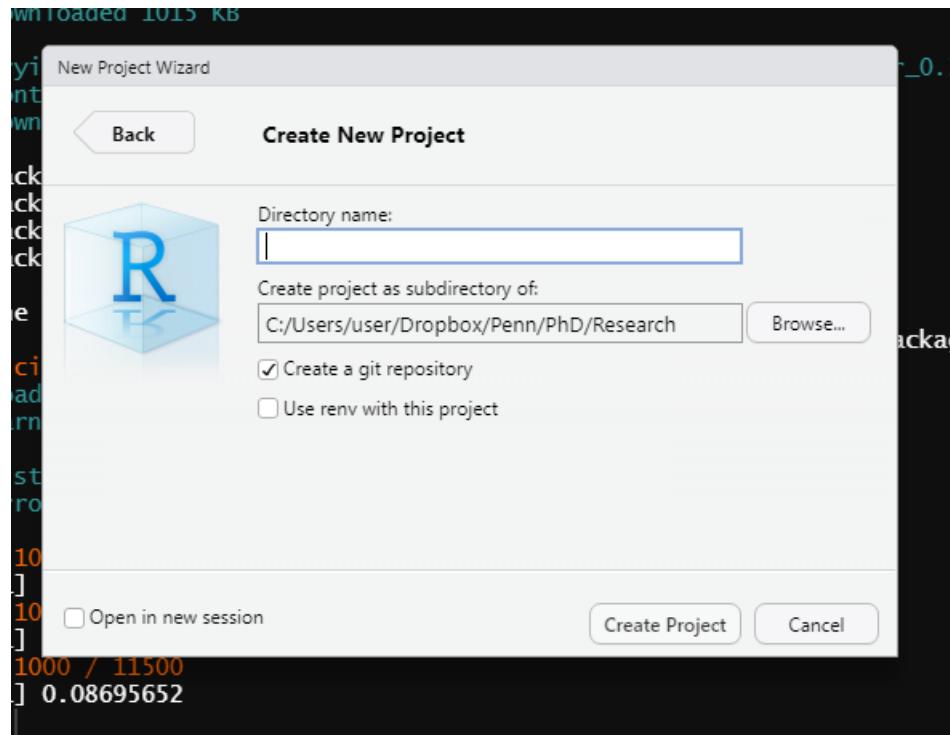


Once you've clicked New Directory, it'll change the window to ask you what type of project you want. The below two figures show all the different types of projects R can make (installing some R packages such as `bookdown` can add more types of projects to this list). R is very versatile and has project types ranging from the standard R Project to books and websites. We just want a standard project so click the New Project button at the top.

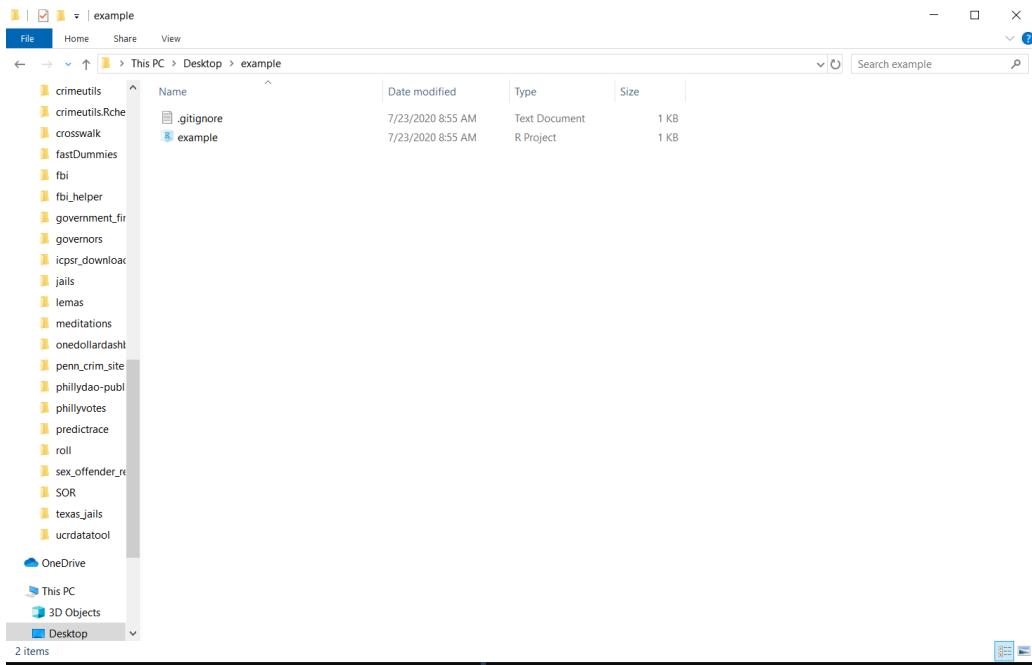




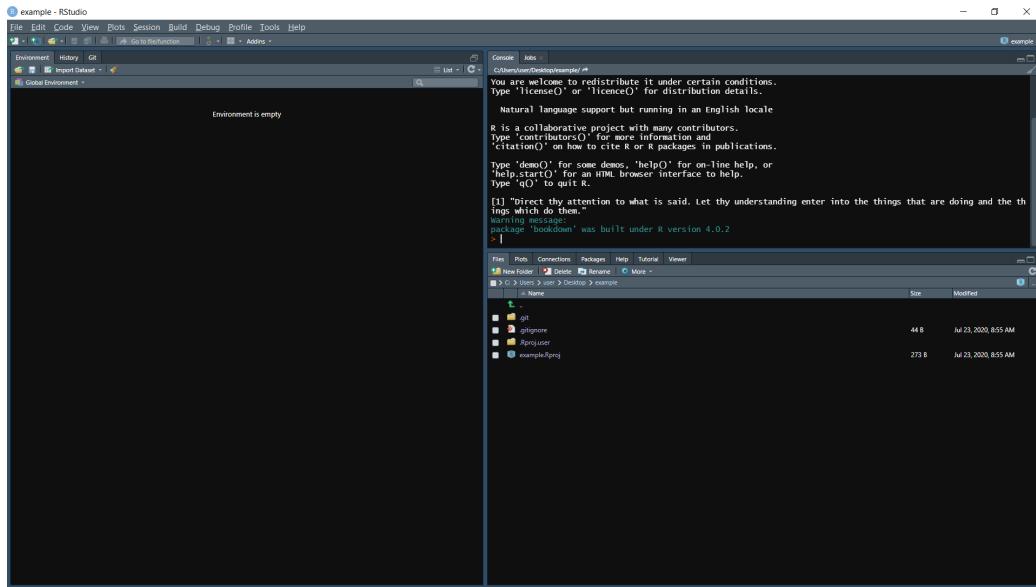
Now it'll have a window that says Create New Project up top. In the Directory name: section you write the name of your R Project. Keep this short and (though not required) follow normal R naming conventions such as all lowercase letters and underscores separating words. This will be the name of your folder so you want it descriptive enough to understand (and for collaborators to understand) what it is for, without being overly long. Once you have a name you can click the Browse... button on the right and go to the folder on your computer where you want to put this folder (ideally, you'll put it in a folder which is backed up by something like Dropbox). Make sure the *Create a git repository* checkbox is selected, and we'll explain why in Chapter 21. Click Create Project and R will make the project folder on your computer and open that project in RStudio.



Below are images of a brand new R Project that I made called *example* that I put in my Desktop folder. The folder is now empty except for two files - `.gitignore` (which we won't talk about here) and `example` which is type "R Project" (and the full name would be `example.Rproj`). This is a **very** important file. Note that its name is the same as the R Project name that I made, and the same as the folder name on my computer. This file is essentially a shortcut that you click to open that R Project. It doesn't do anything more than open the R Project but this is the way you'll access the project every time you want to use it. Double-click this and the Project will open.

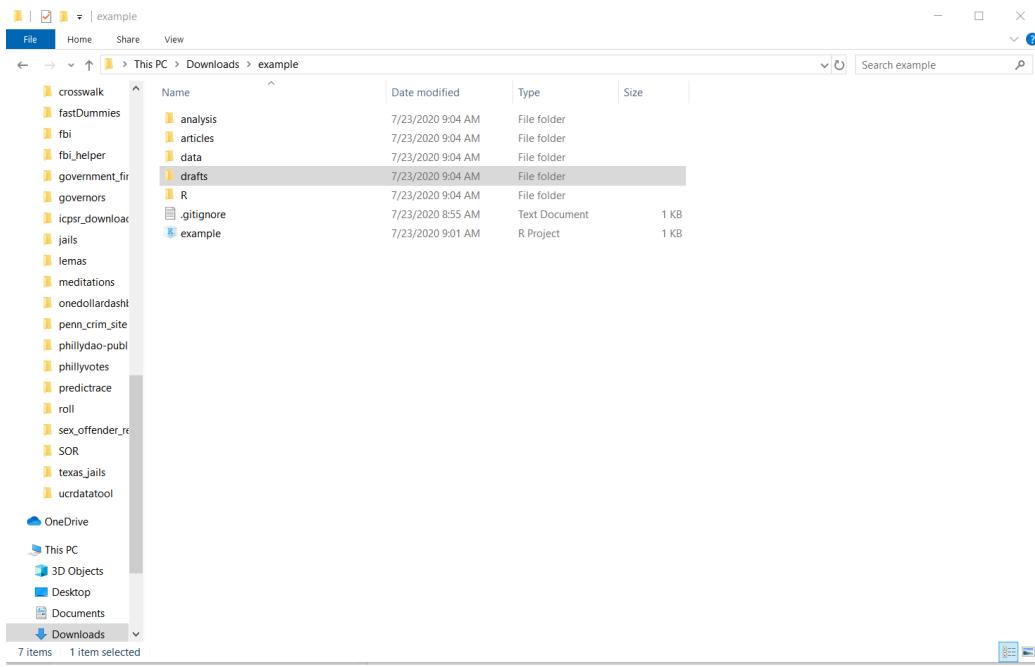


This RStudio session looks nearly identical to other sessions that we've used - and it nearly is identical. A few key differences can be found in the top left corner where it says "example - RStudio", indicating that we're in the *example* R Project. And then directly below the "Console" tab it says "C:/Users/user/Desktop/example/". This is the working directory of this project. I didn't set it, R just knew where it was. If you move this folder to a new folder (say, the Downloads folder) or if someone else downloads it to their computer, R will automatically change the working directory to the right one. You no longer have to worry about it.



18.2.1 Folders

Now that we have the R Project made, we need to start adding some R code and data files to the project so we can get started working. But first, let's talk about proper ways to organize the folder. I've added a few new folders to the new *example* R Project as the basic layout of my work process. This is for a research-oriented project so may not apply in your particular case. Organizing your folders (and as we'll see below, your code) is important so please play around with different ways to organize and find a way that works well for you.



I've added five folders to the R Project folder: analysis, articles, data, drafts, and R (note that I moved it to the Downloads folder, and if I opened the project RStudio would know where the new working directory was). I tend to do my analysis using Stata (primarily because most of my co-authors use Stata instead of R so this is a way we can both work on the analysis) so in this folder I'd keep all of the .do (Stata) files to run the regressions. In articles, I put PDFs of every article I read that I use (or planned to use while reading it) for the paper I'm working on in this project. It's good to keep this organized to share with co-authors or just for easy reference after you've read it. It certainly takes time to find good sources for a lot of research, so you don't want to have to search again because you've forgotten which article you had a particular reference from or that was important to your study. While I recommend writing your papers in R Markdown (see Chapter 11), you will need to create drafts of the paper to send to others (e.g. your collaborators or journals). The "drafts" folder is a good place to keep these versions - some journals require that you submit a Word Document with track-changes for a revise and resubmit so you will need to leave R Markdown occasionally to comply with these rules.

The final two important folders are "R" and "data". In the R folder - as

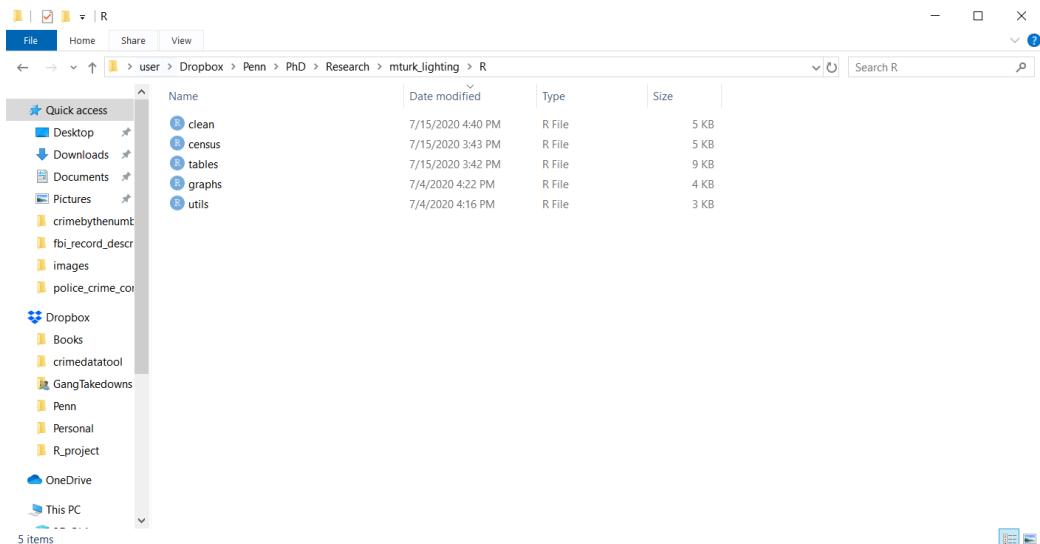
you may have guessed - belong the various R scripts that you write during the project. In Section 18.3, we'll talk in detail as to how to organize these scripts. Inside the “data” folder I made two subfolders: “raw” and “clean”. The raw folder is where you’ll store the data exactly as you got it (for cases where the data is acquired through webscraping, this isn’t necessary). This folder will have, for example, the PDFs that you intend to scrape, and the .csv files downloaded from a different source. It is important to keep this data always unchanged (change only in your R code and save the output to a new file) so you can replicate your results from the original data. In the clean data is that final data output from your work to clean and manipulate (e.g. subset, aggregate) the data. It isn’t strictly necessary to even output a final dataset - you could just rerun your code from the original data each time, and this is fine if your code is very quick to run - but I think it is important both for safekeeping and to be able to share with others. If you collaborate with people, you’ll want to be able to send them the data so they can examine it without having to run all of your code themselves.

This PC > Downloads > example > data				
	Name	Date modified	Type	Size
	clean	7/23/2020 9:04 AM	File folder	
	raw	7/23/2020 9:04 AM	File folder	

18.3 Modular R scripts

If you are like many people who start programming, all of your code will be in a single R script. This is fine when you’re first getting familiar with R and don’t want to go searching for code in places when you’re still uncomfortable with the language. As you become more familiar with R - and as your projects get more complex - you’ll want to start making multiple R scripts in a single project.

When you’re writing a paper you don’t just write one extremely long sentence. You break up ideas into paragraphs and divide groups of paragraphs into larger sections. This is useful in a paper to organize your thoughts and to make it readable for others. It’s also useful when working since you know, for example, “Section 1 is done but I still need to finish Section 2 and the last of Section 3.” This way you don’t confront working on the entire paper at once. You’ll want to follow these lessons in the code you write, with each “section” of code being its own R Script and within a script split up code into particular “paragraphs”. The end goal should be to have modular R scripts, with each script being independent (or relatively so) and the combination of these parts has all the code for your particular project. This is a bit of an abstract concept so let’s use a real example from one of my recent projects.



Above is a folder for the code used to analyze data for a paper examining perceptions of outdoor lighting. There are five R scripts in the folder - clean.R, census.R, tables.R, graphs.R, and utils.R - and these are the only ones used for this project.². Each of these files (utils.R is an exception) has a particular role to play in the analysis of the data. The first file, clean.R is just code that cleans up the survey data and makes it ready to be analyzed and graphed. The census.R file has code that cleans Census data that my co-author and I use to compare our survey sample to the general public. As this is a separate dataset than the survey data, I have it in its own R script. tables.R and

²Analysis was done in Stata so there are separate files for that

graphs.R are the code to make descriptive statistics tables and figures for the paper, respectively. While you could combine the code in these files into a single R script - and the code in each is not very long (shorter than the code in almost every chapter in this book) - it makes it harder to handle (much like reading an article with no space between the lines). I chose these files because they are doing fairly separate tasks, all with the goal of turning raw data into a research paper.

This is an example of how I approach making R scripts, not necessarily the best way to do so. Even here, other decisions could be made. For example, I could have put the code from census.R into clean.R since they're both about cleaning data (and maybe rename clean.R to something like clean_survey.R). While you should try to make separate R scripts for broadly different tasks (regardless of how much code that task requires), you should experiment with how you prefer to separate these scripts, and balance between having one (or a few) super scripts that comprise everything with having too many scripts that do too little - this balance requires experience and experimentation so keep at it!

18.4 Modular code

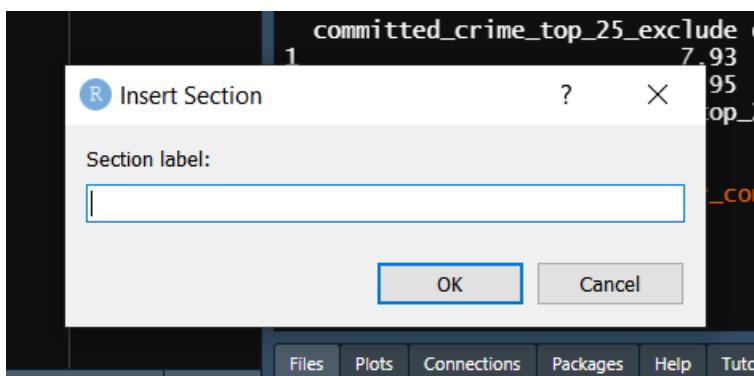
In addition to having separate scripts for each major part of your project, you will want to organize each individual script into relatively modular parts. Whereas each script is like a book chapter, the code inside the script should be like paragraphs, separated into distinct chunks (similar to the chunks in an RMarkdown file). For example, let's say you have some raw data and want to subset it, change some values (e.g. renaming F to Female, M to Male), and then aggregate to a larger geographic area. This is a three-step process - subset, change, aggregate - so you'll want to have three different parts of your R Script dedicated to this. Now, if this is a simple process (and it will always depend on the data and what you want to do with it), you may want to have each step in its own Section (as we'll discuss next). If it's relatively simple and takes only a few lines per step, you'll likely just want to have a line break between steps and identify your choices in comments. It's hard to give precise rules on how to do this as it really does depend on personal preference - I think having more comments and line breaks early in your experience is better as it's a good foundation, and you can alter it to

suit your preferences as you get more experience.

The goal of making modular code is to avoid having a large amount of code without breaks or comments - that'd be like reading a run-on sentence. We'll talk about comments more in Section 19.2.1, but here you should explain your choices (e.g. "Subset to only violent crime and property crime") to inform collaborators (other people and yourself in the future who will likely forget what or why you did something), but without writing too much. Generally the rule of thumb is to have comments for why you did something, not explaining what as the what can be deduced from looking at the code. I think a mix of what and why is helpful as it's quicker than looking at the code, especially if your code is complex. Like a lot of your work, however, this depends on the project and your audience - if you're working with someone new to R, having more comments explaining what you did is helpful.

18.4.1 Section Labels

When you have major parts of a script, you should have something to indicate that this is a distinct section from other parts. RStudio has a handy tool to help make that distinction by creating Sections in your R Script. Press the keys Control+Shift+R (Command+Shift+R in a Mac) and it will open up a window where you can set a section label (note that this will only work in an R Script, not an RMarkdown file). Enter in the name of the section you want and click OK and it'll add that to where your cursor was in the R Script. You can also do this by simply adding four dashes (—) on the end of a comment



Sections are more than just commented parts of a Script. Note that in the

photo below, there is both the Section label in the R Script and that same label in a new section of the Source tab on the right. You can get to this section by clicking on the button on the very top right, the one that looks like a bunch of misaligned lines. In here, it shows all the Sections that exist and clicking the Section name will move to the start of that Section in your R Script. If you have a long script (which is generally unadvised but sometimes can't be helped), this is an easy way to find a particular part of your code.

```
1
2 # Section Label Example -----
3
4 |
5
```

18.4.2 Helper R Scripts

As part of making code organized, I find it helpful to make two R scripts in each (or most) of my projects to hold helper functions or objects - and I call these `utils.R` and `utils_objects.R` (`utils` stands for utility as these are helpful pieces of code for the project). These files should be for code that will be used in multiple R Scripts, so you want them in a single place rather than copying them over in each script where you need them.

In `utils.R`, I keep functions that are either auxiliary (such as code to check data by printing out a set of outputs) or code that is used infrequently (such as loading several files and merging them together at the start of an R Script) where I don't want them in the main file. In `utils_objects`, I put useful objects

such as a vector (`c()`) of values that I will use to subset. For example, if I wanted to subset all violent crimes from a dataset, I would need to know what crimes in that data are considered violent, put them as strings in a vector, and subset to only rows that match those strings. In `utils_objects.R` I could make an object with this vector, such as `violent_crimes <- c("murder", "rape", "robbery", "assault")`.

If you want to run `utils.R` or `utils_objects.R` (or any .R file) in a different R script, you can use the `source()` function which makes R run the entire script inputted in the parentheses. Just put the file name (in quotes) in the parentheses and it will run. For example, if we want to run `utils.R`, we'd write `source("utils.R")`. If that file was in the “data” folder of our R Project, we'd write `source("data/utils.R")` so R knew to look in the “data” folder for the file.

Chapter 19

Collaboration

When you are using R for work, or even for personal research, your work often requires collaborating with others. These collaborators include your research partners, co-workers, yourself in the future, and even people who never code but would like to understand the work that you are doing (such as a supervisor who doesn't code themselves). Collaboration is immensely helpful as it can cut down the work load and make it easier to solve problems by thinking through problems together (and your collaborators may know tricks or shortcuts that you don't). However, it also adds a level of complexity to your work as it requires you to write code that other people can quickly and easily understand (so you don't waste their time). This chapter discusses some of the best practices for working with collaborators (again, yourself in the future is a collaborator!). This is a topic widely discussed in programming books so if you would like more information, please read a book dedicated to this topic.

19.1 Code review

When you collaborate with other people, you will probably each be working on a separate (though related) part of the project and then combine each parts when you are done. Combining your code could be through emailing each other R Scripts - and having one person combine everything - or something more formalized such as using Git, which we discuss in Chapter 21. However you decide to do this, it is important to use a process to review

collaborator's code to check for mistakes. This is a similar process to having a colleague read a paper draft before submitting it.

Code review is a useful technique for reducing the number of mistakes as it is a check on the work before using the code for real. Code review generally involves having one person who writes the code send it to another person who checks the code for any potential mistakes or issues. This check involves ensuring that the code meets the specified style (this is discussed further in Section 19.1.1) and that there are no bugs. For the person having their code reviewed, having comments explaining the what and why of the code (discussed more in Section 19.2.1) will help the reviewer quickly go through the code. The code should also be relatively short, comprising of a specific R Script (or related scripts) and no more than a few hundred lines of code. This is because as code gets more interrelated and complex, it is harder for someone unfamiliar with the code to understand it and see any issues. That means that a reviewer for long code is more likely to miss issues and take longer to review. Reviewing shorter code, even if that means reviewing more often, is often far more efficient for both the reviewer and reviewee and catches more issues.

In cases where you have unit tests written for the code, these tests are an automated form of code review as they too check for mistakes. To save peoples time, you should avoid sending the code for review until it passes all unit tests. However, if you're stuck and can't get certain tests to pass, working with someone else to solve the problem is often faster than doing so yourself because then you have an outside perspective who may see something you lost.

For code review to be most efficient, I recommend developing some rules with your collaborators to specify how and when code review is done. For example, you should determine who reviews certain people's code (ideally with senior people reviewing junior people's code) and how often it is done. I believe that doing code reviews relatively frequently (i.e. after a working draft of some code is ready) is useful as you can catch issues early and not waste anyone's (especially the personal writing the code) time. However, having hard time limits is probably ill-advised as sometimes writing certain code takes far longer than expected and reviewing an unfinished (and potentially far from finished) bunch of code is not efficient for anyone.

When someone is reading a draft of your research paper, they are generally

looking for whether it is correct (i.e. your methods are right, the lit review is thorough, etc.) and how well it flows. Code review is the same. While the primary goal is finding errors, an important aspect is to ensure that it is readable (i.e. proper spacing, how names are written) and consistent across everyone's code in that group. More formally, ensuring that everyone's code is readable and consistent is having people follow a style guideline.

19.1.1 Style guidelines

An important part of reviewing people's code is ensuring that everyone is following the same style guidelines when it comes to writing code. Style guidelines are the grammar rules of writing code. They dictate (or encourage) certain style choices such as whether object names are lowercase, whether they include punctuation, and even when to put long code on a new line. This is equivalent to making sure that people writing in plain language put punctuation and capitalization in the expected place. While you can read !SomETHiNg WrITen. LiKE thIs, it is easier to understand when you it follows adopted and accepted rules.

The important thing here is to be consistent. Consistency makes code much easier to read and helps make code written by multiple people more interchangeable. This book follows the [tidyverse style guide](#) which is one that many R programmers follow, but the exact style you chose is relatively unimportant (choosing more common styles helps when your code may be used by people out of your organization). Feel free to adopt an already made style guide, make any modifications to suit your preferences, or to create an entirely new one yourself. As long as people follow the same format, you'll be able to spend more time on the code, and less time trying to understand it.

19.2 Documentation

An important, though occasionally tedious, part of writing code is documenting your work. We'll talk about documentation in two ways, through comments which focus on specific parts of code, and vignettes which document the project more broadly.

19.2.1 Comments

All the way back in Section 1.1 we introduced comments, which are essentially notes about the code that you include in an R script (by starting a line with the pound key `#`) that isn't run. They are just "comments" to yourself or anyone else reading the code to explain what that code does and why it is there. As is often repeated in explaining the benefit of comments, the main collaborator you will have is yourself in the future.¹ You don't need to comment every single code - and doing so would just make it hard to read - but should comment on important things or chunks of code (i.e. several lines of code that all are for the same purpose). If you write a function, you'll want at least a brief code explaining what it does and what the inputs and parameters do.

Writing comments is not as fun as writing code. Stopping to write a comment on something that seems obvious at the time (after-all, you figured out how to do something you wanted to do and likely were focusing on) interrupts the flow of writing code and slows down your work. And when you have looming deadlines and multiple projects that you're working on, spending the time writing good comments may seem like a bad use of time as the payoff is only in the future. However, the benefits far outweigh the cost. This is true for two reasons. First, when you're collaborating with others, it is much quicker to have text explaining the code than to walk through the code with them (or to have them try to figure it out themselves).² As you work with more people, comments become increasingly important. Writing good comments is also time-efficient when considering that in many cases when you do research you will have to return to a project in the future.

This is best shown when considering a research project that leads to a journal article. For many papers, even if you are fantastically productive and can work nonstop at it without forgetting any decisions, at a certain point you'll need to finish and submit it to a journal. Journal reviews can often take 3-6 months so at that point you'll likely have forgotten many of the (seemingly)

¹I recently worked on a follow-up paper to one I had done a year ago. For some reason, past me decided to name some functions based on the authors of a paper that created that particular method, and didn't leave comments explaining what the code did or why. Past me caused a lot of problems for current me. Please comment your code!

²This is one of the main reasons I wrote this book. After a few years of helping Penn students with the same questions, I decided to write out guides to those topics.

obvious decisions you made in the course of the project.³ Having comments explaining why you made a certain decision (such as including or excluding certain crime types from your analysis) can be a huge time saver when addressing reviewer concerns - you will know why each decision was made and won't have to try to figure out the *why*. This is particularly important when you have to defend a decision in which there is no obvious choice and you want to know your thought process at the time you wrote the code and were immersed in the issues of the data. A lot of data decisions are reasonable at the time based on the quirks of the data but can appear to make no sense if you aren't familiar with the data - comments can remind you of the quirkiness and how you handled it.

19.2.2 Vignettes

Vignettes are essentially a document that explains how to do something with the code you have written. This is common when someone has written an R package and they want to explain in detail important functions from the package. You can think of chapters of this book as vignettes covering particular topics - PDF scraping, webscraping, regular expressions, etc. To make a vignette, you can simply make an R Markdown file (for more information on R Markdown please see Chapter 11) detailing that topic. Since the text you write is included in the document, these files are basically normal R Scripts with extensive comments written in plain language. Often, these comments are more formal than what you'd write in an R Script as they are written as complete sentences or paragraphs and walk through comprehensive ideas rather than focus on discrete chunks of code.

One increasingly prominent method of using R for research is to do everything in an R Markdown file. This allows you to explain your approach - including context on why you did something - and each step you took in plain language in the text of the R Markdown file while still including the code directly in the file - and you can still include comments on that code in the code chunks. Whether you include the code in the output, or just the result of the code, depends on your audience and how far along you are in the project.

If this is for a presentation to update collaborators, for example, it is useful to

³If you're like me and on your 7th rejection for a particular paper, 3-6 months may be optimistic.

include the code as they may notice an issue or give advice based on the code. Including code can also teach your audience something new (I've certainly learned a lot by watching people present using code I wasn't familiar with). If the document is for an audience unfamiliar with R (or programming more generally), or where time to present is limited, you probably won't want to include code.

Whether you do work in an R Script or in an R Markdown file is up to you. If you intend to write up a report anyway, having everything written up in the R Markdown file as you write your code can save you time as you're merging the code and the writing process. However, this loses some nice features in R such as unit tests, which we discussed in detail in Chapter 20. It also depends on how complex your project is. If you have code that is hundreds of lines long and spans multiple R Scripts, putting it all into a single R Markdown file is unfeasible. In this case it'd be better to run the code in the R Script and use the R Markdown file just to present results.

Chapter 20

Tests

20.1 Why test your code?

As you write code, you will inevitably make mistakes. There are two main types of mistakes with coding - those that prevent code from working (i.e. give you an error message and don't run the code) and those that run the code but give you the wrong result. Of these, the first is probably more frustrating as R tends to give fairly unhelpful error messages and you'll feel you hit a roadblock since R just isn't working right. However, the second issue - code is wrong but doesn't tell you it's wrong! - is far more dangerous. This is especially true for research projects.

Let's use examining whether a policy affected murder as an example. In the example data set below, we have two years of data for both murder and theft, and we'll say that the policy changed at the start of the second year. If we want to see if murder changed from 2000 to 2001, we could (overly simply) see if the number of murders in 2001 was different from the number in 2000. And since the data also has theft, we'd want to subset to murder first.

```
example_data <- data.frame(year = c(2000, 2000, 2001, 2001),
                           crime_type = c("murder", "theft", "murder", "theft"),
                           crime_count = c(100, 100, 200, 50))
example_data
#>   year crime_type crime_count
#> 1 2000     murder        100
```

```
#> 2 2000      theft      100
#> 3 2001      murder     200
#> 4 2001      theft      50
```

To see if murder changed, we can subset to the rows where the crime is murder, and then print out the year and crime_count columns to see if there is a change. So our code will be `example_data[example_data$crime_type == "murder", c("year", "crime_count")]`. Below I've accidentally only put one `=` instead of two, this will give us an error and not give any other results. Helpfully, the error message tells us that there's an error with the `=` sign, though not what that exact error is.

```
example_data[example_data$crime_type = "murder", c("year", "crime_count")]
#> Error: <text>:1:38: unexpected '='
#> 1: example_data[example_data$crime_type =
#> ^
```

Now I've made a different mistake. Here, instead of `==`, I've written `!=` which is the opposite of what we want - it'll return all rows that do **not** equal "murder". Now it looks like the policy cut murder in half when in actuality the policy doubled murders! Since we don't print out the type of crime in the output, we wouldn't catch this from the output alone.

```
example_data[example_data$crime_type != "murder", c("year", "crime_count")]
#>   year crime_count
#> 2 2000      100
#> 4 2001      50
```

You may think this is a silly example that is unrealistic. And it is to a degree, it's just one line of code that we're using to evaluate an entire policy. Now think about how you would actually evaluate a policy using data that you're familiar with. Now the code is going to be much more complex. Your code may be hundreds of lines long, deal with multiple data sets that must be joined together, and involve a number of relative subjective (though must be defensible) decisions as to how to deal with your data (e.g. what crimes constitute violent crime, what time unit to analyze), and be written by other people you are collaborating with. The increased complexity with a real analysis increases the likelihood that errors will occur - and even small issues such as an incorrect subset can have large impacts on your results.

So, how do we properly test our code? There are two main methods that I'll refer to as informally and formally. The formal method will be using something called "unit tests" that we're discuss in great detail in the next part of this chapter.

Informal methods are what you've likely been doing already. Essentially, just looking at your data and trying to see if it "looks right". This includes stuff like printing summary statistics (using `summary()`) of important variables and making simple graphs to look at the data. If something is wrong, exploring the data is a fairly good way to discover it. For example, if you are looking at arson data from the FBI, you may find (as this is actually in the data) some cities with billions of car arsons in a month. This is clearly wrong so you know there's an issue - in this case, an issue with not subsetting out obvious outliers. Knowledge about the topic and the data are also important in this approach. If you are familiar with a given topic and your results are similar to that of past studies, that's a good sign that you did things right.¹

You can also take this kind of approach when testing functions - which ideally are the way you write code, so you can use it for more than a single case. For example, if you have a function that takes a number and returns that number + 2, you can test it by checking a few cases. If you input 2, you expect 4. If you input -2, you expect 0. Do this a few times and you can be more confident that the function works properly. Now imagine a function that's more complex - one that calls a different function and uses the result of that function. If you change the underlying function, you'll need to check both that function and the function that calls it. As you have more intertwined pieces in your code, this gets more and more complex. It also takes a lot more time as you'll have a lot of code that only checks a function and will have to run it line by line to see if there's an issue. At this point, relying on informal methods becomes unfeasible and you'll want to use unit tests, a formal way to test your code. Note, however, that this is far better suited for checking functions than for checking data, though it is possible to some degree. We'll discuss formally testing data in Section 20.2.3.1.

¹However, make sure that you don't look less closely just because the results are the way you expect. Past results may be wrong, or you can have a new finding, so make sure to avoid complacency just because you like the results

20.2 Unit tests

A unit test is simply a conditional statement where you have some input, usually a function with some parameters set, and state what you expect the result to be. You are saying “I expect that if I do X, I will get Y”. And if you get a result other than what you expected, R will tell you. In R, you can make a number of unit tests and have R run them all at once and inform you of which ones failed. Each unit test is just a function in R that is specifically for checking whether other functions - or other code or data - are correct. They operate just like a normal function. To use unit tests, we’ll use the R package `testthat` which has a number of functions that make unit testing easier and use some keyboard shortcuts in RStudio that also improve the ease of testing. Please note that these shortcuts will only work if you’re working in an R Package, a normal R Project won’t work. If you don’t have `testthat` installed, do so using `install.packages("testthat")`. For more information on the package, please see the package’s [website](#).

```
library(testthat)
```

In `testthat`, every function follows the same `expect_` format where a type of conditional statement follows the `_`. For example, `expect_equal()` checks if two values are equal, `expect_named()` checks if the name of a data set is correct, and `expect_silent()` makes sure that the code that’s run doesn’t return any warnings, messages, or errors. To use this technique for our above example of the function that adds 2 to an inputted number - which we’ll call `add_2()` - we can use some `expect_equal()` functions. If we input 2, we expect 4. So we’d write `expect_that(add_2(2), 4)`.

```
add_2 <- function(number) { return(number + 2) }
expect_equal(add_2(2), 4)
```

Above is the code that makes the `add_2()` function and one unit test checking it. It doesn’t output anything. That is good. When a test passes, there is no information; when it fails, the function will output a message that it failed. Below is another test, this one intentionally wrong to show what happens when a test fails.

```
expect_equal(add_2(2), 5)
#> Error: add_2(2) not equal to 5.
#> 1/1 mismatches
```

```
#> [1] 4 - 5 == -1
```

It gives an error, telling us that the result of the `add_2(2)` function does not equal 5. Helpfully, it also shows us how much of a difference there is between what we expected and what we got. Note that it says “1/1 mismatches”. That says that all of the expected values - we only expect one value - are incorrect. If we expect more than one result, such as if we expect a function to return a vector, it will check each value and say exactly which ones (in the order we have the resulting vector) are incorrect. This is helpful when diagnosing exactly which part failed.

There are a few different ways to run the unit tests. First, you can run them like a normal R Script by running each line directly or using `source()`. This is fairly inefficient and loses some of the benefits built into RStudio for testing.

The next way is to use the `test_dir()` function from the `testthat` package where you enter the folder directory in the parentheses and it runs every test file in that folder. It’s easier to simply use the `test_path()` function which inserts the correct folder, assuming that you didn’t move folders around after `use_test()` created them. So you’d write `test_dir(test_path())` and it would run all of your tests. This also prints out a nice summary of the results for all of your tests combined, showing the number of tests that passed, that failed (i.e. didn’t pass), that returned warnings, and that were skipped (you can force R to skip some tests which is useful when you change one part of the code and know those tests will fail but still want to test other parts). Note that for this method to work you will first have to run `library(testthat)` so R knows how to run the tests. You will also need to run all the functions or load all of the data that the tests check, otherwise you’ll get an error since R doesn’t implicitly know that the functions/data exist. You can run this the normal way by highlighting and running the function (or the code to load data) in your R Project or use the shortcut Control+Shift+L (Command+Shift+L on a Mac, the L stands for “Load”) which will run every R file in your project (and every line of each file).

The final way is to use the keyboard shortcut Control+Shift+T (Command+Shift+T on a Mac, the T stands for “Test”) which will load all of the files and then run all of the tests. So a quicker way of doing the above method. However, this shortcut only works when using an R Package, not

a normal R Project.

```
> test_dir(test_path())
✓ | OK F W S | Context
✗ | 3 1 | test

test-test.R:5: failure: multiplication works
20 * 2 not equal to 41.
1/1 mismatches
[1] 40 - 41 == -1

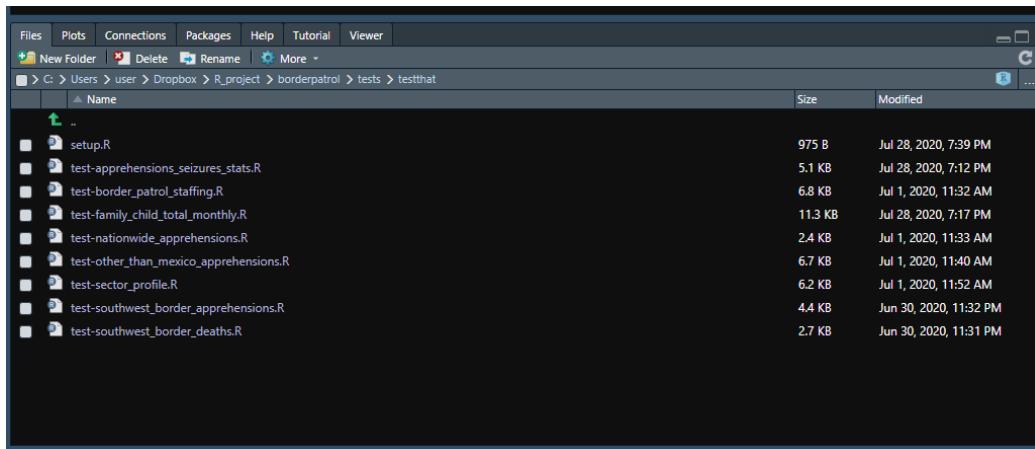
== Results ==
OK: 3
Failed: 1
Warnings: 0
Skipped: 0

I believe in you!
>
```

20.2.1 Modular test scripts

Before getting into exactly how to write a unit test correctly, we'll talk about organizing each testing file. As with your normal R Script, you can have separate testing scripts (a testing script is only a normal R Script which people use specifically for testing code) for each major part of the code that you're testing.² As with the R Scripts for your code, this is simply a way to organize your work, and doesn't affect the testing. Below is an image showing the files I use to test the US Border Patrol scrapers. I have one file per PDF that I scraped.

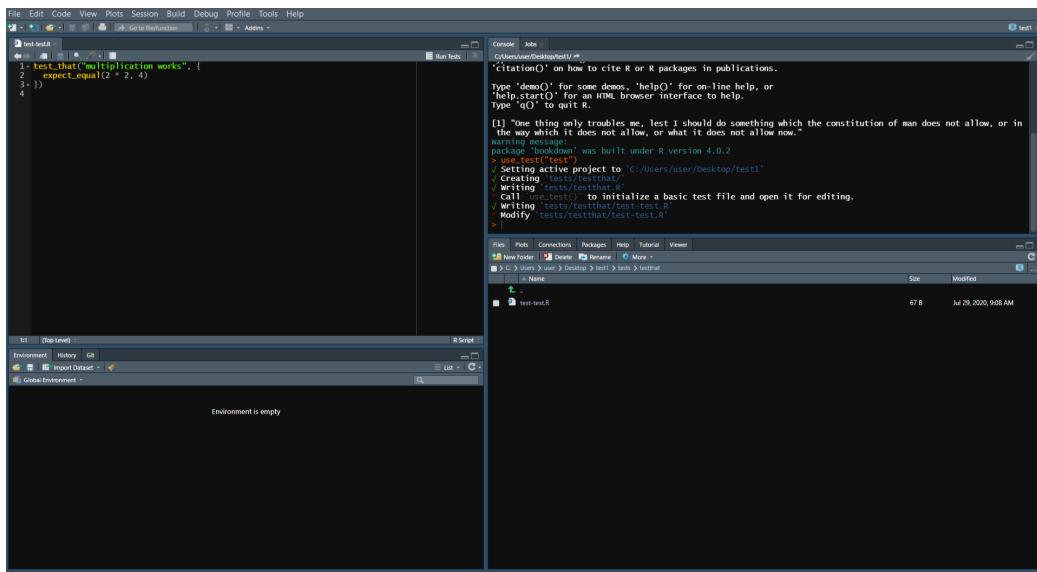
²For more info on having separate R Scripts for each major section of your code, please refer to Section 18.3



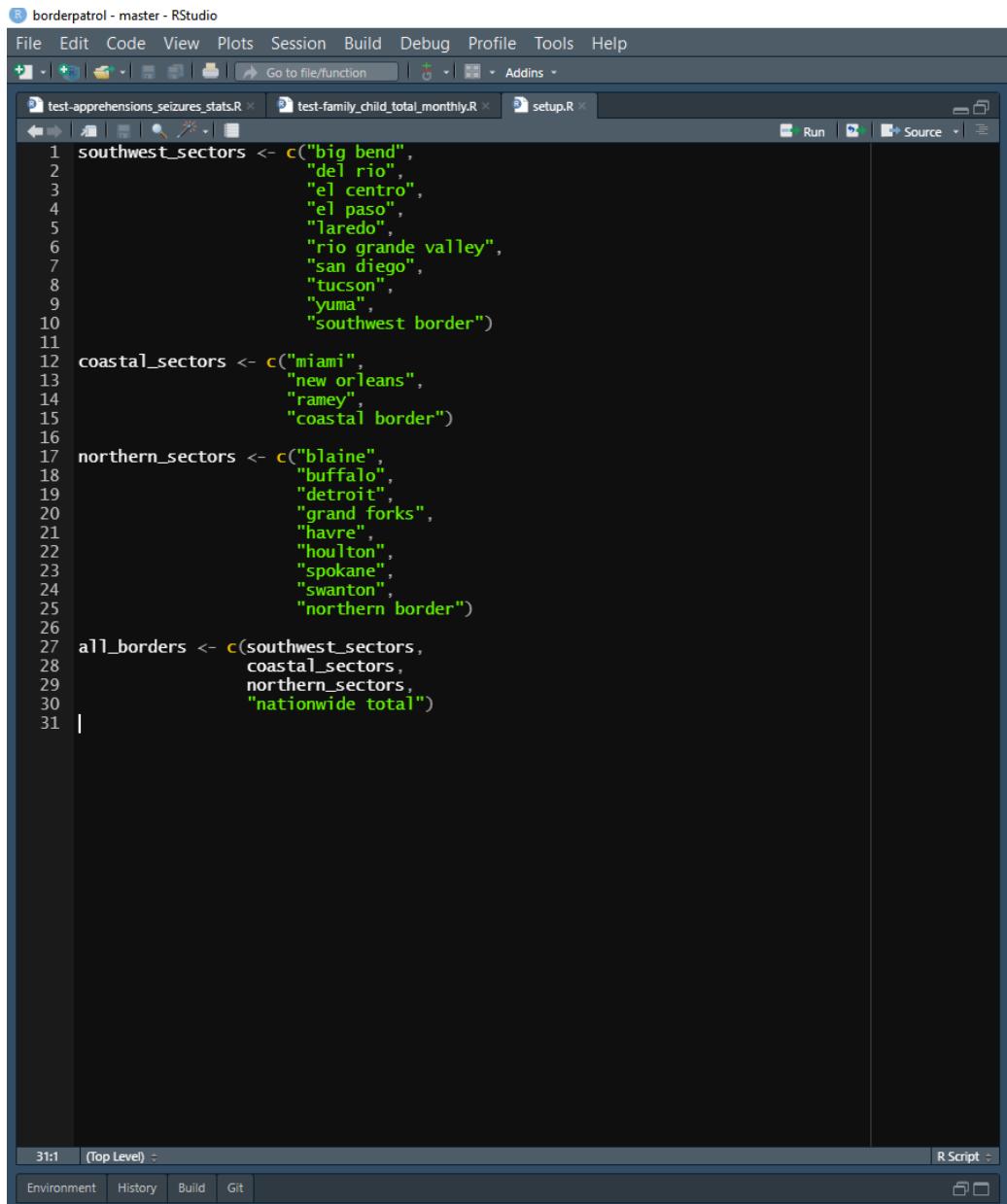
Note where the folder depicted above is located. It’s in a folder called “test-that” in the “tests” folder in the main project folder that I called “border-patrol”. We’ll use a helpful function from the `usethis` package to organize our test files and generate them automatically. If you haven’t installed this package already, do so using `install.packages("usethis")` and then load it with `library(usethis)`.³

You can use the function `use_test()` from the `usethis` package to create a test file inside your R Project. This will automatically create the necessary file and folders (if not created already) so you don’t have to do any more work. Run this function by putting the name of the test file you want to create (in quotes) in the parentheses. It will open the test file in the Source panel (shown in the top left). In the example shown below, I wrote `use_test("test")` to make a new file called “test”. In the Source panel, the file is called “test-test.R” which is just because `usethis` will automatically add “test-” to the name of any test file name you make. `use_test()` will also generate an example of tests, which you modify (or delete entirely) to suit your own needs.

³The `usethis` package is an extremely helpful package that automates a lot of work the you would do primarily for R package development so if you go down that route I recommend exploring the package more through its [website](#).



The first file in the testthat folder is called setup.R which is a file that will automatically run first when you run a test script through R or using RStudio's keyboard shortcut. This file is where you run some code that is used during the tests. In my setup.R file I made several vectors which I use during the tests to subset the data. You won't always need to have a setup.R file, but it's useful when you want to run the same code beforehand for multiple different test scripts.



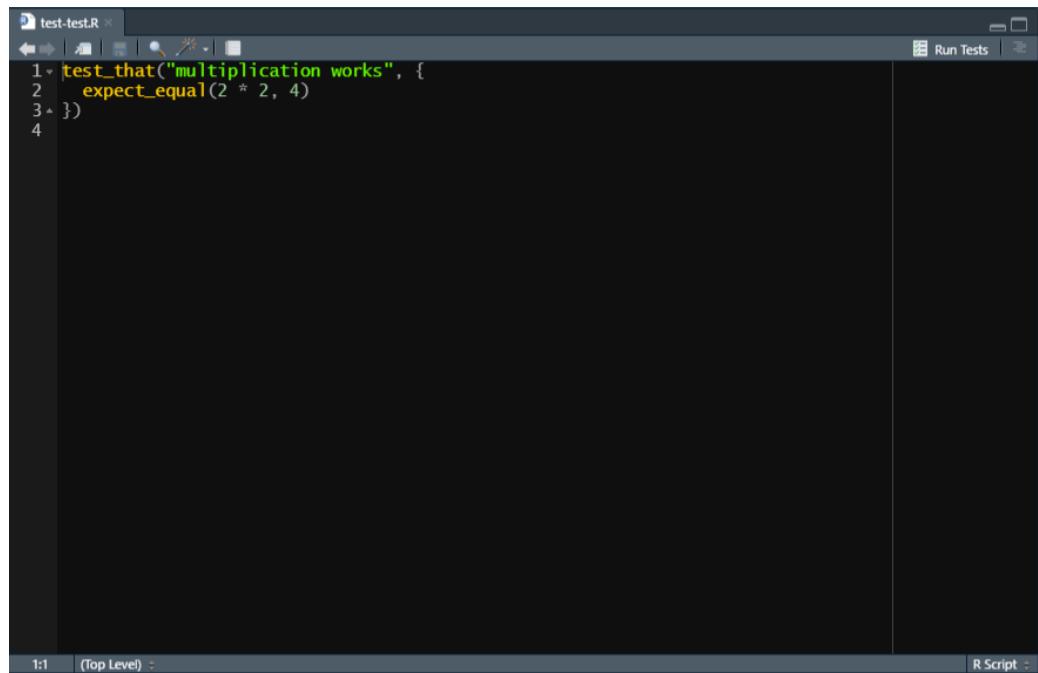
The screenshot shows an RStudio interface with a dark theme. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with various icons. The main workspace contains three tabs: "test-apprehensions_seizures_stats.R", "test-family_child_total_monthly.R", and "setup.R". The "test-apprehensions_seizures_stats.R" tab is active and displays the following R code:

```
1 southwest_sectors <- c("big bend",
2                         "del rio",
3                         "el centro",
4                         "el paso",
5                         "laredo",
6                         "rio grande valley",
7                         "san diego",
8                         "tucson",
9                         "yuma",
10                        "southwest border")
11
12 coastal_sectors <- c("miami",
13                        "new orleans",
14                        "ramey",
15                        "coastal border")
16
17 northern_sectors <- c("blaine",
18                         "buffalo",
19                         "detroit",
20                         "grand forks",
21                         "havre",
22                         "houlton",
23                         "spokane",
24                         "swanton",
25                         "northern border")
26
27 all_borders <- c(southwest_sectors,
28                     coastal_sectors,
29                     northern_sectors,
30                     "nationwide total")
31 |
```

20.2.2 How to write unit tests

We'll start by looking at the default test question example made when using `use_test()` to understand the organization of a test file before getting into

an example of actual tests. In the image below, there are really two pieces. First, we have the actual test on line 2 - `expect_equal(2 * 2, 4)`. This is saying, I expect $2 * 2$ to equal 4, and R will check if that is true. All of your tests will be in this format, just for a specific result from a specific input. Now let's look at the code surrounding that line - `test_that("multiplication works", {})` where the `expect_equal()` line goes inside the {}.

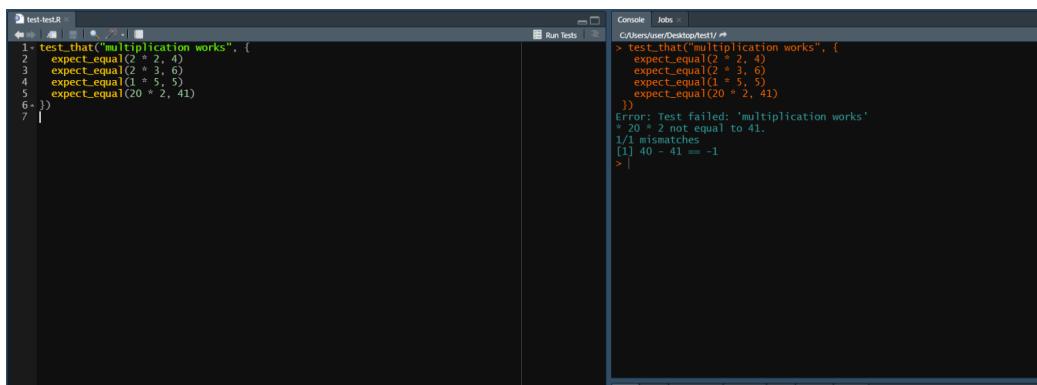


```
test-test.R
1: test_that("multiplication works", {
2:   expect_equal(2 * 2, 4)
3: })
4:
```

The `test_that` code is basically a form of organization within a test file to group similar tests together. In this case it is grouping all of the tests that check if “multiplication works”, though we only have one test written. Below I’ve added three new tests to this “multiplication works” testing group. To run this code, I can either run each `expect_equal()` individually (remember to run `library(testthat)` beforehand or it won’t run) or run the entire `test_that()` group at once (you can do this like running a for loop by either highlighting it all and running or selecting either the top or bottom line [which has the squiggly brackets] and running that line - the entire thing will run).

The benefit of this is that when you run all the tests you write (and you’ll often have many test groups and more individual tests than shown here), if a

test in a group fails, it will tell you exactly which group failed (based on the name of the group which you specify - here, “multiplication works”). Note that the final test in this example is incorrect, and in the Console panel on the right it says that “Test failed: ‘multiplication works’” to tell you where the test failed. The test groups aren’t necessary but they make it easier to organize your tests.



The screenshot shows the RStudio interface with two panes. The left pane contains an R script named 'test-test.R' with the following code:

```

1- test_that("multiplication works", {
2   expect_equal(2 * 2, 4)
3   expect_equal(2 * 6, 12)
4   expect_equal(2 * 5, 10)
5   expect_equal(20 * 2, 41)
6- })
7

```

The right pane is the 'Console' tab, showing the execution of the script and the resulting error message:

```

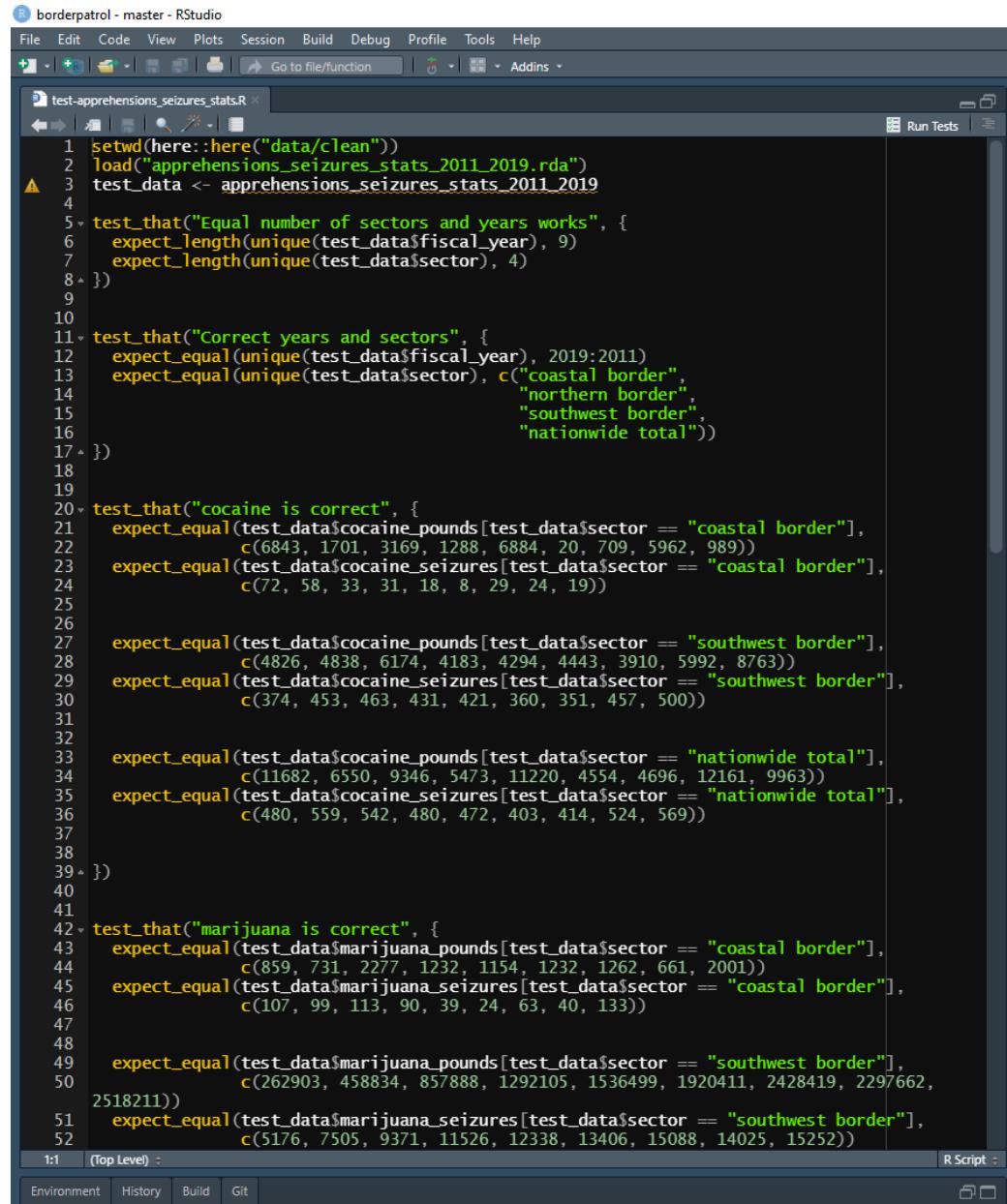
> test_that("multiplication works", {
+   expect_equal(2 * 2, 4)
+   expect_equal(2 * 6, 12)
+   expect_equal(2 * 5, 10)
+   expect_equal(20 * 2, 41)
+ })
Error: Test failed: 'multiplication works'
# 20 * 2 not equal to 41.
1/1 mismatches
[1] 40 - 41 == -1
>

```

As an example of actual tests, we’ll go over the tests that I wrote when I first scraped the US Border Patrol data that we scraped in Chapter 15. This test file is organized almost identically to the example one shown above. At the start I have some code that loads the data that I will test - this isn’t in the setup file since the code is for this specific test script (though it could be in the setup.R file and the results would be the same). While most tests check the result of functions, here I am checking the data that is outputted by the function, and not rerunning the function for each test. I do this because the function is relatively slow to run and I have many tests, but putting the function that gets the data in the test directly will give the exact same results. Then there are several `test_that()` groups with some `expect_equal()` tests inside each.

Since these tests are checking if the code is scraping the PDFs correctly, I determine the expected result by looking at the PDFs and writing down what the values should be (be careful, this must be done by hand but that can mean you mistype - so double-check your work!). We’ll use the test on lines 21-22 as an example. Here I am asking if the values in the “cocaine_pounds” column, for rows where the sector is “coastal border” are equal to the values `c(6843, 1701, 3169, 1288, 6884, 20, 709, 5962, 989)`. If they are, then the scraping was correct (at least for this part of the PDF) and the code worked.

In this case I checked every value that meets the two conditions, but that's just because there were relatively few values. If I had many values that meet those conditions (i.e. many rows of data in that column), I would just check a small number of them.



```

R borderpatrol - master - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
+ - Go to file/function Addins
Run Tests
test-apprehensions_seizures_stats.R
1 setwd(here::here("data/clean"))
2 load("apprehensions_seizures_stats_2011_2019.rda")
3 test_data <- apprehensions_seizures_stats_2011_2019
4
5 test_that("Equal number of sectors and years works", {
6   expect_length(unique(test_data$fiscal_year), 9)
7   expect_length(unique(test_data$sector), 4)
8 })
9
10 test_that("Correct years and sectors", {
11   expect_equal(unique(test_data$fiscal_year), 2019:2011)
12   expect_equal(unique(test_data$sector), c("coastal border",
13     "northern border",
14     "southwest border",
15     "nationwide total"))
16 })
17
18
19 test_that("cocaine is correct", {
20   expect_equal(test_data$cocaine_pounds[test_data$sector == "coastal border"],
21     c(6843, 1701, 3169, 1288, 6884, 20, 709, 5962, 989))
22   expect_equal(test_data$cocaine_seizures[test_data$sector == "coastal border"],
23     c(72, 58, 33, 31, 18, 8, 29, 24, 19))
24
25
26   expect_equal(test_data$cocaine_pounds[test_data$sector == "southwest border"],
27     c(4826, 4838, 6174, 4183, 4294, 4443, 3910, 5992, 8763))
28   expect_equal(test_data$cocaine_seizures[test_data$sector == "southwest border"],
29     c(374, 453, 463, 431, 421, 360, 351, 457, 500))
30
31
32   expect_equal(test_data$cocaine_pounds[test_data$sector == "nationwide total"],
33     c(11682, 6550, 9346, 5473, 11220, 4554, 4696, 12161, 9963))
34   expect_equal(test_data$cocaine_seizures[test_data$sector == "nationwide total"],
35     c(480, 559, 542, 480, 472, 403, 414, 524, 569))
36
37
38 })
39
40
41 test_that("marijuana is correct", {
42   expect_equal(test_data$marijuana_pounds[test_data$sector == "coastal border"],
43     c(859, 731, 2277, 1232, 1154, 1232, 1262, 661, 2001))
44   expect_equal(test_data$marijuana_seizures[test_data$sector == "coastal border"],
45     c(107, 99, 113, 90, 39, 24, 63, 40, 133))
46
47
48   expect_equal(test_data$marijuana_pounds[test_data$sector == "southwest border"],
49     c(262903, 458834, 857888, 1292105, 1536499, 1920411, 2428419, 2297662,
50     2518211))
51   expect_equal(test_data$marijuana_seizures[test_data$sector == "southwest border"],
52     c(5176, 7505, 9371, 11526, 12338, 13406, 15088, 14025, 15252))
53
54 })
55
56 })
57
58 })
59
60 })
61
62 })
63
64 })
65
66 })
67
68 })
69
70 })
71
72 })
73
74 })
75
76 })
77
78 })
79
80 })
81
82 })
83
84 })
85
86 })
87
88 })
89
90 })
91
92 })
93
94 })
95
96 })
97
98 })
99
100 })
101
102 })
103
104 })
105
106 })
107
108 })
109
110 })
111
112 })
113
114 })
115
116 })
117
118 })
119
120 })
121
122 })
123
124 })
125
126 })
127
128 })
129
130 })
131
132 })
133
134 })
135
136 })
137
138 })
139
140 })
141
142 })
143
144 })
145
146 })
147
148 })
149
150 })
151
152 })
153
154 })
155
156 })
157
158 })
159
160 })
161
162 })
163
164 })
165
166 })
167
168 })
169
170 })
171
172 })
173
174 })
175
176 })
177
178 })
179
180 })
181
182 })
183
184 })
185
186 })
187
188 })
189
190 })
191
192 })
193
194 })
195
196 })
197
198 })
199
200 })
201
202 })
203
204 })
205
206 })
207
208 })
209
210 })
211
212 })
213
214 })
215
216 })
217
218 })
219
220 })
221
222 })
223
224 })
225
226 })
227
228 })
229
230 })
231
232 })
233
234 })
235
236 })
237
238 })
239
240 })
241
242 })
243
244 })
245
246 })
247
248 })
249
250 })
251
252 })
253
254 })
255
256 })
257
258 })
259
260 })
261
262 })
263
264 })
265
266 })
267
268 })
269
270 })
271
272 })
273
274 })
275
276 })
277
278 })
279
280 })
281
282 })
283
284 })
285
286 })
287
288 })
289
290 })
291
292 })
293
294 })
295
296 })
297
298 })
299
300 })
301
302 })
303
304 })
305
306 })
307
308 })
309
310 })
311
312 })
313
314 })
315
316 })
317
318 })
319
320 })
321
322 })
323
324 })
325
326 })
327
328 })
329
330 })
331
332 })
333
334 })
335
336 })
337
338 })
339
340 })
341
342 })
343
344 })
345
346 })
347
348 })
349
350 })
351
352 })
353
354 })
355
356 })
357
358 })
359
360 })
361
362 })
363
364 })
365
366 })
367
368 })
369
370 })
371
372 })
373
374 })
375
376 })
377
378 })
379
380 })
381
382 })
383
384 })
385
386 })
387
388 })
389
390 })
391
392 })
393
394 })
395
396 })
397
398 })
399
400 })
401
402 })
403
404 })
405
406 })
407
408 })
409
410 })
411
412 })
413
414 })
415
416 })
417
418 })
419
420 })
421
422 })
423
424 })
425
426 })
427
428 })
429
430 })
431
432 })
433
434 })
435
436 })
437
438 })
439
440 })
441
442 })
443
444 })
445
446 })
447
448 })
449
450 })
451
452 })
453
454 })
455
456 })
457
458 })
459
460 })
461
462 })
463
464 })
465
466 })
467
468 })
469
470 })
471
472 })
473
474 })
475
476 })
477
478 })
479
480 })
481
482 })
483
484 })
485
486 })
487
488 })
489
490 })
491
492 })
493
494 })
495
496 })
497
498 })
499
500 })
501
502 })
503
504 })
505
506 })
507
508 })
509
510 })
511
512 })
513
514 })
515
516 })
517
518 })
519
520 })
521
522 })
523
524 })
525
526 })
527
528 })
529
530 })
531
532 })
533
534 })
535
536 })
537
538 })
539
540 })
541
542 })
543
544 })
545
546 })
547
548 })
549
550 })
551
552 })
553
554 })
555
556 })
557
558 })
559
560 })
561
562 })
563
564 })
565
566 })
567
568 })
569
570 })
571
572 })
573
574 })
575
576 })
577
578 })
579
580 })
581
582 })
583
584 })
585
586 })
587
588 })
589
590 })
591
592 })
593
594 })
595
596 })
597
598 })
599
600 })
601
602 })
603
604 })
605
606 })
607
608 })
609
610 })
611
612 })
613
614 })
615
616 })
617
618 })
619
620 })
621
622 })
623
624 })
625
626 })
627
628 })
629
630 })
631
632 })
633
634 })
635
636 })
637
638 })
639
640 })
641
642 })
643
644 })
645
646 })
647
648 })
649
650 })
651
652 })
653
654 })
655
656 })
657
658 })
659
660 })
661
662 })
663
664 })
665
666 })
667
668 })
669
670 })
671
672 })
673
674 })
675
676 })
677
678 })
679
680 })
681
682 })
683
684 })
685
686 })
687
688 })
689
690 })
691
692 })
693
694 })
695
696 })
697
698 })
699
700 })
701
702 })
703
704 })
705
706 })
707
708 })
709
710 })
711
712 })
713
714 })
715
716 })
717
718 })
719
720 })
721
722 })
723
724 })
725
726 })
727
728 })
729
730 })
731
732 })
733
734 })
735
736 })
737
738 })
739
740 })
741
742 })
743
744 })
745
746 })
747
748 })
749
750 })
751
752 })
753
754 })
755
756 })
757
758 })
759
760 })
761
762 })
763
764 })
765
766 })
767
768 })
769
770 })
771
772 })
773
774 })
775
776 })
777
778 })
779
780 })
781
782 })
783
784 })
785
786 })
787
788 })
789
790 })
791
792 })
793
794 })
795
796 })
797
798 })
799
800 })
801
802 })
803
804 })
805
806 })
807
808 })
809
810 })
811
812 })
813
814 })
815
816 })
817
818 })
819
820 })
821
822 })
823
824 })
825
826 })
827
828 })
829
830 })
831
832 })
833
834 })
835
836 })
837
838 })
839
840 })
841
842 })
843
844 })
845
846 })
847
848 })
849
850 })
851
852 })
853
854 })
855
856 })
857
858 })
859
860 })
861
862 })
863
864 })
865
866 })
867
868 })
869
870 })
871
872 })
873
874 })
875
876 })
877
878 })
879
880 })
881
882 })
883
884 })
885
886 })
887
888 })
889
890 })
891
892 })
893
894 })
895
896 })
897
898 })
899
900 })
901
902 })
903
904 })
905
906 })
907
908 })
909
910 })
911
912 })
913
914 })
915
916 })
917
918 })
919
920 })
921
922 })
923
924 })
925
926 })
927
928 })
929
930 })
931
932 })
933
934 })
935
936 })
937
938 })
939
940 })
941
942 })
943
944 })
945
946 })
947
948 })
949
950 })
951
952 })
953
954 })
955
956 })
957
958 })
959
960 })
961
962 })
963
964 })
965
966 })
967
968 })
969
970 })
971
972 })
973
974 })
975
976 })
977
978 })
979
980 })
981
982 })
983
984 })
985
986 })
987
988 })
989
990 })
991
992 })
993
994 })
995
996 })
997
998 })
999
1000 })
1001
1002 })
1003
1004 })
1005
1006 })
1007
1008 })
1009
1010 })
1011
1012 })
1013
1014 })
1015
1016 })
1017
1018 })
1019
1020 })
1021
1022 })
1023
1024 })
1025
1026 })
1027
1028 })
1029
1030 })
1031
1032 })
1033
1034 })
1035
1036 })
1037
1038 })
1039
1040 })
1041
1042 })
1043
1044 })
1045
1046 })
1047
1048 })
1049
1050 })
1051
1052 })
1053
1054 })
1055
1056 })
1057
1058 })
1059
1060 })
1061
1062 })
1063
1064 })
1065
1066 })
1067
1068 })
1069
1070 })
1071
1072 })
1073
1074 })
1075
1076 })
1077
1078 })
1079
1080 })
1081
1082 })
1083
1084 })
1085
1086 })
1087
1088 })
1089
1090 })
1091
1092 })
1093
1094 })
1095
1096 })
1097
1098 })
1099
1100 })
1101
1102 })
1103
1104 })
1105
1106 })
1107
1108 })
1109
1110 })
1111
1112 })
1113
1114 })
1115
1116 })
1117
1118 })
1119
1120 })
1121
1122 })
1123
1124 })
1125
1126 })
1127
1128 })
1129
1130 })
1131
1132 })
1133
1134 })
1135
1136 })
1137
1138 })
1139
1140 })
1141
1142 })
1143
1144 })
1145
1146 })
1147
1148 })
1149
1150 })
1151
1152 })
1153
1154 })
1155
1156 })
1157
1158 })
1159
1160 })
1161
1162 })
1163
1164 })
1165
1166 })
1167
1168 })
1169
1170 })
1171
1172 })
1173
1174 })
1175
1176 })
1177
1178 })
1179
1180 })
1181
1182 })
1183
1184 })
1185
1186 })
1187
1188 })
1189
1190 })
1191
1192 })
1193
1194 })
1195
1196 })
1197
1198 })
1199
1200 })
1201
1202 })
1203
1204 })
1205
1206 })
1207
1208 })
1209
1210 })
1211
1212 })
1213
1214 })
1215
1216 })
1217
1218 })
1219
1220 })
1221
1222 })
1223
1224 })
1225
1226 })
1227
1228 })
1229
1230 })
1231
1232 })
1233
1234 })
1235
1236 })
1237
1238 })
1239
1240 })
1241
1242 })
1243
1244 })
1245
1246 })
1247
1248 })
1249
1250 })
1251
1252 })
1253
1254 })
1255
1256 })
1257
1258 })
1259
1260 })
1261
1262 })
1263
1264 })
1265
1266 })
1267
1268 })
1269
1270 })
1271
1272 })
1273
1274 })
1275
1276 })
1277
1278 })
1279
1280 })
1281
1282 })
1283
1284 })
1285
1286 })
1287
1288 })
1289
1290 })
1291
1292 })
1293
1294 })
1295
1296 })
1297
1298 })
1299
1300 })
1301
1302 })
1303
1304 })
1305
1306 })
1307
1308 })
1309
1310 })
1311
1312 })
1313
1314 })
1315
1316 })
1317
1318 })
1319
1320 })
1321
1322 })
1323
1324 })
1325
1326 })
1327
1328 })
1329
1330 })
1331
1332 })
1333
1334 })
1335
1336 })
1337
1338 })
1339
1340 })
1341
1342 })
1343
1344 })
1345
1346 })
1347
1348 })
1349
1350 })
1351
1352 })
1353
1354 })
1355
1356 })
1357
1358 })
1359
1360 })
1361
1362 })
1363
1364 })
1365
1366 })
1367
1368 })
1369
1370 })
1371
1372 })
1373
1374 })
1375
1376 })
1377
1378 })
1379
1380 })
1381
1382 })
1383
1384 })
1385
1386 })
1387
1388 })
1389
1390 })
1391
1392 })
1393
1394 })
1395
1396 })
1397
1398 })
1399
1400 })
1401
1402 })
1403
1404 })
1405
1406 })
1407
1408 })
1409
1410 })
1411
1412 })
1413
1414 })
1415
1416 })
1417
1418 })
1419
1420 })
1421
1422 })
1423
1424 })
1425
1426 })
1427
1428 })
1429
1430 })
1431
1432 })
1433
1434 })
1435
1436 })
1437
1438 })
1439
1440 })
1441
1442 })
1443
1444 })
1445
1446 })
1447
1448 })
1449
1450 })
1451
1452 })
1453
1454 })
1455
1456 })
1457
1458 })
1459
1460 })
1461
1462 })
1463
1464 })
1465
1466 })
1467
1468 })
1469
1470 })
1471
1472 })
1473
1474 })
1475
1476 })
1477
1478 })
1479
1480 })
1481
1482 })
1483
1484 })
1485
1486 })
1487
1488 })
1489
1490 })
1491
1492 })
1493
1494 })
1495
1496 })
1497
1498 })
1499
1500 })
1501
1502 })
1503
1504 })
1505
1506 })
1507
1508 })
1509
1510 })
1511
1512 })
1513
1514 })
1515
1516 })
1517
1518 })
1519
1520 })
1521
1522 })
1523
1524 })
1525
1526 })
1527
1528 })
1529
1530 })
1531
1532 })
1533
1534 })
1535
1536 })
1537
1538 })
1539
1540 })
1541
1542 })
1543
1544 })
1545
1546 })
1547
1548 })
1549
1550 })
1551
1552 })
1553
1554 })
1555
1556 })
1557
1558 })
1559
1560 })
1561
1562 })
1563
1564 })
1565
1566 })
1567
1568 })
1569
1570 })
1571
1572 })
1573
1574 })
1575
1576 })
1577
1578 })
1579
1580 })
1581
1582 })
1583
1584 })
1585
1586 })
1587
1588 })
1589
1590 })
1591
1592 })
1593
1594 })
1595
1596 })
1597
1598 })
1599
1600 })
1601
1602 })
1603
1604 })
1605
1606 })
1607
1608 })
1609
1610 })
1611
1612 })
1613
1614 })
1615
1616 })
1617
1618 })
1619
1620 })
1621
1622 })
1623
1624 })
1625
1626 })
1627
1628 })
1629
1630 })
1631
1632 })
1633
1634 })
1635
1636 })
1637
1638 })
1639
1640 })
1641
1642 })
1643
1644 })
1645
1646 })
1647
1648 })
1649
1650 })
1651
1652 })
1653
1654 })
1655
1656 })
1657
1658 })
1659
1660 })
1661
1662 })
1663
1664 })
1665
1666 })
1667
1668 })
1669
1670 })
1671
1672 })
1673
1674 })
1675
1676 })
1677
1678 })
1679
1680 })
1681
1682 })
1683
1684 })
1685
1686 })
1687
1688 })
1689
1690 })
1691
1692 })
1693
1694 })
1695
1696 })
1697
1698 })
1699
1700 })
1701
1702 })
1703
1704 })
1705
1706 })
1707
1708 })
1709
1710 })
1711
1712 })
1713
1714 })
1715
1716 })
1717
1718 })
1719
1720 })
1721
1722 })
1723
1724 })
1725
1726 })
1727
1728 })
1729
1730 })
1731
1732 })
1733
1734 })
1735
1736 })
1737
1738 })
1739
1740 })
1741
1742 })
1743
1744 })
1745
1746 })
1747
1748 })
1749
1750 })
1751
1752 })
1753
1754 })
1755
1756 })
1757
1758 })
1759
1760 })
1761
1762 })
1763
1764 })
1765
1766 })
1767
1768 })
1769
1770 })
1771
1772 })
1773
1774 })
1775
1776 })
1777
1778 })
1779
1780 })
1781
1782 })
1783
1784 })
1785
1786 })
1787
1788 })
1789
1790 })
1791
1792 })
1793
1794 })
1795
1796 })
1797
1798 })
1799
1800 })
1801
1802 })
1803
1804 })
1805
1806 })
1807
1808 })
1809
1810 })
1811
1812 })
1813
1814 })
1815
1816 })
1817
1818 })
1819
1820 })
1821
1822 })
1823
1824 })
1825
1826 })
1827
1828 })
1829
1830 })
1831
1832 })
1833
1834 })
1835
1836 })
1837
1838 })
1839
1840 })
1841
1842 })
1843
1844 })
1845
1846 })
1847
1848 })
1849
1850 })
1851
1852 })
1853
1854 })
1855
1856 })
1857
1858 })
1859
1860 })
1861
1862 })
1863
1864 })
1865
1866 })
1867
1868 })
1869
1870 })
1871
1872 })
1873
1874 })
1875
1876 })
1877
1878 })
1879
1880 })
1881
1882 })
1883
1884 })
1885
1886 })
1887
1888 })
1889
1890 })
1891
1892 })
1893
1894 })
1895
1896 })
1897
1898 })
1899
1900 })
1901
1902 })
1903
1904 })
1905
1906 })
1907
1908 })
1909
1910 })
1911
1912 })
1913
1914 })
1915
1916 })
1917
1918 })
1919
1920 })
1921
1922 })
1923
1924 })
1925
1926 })
1927
1928 })
1929
1930 })
1931
1932 })
1933
1934 })
1935
1936 })
1937
1938 })
1939
1940 })
1941
1942 })
1943
1944 })
1945
1946 })
1947
1948 })
1949
1950 })
1951
1952 })
1953
1954 })
1955
1956 })
1957
1958 })
1959
1960 })
1961
1962 })
1963
1964 })
1965
1966 })
1967
1968 })
1969
1970 })
1971
1972 })
1973
1974 })
1975
1976 })
1977
1978 })
1979
1980 })
1981
1982 })
1983
1984 })
1985
1986 })
1987
1988 })
1989
1990 })
1991
1992 })
1993
1994 })
1995
1996 })
1997
1998 })
1999
2000 })
```

20.2.3 What to test

Now that we've gone over exactly how to make unit tests, let's talk about exactly what to test. When testing functions, you generally want to test every possible parameter in the function, and a variety of inputs. In particular, try to think of ways that the function could be used incorrectly and write a test to catch that. For example, our `add_2()` function will fail if a string (e.g. "2") was inputted instead of a number, so you'll want to add a test for that. You'll also want to make sure that inputting something other than simply a single number, such as a vector of numbers, works correctly. Basically, you want to be thorough and cover all of your bases.

Writing unit tests is one of the most time-efficient things you can do since it helps you avoid making costly (in time and in getting wrong results) mistakes. But don't spend too much time writing tests. If you've tested that `add_2(2)` equals 4, no need to test that `add_2(3)` equals five since you're essentially testing the exact same thing. And consider your audience (even if that is only you). If you know that `add_2()` will only be used by people who know better than to input a string, there's no need to test for that (and testing for that will mean that you'd also want code that handles a string input in the way you want it, such as outputting a message saying strings aren't allowed). In general, I think it's always better to have more tests than fewer, but consider whether writing that test is a good use of your time.

20.2.3.1 Tests for research projects

When you use R for a research project, you'll usually take data that someone else collected, or scrape it yourself, do some work to clean this data (e.g. subset or aggregate the data, standardize values) and then run a regression on it. In these cases there are fewer opportunities to use unit tests to check your code. Indeed, the best checks are often content knowledge about the data and examining the results of the regression to see if it makes sense and fits prior literature.

While testing is most commonly used for functions, you can use it to test data. Writing tests for research data is best if your code is scraping the data (webscrape or PDF scrape) and you want to verify that it is correct, or if you expect the data to change and want to ensure that it is still correct (while exact values will change, you can check broad categories such as whether

certain groups are included). For example, if you know that you only want to look at a certain state, you can write a test that expects the only state in the data to be the one you're analyzing. This way, if you add more data, such as a new release of that data set, the test will catch if there's any other state that you may have forgotten to remove after adding the new data. If you're sure that you will only use a particular data set that never changes, you're better off just writing code in your main R Script (or a specific script for checking the data) to do these checks rather than dedicated tests.

20.2.3.2 Tests for data collection

Our example in this chapter was tests for a data collection process - in our case, PDF scraping - so we've already seen how to test code for gathering data. We'll still talk briefly here about what kind of tests - and how many tests - you will want for this type of code. In normal tests, you don't want to test the exact same thing multiple times (for example, if you test that $2 + 2 = 4$, you don't need to test that $2 + 3 = 5$). This is different when it comes to testing code that collects data from a source, such as through PDF scraping or webscraping (which is different from when you download a machine-readable file like a .csv file).

When testing data collection code, you want to be far more thorough, retesting something in multiple ways. This is because small differences in the data you are scraping may affect the code at different parts of the scrape. For example, imagine a PDF with 10 pages. On nine of those pages each row is a single line, but on a single page there are rows that are multiple lines. Writing tests to check results from a single page will miss the issue 9 out of 10 times. I've often experienced PDF or webscraping where some parts of the data are just "weird" and cause the code to scrape it incorrectly - but often not tell me that there's an issue. So to catch this you'll need far more tests than normal. I prefer to choose a few random pages (more if the PDF/website is longer) and test random rows and columns since that'll give a good coverage of the results. In addition, I look at the PDF or website and try to see if there's anything atypical about a certain part; if there is, I test that specifically. It's easy to over-test (and that's better than under-testing) this kind of work, but there are rapidly diminishing returns. So test comprehensively but not at the cost of having too little time to work on code - again, this is something that requires experience and doesn't have a hard

rule on what constitutes too much (or too little) testing.

20.3 Test-driven development (TDD)

We'll finish this chapter by talking about test-driven development (TDD), a philosophy in programming where you write the tests first and then write the code that meets these tests after. This is really an extension (and personally, an inspiration) to the discussion in Chapter 18 of planning out your project before you start. In Chapter 18 we talked about writing out every step of the project by hand sketching all the figures or tables that you intended to have. With TDD, you write tests for all of the functions you intend to write (and any variations of parameters or inputs for these functions) or data you intend to gather/clean.

Test-driven development is a useful tool to make you really think about the functions that you need to write, and how they interact with each other. This is an excellent way to identify potential issues (I've often realized while writing tests that the approach I was going to do wouldn't work) before you start on the code. However, for this same reason, it is a fairly advanced topic since you need to know exactly (or, mostly) what you need to do, and the likely problems that each approach will face. For that reason, I recommend holding off on using TDD until you're fairly experienced with R or programming in general.

Chapter 21

Git

21.1 What is Git and why do I need it?

As you write R code you will - I hope! - save your R script from time to time (preferably using RStudio's auto-save feature) to avoid losing any code you've written if you close R or shut down your computer. This is important as it'll save everything you've done locally but if your computer crashes, you'll want your work to be backed up elsewhere. While you should have something like Dropbox or Google Drive that keeps backups of your work, here we'll talk about Git which is a version control software that gives you much more control (but requires more work) of the saved work than from something like Dropbox.¹ Before getting into exactly how to use Git, we'll talk first on what it is and how it'll help your work. Git is also a very powerful and complex tool so this guide is going to be touching just a small - but useful to most researchers and R programmers - part of it.

With backup software such as Dropbox, it'll save your work very frequently - so frequently in fact that I turn off Dropbox when I write R since it keeps interrupting me by saving at the moment I'm typing, which stops the typing. Below is the Dropbox page for some R code that I've been working on to scrape COVID data. Notice the timestamps - 4/5 of them are within one

¹This came in handy for me as somehow one of my dissertation papers written in RMarkdown became empty a couple of months before my defense and I couldn't undo that change. My Dropbox backup was older than my Git backup so having Git was a real time saver

minute, showing how often Dropbox is saving changes. This is useful if I need the most recent update - or to share the most recent version with a collaborator. Here's the big issue - and the one that Git solves - I have four versions within a minute of each other, what's the difference between them? Dropbox is saving automatically and doesn't indicate how they're different (clicking on the file shows the complete file, not differences relative to some previous version) which means if I mess up some code a while ago, I can't easily see which version is the one that works. With Git you can essentially wait until you've made enough changes to decide that these changes merit a new "version" of your work (One way to think about this is)

Yesterday			
	webscraping_state_info.R 8:14 PM	Edited by Jacob Kaplan Desktop	136.13 KB <button>Current version</button>
	webscraping_state_info.R 8:14 PM	Edited by Jacob Kaplan Desktop	136.13 KB <button>Restore</button>
	webscraping_state_info.R 8:14 PM	Edited by Jacob Kaplan Desktop	136.13 KB
	webscraping_state_info.R 8:13 PM	Edited by Jacob Kaplan Desktop	136.13 KB
	webscraping_state_info.R 6:09 PM	Edited by Jacob Kaplan Desktop	136.13 KB

If you're ever used the track changes feature on a Word Document(or Google Doc or Overleaf, etc.), the concept is similar. When you have this setting in a Word or Google Doc every time you (or anyone else) makes changes in that document, those changes, who made them, and when they occurred, is tracked. This makes it easy to see exactly what part of the file was changed and to undo that change if necessary. Below is an example of this feature on one of my drafts on Overleaf (basically a way to collaborate using LaTeX which is similar to RMarkdown). You can see each change that my co-author Aaron Chalfin made in the draft in the purpose changes the main part of the photo. The parts that were rewritten or added are highlighted in purple while the parts that were deleted have a purple . What is shown in purple isn't all of the history of changes for this paper. If you look at the part on the right, highlighted in green, it shows what files were edited, by whom, and at what time. If you don't like a change - or in R's case more commonly, broke some code by accident - you can go back in the history of changes and return to an older version.

The way that R - and many other programming languages (and technically

you can use this for any file or folder) does this “version control” is through Git.

The times you see are ones that Overleaf automatically set each change to.

The screenshot shows the Overleaf interface with a sidebar on the left listing files: main.tex (edited), main.bib, chicago.old_to_ne... (disabled), chicago_street_re... (disabled), chicago_street_o... (disabled), googletrends.pdf (disabled), googletrends.cri... (disabled), heterogeneity.tex (disabled), lighting_survey_1... (disabled), lighting_survey_p... (disabled), main_regression.tex (disabled), mprobit.tex (disabled), pilot.tex (disabled), predictor_of_safet... (disabled), regression_weight... (disabled), sample_general_pu... (disabled), treatment_control... (disabled), willingness_to_pa... (disabled), willingness_to_pa... (disabled), and wtp_hetero.tex (disabled). The main area displays a document with several numbered footnotes. A vertical timeline on the right lists changes:

- Edited main.bib at 11:50 am • achalfin
- Edited main.tex at 11:45 am • achalfin
- Edited main.tex at 11:39 am • achalfin
- Edited main.bib at 11:34 am • achalfin
- Edited main.bib at 11:28 am • achalfin
- Edited main.tex at 11:23 am • achalfin
- Edited main.tex at 11:18 am • achalfin
- Edited main.tex at 11:10 am • achalfin
- Edited main.tex at 11:05 am • achalfin
- Edited main.tex at 11:05 am • achalfin

A note at the bottom of the timeline states: "We also ask a question that elicits how much time respondents plan to spend outdoors during nighttime hours given the photo who 35 more updates below to view. The purpose of this function is to break down a block that has limited the interactivity of the panelized literature on safety."

You make changes to your code or RMarkdown file and the computer will track these changes.

21.2 Git basics

There are four main processes you need to know for a basic understanding of Git: checkout, add and commit, push, and pull. We’ll use the example of getting a book from the library to walk through using Git. The steps for this are simple, we go to the library, pick a book we want, check it out from the librarian, read it, and eventually return it. Using Git adds one wrinkle to this, we will want to write in the book and see what other people write too. Of course, when the book is checked out, no one else could write in our version, and no one can see what we write. So anything we write has to be done before we return the book to the library, then we check-out the book again to see what other people have written. When we want another book, we simply redo these steps.

Library Steps	Git steps	Git code
Go to library		
Find book and check-out book	Clone (usually will just be done once per project). RStudio helps with this. The code you see in the next cell is a little more complicated than normal since we're using RStudio Server.	git clone <i>path to repo</i> , <i>can be GitHub link</i>
Read or write in book	This is done in R, not in Git	No git code, this is going to be whatever code we write in R. Also includes any outputs such as making a graph that is saved, RMarkdown outputs like a PDF, or even new R files.
Return book	Add & commit Push	git add . git commit -m "message" indicating what we wrote" git push
Check-out book again (to see what other people have written in it)	Pull	git pull

Another way to think about commit vs push is that of writing an email. When you write an email, you're essentially editing a blank document by adding the words of the email. When you save (but don't send) the email, you are making a commit (essentially "committing" or promising to make a change). When you send the email you are making a push (taking something that you have written and changed and sending it to the main repository).

While emails let you correspond directly between two or more people, how Git works is like sending the email to a central server (or a Post Office) and anyone who wants to read it has to go there. And when someone reads it and responds their email also goes to this central server. You have to go there to get their response (called a “pull” in Git terms) which is essentially an addition to your initial email.

21.3 When to commit

21.4 Code review

Part V

Data

Chapter 22

Introduction

At this point you have learned how to read in data, manipulate it to get just the parts you want or to aggregate it to the level you want, and visualize it through maps or graphs. You've done so using data sets that are commonly used in criminological research.

In the next several chapters we will be introducing a number of other data sets - or looking deeper into data we've already seen - that are common in criminology. While these chapters do use R a bit to explore or read in the data, they are primarily a discussion of the trade-offs of using each data set. Some of the data sets are difficult to read into R, requiring more steps than you may be used to, so these chapters will also discuss how to get that data into R.

Chapter 23

Uniform Crime Report (UCR) - Offenses Known and Clearances by Arrest

The Uniform Crime Report (UCR) Program Data are an collection of agency-level crime data published by the FBI. There are a number of different data sets included in the UCR including data on crime, arrests, hate crimes, arson, and stolen property. We'll be using the Offenses Known and Clearances by Arrest data set (the "crime" data set), which is the most commonly used data set in the UCR and is sometimes used as a shorthand for UCR data. In this lesson we'll use "UCR" and "Offenses Known and Clearances by Arrest" interchangeably but keep in mind that doing so is technically incorrect.

You can read more about the UCR program and all of the data sets it includes on the National Archive of Criminal Justice Data (NACJD) page [here](#). You can also check out my site [Crime Data Tool](#) which visualizes several of the UCR data sets and has info in the [FAQ page](#) explaining the data.

Nearly every police agency in the United States - approximately 18,000 agencies - now report their data to the FBI which compiles and releases the UCR data (some states have their agencies report to a state department which then sends the data to the FBI). This data is available since 1960 though early years have many fewer agencies reporting than do so in later years.

The data file has annual data on the number of crimes reported, the number

of crimes cleared, the number cleared where all offenders are under age 18, and the number of unfounded crimes. We'll discuss each of these a bit further as we dive into the data. Agencies report the monthly number of each crime though the data we'll work with has aggregated that to annual counts.

Due to its longevity (it has data since 1960) and ubiquity (almost every agency reports and has done so for many years) it is a popular data set for criminologists.

23.1 Exploring the UCR data

We are going to look at data with the combined annual count of crimes for every year available - 1960-2017 - which I've made available [here](#). The FBI releases the data as a single file per year and each file has monthly counts of crime. This data set does some cleaning for us by aggregating yearly and making it a single file for the whole time period. The first step when working with this UCR data is loading it into R. As with loading any data, make sure that your working directory path is correctly set using `setwd()` so R knows which folder the data is in.

```
load("data/offenses_known_yearly_1960_2017.rda")
```

Let's start with a basic examination of the data. First, how big it is, what variables it has, and what the units are.

```
ncol(offenses_known_yearly_1960_2017)
#> [1] 159
```

```
nrow(offenses_known_yearly_1960_2017)
#> [1] 959010
```

```
names(offenses_known_yearly_1960_2017)
#> [1] "ori"                                "orig"
#> [3] "agency_name"                         "state"
#> [5] "state_abb"                            "year"
#> [7] "number_of_months_reported"           "fips_state_code"
#> [9] "fips_county_code"                     "fips_state_county_code"
#> [11] "fips_place_code"                     "fips_state_place_code"
#> [13] "agency_type"                         "agency_subtype_1"
```

```

#> [15] "agency_subtype_2"
#> [17] "census_name"
#> [19] "population_group"
#> [21] "juvenile_age"
#> [23] "last_update"
#> [25] "followup_indication"
#> [27] "covered_by_ori"
#> [29] "date_of_last_update"
#> [31] "special_mailing_group"
#> [33] "first_line_of_mailing_address"
#> [35] "third_line_of_mailing_address"
#> [37] "officers_killed_by_felony"
#> [39] "officers_assaulted"
#> [41] "actual_manslaughter"
#> [43] "actual_rape_by_force"
#> [45] "actual_robbery_total"
#> [47] "actual_robbery_with_a_knife"
#> [49] "actual_robbery_unarmed"
#> [51] "actual_assault_with_a_gun"
#> [53] "actual_assault_other_weapon"
#> [55] "actual_assault_simple"
#> [57] "actual_burg_force_entry"
#> [59] "actual_burg_attempted"
#> [61] "actual_mtr_veh_theft_total"
#> [63] "actual_mtr_veh_theft_truck"
#> [65] "actual_all_crimes"
#> [67] "actual_index_violent"
#> [69] "actual_index_total"
#> [71] "tot_clr_manslaughter"
#> [73] "tot_clr_rape_by_force"
#> [75] "tot_clr_robbery_total"
#> [77] "tot_clr_robbery_with_a_knife"
#> [79] "tot_clr_robbery_unarmed"
#> [81] "tot_clr_assault_with_a_gun"
#> [83] "tot_clr_assault_other_weapon"
#> [85] "tot_clr_assault_simple"
#> [87] "tot_clr_burg_force_entry"
"crosswalk_agency_name"
"population"
"country_division"
"core_city_indication"
"fbi_field_office"
"zip_code"
"agency_count"
"month_included_in"
"special_mailing_address"
"second_line_of_mailing_address"
"fourth_line_of_mailing_address"
"officers_killed_by_accident"
"actual_murder"
"actual_rape_total"
"actual_rape_attempted"
"actual_robbery_with_a_gun"
"actual_robbery_other_weapon"
"actual_assault_total"
"actual_assault_with_a_knife"
"actual_assault_unarmed"
"actual_burg_total"
"actual_burg_nonforce_entry"
"actual_theft_total"
"actual_mtr_veh_theft_car"
"actual_mtr_veh_theft_other"
"actual_assault_aggravated"
"actual_index_property"
"tot_clr_murder"
"tot_clr_rape_total"
"tot_clr_rape_attempted"
"tot_clr_robbery_with_a_gun"
"tot_clr_robbery_other_weapon"
"tot_clr_assault_total"
"tot_clr_assault_with_a_knife"
"tot_clr_assault_unarmed"
"tot_clr_burg_total"
"tot_clr_burg_nonforce_entry"

```

364CHAPTER 23. UNIFORM CRIME REPORT (UCR) - OFFENSES KNOWN AND CLEA

```
#> [89] "tot_clr_burg_attempted"
#> [91] "tot_clr_mtr_veh_theft_total"
#> [93] "tot_clr_mtr_veh_theft_truck"
#> [95] "tot_clr_all_crimes"
#> [97] "tot_clr_index_violent"
#> [99] "tot_clr_index_total"
#> [101] "clr_18_manslaughter"
#> [103] "clr_18 Rape_by_force"
#> [105] "clr_18_robbery_total"
#> [107] "clr_18_robbery_with_a_knife"
#> [109] "clr_18_robbery_unarmed"
#> [111] "clr_18_assault_with_a_gun"
#> [113] "clr_18_assault_other_weapon"
#> [115] "clr_18_assault_simple"
#> [117] "clr_18_burg_force_entry"
#> [119] "clr_18_burg_attempted"
#> [121] "clr_18_mtr_veh_theft_total"
#> [123] "clr_18_mtr_veh_theft_truck"
#> [125] "clr_18_all_crimes"
#> [127] "clr_18_index_violent"
#> [129] "clr_18_index_total"
#> [131] "unfound_manslaughter"
#> [133] "unfound_Rape_by_force"
#> [135] "unfound_robbery_total"
#> [137] "unfound_robbery_with_a_knife"
#> [139] "unfound_robbery_unarmed"
#> [141] "unfound_assault_with_a_gun"
#> [143] "unfound_assault_other_weapon"
#> [145] "unfound_assault_simple"
#> [147] "unfound_burg_force_entry"
#> [149] "unfound_burg_attempted"
#> [151] "unfound_mtr_veh_theft_total"
#> [153] "unfound_mtr_veh_theft_truck"
#> [155] "unfound_all_crimes"
#> [157] "unfound_index_violent"
#> [159] "unfound_index_total"
"tot_clr_theft_total"
"tot_clr_mtr_veh_theft_car"
"tot_clr_mtr_veh_theft_other"
"tot_clr_assault_aggravated"
"tot_clr_index_property"
"clr_18_murder"
"clr_18_Rape_total"
"clr_18_Rape_attempted"
"clr_18_robbery_with_a_gun"
"clr_18_robbery_other_weapon"
"clr_18_assault_total"
"clr_18_assault_with_a_knife"
"clr_18_assault_unarmed"
"clr_18_burg_total"
"clr_18_burg_nonforce_entry"
"clr_18_theft_total"
"clr_18_mtr_veh_theft_car"
"clr_18_mtr_veh_theft_other"
"clr_18_assault_aggravated"
"clr_18_index_property"
"unfound_murder"
"unfound_Rape_total"
"unfound_Rape_attempted"
"unfound_robbery_with_a_gun"
"unfound_robbery_other_weapon"
"unfound_assault_total"
"unfound_assault_with_a_knife"
"unfound_assault_unarmed"
"unfound_burg_total"
"unfound_burg_nonforce_entry"
"unfound_theft_total"
"unfound_mtr_veh_theft_car"
"unfound_mtr_veh_theft_other"
"unfound_assault_aggravated"
"unfound_index_property"
```

We can see this is a very big file - 159 columns and nearly a million rows! Normally we'd use the `head()` function to see the first 6 rows of every column but since this data has so many columns we won't do that as it'd be hard to read. Instead we can use `View()` to open what's essentially an Excel file showing our data. This is useful to quickly glance at the data but is limited as it can bias us to believe that the first several rows are representative of the data (an issue also present with `head()`). But, for a first glance it is useful and will be supplemented by better checks below.

```
View(offenses_known_yearly_1960_2017)
```

From looking at the data in `View()` we can see that the units are agency-years. Each row is a single agency for a single year. This is useful because it tells us we will have crime in agencies over time, which is a very common unit of crime data. Let's take a look at how many agencies report each year using the `table()` function which says how many times each value occurs for the column we select. This is also a useful check on if every year from 1960 to 2017 is actually available - don't just trust that the data has what it says it has!

```
table(offenses_known_yearly_1960_2017$year)
#>
#> 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972
#> 8452 8456 7825 8713 9038 9097 9147 9275 9398 9477 9835 10509 11302
#> 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985
#> 12002 12510 13516 14518 15230 15770 16176 16413 16614 16792 16913 17037 17267
#> 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998
#> 17441 17527 17298 17430 17608 17852 18012 18195 18367 18482 18536 18921 18510
#> 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011
#> 18778 19655 19820 20214 20388 20585 20739 21011 21219 21353 21583 21771 21897
#> 2012 2013 2014 2015 2016 2017
#> 22049 22202 22332 22524 22645 22784
```

From these results it's clear that there are huge differences in how many agencies report in early years compared to more recent years. Is this an issue in an analysis? From the above table it is concerning but not entirely clear there is an issue depending on our specific analysis. If we only care about recent years then it wouldn't matter. If we only use large agencies, then knowing that relatively few agencies reported in 1960 doesn't mean

that few large agencies reported. For that you'd have to look closer at only the agencies you want to study - we won't do that here but keep it in mind.

23.2 ORIs - Unique agency identifiers

In the UCR and other FBI data sets, agencies are identified using ORiginating Agency Identifiers or ORIs. These are unique ID codes used to identify an agency. If we used the agency's name we'd end up with some duplicates. For example, if you looked for the Philadelphia Police Department using the agency name, you'd find both the "Philadelphia Police Department" in Pennsylvania and the one in Mississippi.

```
head(offenses_known_yearly_1960_2017$ori)
#> [1] "AK00101" "AK00101" "AK00101" "AK00101" "AK00101" "AK00101"
```

Each ORI is a 7-digit value starting with the state abbreviation (for some reason the FBI incorrectly puts the abbreviation for Nebraska as NB instead of NE) followed by 5 numbers. In the NIBRS data (another FBI data set) the ORI uses a 9-digit code - expanding the 5 numbers to 7 numbers. When dealing with specific agencies, make sure to use the ORI rather than the agency name to avoid any mistakes.

For an easy way to find the ORI number of an agency, use this [site](#). Type an agency name or an ORI code into the search section and it will return everything that is a match.

23.3 Hierarchy Rule

The UCR has what is called the Hierarchy Rule where only the most serious crime in an incident is reported (except for motor vehicle theft which is always included). For example if there is an incident where the victim is robbed and then murdered, only the murder is counted as it is considered more serious than the robbery.

How much does this affect our data in practice? Actually very little. Though the Hierarchy Rule does mean this data is an under-count, data from other sources indicate that it isn't much of an under count. The FBI's other data set, the National Incident-Based Reporting System (NIBRS) contains every

crime that occurs in an incident (i.e. it doesn't use the Hierarchy Rule). Using this we can measure how many crimes the Hierarchy Rule excludes (Most major cities do not report to NIBRS so what we find in NIBRS may not apply to them). In over 90% of incidents, only one crime is committed. Additionally, when people talk about "crime" they usually mean murder which, while incomplete to discuss crime, means the UCR data here is accurate on that measure.

23.4 Which crimes are included?

If you look back at the output when we ran `names(offenses_known_yearly_1960_2017)` you'll see that it produced five broad categories of columns. The first was information about the agency including population and geographic info, then came four columns with the same values except starting with "actual", "tot_clr", "clr_18", and "unfound". Following these starting values were 30 crime categories. We'll discuss what each of those starting values mean in a bit, let's first talk about which crimes are included and what that means for research.

23.4.1 Index Crimes

The Offenses Known and Clearances by Arrest data set contains information on the number of "Index Crimes" (sometimes called Part I crimes) reported to each agency. These index crimes are a collection of eight crimes that, for historical reasons based largely by perceived importance in the 1920's when the UCR program was first developed, are used as the primary measure of crime today. Other data sets in the UCR, such as the Arrests by Age, Sex, and Race data and the Hate Crime data have more crimes reported.

The crimes are, in order by the Hierarchy Rule -

1. Homicide
 - Murder and non-negligent manslaughter
 - Manslaughter by negligence
2. Rape

368CHAPTER 23. UNIFORM CRIME REPORT (UCR) - OFFENSES KNOWN AND CLEA.

- Rape
 - Attempted rape
3. Robbery
- With a firearm
 - With a knife or cutting instrument
 - With a dangerous weapon not otherwise specified
 - Unarmed - using hands, fists, feet, etc.
4. Aggravated Assault (assault with a weapon or causing serious bodily injury)
- With a firearm
 - With a knife or cutting instrument
 - With a dangerous weapon not otherwise specified
 - Unarmed - using hands, fists, feet, etc.
5. Burglary
- With forcible entry
 - Without forcible entry
 - Attempted burglary with forcible entry
6. Theft (other than of a motor vehicle)
7. Motor Vehicle Theft
- Cars
 - Trucks and buses
 - Other vehicles

8. Arson
9. Simple Assault

For a full definition of each of the index crimes see the FBI's Offense Definitions page [here](#).

Arson is considered an index crime but is not reported in this data - you need to use the separate Arson data set of the UCR to get access to arson counts. The ninth crime on that list, simple assault, is not considered an index crime but is nevertheless included in this data.

Each of the crimes in the list above, and their subcategories, are included in the UCR data. In most reports, however, you'll see them reported as the total number of index crimes, summing up categories 1-7 and reporting that as "crime". These index crimes are often divided into violent index crimes - murder, rape, robbery, and aggravated assault - and property index crimes - burglary, theft, motor vehicle theft.

23.4.2 The problem with using index crimes

The biggest problem with index crimes is that it is simply the sum of 8 (or 7 since arson data usually isn't available) crimes. Index crimes have a huge range in their seriousness - it includes both murder and theft. This is clearly wrong as 100 murders is more serious than 100 thefts. This is especially a problem as less serious crimes (theft mostly) are far more common than more serious crimes (in 2017 there were 1.25 million violent index crimes in the United States. That same year had 5.5 million thefts.). So index crimes under-count the seriousness of crimes. Looking at total index crimes is, in effect, mostly just looking at theft.

This is especially a problem because it hides trends in violent crimes. San Francisco, as an example, has had a huge increase in index crimes in the last several years. When looking closer, that increase is driven almost entirely by the near doubling of theft since 2011. During the same years, violent crime has stayed fairly steady. So the city isn't getting more dangerous but it appears like it is due to just looking at total index crimes.

Many researchers divide index crimes into violent and nonviolent categories,

which helps but is still not entirely sufficient. Take Chicago as an example. It is a city infamous for its large number of murders. But as a fraction of index crimes, Chicago has a rounding error worth of murders. Their 653 murders in 2017 is only 0.5% of total index crimes. For violent index crimes, murder makes up 2.2%. What this means is that changes in murder are very difficult to detect. If Chicago had no murders this year, but a less serious crime (such as theft) increased slightly, we couldn't tell from looking at the number of index crimes.

23.4.3 Rape definition change

Starting in 2013, rape has a new, broader definition in the UCR to include oral and anal penetration (by a body part or object) and to allow men to be victims. The previous definition included only forcible intercourse against a woman. As this revised definition is broader than the original one post-2013, rape data is not comparable to pre-2013 data.

23.5 Actual offenses, clearances, and unfounded offenses

For each crime we have four different categories indicating the number of crimes actually committed, the number cleared, and the number determined to not have occurred.

23.5.1 Actual

This is the number of offenses that occurred, simply a count of the number of crimes that month. For example if 10 people are murdered in a city the number of “actual murders” would be 10.

23.5.2 Total Cleared

A crime is cleared when an offender is arrested or when the case is considered cleared by exceptional means. When a single offender for a crime is arrested, that crime is considered cleared. If multiple people committed a crime, only a single person must be arrested for it to be cleared, and as the UCR data is at the offense level, making multiple arrests for an incident only counts as

one incident cleared. So if 10 people committed a murder and all 10 were arrested, it would report one murder cleared not 10. If only one of these people are arrested it would still report one murder cleared - the UCR does not even say how many people commit a crime.

A crime is considered exceptionally cleared if the police can identify the offender, have enough evidence to arrest the offender, know where the offender is, but is unable to arrest them. Some examples of this are the death of the offender or when the victim refuses to cooperate in the case.

Unfortunately this data does not differentiate between clearances by arrest or by exceptional means. For a comprehensive report on how this variable can be exploited to exaggerate clearance rates, see [this report by ProPublica](#) on exceptional clearances with rape cases.

23.5.3 Cleared Where All Offenders Are Under 18

This variable is very similar to Total Cleared except is only for offenses in which **every** offender is younger than age 18.

23.5.4 Unfounded

An unfounded crime is one in which a police investigation has determined that the reported crime did not actually happen. For example if the police are called to a possible burglary but later find out that a burglary did not occur, they would put it down as 1 unfounded burglary. This is based on police investigation rather than the decision of any other party such as a coroner, judge, jury, or prosecutor.

23.6 Number of months reported

UCR data is reported monthly though even agencies that decide to report their data may not do so every month. As we don't want to compare an agency which reports 12 months to one that reports fewer, the variable *number_of_months_reported* is way keep only agencies that report 12 months, or deal with those that report fewer.

372 CHAPTER 23. UNIFORM CRIME REPORT (UCR) - OFFENSES KNOWN AND CLEA.

```
table(offenses_known_yearly_1960_2017$number_of_months_reported)
#>
#>      0      1      2      3      4      5      6      7      8      9
#> 229843  2514  2522  2926  2633  2875  3876  3350  4013  4693  6
#>     11     12
#> 14940 678409
```

From our `table()` output it seems that when agencies do report, they tend to do so for all 12 months of the year. However, this variable is seriously flawed, and its name is quite misleading. In reality this variable is actually just whichever the last month reported was. If an agency reported every month of the year, meaning December is the last month, they would have a value of 12. If the agency **only** reported in December, they would also have a value of 12. While there are ways in the monthly data to measure actual number of months reported, these ways are also flawed. So be cautious about this data and particularly the value of this variable.

Appendix A

Useful resources

A.1 Learning R and coding issues

[R for Data Science](#) - This free online book provides a good introduction for R though it differs in several important ways from this book.

[Stack Overflow](#) - Stack Overflow is a website that answers programming-related questions. It's like the Yahoo Answers of programming. That said, a lot of the answer are bad. Some answers are overly confusing or provide code that you may not understand. You can use this source, but don't rely too heavily on it. Its search function isn't great so it's better to Google your question and choose the stackoverflow.com result.

A.2 Data

[National Archive of Criminal Justice Data \(NACJD\) - Crime Data](#) - NACJD is a site where you can download crime data, including many of the data sets we've worked on throughout this book. Nearly all FBI and BJS data sets are available on this site. If you need data for a research question, NACJD is a good place to start looking.

[Inter-university Consortium for Political and Social Research \(ICPSR\) - General Data](#) - Like NACJD, ICPSR allows you to download data from a variety of government and non-government sources for research. ICPSR holds a

wider variety of data than NACJD does (NACJD is part of ICPSR) so includes data that is not specifically crime-related.

Crime Data Tool - cleaned crime data - This is my personal site where you can look at graphs and tables of (primarily) agency-level data including offenses, arrests, assaults on officers, and prison information. On each page you can download a .csv file with the information displayed. On the [data page](#), there are links to all of the data sets I have cleaned and made available - this data is mostly UCR data but includes some other data such as the alcohol consumption data we explored in Chapter @ref(#graphing-intro).

Local crime data - The Police Foundation has put together a list of data sets made public by police agencies. This data is primarily data sets of crime incidents, calls for service, arrests, and officer-involved shootings though some other categories such as workforce demographics and assaults on officers are available.

Bibliography

- Jain, H. C., Singh, P., and Agocs, C. (2000). Recruitment, selection and promotion of visible-minority and Aboriginal police officers in selected Canadian police services. *Canadian Public Administration*, 43(1):46–74.
- Reaves, B. (1993). *Using NIBRS data to analyze violent crime*. US Department of Justice, Office of Justice Programs, Bureau of Justice.